Advances in Database Technology — EDBT 2020

23rd International Conference
on Extending Database Technology
Copenhagen, Denkmark, March 30–April 2, 2020
Proceedings

Editors

Angela Bonifati
Yongluan Zhou
Marcos Antonio Vaz Salles
Alexander Böhm
Dan Olteanu
George Fletcher
Arijit Khan
Bin Yang

open proceedings

Advances in Database Technology — EDBT 2020
Proceedings of the 23rd International Conference
on Extending Database Technology
Copenhagen, Denkmark, March 30–April 2, 2020

Series ISSN: 2367-2005

Editors

Angela Bonifati, Lyon 1 University, France
Yongluan Zhou, University of Copenhagen, Denmark
Marcos Antonio Vaz Salles, University of Copenhagen, Denmark
Alexander Böhm, SAP Research, Germany
Dan Olteanu, University of Oxford, United Kingdom
George Fletcher, Eindhoven University of Technology, The Netherlands
Arijit Khan, Nanyang Technnological University, Singapore
Bin Yang, Aalborg University, Denmark

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

OpenProceedings ISBN: 978-3-89318-083-7

DOI of this front matter: `10.5441/002/edbt.2020.01`

Foreword

The International Conference on Extending Database Technology (EDBT) is a leading international forum for database researchers, developers, and users to present and discuss novel and cutting-edge ideas and techniques, and to showcase tools and experiences related to data management. Data management is an essential enabling technology that has applications in several scientific, business and social communities, and runs on diverse technical platforms associated with the web, enterprises, clouds and mobile devices. The database community has a continuing tradition of contributing with models, algorithms and architectures to the set of tools and applications that enable day-to-day functioning of our societies. Faced with the broad challenges of today's applications, data management technology constantly broadens its reach, exploiting new hardware and software to achieve innovative results and embracing new challenges in the years to come.

EDBT 2020 solicited submissions of original research contributions, descriptions of industrial solutions and applications, and proposals for tutorials and software demonstrations. We encouraged submissions of research papers related to all aspects of data management. We also encouraged submissions of visionary papers as well as innovative system papers and experimental analyses papers. In addition to long research paper submissions, EDBT 2020 again encouraged the submission of short research papers, which provide an excellent opportunity to describe significant work or research in progress that can foster the discussions at the conference. Short papers are presented as posters at plenary poster sessions of the conference. This year, they will also be communicated at a plenary lightning talks session.

The program committees of EDBT accepted 30 out of 151 submitted regular research papers, resulting in an acceptance rate of 20% for the research track; 26 out of 85 submitted short research papers, resulting in an acceptance rate of 31% for short research papers; 16 out of 37 demos, resulting in an acceptance rate of 43% for the demonstration track; and 10 out of 37 industrial and application papers, resulting in an acceptance rate of 27% for industrial and application papers. The papers will be presented in eight research paper sessions, three industrial and application sessions, as well as two plenary poster and demonstration sessions.

The program additionally features five workshops, one of which is the well-established DOLAP workshop that has successfully been co-located with EDBT since many years. Finally, the conference program includes four tutorials and an EDBT and ICDT joint session on climate change.

I would like to thank all authors for their contributions: a successful conference crucially depends on high-quality submissions. I also would like to thank all senior reviewers and reviewers for serving on the EDBT 2020 program committee, in particular for the high quality and timely handling of all reviews and discussions. This community service requires a lot of work on a tight schedule, and is what makes our research community function and ensures the sustained impact of our research. Thanks to their valuable effort we can look forward to an exciting program and attractive EDBT conference in Copenhagen from March 30–April 2, 2020.

A warm thanks to Anastasia Alaimaki and Tamer Özsu for serving on the Test-of-Time Award committee to select the paper from EDBT 2010 that has had the most lasting influence. Wook-shin Han, Erhard Rahm and Nesime Tatbul generously accepted to serve on the Best Paper committee. The EDBT 2020 program is the result of the joint effort of many people who shared their experience and time to contribute to the EDBT 2020 program and make the conference a great success. Alexander Boehm served as PC chair for industrial and application papers; George Fletcher as PC chair for the demonstration track; Dan Olteanu as tutorial chair; Alexandra Poulovassilis as workshop chair. My warmest thanks to all these people. The general chairs, Yongluan Zhou and Marcos Vaz Salles and the other local organizers worked hard to make all necessary arrangements for a successful event. Special thanks to Arijt Khan, the EDBT proceedings chair; Davide Mottin, the publicity chair; Bin Yang, the local executive chair; Boris Düdder, the sponsorship chair; Nanna Højholt, the finance chair and Yiwen Wang, the website chair, for tirelessly finding solutions for all our requests. Norman Paton was most helpful in advising and coordinating with the EDBT Executive Board. Last but not least, I would like to thank Marc H. Scholl for assembling the EDBT proceedings on openproceedings.org I hope that you find EDBT 2020 inspiring, enriching, and enjoyable and look forward to meeting you in Copenhagen.

Angela Bonifati
EDBT 2020 Program Chair

Program Committee Members

Research Program Committee

Angela Bonifati (Lyon 1 U, France) – Chair

Senior Program Committee Members

Karl Aberer (EPFL Lausanne, Switzerland)
Walid Aref (Purdue U, USA)
Michael Benedikt (U Oxford, UK)
Michael Böhlen (U Zurich, Switzerland)
K. Selcuk Candan (Arizona State U, USA)
Kevin Chang (U Illinois at Urbana-Champaign, USA)
Vassilis Christophides (INRIA, France)
Daniel Deutch (Tel Aviv U, Israel)
Floris Geerts (U Antwerp, Belgium)
Jan Hidders (VU Brussel, Belgium)
Katja Hose (Aalborg U, Denmark)
Christoph Koch (EPFL Lausanne, Switzerland)
Georgia Koutrika (Athena Research Center, Greece)
Ulf Leser (Humboldt-U Berlin, Germany)
Guoliang Li (Tsinghua U, China)
Chengkai Li (U Texas at Arlington, USA)
Eric Lo (Chinese U Hong Kong, China)
Evaggelia Pitoura (U Ioannina, Greece)
Louiqa Raschid (U Maryland, USA)
Sherif Sakr (U Tartu, Estonia)
Semih Salihoglu (U Waterloo, Canada)
Kai-Uwe Sattler (TU Ilmenau, Germany)
Arash Termehchy (Oregon State U, USA)
Riccardo Torlone (Roma Tre U, Italy)
Peter Triantafillou (U Warwick, UK)
Yannis Velegrakis (Utrecht U, The Netherlands)

Program Committee Members

Bernd Amann (Sorbonne U – LIP6, France)
Akhil Arora (EPFL Lausanne, Switzerland)
Elena Baralis (Politecnico di Torino, Italy)
Denilson Barbosa (U Alberta, Canada)
Senjuti Basu Roy (New Jersey Inst. of Techn., USA)
Luigi Bellomarini (Banca d'Italia, Italy)
Sonia Bergamaschi (U Modena Reggio Emilia, Italy)
Laure Berti-Equille (IRD, France)
Arnab Bhattacharya (IIT Kanpur, India)
Luc Bouganim (INRIA-UVSQ, France)
Andrea Calì (U London, Birkbeck College, UK)
Bogdan Cautis (Paris Sud U, France)
Lei Chen (Hong Kong U Sc. & Techn., China)
Dario Colazzo (Paris Dauphine U, France)
Bin Cui (Peking U, China)
Alfredo Cuzzocrea (U Calabria, Italy)
Sabrina De Capitani di Vimercati (U Milan, Italy)
Stefania Dumbrava (ENSIIE Rennes, France)
Donatella Firmani (Roma Tre U, Italy)
Rainer Gemulla (U Mannheim, Germany)
Paolo Guagliardo (U Edinburgh, UK)
Xi He (U Waterloo, Canada)
Xin Huang (Hong Kong Baptist U, China)
Zsolt Istvan (IMDEA Software Institute, Spain)
Panos Kalnis (King Abdullah UST, Saudi Arabia)
Vana Kalogeraki (Athens U Ec. & Busin., Greece)
Verena Kantere (U Ottawa, Canada)
Yaron Kanza (AT&T Labs – Research, USA)
Asterios Katsifodimos (Delft UT, The Netherlands)
Xiang Lian (Kent State U, USA)
Ping Lu (Beihang U, China)
Paolo Missier (Newcastle U, USA)
Davide Mottin (Aarhus U, Denmark)
Behrooz Omidvar-Tehrani (Grenoble Alpes, France)
Eric Peukert (Leipzig U, Germany)
Holger Pirk (Imperial College London, UK)
Dimitris Plexousakis (Institute of CS, FORTH, Greece)
Giuseppe Polese (U Salerno, Italy)
Arnau Prat (U Politècn. Catalunya, Spain)
Mohammad Sadoghi (UC Davis)
Carlo Sartiani (U della Basilicata, Italy)
Stefanie Scherzinger (OTH Regensburg, Germany)
Petra Selmer (Neo4j, UK)
Juan F. Sequeda (Capsenta Labs, USA)
Hala Skaf-Molli (U Nantes, France)
Kostas Stefanidis (U Tampere, Finland)
Gábor Szárnyas (Budapest U Tech. & Eco., Hungary)
Ernest Teniente (U Politècn. Catalunya, Spain)
Jens Teubner (TU Dortmund, Germany)
Farouk Toumani (U Clermont Auvergne, France)
Anthony K. H. Tung (National U Singapore)
Wendy Hui Wang (Stevens Inst. of Techn., USA)
Nikolay Yakovets (TU Eindhoven, The Netherlands)
Demetrios Zeinalipour (U Cyprus, Greece)

Conference Organization

General Chairs
Yongluan Zhou, University of Copenhagen, Denmark
Marcos Antonio Vaz Salles, University of Copenhagen, Denmark

EDBT Program Chair
Angela Bonifati, Lyon 1 University, France

ICDT Program Chair
Carsten Lutz, University of Bremen, Germany

EDBT Industrial/Application Chair
Alexander Böhm, SAP Research, Germany

EDBT Demonstrations Chair
George Fletcher, Eindhoven University of Technology, The Netherlands

Tutorial Chair
Dan Olteanu, University of Oxford, United Kingdom

Workshops Chair
Alex Poulovassilis, Birbeck University of London, United Kingdom

EDBT Proceedings Chair
Arijt Khan, Nanyang Technnological University, Singapore

ICDT Proceedings Chair
Jean Christoph Jung, University of Bremen, Germany

Local Executive Chair
Bin Yang, Aalborg University, Denmark

Sponsorship Chairs
Boris Düdder, University of Copenhagen, Denmark

Publicity Chair
Davide Mottin, Aarhus University, Denmark

Finance Chair
Nanna Højholt, University of Copenhagen, Denmark

Website Chair
Yiwen Wang, University of Copenhagen, Denmark

Test-of-Time Award

Established in 2014, the Test-of-Time Award awarded by the Extended Database Technology (EDBT) Conference recognizes papers presented at 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 EDBT 2020 Test of Time Award committee was formed by Anastasia Ailamaki (Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland), Tamer Oszu (University of Waterloo, Canada), and Angela Bonifati (Lyon 1 University, France).

After careful consideration, the committee has decided to select the following paper from the EDBT 2010 conference as the EDBT ToT Award winner for 2020:

Optimizing joins in a map-reduce environment

by Foto Afrati and Jeff Ullman

published in EDBT 2010 Proceedings, pp. 99–110, DOI: 10.1145/1739041.1739056.

This paper presented optimization strategies for executing multi-way joins in a map-reduce environment. It focused on large-scale data and provided algorithms to choose the number of map-keys and shares in order to minimize the communication cost among the map and reduce processes.
The committee members agreed that this paper clearly pioneered the field of join processing in map-reduce environments. It has triggered substantial follow-up research and impact on big data processing in parallel and distributed architectures.

The EDBT Test-of-Time award for 2020 will be presented during the EDBT/ICDT 2020 Conference in Copenhagen, Denmark, as part of the Awards session on March 31, 2020.

Best Paper Award

The EDBT 2020 Best Paper Award committee was formed by Wook-shin Han (Postech, Korea), Erhard Rahm (University of Leipzig, Germany), Nesime Tatbul (Intel & MIT, USA), and Angela Bonifati (Lyon 1 University, France). After careful consideration, the committee has decided to select the following paper as the EDBT Best Paper for 2020:

Provenance for Probabilistic Logic Programs

by Shaobo Wang, Hui Lyu, Jaichi Zhang, Chenyuan Wu, Xinyi,Chen, Wenchao,Zhou, Boon Thau Loo, Susan B. Davidson, Chen Chen.

DOI: 10.5441/002/edbt.2020.14

Abstract: Despite the emergence of probabilistic logic programming (PLP) languages for data driven applications, there are currently no debugging tools based on provenance for PLP programs. In this paper, we propose a novel provenance model and system, called P3 (Provenance for Probabilistic logic Programs) for analyzing PLP programs. P3 enables four types of provenance queries: tra- ditional explanation queries, queries for finding the set of most important derivations within an approximate error, top-K most influential queries, and modification queries that enable us to modify tuple probabilities with fewest modifications to program or input data. We apply these queries into real-world scenar- ios and present theoretical analysis and practical algorithms for such queries. We have developed a prototype of P3, and our evaluation on real-world data demonstrates that the system can support a wide-range of provenance queries with explainable results. Moreover, the system maintains provenance and execute queries efficiently with low overhead.

The EDBT Best Paper Awards for 2020 will be presented during the EDBT/ICDT 2020 Conference in Copenhagen, Denmark, as part of the Awards session on March 31, 2020.

Table of Contents

Foreword . . . i

Program Committee Members . . . ii

Conference Organization . . . iv

Test-of-Time Award . . . v

Best Paper Award . . . vi

Table of Contents . . . vii

Research Papers

Automatic Canonical Utterance Generation for Task-Oriented Bots from API Specifications
Mohammad-Ali Yaghoub-Zadeh-Fard, Boualem Benatallah, Shayan Zamanirad . . . 1

Efficient Continuous Multi-Query Processing over Graph Streams
Lefteris Zervakis, Vinay Setty, Christos Tryfonopoulos, Katja Hose . . . 13

Zooming Out on an Evolving Graph
Amir Aghasadeghi, Vera Moffitt, Sebastian Schelter, Julia Stoyanovich . . . 25

Efficient Search for Multi-Scale Time Delay Correlations in Big Time Series Data
Nguyen Ho, Torben Bach Pedersen, Van Long Ho, Mai Vu . . . 37

Dynamic Query Refinement for Interactive Data Exploration
Alexander Kalinin, Ugur Cetintemel, Zheguang Zhao, Stanley Zdonik . . . 49

Micro Analysis to Enable Energy-Efficient Database Systems
Chen Yang, Yongjie Du, Zhihui Du, Meng Xiaofeng . . . 61

Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication
Thamir Qadah, Suyash Gupta, Mohammad Sadoghi . . . 73

Ensemble Grammar Induction For Detecting Anomalies in Time Series
Yifeng Gao, Jessica Lin, Constantin Brif . . . 85

Incremental Based Framework for Efficient Top-K Similarity Search in Interactive Data Analysis Sessions
Amit Somech, Tova Milo, Oded Elbaz . . . 97

pgFMU: Integrating Data Management with Physical System Modelling
Olga Rybnytska, Laurynas Šikšnys, Torben Bach Pedersen, Bijay Neupane . . . 109

Differentially-Private Next-Location Prediction with Neural Networks
Ritesh Ahuja, Gabriel Ghinita, Cyrus Shahabi . . . 121

Explaining Differences Between Unaligned Table Snapshots
Manuel Fink, Christian Meilicke, Heiner Stuckenschmidt . . . 133

Provenance for Probabilistic Logic Programs
Shaobo Wang, Hui Lyu, Jiachi Zhang, Chenyuan Wu, Xinyi Chen, Wenchao Zhou, Boon Thau Loo, Susan B. Davidson, Chen Chen . . . 145

Improved Cardinality Estimation by Learning Queries Containment Rates
Rojeh Hayek, Oded Shmueli . . . 157

Automated Management of Indexes for Dataflow Processing Engines in IaaS Clouds
Herald Kllapi, Ilia Pietri, Verena Kantere, Yannis Ioannidis . . . 169

Optimal Histograms with Outliers
Rachel Behar, Sara Cohen 181

BalanSiNG: Fast and Scalable Generation of Realistic Signed Networks
Jinhong Jung, Ha-Myung Park, U Kang 193

Distributed Similarity Joins over Top-K Rankings
Evica Milchevski, Sebastian Michel 205

MV-PBT: Multi-Version Indexing for Large Datasets and HTAP Workloads
Christian Riegger, Tobias Vincon, Robert Gottstein, Ilia Petrov 217

Lineage-Preserving Anonymization of the Provenance of Collection-Based Workflows
Khalid Belhajjame 229

Sharing Computations for User-Defined Aggregate Functions
Chao Zhang, Farouk Toumani 241

Tracing nested data with structural provenance for big data analytics
Ralf Diestelkämper, Melanie Herschel 253

A Parallel and Distributed Approach for Diversified Top-k Best Region Search
Hamid Shahrivari, Matthaios Olma, Odysseas Papapetrou, Dimitrios Skoutas, Anastasia Ailamaki 265

Data Curation with Deep Learning
Saravanan Thirumuruganathan, Nan Tang, Mourad Ouzzani, Anhai Doan 277

Fairness in Clustering with Multiple Sensitive Attributes
Savitha Abraham, Deepak P, Sowmya S Sundaram 287

Ontology-Based RDF Integration of Heterogeneous Data
Maxime Buron, Francois Goasdoue, Ioana Manolescu, Marie-Laure Mugnier 299

Manually Detecting Errors for Data Cleaning Using Adaptive Crowdsourcing Strategies
Haojun Zhang, Chengliang Chai, Anhai Doan, Paris Koutris, Esteban Arcaute 311

Publishing Video Data with Indistinguishable Objects
Han Wang, Yuan Hong, Yu Kong, Jaideep Vaidya 323

Efficient PrefDiv Algorithms for Effective Top-k Result Diversification
Xiaoyu Ge, Panos Chrysanthis 335

Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins
Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Christoph Anneser, Eleni Tzirita Zacharatou, Harish Doraiswamy, Peter Boncz, Thomas Neumann, Alfons Kemper 347

Short papers

DISGD: A Distributed Shared-nothing Matrix Factorization for Large Scale Online Recommender Systems
Heidy Hazem, Ahmed Awad, Ahmed Hassan, Sherif Sakr 359

Forming Compatible Teams in Signed Networks
Ioannis Kouvatis, Konstantinos Semertzidis, Maria Zerva, Evaggelia Pitoura, Panayiotis Tsaparas 363

A Learning Based Approach to Predict Shortest-Path Distances
Jianzhong Qi, Wei Wang, Rui Zhang, Zhuowei Zhao 367

Optimizing Data Movement with Near-Memory Acceleration of In-memory DBMS
Donghun Lee, Andrew Chang, Minseon Ahn, Jongmin Gim, Jungmin Kim, Jaemin Jung, Kangwoo Choi, Vincent Pham, Oliver Rebholz, Krishna Malladi, Yang-Seok Ki 371

ODSA: Open Database Storage Access
James Wagner, Alexander Rasin, Dai Hai Ton That, Tanu Malik, Jonathan Grier 375

An Integrated Graph Neural Network for Supervised Non-obvious Relationship Detection in Knowledge Graphs
Phillipp Müller, Xiao Qin, Balaji Ganesan, Nasrullah Sheikh, Berthold Reinwald 379

Outlier detection in multivariate functional data based on a geometric aggregation
Clément Lejeune, Josiane Mothe, Olivier Teste 383

REMI: Mining Intuitive Referring Expressions on Knowledge Bases
Luis Galárraga, Julien Delaunay, Jean-Louis Dessalles 387

The Case for Hybrid Succinct Data Structures
Christoph Anneser, Andreas Kipf, Harald Lang, Thomas Neumann, Alfons Kemper 391

FairPrep: Promoting Data to a First-Class Citizen in Studies on Fairness-Enhancing Interventions
Sebastian Schelter, Yuxuan He, Jatin Khilnani, Julia Stoyanovich 395

Partially Materializable Delta Trees for Efficient Data Wrangling of Semi-Structured Contents
Nico Schäfer, Sebastian Michel 399

Towards Fine-Grained Data Access Control Through Active Peer Probing
Yael Amsterdamer, Osnat Drien 403

The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries
Angjela Davitkova, Evica Milchevski, Sebastian Michel 407

Retro: Relation Retrofitting For In-Database Machine Learning on Textual Data
Michael Günther, Maik Thiele, Wolfgang Lehner 411

Revisiting the Theory and Practice of Database Cracking
Fatemeh Zardbani, Peyman Afshani, Panagiotis Karras 415

Boosting Blocking Performance in Entity Resolution Pipelines: Comparison Cleaning using Bloom Filters
Leonardo Gazzarri, Melanie Herschel 419

Disco: Efficient Distributed Window Aggregation
Lawrence Benson, Philipp M. Grulich, Steffen Zeuch, Volker Markl, Tilmann Rabl 423

Explaining Missing Query Results in Natural Language
Daniel Deutch, Nave Frost, Amir Gilad, Tomer Haimovich 427

A Context-based Approach for Partitioning Big Data
Sara Migliorini, Alberto Belussi, Elisa Quintarelli, Damiano Carra 431

SlideSide: A fast Incremental Stream Processing Algorithm for Multiple Queries
Georgios Theodorakis, Peter Pietzuch, Holger Pirk 435

Efficient Enumeration of Four Node Graphlets at Trillion-Scale
Yudi Santoso, Venkatesh Srinivasan, Alex Thomo 439

Band Joins for Interval Data
Panagiotis Bouros, Konstantinos Lampropoulos, Dimitrios Tsitsigkos, Nikos Mamoulis, Manolis Terrovitis 443

Personalized Page Rank on Knowledge Graphs: Particle Filtering is all you need!
Denis Gallo, Matteo Lissandrini, Yannis Velegrakis 447

Towards Planning of Regular Queries with Memory
Thomas Mulder, Nikolay Yakovets, George Fletcher 451

Accurate Demand Forecasting for Retails with Deep Neural Networks
Shanhe Liao, Weixiong Rao 455

Efficient Skyline Computation in High-Dimensionality Domains
Rui Liu, Dominique Li 459

Industry and Applications Papers

Entity Matching with Transformer Architectures - A Step Forward in Data Integration
Ursin Brunner, Kurt Stockinger . . . 463

Gallery: A Machine Learning Model Management System at Uber
Chong Sun, Nader Azari, Chintan Turakhia . . . 474

Diverse User Selection for Opinion Procurement
Yael Amsterdamer, Oded Goldreich . . . 486

Elastic Scaling in VectorH
Steffen Kläbe, Kai-Uwe Sattler, Stephan Baumann, Michael Rink . . . 498

Fairness in Online Jobs: A Case Study on TaskRabbit and Google
Sihem Amer-Yahia, Shady Elbassuoni, Ahmad Ghizzawi, Ria Mae Borromeo, Emilie Hoareau, Philippe Mulhem . . 510

Inventory Reduction via Maximal Coverage in E-Commerce
Shay Gershtein, Tova Milo, Slava Novgorodov . . . 522

Cost Estimation Across Heterogeneous SQL-Based Big Data Infrastructures in Teradata IntelliSphere
Kassem Awada, Mohamed Eltabakh, Conrad Tang, Mohammed Al-Kateb, Sanjay Nair, Grace Au . . . 534

Fast Entropy Maximization for Selectivity Estimation of Conjunctive Predicates on CPUs and GPUs
Diego Havenstein, Peter Lysakovski, Norman May, Guido Moerkotte, Gabriele Steidl . . . 546

Weaving Enterprise Knowledge Graphs: The Case of Company Ownership Graphs
Paolo Atzeni, Luigi Bellomarini, Michela Iezzi, Emanuel Sallinger, Adriano Vlad . . . 555

Expanding Query Answers on Medical Knowledge Bases
Chuan Lei, Vasilis Efthymiou, Rebecca Geis, Fatma Ozcan . . . 567

Demonstrations

VAP: A Visual Analysis Tool for Energy Consumption Spatio-temporal Pattern Discovery
Xiufeng Liu, Zhibin Niu, Yanyan Yang, Junqi Wu, Dawei Cheng, Xin Wang . . . 579

Chronos: The Swiss Army Knife for Database Evaluations
Marco Vogt, Alexander Stiemer, Sein Coray, Heiko Schuldt . . . 583

skyex: an R Package for Entity Linkage
Suela Isaj, Torben Bach Pedersen . . . 587

Data Quality Checking for Machine Learning with MeSQuaL
Ugo Comignani, Noël Novelli, Laure Berti-Equille . . . 591

MALOS: A Movement-Aware Location Selection System
Di Yang, Hui Li, Meng Wang, Dan Li, Jiangtao Cui . . . 595

RRAMEN: An Interactive Tool for Evaluating Choices and Changes in Transportation Networks
Camila Ferreira Costa, Theodoros Chondrogiannis, Mario A. Nascimento, Panagiotis Bouros . . . 599

JedAI^3 : beyond batch, blocking-based Entity Resolution
George Papadakis, Leonidas Tsekouras, Emmanouil Thanos, Nikiforos Pittaras, Giovanni Simonini, Dimitrios Skoutas, Paul Isaris, George Giannakopoulos, Themis Palpanas, Manolis Koubarakis . . . 603

RetroLive: Analysis of Relational Retrofitted Word Embeddings
Michael Günther, Maik Thiele, Erik Nikulski, Wolfgang Lehner . . . 607

RulER: Scaling Up Record-level Matching Rules
Luca Gagliardelli, Giovanni Simonini, Sonia Bergamaschi . . . 611

Schema Mapping Generation in the Wild: A Demonstration with Open Government Data
Lacramioara Mazilu, Nikolaos Konstantinou, Norman Paton, Alvaro A. A. Fernandes 615

EPIQUE: Extracting Meaningful Science Evolution Patterns from Large Document Archives
Ke Li, Hubert Naacke, Bernd Amann 619

Task-Tuning in Privacy-Preserving Crowdsourcing Platforms
Joris Duguépéroux, Antonin Voyez, Tristan Allard 623

Scaling a Public Transport Monitoring System to Internet of Things Infrastructures
Haralampos Gavriilidis, Adrian Michalke, Laura Mons, Steffen Zeuch, Volker Markl 627

Governor: Operator Placement for a Unified Fog-Cloud Environment
Ankit Chaudhary, Steffen Zeuch, Volker Markl 631

Human-in-the-Loop Schema Inference for Massive JSON Datasets
Mohamed-Amine Baazizi, Clément Berti, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani 635

SQL Query Processing Using an Integrated FPGA-based Near-Data Accelerator in ReProVide
Lekshmi B.G., Andreas Becher, Klaus Meyer-Wegener, Stefan Wildermann, Jürgen Teich 639

Tutorials

Declarative Languages for Big Streaming Data
Riccardo Tommasini, Sherif Sakr, Emanuele Della Valle, Hojjat Jafarpour 643

Entity Resolution: Past, Present and Yet-to-Come
George Papadakis, Ekaterini Ioannou, Themis Palpanas 647

Fairness in Rankings and Recommenders
Evaggelia Pitoura, Georgia Koutrika, Kostas Stefanidis 651

NoSQL Schema Evolution and Data Migration: State-of-the-Art and Opportunities
Uta Störl, Meike Klettke, Stefanie Scherzinger 655

Errata Notes

Erratum for Discovering Order Dependencies through Order Compatibility (EDBT 2019)
Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, Divesh Srivastava 659

Automatic Canonical Utterance Generation for Task-Oriented Bots from API Specifications

Mohammad-Ali Yaghoub-Zadeh-Fard
m.yaghoubzadehfard@unsw.edu.au
University of New South Wales, Sydney
Sydney, NSW

Boualem Benatallah
b.benatallah@cse.unsw.edu.au
University of New South Wales, Sydney
Sydney, NSW

Shayan Zamanirad
shayanz@cse.unsw.edu.au
University of New South Wales, Sydney
Sydney, NSW

ABSTRACT

With the mind-blowing development of REST (REpresentational State Transfer) APIs (Application Programming Interfaces), many applications have been designed to harness their potential. As such, bots have recently become interesting interfaces to connect humans to APIs. Supervised approaches for building bots rely upon a large set of user utterances paired with API methods. Collecting such pairs is typically done by obtaining initial utterances for a given API method and paraphrasing them to obtain new variations. However, existing approaches for generating initial utterances (e.g., creating sentence templates) do not scale and are domain-specific, making bots expensive to maintain. The automatic generation of initial utterances can be considered as a supervised translation task in which an API method is translated into an utterance. However, the key challenge is the lack of training data for training domain-independent models. In this paper, we propose *API2CAN*, a dataset containing 14,370 pairs of API methods and utterances. The dataset is built by processing a large number of public APIs. However, deep-learning-based approaches such as sequence-to-sequence models require larger sets of training samples (ideally millions of samples). To mitigate the absence of such large datasets, we formalize and define resources in REST APIs, and we propose a delexicalization technique (by converting an API method and initial utterances to tagged sequences of resources) to let deep-learning-based approaches learn from such datasets.

1 INTRODUCTION

Much of the information we receive about the world is API-regulated. Essentially, APIs are used for connecting devices, managing data, and invoking services [1–3]. In particular, because of its simplicity, REST is the most dominant approach for designing Web APIs [4–6]. Meanwhile, thanks to the advances in machine learning and availability of web services, building natural language interfaces has gained attention by both researchers and organizations (e.g., Apple's Siri, Google's Virtual Assistant, IBM's Watson, Microsoft's Cortana). Natural language interfaces and virtual assistants serve a wide range of tasks by mapping user utterances (also called user expressions) into appropriate operations. Examples include reporting weather, booking flights, controlling home devices, and querying databases [1, 7–9]. Increasingly, organizations have started or plan to use capabilities arising from advances in cognitive computing to increase productivity, automate business processes, and extend the breadth of their business offering.

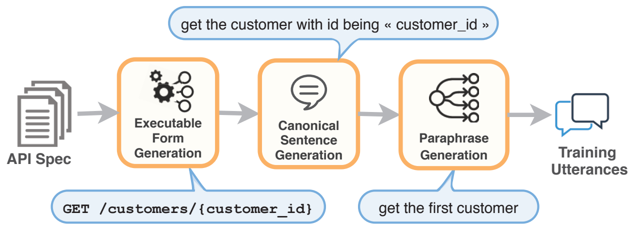
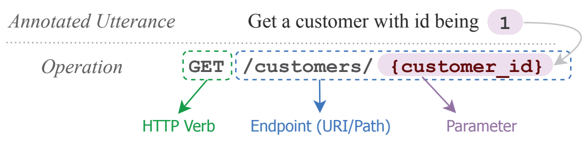

get the customer with id being « customer_id »
API Spec
Executable Form Generation
Canonical Sentence Generation
Paraphrase Generation
Training Utterances
GET /customers/{customer_id}
get the first customer

Figure 1: Classical Training Data Generation Pipeline

To serve users' requests, virtual assistants often employ supervised models which require a large set of natural language utterances (e.g., *"get a customer with id being 1"*) paired with their corresponding executable forms (e.g., SQL queries, API calls, logical forms). The training pairs are used to learn the mappings between user utterances and executable forms. Given the popularity of REST APIs (based on the well-known HTTP protocol), we focus on one of the most common types of executable forms called *operations*. In REST APIs, an operation (also called API method) consists of an HTTP verb (e.g., GET, POST), an endpoint (e.g., `/customers`), and a set of parameters[1] (e.g., query parameters). Figure 2 shows different parts of a REST request in HTTP.

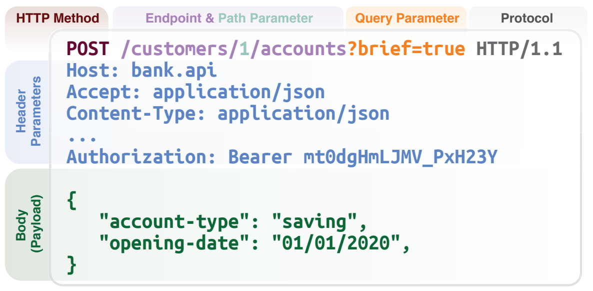
An annotated utterance is a corresponding natural language expression to an operation in which API parameters are labeled:



As shown in Figure 1, collecting such pairs is typically done in two steps: (i) obtaining initial utterances for each operation; and (ii) paraphrasing the initial utterances either automatically or manually (e.g., crowdsourcing) to new variations in order to live up to the richness in human languages [1, 7, 8]. Paraphrasing approaches (e.g., crowdsourcing, automatic paraphrasing systems) have made the second step less costly [7, 8, 10], but existing approaches for generating the initial sentences are still limited, and they are not scalable [8].

Existing solutions for generating initial utterances (also called canonical utterances) often involve employing domain experts to generate hand-crafted domain-specific grammars or templates [1, 8, 11]. Almond virtual assistant, as an example, relies on hand-crafted rules to generate initial utterances [8]. Such approaches

[1]In this paper, to show parameters of an operation, we use curly brackets with two parts separated by semicolon (e.g., {customer_id:1}): the first part gives the name of the parameter and the second part indicates a sample value for the parameter



```
POST /customers/1/accounts?brief=true HTTP/1.1
Host: bank.api
Accept: application/json
Content-Type: application/json
...
Authorization: Bearer mt0dgHmLJMV_PxH23Y

{
   "account-type": "saving",
   "opening-date": "01/01/2020",
}
```

Figure 2: Example of an HTTP POST Request

are domain-specific and costly since rules are generated by experts [1, 8, 11]. In other words, adding new APIs to a particular virtual assistant requires manual efforts for revising hand-crafted grammars to generate training samples for new domains. With the growing number of APIs and modifications of existing APIs, automated bot development has become paramount, especially for virtual assistants which aim at servicing a wide range of tasks [1, 8].

Supervised approaches such as sequence-to-sequence models can be used for translating *operations* to *canonical utterances*. However, the key challenge is the lack of training data (pairs of operations and canonical utterances) for training domain-independent models. In this paper, we propose *API2CAN*, a dataset containing 14,370 pairs of operations and canonical utterances. The dataset is generated automatically by processing a large set of OpenAPI specifications[2] (based on the description/summary of each operation). However, deep-learning-based approaches such as sequence-to-sequence models require much larger sets of samples to train from (ideally millions of training samples). That is to say, sequence-to-sequence models are easy to overfit small training datasets, and issues such as out of vocabulary words (OOV) can negatively impact their performance. To overcome such issues, we propose a delexicalization technique to convert an operation to a sequence of predefined tags (e.g., singleton, collection) based on RESTful principles and design guidelines (e.g., use of plural names for a collection of resources, using HTTP verbs). In summary, our contribution is three-folded:

- **A Dataset.** We propose a dataset called *API2CAN*, containing annotated canonical templates (a canonical utterance in which parameter values have been replaced with placeholders e.g., *"get a customer with id being «id»"*) for 14,370 operations of 985 REST APIs. We automatically built the dataset by processing a large set of OpenAPI specifications, and we converted operation descriptions to canonical templates based on a set of heuristics (e.g., extracting a candidate sentence, injecting parameter placeholders in the method descriptions, removing unnecessary words). We then split the dataset into three parts (test, train, and validation sets).
- **A Delexicalization Technique.** Deep-learning algorithms such as sequence-to-sequence models require millions of training pairs to learn from. To assist such models to learn from smaller datasets, we propose a delexicalization technique to convert input (operation) and output (canonical template) of such models to a sequence of predefined tags called *resource identifiers*. The proposed approach is based on the concept of *resource* in RESTful design. Particularly, we formalize various kinds of resources (e.g., collection, singleton) in REST APIs. Next, using the identified resource types, we propose a delexicalization technique to replace mentions of each *resource* (e.g., customers) with a corresponding resource identifier (e.g., Collection_1). As such, for a given operation (e.g., `GET /customers/{customer_id}`), the model learns to translate the delexicalized operation (e.g., `GET Collection_1 Singleton_1`) to a delexicalized canonical templates (e.g., "get a Collection_1 with Singleton_1 being «Singleton_1»"). A resource identifier consists of two parts: (1) the type of resource and (2) a number n which indicates n-th occurrence of a resource type in a given operation. Resource identifiers are then used in time of translation to lexicalize the output of the sequence-to-sequence model (e.g., "get a Collection_1 with Singleton_1 being «Singleton_1»") to generate a canonical template (e.g., "get a customer with customer id being «customer_id»"). Delexicalization is done to reduce the impact of OOV and force the model to learn the pattern of translating resources in an operation to a canonical template (rather than translating a sequence of words).
- **Analysis of Public REST APIs.** We analyze and give insight into a large set of public REST APIs. It includes how REST APIs are designed in practice and drifts from the RESTful principles (design guidelines such as using plural names, appropriate use of HTTP verbs). We also provide inside into distribution of parameters (e.g., parameter types and location) and how values can be sampled various types of parameters to generate canonical utterances out of canonical templates using API specifications (e.g., example values, similar parameters with sample values). Automatic sampling values for parameters is essential for automatic generation of canonical utterances because current bot development platforms (e.g., IBM Watson) require annotated utterances (not canonical templates with placeholders).

2 RELATED WORK

REST APIs. REST is an architectural style and a guideline of how to use the HTTP protocol[3] for designing Web services [12]. RESTful web services leverage HTTP using specific architectural principles (i,e., addressability, uniform interface) [13]. Since REST is just a guideline without standardization, it is not surprising that API developers only partially follow the guidelines or interpret REST in their own ways [5]. In particular, this paper is built upon one of the most important principles in REST, namely the *uniform interface* principle. According to this principle, resources must be accessed and manipulated using proper HTTP methods (e.g., DELETE, GET) and status codes (e.g. using "201" to show a resource is created, and "404" to show resource does not exist). The uniform interface requires API to be developed uniformly to ensure that API users can understand the functionality of each operation without reading tedious and long descriptions. To ensure uniform interface, API developers are required to follow design patterns (e.g., using plural names to name collection of resources, using lowercase letters in paths). Existing works have listed not only those patterns but also anti-patterns in designing interfaces of REST APIs [5, 14, 15]. Examples of anti-patterns

[2]previously known as Swagger specification

[3]REST isn't protocol-specific, but it is designed over HTTP nowadays

also include using underline in paths and adding file extensions in paths [4, 6].

In this paper, we build upon existing works on designing interfaces for REST APIs. In particular, we formalize resource types based on patterns and anti-patterns recognized in prior works and built a resource tagger to annotate the segments of a given operation with resource types.

Conversational Agents and Web APIs. Research on conversational agents (e.i., bots, chatbots, dialog systems, virtual assistants) dates back to decades ago [16]. However, there have been only a few targeting web APIs, particularly because of the lack of training samples [1–3]. In absence of training data, operations descriptions (e.g., having long descriptions containing unnecessary information) have been used for detecting the user's intent [3]. However, operations often lack proper descriptions, and operations descriptions may share the same vocabularies in a single API, making it difficult for the bot to differentiate between operations [3]. Moreover, these descriptions are rarely similar to the natural language utterances which are used by bot users to interact with bots. That is to say, these descriptions are originally written to document operations (not intended to be used for training bots) [2, 3].

Other approaches rely on domain experts for generating initial utterances [1, 7, 8]. These approaches include (i) natural language templates (a canonical utterance with placeholders) which are written by experts [17], and (ii) domain-specific grammars such as rules written for semantic parsers [1, 8]. Thus in either approach, manual effort is required to modify the templates of grammar if API specifications are changed. In the template-based approach, for each operation, a few templates are created in which entities are replaced with placeholders (e.g., *"search for a flight from* `ORIGIN` *to* `DESTINATION`"). Next, by feeding values (e.g., `ORIGIN=[Sydney]` and `DESTINATION=[Houston]`) to the placeholders canonical utterances are generated (e.g., *"search for a flight from Sydney to Houston"*). Likewise, generative grammars have been used by semantic parsers for generating canonical utterances [1, 17, 18]. In this approach, logical forms are automatically generated based on the expert-written grammar rules. The grammar is used to automatically produce canonical utterances for the randomly generated logical forms [1]. Both generative grammar and template-based approaches require human efforts, making them hard and costly to scale.

In our work, by adopting ideas from the principles of RESTful design and machine translation techniques, we tackle the main issue which is creating the canonical utterances for RESTful APIs. As opposed to current techniques such as generative-grammar-based or template-based approaches, the proposed approach is domain-independent and can automatically generate initial utterances without human efforts. We thus pave the way for automating the process of building virtual assistants, which serve a large number of tasks, by automating the process of training datasets for new/updated APIs.

User Utterance Acquisition Methods. Current approaches for obtaining training utterances usually involves three main paradigms: launching a prototype to get utterances from end-users, employing crowd workers, and using automatic paraphrasing techniques to paraphrase existing utterances [19].

In the prototype-based approach, a bot is built without any (rule-based methods) or with a small number of annotated utterances. Such prototypes are able to obtain utterances from users to further improve the bots based on supervised machine learning techniques [20]. However, in case of using supervised machine learning methods in building the prototype, collecting initial annotated user utterances is still needed. Collecting an initial set of training samples is essential since the prototype bot must be accurate enough to serve existing user's requests without turning them away from the bot.

```
paths:
  /customers/{customer_id}:
    get:
      description: gets a customer by its id,
      summary: returns a customer by its id,
      parameters:
      - {
          name: customer_id,
          in: path,
          description: customer identifier,
          required: true,
          type: string
        }
```

Figure 3: Excerpt of an OpenAPI Specification

Crowdsourcing has been also used extensively to obtain natural language corpora for conversational agents [1, 8, 17, 18]. In this approach, a canonical utterance is provided as a starting point, and workers are asked to paraphrase the expression to new variations. Automatic paraphrasing techniques have also been employed to automatically generate training data [21–24]. This is done by paraphrasing canonical utterances to obtain new utterances automatically. However, while automatic paraphrasing is scalable and potentially cheaper, even the state-of-art models fall short in producing sufficiently diverse paraphrasing [25], and fail in producing multiple semantically-correct paraphrases for a single expression [26–28]. Nevertheless, these automatic approaches are still beneficial for bootstrapping a bot.

In this paper, we propose a dataset and an automated, scalable, and domain-independent approach for generating canonical utterances. Generated canonical utterances can be next fed to either automatic paraphrasing systems or crowdsourcing techniques to generate training samples for task-oriented bots.

3 THE API2CAN DATASET

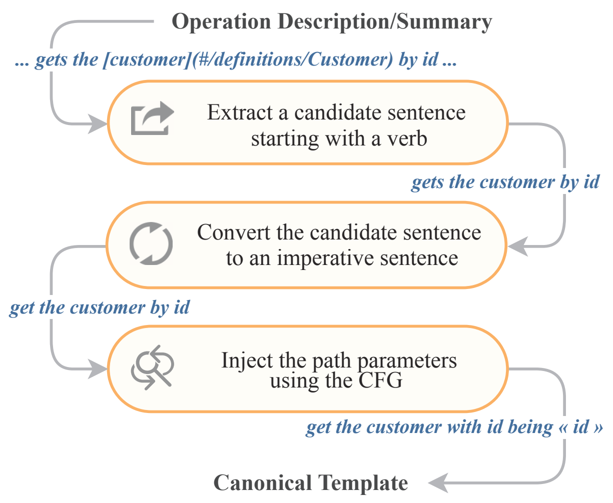
In this section, we explain the process of building the API2CAN dataset, and we provide its statistics (e.g., size).

3.1 API2CAN Generation Process

To generate the training dataset (pairs of operations and canonical utterances), we obtained OpenAPI specifications indexed in OpenAPI Directory[4]. OpenAPI Directory is a Wikipedia for REST APIs[5], and OpenAPI specification is a standard documentation format for REST APIs. As shown in Figure 3, the OpenAPI specification includes description, and information about the parameters (e.g., data types, examples) of each operation. We obtained the latest version of each API index in OpenAPI Directory, and totally collected 983 APIs, containing 18,277 operations in total (18.59 operation per an API on average). Finally, we generated canonical utterances for each of the extracted operations as described in the rest of this section and illustrated in Figure 4.

Candidate Sentence Extraction. We extract a candidate sentence from either the summary or description of the operation specification. For a given operation, the description (and summary) of the operation (e.g., *"gets a [customer] (#/definitions/-Customer) by id. The response contains ..."*) is pre-processed by

[4]https://github.com/APIs-guru/openapi-directory/tree/master/APIs
[5]https://apis.guru/browse-apis/



Operation Description/Summary
... gets the [customer](#/definitions/Customer) by id ...
Extract a candidate sentence starting with a verb
gets the customer by id
Convert the candidate sentence to an imperative sentence
get the customer by id
Inject the path parameters using the CFG
get the customer with id being « id »
Canonical Template

Figure 4: Process of Canonical Utterance Extraction

removing HTML tags, lowercasing, and removing hyperlinks (e.g., *"gets a customer by id. the response contains ..."*) and then it is split into its sentences (e.g., "gets a customer by id.", "the response contains ..."). Next, the first sentence starting with a verb (e.g., *"gets a customer by id"*) is chosen as a potential canonical utterance, and its verb is converted to its imperative form (e.g., *"get a customer by id"*).

Parameter Injection While the extracted sentence is usually a proper English sentence, it cannot be considered as a user utterance. That is because the sentence often points to the parameters of the operation without specifying their values. For example, given an operation like "`GET /customers/{customer_id}`" the extracted sentence is often similar to sentences like *"get a customer by id"* or *"return a customer"*. However, we are interested in annotated canonical utterances such as *"get the customer with id being «id»"*, and *"get the customer when its id is «id»"*; where "«id»" is a sampled valued for `customer_id`. To consider parameter values in the extracted sentence, we created a context-free grammar (CFG) as briefly shown in Table 1. This grammar has been created based on our observations of how operation descriptions are written (how parameters are mentioned in the extracted candidate sentences) by API developers. With this grammar, a list of possible mentions of parameters in the operation description is generated (e.g., *"by customer id"*, *"based on id"*, *"with the specified id"*). Then the lengthiest mention found in the sentence is replaced with *"with NPN being «PN»"*, where *NPN* and *PN* are human-readable version of the parameter name (e.g., customer_id ⟶ customer id) and its actual name respectively (e.g., *"get a customer with customer id being «customer_id»"*).

We also observed that path parameters are not usually mentioned in operation descriptions in API specifications. For example, in an operation description like *"returns an account for a given customer"* the path parameter *accountId* and *customerId* are absent, but the lemmatized name of collections *"customer"* and *"account"* are present. By using the information obtained from detecting such resources (see Section 4.2), it is possible to convert the description into *"return an account with id being «customer_id» for a given customer with id being «account_id»"*.

In the process of generating the *API2CAN* dataset, a few types of parameters were automatically ignored. As such, we did not include *header parameters*[6] since they are mostly used for authentication, caching, or exchanging information such as *Content-Type* and *User-Agent*. Thus such parameters do not specify entities of users' intentions. Likewise, using a list of predefined parameter names (e.g., auth, v1.1), we automatically ignored *authentication* and *versioning parameters* because bot users are not expected to directly specify such parameters while talking to a bot. Moreover, since the payload of an operation can contain inner objects, we assume that all attributes in the expected payload of an operation are flattened. This is done by concatenating the ancestors' attributes with the inner objects' attributes. For instance, the parameters in the following payload are flattened to "customer name" and "customer surname":

```
{
  "customer": {
    "name": "string",
    "surname": "string"
  }
}
```

As such, we convert complex objects to a list of parameters that can be asked from a user during a conversation.

Table 1: Parameter Replacement Context Free Grammar

Rule	
$N \longrightarrow \{PN\}\|\{NPN\}\|\{LPN\}\|\{RN\}\|\{NRN\}\|\{LRN\}$	
$CPX \longrightarrow$ 'by' | 'based on' | 'by given' | 'based on given' | ...	
$R \longrightarrow N$ | CPX N | N CPX N	
{PN}	Parameter Name (e.g., "customer_id", "CustomerID", "CustomersID")
{NPN}	Normalized PN by splitting concatenated words and lowercasing (e.g., "customer id", "customers id")
{LPN}	Lemmatized NPN (e.g., "customer id")
{RN}	Resource Name (e.g., "Customers")
{NRN}	Normalized RN (e.g., "customers")
{RN}	Lemmatized NRN (e.g., "customer")

3.2 Dataset Statistics

By processing all API specifications, we were able to automatically generate a dataset called *API2CAN*[7] which includes 14,370 pairs of operations and their corresponding canonical utterances. We next divided the dataset into three parts as summarized in Table 2, and manually checked and corrected extracted utterances in the test dataset to ensure a fair assessment of models learned on the dataset[8].

Table 2: *API2CAN* Statistics

Dataset	APIs	Size
Train Dataset	858	13029
Validation Dataset	50	433
Test Dataset	50	908

Figure 5 shows the number of operations in *API2CAN* based on the HTTP verbs (e.g., GET, POST). As shown in Figure 5, the

[6]Header fields are components of the header section of request in the Hypertext Transfer Protocol (HTTP).
[7]https://github.com/mysilver/API2CAN
[8]Train and validation datasets will be also manually revised in near future

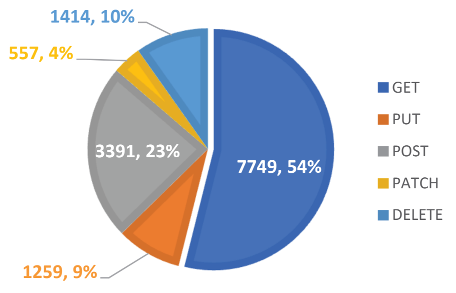


Figure 5: API2CAN Breakdown by HTTP Verb

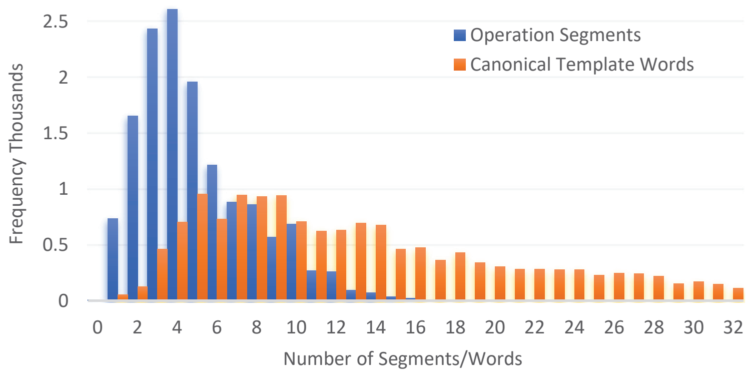


Figure 6: API2CAN Breakdown by Length

majority of operations are of GET methods which are usually used for retrieving information (e.g., *"get the list of customers"*), followed by POST methods which are usually used for creating resources (e.g., *"creating a new customer"*). The DELETE, PUT, and PATCH methods are also used for removing (e.g., *"delete a customer by id being «id»"*), replacing (e.g., *"replace a customer by id being «id»"*), and partially updating (e.g., *"update a customer by id being «id»"*) a resource.

Figure 6 also represents the distribution of number of segments in the operations[9] as well as the number of words in the generated canonical templates. As shown in Figure 6, many of the operations consist of less than 14 segments by 4 being the most common. Given the typical number of segments in the operations, Neural Machine Translation (NMT)-based approaches can be used for the generation of canonical sentences [29, 30]. On the other hand, the canonical sentences in the *API2CAN* dataset are longer. The reason behind having such lengthier utterances is the existence of parameters, and operations with more parameters tend to be lengthier. However, given the maximum length of canonical sentences, NMT-based approaches can still perform well [30].

4 NEURAL CANONICAL SENTENCE GENERATION

Neural Machine Translation (NMT) systems are usually based on encoder-decoder architecture to directly translate a sentence in one language to a sentence in a different language. As shown in Figure 7, generating a canonical template for a given operation can be also considered as a translation task. As such, the operation is encoded into a vector, and the vector is next decoded into an annotated canonical template. However, the main challenge in building such a translation model is the lack of a large training dataset. Since deep-learning models are data thirty, training requires a very large and diverse set of training samples (ideally millions of pairs of operations and their associating user utterances). As mentioned in the previous section, we automatically generated a dataset called *API2CAN*. However, such a dataset is still not large enough for training sequence-to-sequence models.

Having a large set of training samples requires a very large diverse set of operations as well. However, such a large set of APIs and operations is not available. One of the serious repercussions of the lack of such a set of operations is that training samples lack a very large number of possible words that can possibly appear in the operations (but did not appear in the training dataset). As a result, the models trained on such datasets will face many out-of-vocabulary words at runtime. To address this issue, we propose a delexicalization technique called *resource-based delexicalization*. As such, we reduce the impact of the out-of-vocabulary problem and force the model to learn the pattern of translating resources in an operation to a canonical template (instead of translating a sequence of words).

[9]For example, "`GET /customers/{customer_id}`" has two segments: "`customers`" and "`{customer_id}`"

4.1 Resources in REST

In RESTful design, primary data representation is called *resource*. A resource is an object with a type, associated data, relationships to other resources, and a set of HTTP verbs (e.g., GET, POST) that operate on it. Designing RESTful APIs often involves following conventions in structuring URIs (endpoints) and naming resources. Examples include using plural nouns for naming resources, using the "GET" method for retrieving a resource and using the "POST" method for creating a new resource.

In RESTful design, resources can be of various types. Most commonly, a resource can be a document or a collection. A document, which is also called *singleton resource*, represents a single instance of the resource. For example, "`/customers/{customer_id}`" represents a customer that is identified by a path parameter ("`customer_id`"). On the other hand, a *collection resource* represents all instances of a resource type such as "`/customers`". Resources can be also nested. As such, a resource may also contain a sub-collection ("`/customers/{customer_id} /accounts`"), or a singleton resource (e.g., "`/customers/{customer_id} /accounts/ {account_id}`"). In RESTful design, CRUD actions (create, retrieve, update and delete) over resources are shown by HTTP verbs (e.g., GET, POST). For example, "`GET /customers`" represents the action of getting the list of customers, and "`POST /customers`" indicates the action of creating a new customer. However, some actions might not fit into the world of conventional CRUD operations. In such cases, *controller* resources are used. *Controller* resources are like executable functions, with inputs and return-values. REST APIs rely on *action controllers* to perform application specific actions that cannot be logically mapped to one of the standard HTTP verbs. For example, an operation such as "`GET /customers/{customer_id}/activate`" can be used to activate a customer. Moreover, while it is unconventional, adjectives also are occasionally used for filtering resources. For example, "`GET /customers/activated`" means getting the list of all activated customers. In this paper, such adjectives are called *attribute controllers*.

While above-mentioned principles are followed by many API developers, there are still many APIs violate these principles. By manually exploring APIs and prior works [5, 14, 15], we identified some unconventional resource types used in designing operations as summarized in Table 3. A common drift from RESTful principles is the use of programming conventions in naming resources (e.g., "createActor", "get_customers"). Aggregation functions (e.g., sum, count) and expected output format of an operation (e.g., "json", "tsb", "txt") are also used in designing endpoints. Words similar to "search" (e.g. "query", "item-search") are used to indicate that the operation looks for resources based on given criteria. Moreover, collections are occasionally filtered/sorted by using keywords such as "filtered-by", "sort-by", or appending a resource name to "By" (e.g., "ByName", "ByID"). Segments in the endpoints may also indicate API versions (e.g., v1.12), or authentication endpoints (e.g., auth, login). Even though the aforementioned types of resources are against the conventional design guidelines of RESTful design, they are important to detect since still they are used by API developers in practice.

Table 3: Resource Types

Resource Type	Example
Collection	`/customers`
Singleton	`/customers/{customer_id}`
Action Controller	`/customers/{customer_id}/activate`
Attribute Controller	`/customers/activated`
API Specs	`/api/swagger.yaml`
Versioning	`/api/v1.2/search`
Function	`/AddNewCustomer`
Filtering	`/customers/ByGroup/{group-name}`
Search	`/customers/search`
Aggregation	`/customers/count`
File Extension	`/customers/json`
Authentication	`/api/auth`

4.2 Resource-based Delexicalization

In resource-based delexicalization, the input (API call) and output (canonical template) of the sequence-to-sequence model are converted to a sequence of resource identifiers as shown in Figure 7. This is done by replacing mentions of resources (e.g., customers, customer) with a corresponding resource identifier (e.g., Collection_1). A resource identifier consists of two parts: (i) the type of resource and (ii) a number *n* which indicates *n*-th occurrence of a resource type in a given operation. This number later is used in the lexicalization of the output of the sequence-to-sequence model to generate a canonical template.

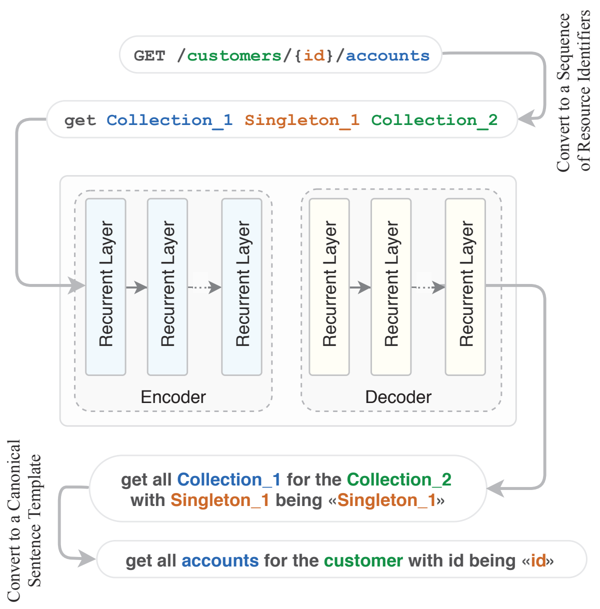
To detect resource types, we used the *Resource Tagger* shown in Algorithm 1. We convert the raw sequence of words in a given operation (e.g., "`GET /customers/{customer_id}/accounts`") to a sequence of resource identifiers (e.g., "`get Collection_1 Singleton_1 Collection_2`"). Likewise, mentions of resources in the canonical templates are replaced with their corresponding resource identifiers (e.g., "get all `Collection_1` for the `Collection_2` with `Singleton_1` being `Singleton_1`"). The intuition behind the conversions is to help the model to focus on translating a sequence of resources instead of words.

Algorithm 1: Resource Tagger

```
Input: segments of the operation
Result: List of resources
resources ⟵ [];
i ⟵ size(segments);
for i ← length(segments) down to 1 do
    current ⟵ segments[i];
    resource ⟵ new Resource();
    resource.name ⟵ current;
    previous ⟵ $\phi$;
    if $i > 1$ then
        previous ⟵ segments[$i - 1$];
    end
    resource.type ⟵ "Unknown";
    if current is a path parameter then
        if previous is a plural noun and an identifier then
            resource.type ⟵ "Singleton";
            resource.collection ⟵ previous;
        else
            resource.type ⟵ "Unknown Param";
        end
    else
        if current starts with "by" then
            resource.type ⟵ "Filtering";
        else if current in ["count", "min", ...] then
            resource.type ⟵ "Aggregation";
        else if current in ["auth", "token", ...] then
            resource.type ⟵ "Authentication";
        else if current in ["pdf", "json", ...] then
            resource.type ⟵ "File Extension";
        else if current in ["version", "v1", ...] then
            resource.type ⟵ "Versioning";
        else if current in ["swagger.yaml", ...] then
            resource.type ⟵ "API Specs";
        else if any of ["search", "query", ...] in current then
            resource.type ⟵ "Search";
        else if current is a phrase and starts with a verb then
            resource.type ⟵ "Function";
        else if current is a plural noun then
            resource.type ⟵ "Collection"
        else if current is a verb then
            resource.type ⟵ "Action Controller";
        else if current is an adjective then
            resource.type ⟵ "Attribute Controller";
    end
    resources.append(resource);
end
return reversed(resources)
```

In the time of using the model for generating canonical templates, the tagged resource identifiers are replaced with their corresponding resource names (e.g., Collection_2 ⟶ customers). Meanwhile, in the process of replacing resource tags, occasionally grammatical errors might happen such as having plural nouns instead of singular forms. To make the final generated canonical template more robust, we used LanguageTool [10] (an

[10]https://languagetool.org



GET /customers/{id}/accounts
Convert to a Sequence of Resource Identifiers
get Collection_1 Singleton_1 Collection_2
Recurrent Layer
Recurrent Layer
Recurrent Layer
Encoder
Recurrent Layer
Recurrent Layer
Recurrent Layer
Decoder
get all Collection_1 for the Collection_2 with Singleton_1 being «Singleton_1»
Convert to a Canonical Sentence Template
get all accounts for the customer with id being «id»

Figure 7: Canonical Template Generation via Resource-based Delexicalization

open-source tool for automatically detecting and correcting linguistically errors) to correct linguistic errors in the generated canonical templates.

5 PARAMETER VALUE SAMPLING

To obtain canonical utterances, values must be sampled for the parameters (placeholders) inside a given canonical template. The sampled values help to generate canonical utterances which are understandable sentences without any placeholders. Canonical utterances can be paraphrased later either automatically or manually by crowd-workers to diversify the training samples. In this section, we investigate how values can be sampled for parameters of REST APIs. More specifically, we identified five main sources as follows.

(1) **Common Parameters.** Parameters such as identifiers (e.g., customer_id), emails, and dates are ubiquitous in REST APIs. We built a set of such parameters paired with values. As such, a short random string or numeric value is generated for identifiers based on the parameter data type. Likewise, mock email addresses and dates are generated automatically.

(2) **API Invocation.** By invocation of API methods that return a list of resources (e.g., "`GET /customers`"), we can obtain a large number of values for various attributes (e.g., customer names, customer ids) of the resource. Such values are reliable since they correspond to real values of entities in the retrieved resources. Thus they can be used reliably to generate canonical utterances out of canonical templates.

(3) **OpenAPI Specification.** An OpenAPI specification may include an example or default values[11] for parameters of each operation. Since these values are generated by API owners, they are reliable. Moreover, API specification specifies the data-types of parameters. This can also be used to automatically generate values for parameters in the absence of example and default values. For example, in the case of *enumeration types* (e.g., gender ⟶ [MALE, FEMALE]), one of the elements is randomly selected as a parameter value. In the case of numeric parameters (e.g. size), a random number is generated within the specified range (e.g., between 1 to 10) in the API specification. Likewise, for the parameters whose values follow regular expressions (e.g., "[0-9]%" indicates a string that has a single-digit before a percent sign), random sequences are generated to fulfill the given pattern in the API specification (e.g., "8%").

(4) **Similar Parameters.** Having a large set of API specifications, example values can be found from similar parameters (sharing the same name and datatype). This can be possible by processing parameters of API repositories such as OpenAPI directory.

(5) **Named Entities.** Knowledge graphs provide information about various entities (e.g., cities, people, restaurants, books, authors). Examples of such knowledge graphs include for Freebase [31], DBpedia [32], Wikidata[33], and YAGO [34]. For a given entity type such as *"restaurant"* in the restaurant domain, these knowledge graphs might contain numerous entities (e.g., KFC, Domino's). Such knowledge bases can be used to sample values for a given parameter if the name of the parameter matches an entity type. In this paper, we use Wikidata to sample values for entity types. Wikidata is a knowledge graph which is populated by processing Wikimedia projects such as Wikipedia.

[11] An example illustrates what the value is supposed to be for a given parameter. But a default value is what the server uses if the client does not provide the value.

6 EXPERIMENTS & RESULTS

Before delving into the experiments, we briefly explain the training process in the case of using neural translation methods. We trained the neural models using the Adam optimizer [35] with an initial learning rate of 0.998, a dropout of 0.4 between recurrent layers (e.g., LSTM, BiLSTM), and a batch size of 512. It is worth noting that the hyper-parameters are initial configurations set based on the size of the dataset and values suggested in the literature, and finding optimized values requires further studies. Furthermore, in case of not using delexicalization, we also populate word embeddings of the model with GloVe [36].

In the time of translation, we used beam search with a beam size of 10 to obtain multiple translations for a given operation, and then the first translation with the same number of placeholders as the number of the parameters in the given operation is considered as its canonical template. Moreover, we replaced the generated *unknown* tokens with the source token that had the highest attention weight to avoid the out-of-vocabulary problem.

6.1 Translation Methods

We trained translation models using different sequence-to-sequence architectures and we also built a rule-based translator as described next. Given the size of the API2CAN dataset, we configured the models using two layers for both encoding and decoding parts at the most.

GRU. This model consists of two layers (each having 256 units) of Gated Recurrent Units (GRUs) [37] for both encoding and decoding layers using the attention mechanism [38].

Table 4: Excerpt of Transformation Rules

#		Resources Sequence	Transformation Rule
1	***Rule***	`GET /{c}/`	get list of {c.name}
	Example	`GET /customers`	get list of customers
2	***Rule***	`DELETE /{c}/`	delete all {c.name}
	Example	`DELETE /customers`	delete all customers
3	***Rule***	`GET /{c}/{s}/`	get the {$singular(c.name)$} with {s.name} being {s.name}
	Example	`GET /customers/{id}`	get the customer with id being <id>
4	***Rule***	`DELETE /{c}/{s}/`	delete the {$singular(c.name)$} with {s.name} being <{s.name}>
	Example	`DELETE /customers/{id}`	delete the customer with id being <id>
6	***Rule***	`PUT /{c}/{s}/`	replace the {$singular(c.name)$} with {s.name} being <{s.name}>
	Example	`PUT /customers/{id}`	replace the customer with id being <id>
7	***Rule***	`GET /{c}/{a}/`	get {a.name} {$singular(c.name)$}
	Example	`GET /customers/first`	get first customer
8	***Rule***	`GET /{c1}/{s}/{c2}/`	get the list of {c2.name} of the {$singular(c1.name)$} with {s.name} being {s.name}
	Example	`GET /customers/{id}/accounts`	get the list of accounts of the customer with id being <id>

LSTM. This model consists of two layers (each having 256 units) of two layers of LSTM for both encoding and decoding using the attention mechanism [38].

CNN. We also built a sequence-to-sequence model based on Convolutional Neural Network (CNN) as proposed in [39]. In particular, we used 3x3 convolutions (one layer of 256 units) with the attention mechanism [38].

BiLSTM-LSTM. This model consists of two layers (each having 256 units) of Bidirectional Long-Short Term Memory (BiLSTM) [40] for encoding, and two layers (each having 256 units) of Long-Short Term Memory (LSTM) [41] for the decoder using the attention mechanism [38].

Transformer. The Transformer architecture [42] has been shown to perform very strong in machine translation tasks [43, 44]. We used the Transformer model implemented by OpenNMT [45] using the same hyper-parameters as the original paper [42]. For an in-depth explanation of the model, we refer the interested reader to the original paper [42].

Rule-based (RB) Translator. Based on the concept of resource in REST APIs, we also built a rule-based translation system to translate operations to canonical templates (shown in Algorithm 2). First, the algorithm extracts the resources of a given operation based on the resource types extracted by the *Resource Tagger* algorithm (see Algorithm 1). Next, the algorithm iterates over an ordered set of *transformation rules* to transform the operation to a canonical template. A *transformation rule* is a hand-crafted Python function which is able to translate a specific HTTP method (e.g., GET) and sequence of resource types (e.g., a *collection* resource followed by a *singleton* resource) to a canonical template. We created 33 *transformation rules* by the time of writing this paper, some of which are listed in Table 4. In this table, `{c}`, `{s}`, and `{a}` stands for collection, singleton, and attribute controller respectively. And the $singular(.)$ function returns the singular form of a given name. For instance, in case of an operation like "`GET /customers`", given that the bot user requests a collection of customers, the provided transformer (rule number 1 in Table 4) is able to generate a canonical template as *"get the list of customers"*. Following Python function also presents the *transformation rule* implementation which is able to translate such operations (a single collection resource when the HTTP method is "GET"):

```
def transform(resources, verb):
 if verb != "GET" or len(resources) != 1:
    return
 if resources[0].type != "Collection":
    return
 collection = resources[0]
 return "get the list of {}".format(collection.name)
```

A transformer is written based on the assumption that a sequence of resource types has special meaning. For example, considering "`GET /customers/{id}/accounts`" and "`GET /users/{user_id}/aliases`", both operations share the same HTTP verbs and sequence of resource types (a singleton followed by a collection). In such cases, possible canonical templates are "*get accounts of a customer when its id is «id»*" and "*get aliases of a user when its user id is «user_id»*". Thus such a sequence of resource types can be converted to a rule like: *"get {collection} of a {singleton.collection} when its {singleton.name} is «{singleton.name}»"*; in which "{}" represents placeholders and *singleton.collection* represents the name of the collection for the given singleton resource (e.g., customers, users). Thus adding a new transformation rule would mean generalizing a specific sequence of resources types that is not considered in the existing translators. However, as discussed earlier, since many APIs do not conform to the RESTful principles, creating a comprehensive set of transformation rules is very challenging.

6.2 Canonical Utterance Generation

Quantitative Analysis. For each of the aforementioned NMT architectures, we trained models with and without using the

Algorithm 2: Rule-Based Translator

```
   Input: operation, transformation_rules written by experts
   Result: A canonical template
1  resources ⟵ ResourceTagger(operation);
2  foreach $t \in transformation\_rules$ do
3      canonical ⟵ t.transform(resources, operation.verb);
4      if $canonical \neq \phi$ then
5          param_clause ⟵ to_clause(operation.parameters) ;
6          canonical ⟵ canonical + " " + param_clause ;
7          return canonical;
8      end
9  end
10 return $\phi$
```

proposed *resource-based delexicalization* approach as described in Section 4.2. In these experiments, we did not tune any hyper parameters and trained the models on the training dataset. For each baseline, we saved the model after 10000 steps and used the model with the minimum perplexity based on the validation set to compare with other configurations. Table 5 presents the performance of each model in terms of machine translation metrics: bilingual evaluation understudy (BLEU) [46], Google's BLEU Score (GLEU) [47], and Character n-gram F-score (CHNF) [48].

In the case of using the RB-Translator, hand-crafted transformation rules are able to generate canonical templates for 26% of the operations. Creating such transformation rules is very challenging for lengthy operations as well as those not following RESTful principles. We did not include RB-Translators' performance in Table 5 because the results are not comparable to the rest. Our experiments indicate that RB-Translator performs reasonably well (BLEU=0.744, GLEU=0.746, and CHRF=0.850). However, the BiLSTM-LSTM model built on the proposed dataset using the resource-based delexicalization technique outperforms the RB-Translator (BLEU=0.876, GLEU=0.909, and CHRF=0.971), ignoring the operations which RB-Translator could not translate. As experiments indicate, *Delexicalized BiLSTM-LSTM* outperforms the rest of the translation systems, and resource-based delexicalization improves the performance of NMT systems by large.

Table 5: Translation Performance

Translation-Method	BLEU	GLEU	CHRF
Delexicalized BiLSTM-LSTM	**0.582**	**0.532**	**0.686**
Delexicalized Transformer	0.511	0.462	0.619
Delexicalized LSTM	0.503	0.470	0.652
Delexicalized CNN	0.507	0.458	0.601
Delexicalized GRU	0.481	0.450	0.623
BiLSTM-LSTM	0.318	0.266	0.421
Transformer	0.295	0.248	0.397
LSTM	0.278	0.226	0.381
CNN	0.271	0.220	0.379
GRU	0.251	0.198	0.347

Qualitative Analysis. Table 6 gives a few examples of canonical templates generated by the proposed translator (Delexicalized BiLSTM-LSTM). While the machine-translation metrics do not show very strong translation performance in Table 5, our manual inspections revealed that these metrics do not reflect the actual performance of the proposed translators. Therefore, we conducted another experiment to manually evaluated the translated operations. For this reason, we asked two experts to rate the generated canonical templates manually using a Likert scale (in a range of 1 to 5 with 5 showing the most appropriate canonical sentence). In the experiment, the experts were given pairs of generated canonical utterances and operations (including the description of the operation in the API specification). Next, they were asked to rate the generated canonical templates in a range of 1 to 5.

Figure 8 shows the Likert assessment for the best performing models in Table 5. Based on this experiment, canonical templates generated by RB-Translator are rated 4.47 out 5, and those of the delexicalized BiLSTM-LSTM are rated 4.06 out of 5 (by averaging the scores given by the annotators). The overall Kappa test showed a high agreement coefficient between the raters by Kappa being 0.86 [49]. Based on manual inspections, as also shown in Table 6, we observed that when APIs are designed based on the RESTful principles the delexicalized Delexicalized performs as good as RB-Translator.

Figure 8 also shows how the automatically generated dataset (*API2CAN*) represents their corresponding operations. Based on the rates given by the annotators, the dataset (training part) is also of decent quality while being noisy, indicating that the proposed set of heuristics for generating the dataset are well-defined. Given the promising quality of generated canonical templates, we concluded that the noises in the dataset can be ignored. However, yet it is desirable to manually clean the dataset.

Table 6: Examples of Generated Canonical Templates

	Sample
Operation	`GET /v2/taxonomies/`
Canonical	fetch all taxonomies
Operation	`PUT /api/v2/shop_accounts/{id}`
Canonical	update a shop account with id being <id>
Operation	`DELETE /api/v1/user/devices/{serial}`
Canonical	delete a device with serial being <serial>
Operation	`GET /user/ratings/query`
Canonical	get a list of ratings that match the query
Operation	`GET /v1/getLocations`
Canonical	get a list of locations
Operation	`POST /series/{id}/images/query`
Canonical	query the images of the series with id being <id>
Operation	`PUT /api /hotel /v0 /hotels /{hotelId} /rateplans/batch/$rates`
Canonical	set rates for rate plans of a hotel with hotel id being <hotelId>

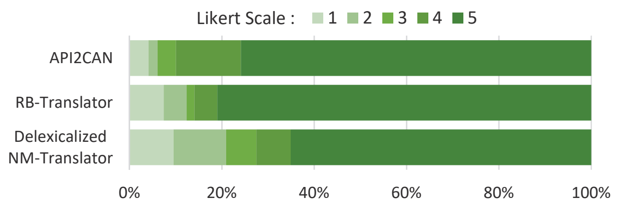


Figure 8: Assessment of Generated Canonical Templates

Error Analysis. Even though the proposed method outperforms the baselines, it still occasionally fails in generating high-quality canonical templates. Based on our investigations, there are three main sources of error in generating the canonical templates: (i) detecting resource types, (ii) translating APIs which do not conform to RESTful principles, and (iii) lengthy operations with many segments.

Detecting resource types requires natural language processing tools to detect parts of speech (POS) of a word (e.g., verb, noun, adjective), and to detect if a given noun is plural or singular (particularly for unknown words or phrases and uncountable nouns). However, these tools occasionally fail. Specifically, POS taggers are built for detecting parts of speech for words inside a sentence. Thus it is not surprising if they fail in detecting if a word like "rate" is a verb or noun in a given operation. For example, an operation like `GET /participation/rate` can indicate both "get the rate of participations" and "rate the participants". Another source of such issues is tokenization. It is common in APIs to concatenate words (e.g., whoami, addons, registrierkasseuuid, AddToIMDB). While it seems trivial for an individual to split these words, existing tools frequently fail. Such issues affect the process of detecting resources and consequently impact the generation of canonical templates negatively.

Unconventional API design (not conforming to RESTful principles) also extensively impacts the quality of generated canonical templates. Common drifts from RESTful principles includes using wrong HTTP verb (e.g., "`POST`" for retrieving information), using singular nouns for showing collections (e.g. `/customer`), adding non-resource parts to the path of the operation (e.g., adding *response format* like "json" in `/customers/json`. Since those API developers (who do not conform to design guidelines) follow their own thoughts instead of accepted rules, the automatic generation of canonical templates is challenging.

Lengthy operations (those with roughly more than 10 segments) naturally are rare in REST APIs. Such lengthy operations convey more complex intents than those with a lesser number of segments. As shown in Figure 6, unfortunately, such operations are also rare in the proposed dataset (*API2CAN*), impacting translation of lengthy operations.

6.3 Parameter Value Sampling

This section provides an analysis of parameters in the RESTful APIs and evaluates the proposed parameter sampling approach which is used for generating canonical utterances out of canonical templates. To this end, we processed API specifications which are indexed in OpenAPI Directory. Based on our analysis, the dataset contains 145971 parameters in total, which indicates that an operation has 8.5 parameters on average.

Figure 9 presents statistics of parameters in the whole list of API specifications in the OpenAPI Directory. As shown in the right-hand pie chart, most of the parameters are located in the payload (body) of APIs, followed by query and path parameters. Figure 9 also shows the percentages of parameter data types in the collection with *strings* being the most common type of parameters. About 1.5% of string parameters are defined by regular expressions, and 4.8% of them can be associated with an entity type[12]. String parameters are followed by integers, booleans, numbers, and enumerations. Moreover, some parameters are left without any type, or they are given general parameter types such as *"object"* without any schemes. These parameters are combined together in the left-hand pie chart in Figure 9 with a single label–*"others"*. Moreover, 28% of parameters are required parameters (not optional), 10.6% of parameters have not assigned any value in the API specifications, and 26% of all parameters are identifiers (e.g., id, UUID). Thus, sampling values is required only for less than 10.6% of parameters (those without any values). In particular, value sampling for string parameters requires more attention. That is because string parameters are widely used, and they are more difficult to automatically be assigned values in comparison to other types of parameters (e.g., integers, enumerations).

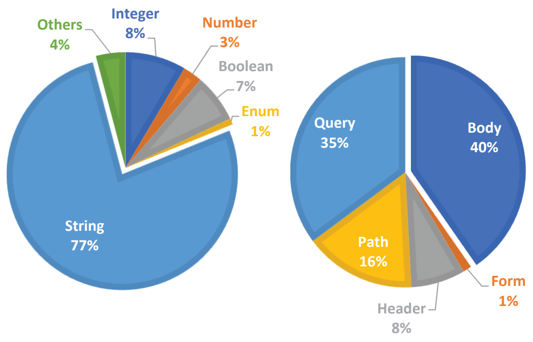


Figure 9: Parameter Type and Location Statistics

To evaluate how well the proposed method generates sample values for parameters, we conducted an experiment. Since generating sample values for data types such as numbers and enumerations is straightforward, we only considered string parameters in this experiment. To this end, we randomly selected 200 parameters and asked an expert to annotate if a sampled value is appropriate for the given value or not. The results indicate that 68 percent of sampled values are appropriate for given parameters. The main reason for inappropriate sampled values is noises in the API specifications. For instance, developers occasionally describe the parameters in the *example* part instead of the *description* part of the documentation. For instance, for a string parameter like "`customer_id`", the example part may be filled by "a valid customer id". Moreover, sometimes the same parameter name is used in different contexts for different purposes. For example, the parameter name like *"name"* which can be used for representing the name of a person, school, or any object.

7 CONCLUSION & FUTURE WORK

This paper aimed at addressing an important shortcoming in current approaches for acquiring canonical utterances. In this paper, we demonstrated that the generation of canonical utterances can be considered as a machine translation task. As such, our work also aimed at addressing an important challenge in training supervised neural machine translators, namely the lack

[12]We looked up the parameter name in Wikidata to find if there is associating entity type

of training data for translating operations to canonical templates. By processing a large set of API specifications and based on a set of heuristics, we build a dataset called *API2CAN*. However, deep-learning-based approaches require larger sets of training samples to train domain-independent models. Thus, by formalizing and defining resources in REST APIs, we proposed a delexicalization technique to convert an operation to a tagged sequence of resources to help sequence-to-sequence models to learn from such a dataset. In addition, we showed how parameter values can be sampled to feed placeholders in a canonical template and generate canonical utterances. We also gave a systematic analysis of web APIs and their parameters, indicating the importance of string parameters in automating the generation of canonical utterances.

In our future work, we will be working on improving the dataset (*API2CAN*). Moreover, given that fulfilling complex intents usually requires a combination of operations [8, 50], we will be working on compositions between operations. To achieve this, it is required to detect the relations between operations and generate canonical templates for complex tasks (e.g., tasks requiring conditional operations or compositions of multiple operations). In future work, we will target these problems, together with many other exciting opportunities as extensions to this work.

ACKNOWLEDGEMENTS

This research was supported fully by the Australian Government through the Australian Research Council's Discovery Projects funding scheme (project DP1601104515).

REFERENCES

[1] Y. Su, A. H. Awadallah, M. Khabsa, P. Pantel, M. Gamon, and M. Encarnacion, "Building natural language interfaces to web apis," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM '17. New York, NY, USA: ACM, 2017, pp. 177–186. [Online]. Available: http://doi.acm.org/10.1145/3132847.3133009

[2] P. Babkin, M. F. M. Chowdhury, A. Gliozzo, M. Hirzel, and A. Shinnar, "Bootstrapping chatbots for novel domains," in *Workshop at NIPS on Learning with Limited Labeled Data (LLD)*, 2017.

[3] M. Vaziri, L. Mandel, A. Shinnar, J. Siméon, and M. Hirzel, "Generating chat bots from web api specifications," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2017. New York, NY, USA: ACM, 2017, pp. 44–57. [Online]. Available: http://doi.acm.org/10.1145/3133850.3133864

[4] F. Palma, J. Gonzalez-Huerta, N. Moha, Y.-G. Guéhéneuc, and G. Tremblay, "Are restful apis well-designed? detection of their linguistic (anti)patterns," in *Service-Oriented Computing*, A. Barros, D. Grigori, N. C. Narendra, and H. K. Dam, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 171–187.

[5] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella, "Rest apis: A large-scale analysis of compliance with principles and best practices," in *Web Engineering*, A. Bozzon, P. Cudre-Maroux, and C. Pautasso, Eds. Cham: Springer International Publishing, 2016, pp. 21–39.

[6] F. Palma, J. Gonzalez-Huerta, M. Founi, N. Moha, G. Tremblay, and Y.-G. GuÃĺhÃĺneuc, "Semantic analysis of restful apis for the detection of linguistic patterns and antipatterns," *International Journal of Cooperative Information Systems*, p. 1742001, 05 2017.

[7] M.-A. Yaghoub-Zadeh-Fard, B. Benatallah, M. Chai Barukh, and S. Zamanirad, "A study of incorrect paraphrases in crowdsourced user utterances," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 295–306. [Online]. Available: https://www.aclweb.org/anthology/N19-1026

[8] G. Campagna, R. Ramesh, S. Xu, M. Fischer, and M. S. Lam, "Almond: The architecture of an open, crowdsourced, privacy-preserving, programmable virtual assistant," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2017, pp. 341–350. [Online]. Available: https://doi.org/10.1145/3038912.3052562

[9] T.-H. K. Huang, W. S. Lasecki, and J. P. Bigham, "Guardian: A crowd-powered spoken dialog system for web apis," in *Third AAAI conference on human computation and crowdsourcing*, 2015.

[10] M. Negri, Y. Mehdad, A. Marchetti, D. Giampiccolo, and L. Bentivogli, "Chinese whispers: Cooperative paraphrase acquisition," in *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*. Istanbul, Turkey: European Language Resources Association (ELRA), May 2012, pp. 2659–2665. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2012/pdf/772_Paper.pdf

[11] Y. Wang, J. Berant, and P. Liang, "Building a semantic parser overnight," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, Jul. 2015, pp. 1332–1342. [Online]. Available: https://www.aclweb.org/anthology/P15-1129

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1," 1999.

[13] C. Pautasso, "Restful web services: principles, patterns, emerging technologies," in *Web Services Foundations*. Springer, 2014, pp. 31–51.

[14] F. Palma, J. Dubois, N. Moha, and Y.-G. Guéhéneuc, "Detection of rest patterns and antipatterns: A heuristics-based approach," in *Service-Oriented Computing*, X. Franch, A. K. Ghose, G. A. Lewis, and S. Bhiri, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 230–244.

[15] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc, "Are rest apis for cloud computing well-designed? an exploratory study," in *Service-Oriented Computing*, Q. Z. Sheng, E. Stroulia, S. Tata, and S. Bhiri, Eds. Cham: Springer International Publishing, 2016, pp. 157–170.

[16] J. Weizenbaum, "Eliza&mdash;a computer program for the study of natural language communication between man and machine," vol. 9, no. 1. New York, NY, USA: ACM, Jan. 1966, pp. 36–45. [Online]. Available: http://doi.acm.org/10.1145/365153.365168

[17] W. Y. Wang, D. Bohus, E. Kamar, and E. Horvitz, "Crowdsourcing the acquisition of natural language corpora: Methods and observations," in *2012 IEEE Spoken Language Technology Workshop (SLT)*, Dec 2012, pp. 73–78.

[18] A. Ravichander, T. Manzini, M. Grabmair, G. Neubig, J. Francis, and E. Nyberg, "How would you say it? eliciting lexically diverse dialogue for supervised semantic parsing," in *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue*, 2017, pp. 374–383.

[19] J. Fang and J. Kessler, "System and method to acquire paraphrases," Jun. 6 2017, uS Patent 9,672,204.

[20] K. Dhamdhere, K. S. McCurley, R. Nahmias, M. Sundararajan, and Q. Yan, "Analyza: Exploring data with conversation," in *Proceedings of the 22Nd International Conference on Intelligent User Interfaces*, ser. IUI '17. New York, NY, USA: ACM, 2017, pp. 493–504. [Online]. Available: http://doi.acm.org/10.1145/3025171.3025227

[21] N. Duan, D. Tang, P. Chen, and M. Zhou, "Question generation for question answering," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sep. 2017, pp. 866–874. [Online]. Available: https://www.aclweb.org/anthology/D17-1090

[22] L. Dong, J. Mallinson, S. Reddy, and M. Lapata, "Learning to paraphrase for question answering," *arXiv preprint arXiv:1708.06022*, 2017.

[23] J. Mallinson, R. Sennrich, and M. Lapata, "Paraphrasing revisited with neural machine translation," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, vol. 1, 2017, pp. 881–893.

[24] R. Bapat, P. Kucherbaev, and A. Bozzon, "Effective crowdsourced generation of training data for chatbots natural language understanding," in *Web Engineering*, T. Mikkonen, R. Klamma, and J. Hernández, Eds. Cham: Springer International Publishing, 2018, pp. 114–128.

[25] S. A. Hasan, B. Liu, J. Liu, A. Qadir, K. Lee, V. Datla, A. Prakash, and O. Farri, "Neural clinical paraphrase generation with attention," in *Proceedings of the Clinical Natural Language Processing Workshop (ClinicalNLP)*, 2016, pp. 42–53.

[26] A. Gupta, A. Agarwal, P. Singh, and P. Rai, "A deep generative framework for paraphrase generation," *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[27] Q. Xu, J. Zhang, L. Qu, L. Xie, and R. Nock, "D-page: Diverse paraphrase generation," *CoRR*, vol. abs/1808.04364, 2018.

[28] F. Brad and T. Rebedea, "Neural paraphrase generation using transfer learning," in *Proceedings of the 10th International Conference on Natural Language Generation*, 2017, pp. 257–261.

[29] J. Pouget-Abadie, D. Bahdanau, B. van Merriënboer, K. Cho, and Y. Bengio, "Overcoming the curse of sentence length for neural machine translation using automatic segmentation," in *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 78–85. [Online]. Available: https://www.aclweb.org/anthology/W14-4009

[30] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder–decoder approaches," in *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 103–111. [Online]. Available: https://www.aclweb.org/anthology/W14-4012

[31] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: A collaboratively created graph database for structuring human knowledge," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 1247–1250.

[Online]. Available: http://doi.acm.org/10.1145/1376616.1376746

[32] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer *et al.*, “Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia,” *Semantic Web*, vol. 6, no. 2, pp. 167–195, 2015.

[33] D. Vrandečić and M. Krötzsch, “Wikidata: A free collaborative knowledgebase,” *Commun. ACM*, vol. 57, no. 10, pp. 78–85, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629489

[34] T. Rebele, F. Suchanek, J. Hoffart, J. Biega, E. Kuzey, and G. Weikum, “Yago: A multilingual knowledge base from wikipedia, wordnet, and geonames,” in *The Semantic Web – ISWC 2016*, P. Groth, E. Simperl, A. Gray, M. Sabou, M. Krötzsch, F. Lecue, F. Flöck, and Y. Gil, Eds. Cham: Springer International Publishing, 2016, pp. 177–185.

[35] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *International Conference on Learning Representations*, 12 2014.

[36] J. Pennington, R. Socher, and C. Manning, “Glove: Global vectors for word representation,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. [Online]. Available: https://www.aclweb.org/anthology/D14-1162

[37] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using RNN encoder-decoder for statistical machine translation,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, 2014, pp. 1724–1734. [Online]. Available: https://www.aclweb.org/anthology/D14-1179/

[38] T. Luong, H. Pham, and C. D. Manning, “Effective approaches to attention-based neural machine translation,” in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 1412–1421. [Online]. Available: https://www.aclweb.org/anthology/D15-1166

[39] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, “Convolutional sequence to sequence learning,” in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML’17. JMLR.org, 2017, pp. 1243–1252. [Online]. Available: http://dl.acm.org/citation.cfm?id=3305381.3305510

[40] A. Graves, S. Fernández, and J. Schmidhuber, “Bidirectional lstm networks for improved phoneme classification and recognition,” in *Artificial Neural Networks: Formal Models and Their Applications – ICANN 2005*, W. Duch, J. Kacprzyk, E. Oja, and S. Zadrożny, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 799–804.

[41] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[42] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS’17. USA: Curran Associates Inc., 2017, pp. 6000–6010. [Online]. Available: http://dl.acm.org/citation.cfm?id=3295222.3295349

[43] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *NAACL-HLT*, 2019.

[44] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, “Improving language understanding by generative pre-training,” *URL https://s3-us-west-2. amazonaws. com/openai-assets/research-covers/languageunsupervised/language understanding paper. pdf*, 2018.

[45] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. Rush, “OpenNMT: Open-source toolkit for neural machine translation,” in *Proceedings of ACL 2017, System Demonstrations*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 67–72. [Online]. Available: https://www.aclweb.org/anthology/P17-4012

[46] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: A method for automatic evaluation of machine translation,” in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL ’02. Stroudsburg, PA, USA: Association for Computational Linguistics, 2002, pp. 311–318. [Online]. Available: https://doi.org/10.3115/1073083.1073135

[47] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, “Google’s neural machine translation system: Bridging the gap between human and machine translation,” *arXiv preprint arXiv:1609.08144*, 2016.

[48] M. Popović, “chrF: character n-gram f-score for automatic MT evaluation,” in *Proceedings of the Tenth Workshop on Statistical Machine Translation*. Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 392–395. [Online]. Available: https://www.aclweb.org/anthology/W15-3049

[49] M. L. McHugh, “Interrater reliability: the kappa statistic,” *Biochemia medica: Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[50] E. Fast, B. Chen, J. Mendelsohn, J. Bassen, and M. S. Bernstein, “Iris: A conversational agent for complex tasks,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’18. New York, NY, USA: ACM, 2018, pp. 473:1–473:12. [Online]. Available: http://doi.acm.org/10.1145/3173574.3174047

Efficient Continuous Multi-Query Processing over Graph Streams

Lefteris Zervakis[§†], Vinay Setty[§‡], Christos Tryfonopoulos[†], Katja Hose[§]
§ Aalborg University, Aalborg, Denmark
† University of the Peloponnese, Tripolis, Greece
‡ University of Stavanger, Stavanger, Norway
{lefteris,vinay,khose}@cs.aau.dk,{zervakis,trifon}@uop.gr,vsetty@acm.org

ABSTRACT

Graphs are ubiquitous and ever-present data structures that have a wide range of applications involving social networks, knowledge bases and biological interactions. The evolution of a graph in such scenarios can yield important insights about the nature and activities of the underlying network, which can then be utilized for applications such as news dissemination, network monitoring, and content curation. Capturing the continuous evolution of a graph can be achieved by long-standing sub-graph queries. Although, for many applications this can only be achieved by a set of queries, state-of-the-art approaches focus on a single query scenario. In this paper, we therefore introduce the notion of continuous multi-query processing over graph streams and discuss its application to a number of use cases. To this end, we designed and developed a novel algorithmic solution for efficient multi-query evaluation against a stream of graph updates and experimentally demonstrated its applicability. Our results against two baseline approaches using real-world, as well as synthetic datasets, confirm a two orders of magnitude improvement of the proposed solution.

1 INTRODUCTION

In recent years, graphs have emerged as prevalent data structures to model information networks in several domains such as social networks, knowledge bases, communication networks, biological networks and the World Wide Web. These graphs are *massive* in scale and *evolve* constantly due to frequent updates. For example, according to its latest quarterly update, Facebook has over 1.52B daily active users who generate over 500K posts/comments and four million likes every minute resulting in massive updates to the Facebook social graph.

To gain meaningful and up-to-date insights in such frequently updated graphs, it is essential to be able to monitor and detect continuous patterns of interest. There are several applications from a variety of domains that may benefit from such monitoring. In social networks, such applications may involve targeted advertising, spam detection [3, 40], and fake news propagation monitoring based on specific patterns [34]. Similarly, other applications like (i) protein interaction patterns in biological networks [37, 45], (ii) traffic monitoring in transportation networks, (iii) attack detection (e.g., distributed denial of service attacks in computer networks), (iv) question answering in knowledge graphs [2], and (v) reasoning over RDF graphs may also benefit from such pattern detection.

For the applications mentioned above it is necessary to express the required patterns as *continuous sub-graph queries* over (one or many) streams of graph updates and appropriately *notify* the subscribed users for any patterns that match their subscription. Detecting these query patterns is fundamentally a sub-graph isomorphism problem which is known to be NP-complete due to the exponential search space resulting from all possible sub-graphs [18, 33]. The typical solution to address this issue is to pre-materialize the necessary sub-graph views for the queries and perform exploratory joins [36]; an expensive operation even for a single query in a static setting.

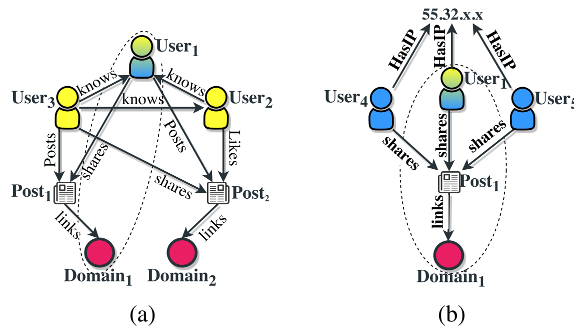


Figure 1: Spam detection: Users sharing and liking content with links to flagged domains. (a) A clique of users who know each other, and (b) Users sharing the same IP address.

These applications deal with graph streams in such a setup that is often essential to be able to support hundreds or thousands of continuous queries simultaneously. This leads to several challenges that require: (i) quickly detecting the affected queries for each update, (ii) maintaining a large number of materialized views, and (iii) avoiding the expensive join and explore approach for large sets of queries.

To better illustrate the remarks above, consider the application of spam detection in social networks. Fig. 1 shows an example of two graph patterns that may emerge from malicious user activities, i.e., users posting links to domains that have been flagged as fraudulent. Notice that malicious behavior could be caused either because a group of users that know each other share and like each other's posts containing content from a flagged domain (Fig. 1(a)), or because the group of users shared the same flagged post several times from the same IP (Fig. 1(b)). Even though these two queries are fundamentally different and produce different matching patterns, they share a common sub-graph pattern, i.e., "$\text{User1} \xrightarrow{\text{shares}} \text{Post1} \xrightarrow{\text{links}} \text{Domain1}$". If these two queries are evaluated independently, all the computations for processing the common pattern have to be executed twice. However, by identifying common patterns in query sets, we can amortize the costs of processing and answering them.

One simple approach to avoid processing all the (continuous) queries upon receiving a graph update is to index the query graphs using an inverted-index at the granularity of edges. While this approach may help us quickly detect all the affected queries for a given graph update, we still need to perform several exploratory joins to answer the affected queries. For example, in Fig. 1, we would need to join and explore the edges matching the pattern "User1 $\xrightarrow{\text{Shares}}$ Post1 and Post1 $\xrightarrow{\text{Links}}$ Domain1" upon each update to process the two queries. On the contrary, if we first identify the maximal sub-graph patterns shared among the queries instead, we can minimize the number of operations necessary to answer the queries. Therefore, a solution which groups queries based on their shared patterns would be expected to deliver significant performance gains. To the best of our knowledge, none of the existing works provide a solution that exploits common patterns for continuous multi-query answering.

In this paper, we address this gap by proposing a novel *algorithmic solution*, coined TRIC (TRIe-based Clustering) to index and cluster continuous graph queries. In TRIC, we first decompose queries into a set of directed paths such that each vertex in the query graph pattern belongs to at least one path (path covering problem [11]). However, obtaining such paths leads to redundant query edges and vertices in the paths; this is undesirable since it affects the performance of the query processing. Therefore, we are interested in finding paths which are shared among different queries, with minimal duplication of vertices. The paths obtained are then indexed using 'tries' that allow us to minimize query answering time by (i) quickly identifying the affected queries, (ii) sharing materialized views between common patterns, and (iii) efficiently ordering the joins between materialized views affected from the update. To this end, our *contributions* are:

- We formalize the problem of continuous multi-query answering over graph streams (Section 3).
- We propose a novel query graph clustering algorithm that is able to efficiently handle large numbers of continuous graph queries by resorting on (i) the decomposition of continuous query graphs to minimum covering paths and (ii) the utilization of tries for capturing the common parts of those paths (Section 4).
- Since no prior work in the literature has considered continuous multi-query answering in the context of graph streams, we designed and developed two algorithmic solutions that utilize inverted indexes for the graph query answering. Additionally, we deploy and extend Neo4j [43], a well-established graph database solution, to support our proposed paradigm. To this end, the proposed solutions will serve as baselines approaches during the experimental evaluation (Section 5).
- We experimentally evaluate the proposed solution using three different datasets from social networks, transportation, and biology domains, and compare the performance against the three baselines. In this context, we show that our solution can achieve up to two orders of magnitude improvement in query processing time (Section 6).

2 RELATED WORK

Structural graph pattern search using graph isomorphism has been studied in the literature before [18, 33]. In [17], the authors propose a solution that aims at reducing the search space for a single query graph. The solution identifies candidate regions in the graph that can contain query embeddings, while it is coupled with a neighborhood equivalence locating strategy to generate enumerations. In the same spirit [30] aims at reducing the search space in the graph by exploiting syntactic similarities present on vertex relationships. [31] considers the sub-graph isomorphism problem when multiple queries are answered simultaneously. However, these techniques are designed for static graphs and are not suitable for processing continuous graph queries on evolving graphs.

Continuous sub-graph matching has been considered in [41] but the authors assume a static set of sub-graphs to be matched against update events, use approximate methods that yield false positives, and small (evolving) graphs. An extension to this work considers the problem of uncertain graph streams [7], over wireless sensor networks and PPIs. The work in [15] considers a setup of continuous graph pattern matching over knowledge graph streams. The proposed solution utilizes finite automatons to represent and answer the continuous queries. However, this approach can support a handful of queries, since, each query is evaluated separately, while, it generates false positives due to the adopted sliding window technique. These solutions are not suitable for answering large number of continuous queries on graphs with high update rates.

There are a few publish/subscribe solutions on ontology graphs proposed in [29, 42], but they are limited to the RDF data model. Distributed pub/sub middleware for graphs has recently been proposed in [5], however, the main focus is on node constraints (attributes) while ignoring the graph structure.

The work in [9] provides an exact subgraph search algorithm that exploits the *temporal* characteristics of representative queries for online news or social media exploration. The algorithm exploits the structural and semantic characteristics of the graph through a specialized data structure. Where the authors consider continuous query answering with graph patterns over dynamic multi-relation graphs. In [36] the authors perform subgraph matching over a billion node graph by proposing graph exploration methods based on cloud technologies. While the aforementioned works are similar to the query evaluation scenario, the emphasis is on efficient search mechanisms, rather than continuous answering over streaming graphs.

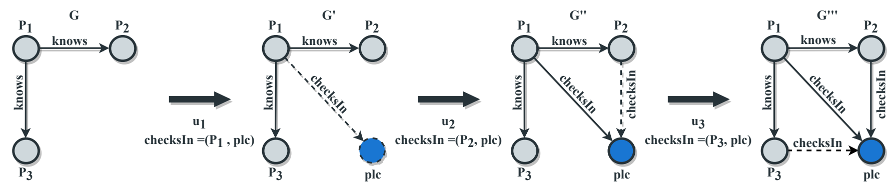
In the graph streams domain; [28] proposes algorithms to identify correlated graphs from a graph stream. This differs from our setup since a sliding window that covers a number of batches of data is used, and the main focus is set on identifying subgraphs with high Pearson correlation coefficients. In [14], the authors propose continuous pattern detection in graph streams with snapshot isolation. However, this solution considers only isolated queries (i.e., one query at a time) and the patterns detected are approximate. Finally, in [13] the authors propose a solution over a distributed computational environment, while the solution operates under the assumption of a static and limited query set.

Finally, some of the techniques used for increasing the efficiency of the proposed algorithm employ standard data indexing practices from a variety of domains, although the assumed setup and target applications differ significantly: (i) the representation of materialized views bears similarities with techniques from centralized RDF query processing [24, 32, 39] and statement/property tables as in Jena1 and Jena2 [44], (ii) the problem of maintaining the materialized views of (graph) queries relates to incremental view maintenance [4, 8, 25, 47] in database and warehousing environments, (iii) while query decomposition and tree-based clustering is typical in a variety of domains and applications [31, 38].

In general, solutions like the proposed one on continuous subgraph pattern matching can be applied in a wide range of domains such as social networks, protein-protein interactions (PPI), cyber-security, knowledge graphs, road network monitoring, and co-authorship graphs. Social network graphs emerge naturally

Update stream S
$u_1 = (checksIn = (P_1, \text{plc}))$
$u_2 = (checksIn = (P_2, \text{plc}))$
$u_3 = (checksIn = (P_3, \text{plc}))$

(a)



(b)

Figure 2: (a) An update stream S and (b) the evolution of graph G after inserting $u_i \in S$.

from the evolving interactions and activities of the users, while applications such as advertising, recommendation systems, and information discovery aim at exploiting these interactions. Social network applications may benefit from continuous pattern matching, and can leverage on already observed patterns in content propagation [21, 22, 46] and influential user discovery [6, 9]. PPIs are data repositories [35, 37] that index proteins (graph vertices) and the interactions (graph edges) between them. PPI graphs are constantly updated due to additions and invalidations of interactions, while scientists manually query PPIs to discover new patterns. In such scenarios, subgraph matching could enhance information discovery through appropriate graphical user interface tools. In cyber-security, subgraph matching could be applied for network motoring, denial of service, and exfiltration attacks [20], while subgraph matching over road networks could capture traffic events, and taxi route pricing. Finally, in the domain of co-authorship graphs, users may utilize continuous query evaluation in services similar to Google Scholar Alerts, when requesting to be notified about newly published content.

3 DATA MODEL AND PROBLEM DEFINITION

In this section we outline the data (Section 3.1) and query model (Section 3.2) that our approach builds upon.

3.1 Graph Model

In this paper, we use attribute graphs [10] (Definition 3.1), as our data model, as they are used natively in a wide variety of applications, such as social network graphs, traffic network graphs, and citation graphs. Datasets in other data models can be mapped to attribute graphs in a straightforward manner so that our approach can be applied to them as well.

Definition 3.1. An *attribute graph* G is defined as a directed labeled multigraph:

$$G = (V, E, l_V, l_E, \Sigma_V, \Sigma_E)$$

where V is the set of vertices and E the set of edges. An edge $e \in E$ is an ordered pair of vertices $e : (s, t)$, where $s, t \in V$ represent source and target vertices. $l_V : V \longrightarrow \Sigma_V$ and $l_E : E \longrightarrow \Sigma_E$ are labeling functions assigning labels to vertices and edges from the label sets Σ_V and Σ_E.

For ease of presentation, we denote an edge e as $e = (s, t)$, where e, s and t are the labels of the edge($l_E(e)$), source vertex ($l_V(s)$) and target vertex ($l_V(t)$) respectively.

As our goal is to facilitate efficient continuous multi-query processing over graph streams, we also provide formal definitions for updates and graph streams (Definitions 3.2 and 3.3).

Definition 3.2. An update u_t on graph G is defined as an *addition* (e) of an edge e at time t. An addition leads to new edges between vertices and possibly the creation of new vertices.

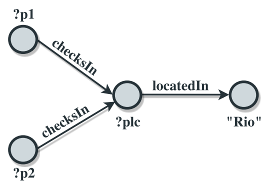


Figure 3: Example query graph pattern.

Definition 3.3. A *graph stream* $S = (u_1, u_2, \dots, u_t)$ of graph G is an *ordered sequence* of updates.

Fig. 2(a) presents an update stream S consisting of three graph updates u_1, u_2, and u_3 generated from social network events. While, Fig. 2 (b) shows the initial state of graph G and its evolution after inserting sequentially the three updates.

3.2 Query Model

For our query model we assume that users (or services operating on their behalf) are interested to learn when certain patterns emerge in an evolving graph. Definition 3.4 formalizes *query graph patterns* that define structural and attribute constraints on graphs.

Definition 3.4. A *query graph pattern* Q_i is defined as a directed labeled multigraph:

$$Q_i = (V_{Q_i}, E_{Q_i}, vars, l_V, l_E, \Sigma_V, \Sigma_E)$$

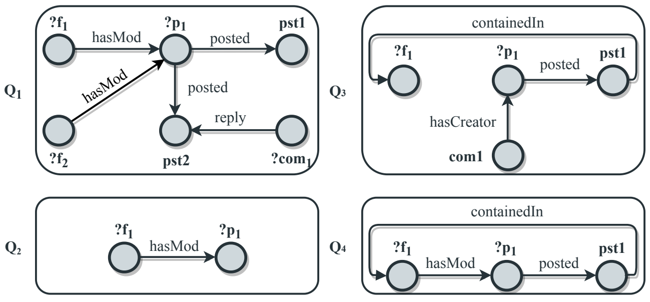
where V_{Q_i} is a set of vertices, E_{Q_i} a set of edges, and $vars$ a set of variables. $l_V : V \longrightarrow \{\Sigma_V \cup vars\}$ and $l_E : E \longrightarrow \Sigma_E$ are labeling functions assigning labels (and variables) to vertices and edges.

Let us consider an example where a user wants to be notified when his friends visit places nearby. Fig. 3 shows the corresponding query graph pattern that will result in a user notification when two people check in at the same place/location in Rio.

Based on the above definitions, let us now define the problem of *multi-query processing over graph streams*.

Problem Definition. Given a set of query graph patterns $Q_{DB} = \{Q_1, Q_2, \dots, Q_k\}$, an initial attribute graph G, and a graph stream S with continuous updates $u_t \in S$, the problem of *multi-query processing over graph streams* consists of continuously identifying all satisfied query graph patterns $Q_i \in Q_{DB}$ when applying incoming updates.

Query Set and Graph Modifications. A set of query graph patterns Q_{DB} is subject to modifications (i.e., additions and deletions). In this work, we focus on streamlining the query indexing phase, while developing techniques that allow processing each incoming query graph pattern separately, thus supporting continuous additions in Q_{DB}. In the same manner, a graph G is subject to edge additions and deletions, our main objective is to efficiently determine the queries satisfied by an edge addition. The proposed model does not require indexing the entire graph G and retains solely the necessary parts of G for the query answering. To this



(a)

Query ID	Set of Covering Paths
Q_1	$P_1 = \{?var \xrightarrow{hasMod} ?var \xrightarrow{posted}$ "$pst1$"$\}$ $P_2 = \{?var \xrightarrow{hasMod} ?var \xrightarrow{posted}$ "$pst2$"$\}$ $P_3 = \{?var \xrightarrow{reply}$ "$pst2$"$\}$
Q_2	$P_1 = \{?var \xrightarrow{hasMod} ?var\}$
Q_3	$P_1 = \{$"$com1$" $\xrightarrow{hasCreator} ?var \xrightarrow{posted}$ "$pst1$" $\xrightarrow{containedIn} ?var\}$
Q_4	$P_1 = \{?var \xrightarrow{hasMod} ?var \xrightarrow{posted}$ "$pst1$" $\xrightarrow{containedIn} ?var\}$

(b)

Figure 4: (a) Four query graph patterns that capture events generated inside a social network and (b) their covering paths.

end, we do not further discuss deletions on Q_{DB} and G, as we focus on providing high performance query answering algorithms.

4 TRIE-BASED CLUSTERING

To solve the problem defined in the previous section, we propose TRIC (TRIe-based Clustering). As motivated in Section 1, the *key idea* behind TRIC lies in the fact that query graph patterns overlap in their structural and attribute restrictions. After identifying and indexing these shared characteristics (Section 4.1), they can be exploited to batch-answer the indexed query set and in this way reduce query response time (Section 4.2).

4.1 Query Indexing Phase

TRIC indexes each query graph pattern Q_i by applying the following two steps:

1. Transforming the original query graph pattern Q_i into a set of path conjuncts, that cover all vertices and edges of Q_i, and when combined can effectively re-compose Q_i.
2. Indexing all paths in a trie-based structure along with unique query identifiers, while clustering all paths of all indexed queries by exploiting commonalities among them.

In the following, we present each step of the query indexing phase of Algorithm TRIC, give details about the data structures utilized and provide its pseudocode (Fig. 5).

$\underline{\mathcal{S}tep\ 1}$: **Extracting the Covering Paths.** In the first step of the query indexing process, Algorithm TRIC decomposes a query graph pattern Q_i and extracts a set of paths $CP(Q_i)$ (Fig. 5, line 1). This set of paths, covers all vertices $V \in Q_i$ and edges $E \in Q_i$. At first, we give the definition of a path and subsequently define and discuss the covering path set problem.

Definition 4.1. A *path* $P_i \in Q_i$ is defined as a list of vertices $P_i = \{v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \dots v_k \xrightarrow{e_k} v_{k+1}\}$ where $v_i \in Q_i$, such that two sequential vertices $v_i, v_{i+1} \in P_i$ have exactly one edge $e_i \in Q_i$ connecting them, i.e., $e_k = (v_k, v_{k+1})$.

Definition 4.2. The *covering paths* [1] *CP* of a query graph Q_i is defined as a set of paths $CP(Q_i) = \{P_1, P_2, \dots, P_k\}$ that cover all *vertices* and *edges* of Q_i. In more detail, we are interested in the least number of paths while ensuring that for every vertex $v_i \in Q_i$ there is at least one path P_j that contains v_i, i.e., $\forall i \exists j : v_i \in P_j$, $v_i \in Q_i$. In the same manner, for every edge $e_i \in Q_i$ there is at least one path P_j that contains e_i, i.e., $\forall i \exists j : e_i \in P_j$.

Obtaining the Set of Covering Paths. The problem of obtaining a set of paths that covers all vertices and edges is a graph optimization problem that has been studied in literature [1, 27]. In our approach, we choose to solve the problem by applying a greedy

```
       Input: Query $Q_i = (V_{Q_i}, E_{Q_i}, vars, l_V, l_E, \Sigma_V, \Sigma_E)$
       Output: $Q_{DB} \leftarrow Q_{DB} \cup Q_i$
Step 1 {  1  $Paths \leftarrow CP(Q_i)$; // Obtain the set of covering paths
Step 2 {  2  foreach $P_i \in Paths$ do // For each covering path $P_i$ of $Q_i$
          3      foreach trie $T_i$ with $root(T_i) = e_1 : e_1 \in P_i$ do
          4          $depthFirstSearch(T_i)$; // Traverse trie in DFS
                     // If there exists a trie that can store $P_i$
          5          if $\exists \{n_0 \rightarrow \dots n_i \rightarrow \dots n_k\} \subseteq P_i$ then
                         // Store the trie path positions
          6              $positions \leftarrow \{n_0 \rightarrow \dots n_i \rightarrow \dots n_k\}$;
                 // If all edges $e_i \in P_i$ cannot be indexed, create
                    additional trie nodes to index them
          7      if $positions \cap P_i \neq \emptyset$ then
          8          $create\_nodes(P_i \setminus positions)$;
          9      $last(positions) \leftarrow id(Q_i)$; // Store the query id
                 // Keep a reference to the last trie node
         10      $pathPositions \rightarrow pathPositions \cup last(positions)$;
                 // Store tries $T_i$ under which, edge $e_i$ is indexed
         11      foreach $e_i \in P_i$ do
         12          $edgeInd[e_i] \leftarrow T_i$;
             // Store the nodes that $Q_i$ was indexed under
         13  $queryInd[id(Q_i)] \leftarrow pathPositions$
```

Figure 5: Query indexing phase of Algorithm TRIC.

algorithm, as follows: For all vertices v_i in the query graph Q_i execute a depth-first walk until a leaf vertex (no outgoing edge) of the graph is reached, or there is no new vertex to visit. Subsequently, repeat this step until all vertices and edges of the query graph Q_i have been visited at least once and a list of paths has been obtained. Finally, for each path in the obtained list, check if it is a sub-path of an already discovered path, and remove it from the list of covering paths. The end result of this procedure yields the *set of covering paths*.

Example 4.3. In Fig. 4(a) we present four query graph patterns. These query graph patterns capture activities of users inside a social network. By applying Definition 4.2 on the four query graph patterns presented, Algorithm TRIC extracts four sets of covering paths, presented in Fig. 4(b).

Obtaining a set of paths serves two purposes: (a) it gives a less complex representation of the query graph that is easier to manage, index and cluster, as well as (b) it provides a streamlined approach on how to perform the materialization of the subgraphs that match a query graph pattern, i.e., the query answering during the evolution of the graph.

Materialization. Each edge e_i that is present in the query set has a materialized view that corresponds to its $matV[e_i]$. The materialized view of e_i stores all the updates u_i that contain e_i. In order to obtain the subgraphs that satisfy a query graph pattern Q_i all edges $e_i \in Q_i$ must have a non-empty materialized view (i.e., $matV \neq \emptyset$) and the materialized views should be joined as defined by the query graph pattern.

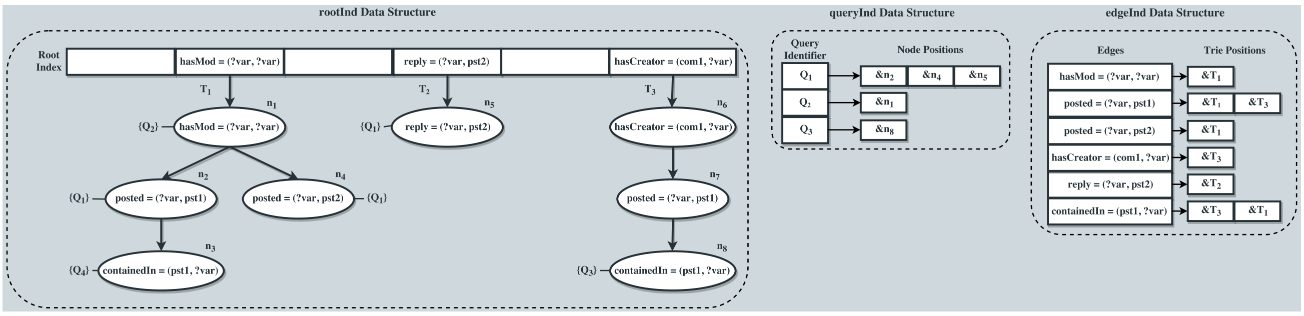
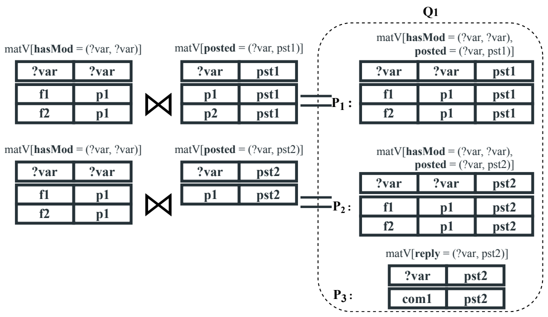


Figure 6: Data structures utilized by Algorithm TRIC to cluster query graph patterns.



matV[**hasMod** = (?var, ?var)]

?var	?var
f1	p1
f2	p1

matV[**posted** = (?var, pst1)]

?var	pst1
p1	pst1
p2	pst1

matV[**hasMod** = (?var, ?var), **posted** = (?var, pst1)]

?var	?var	pst1
f1	p1	pst1
f2	p1	pst1

matV[**hasMod** = (?var, ?var)]

?var	?var
f1	p1
f2	p1

matV[**posted** = (?var, pst2)]

?var	pst2
p1	pst2

matV[**hasMod** = (?var, ?var), **posted** = (?var, pst2)]

?var	?var	pst2
f1	p1	pst2
f2	p1	pst2

matV[**reply** = (?var, pst2)]

?var	pst2
com1	pst2

Figure 7: Materialized views of Q_1.

In essence, the query graph pattern determines the *execution plan* of the query. However, given that a query pattern in itself is a graph there is a high number of possible execution plans available. A path $P_i = \{v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \dots v_k\}$ serves as a model that defines the order in which the materialization should be performed. Thus, starting from the source vertex $v_1 \in P_i$ and joining all the materialized views from v_1 to the leaf vertex $v_k \in P_i : |P| = k$ yields all the subgraphs that satisfy the path P_i. After all paths P_i that belong in Q_i have been satisfied, a final join operation must be performed between all the paths. This join operation will produce the subgraphs that satisfy the query graph Q_i. To achieve this path joining set, additional information is kept about the intersection of the paths $P_i \in Q_i$. The intersection of two paths P_i and P_j are their common vertices.

Example 4.4. Fig. 7 presents some possible materialized views that correspond to the covering paths of query graph Q_1 (Fig. 4 (b)). In order to locate all subgraphs that satisfy the structural and attribute restrictions posed by paths P_1, P_2 and P_3 their materialized views should be calculated. More specifically, path $P_1 = \{?var \xrightarrow{hasMod} ?var \xrightarrow{posted}$ "$pst1$"$\}$, is formulated by two edges, edges $hasMod = (?var, ?var)$ and $posted = (?var, pst1)$, thus, their materialized views $matV[hasMod = (?var, ?var)]$ and $matV[posted = (?var, pst1)]$ must be joined. These two views contains all updates u_i that correspond to them, while the result of their join operation will be a new materialized view $matV[hasMod = (?var, ?var), posted = (?var, pst1)]$ as shown in Fig. 7. In a similar manner, the subgraphs that satisfy path P_2 are calculated, while P_3 that is formulated by a single edge does not require any join operations. Finally, in order to calculate the subgraphs that match Q_1 all materialized views that correspond to paths P_1, P_2 and P_3 must be joined.

$\underline{\mathcal{S}tep\ 2}$: **Indexing the Paths.** Algorithm TRIC proceeds into indexing all the paths, extracted in $\mathcal{S}tep$ 1, into a trie-based data structure. For each path $P_i \in CP(Q_i)$, TRIC examines the forest for trie roots that can index the first edge $e_1 \in P_i$ (Fig. 5, lines 3 − 6). To access the trie roots, TRIC utilizes a hash table (namely *rootInd*) that indexes the values of the root-nodes (keys) and the references to the root nodes (values). If such trie T_i is located, T_i is traversed in a *DFS* manner to determine in which sub-trie path P_i can be indexed (Fig. 5, line 4). Thus, TRIC traverses the forest to locate an existing trie-path $\{n_1 \rightarrow \dots n_i \rightarrow \dots n_k\}$ that can index the ordered set of edges $\{e_1, \dots, e_k\} \in P_i$. If the discovered trie-path can index P_i partially (Fig. 5, line 7), TRIC proceeds into creating a set of new nodes under n_k that can index the remaining edges (Fig. 5, line 8). Finally, the algorithm stores the identifier of Q_i at the last node of the trie path (Fig. 5, line 9).

Algorithm TRIC makes use of two additional data structures, namely *edgeInd* and *queryInd*. The former data structure is a hash table that stores each edge $e_i \in P_i$ (key) and a collection of trie roots T_i which index e_i as the hash table's value (Fig. 5, lines 11 − 12). Finally, TRIC utilizes a matrix *queryInd* that indexes the query identifier alongside the set of nodes under which its covering paths $P_i \in CP(Q_i)$ was indexed (Fig. 5, line 13).

Example 4.5. Fig. 6 presents an example of *rootInd*, *queryInd* and *edgeInd* of Algorithm TRIC when indexing the set of covering paths of Fig. 4 (b). Notice that TRIC indexes paths $P_1, P_2 \in Q_1$, path $P_1 \in Q_2$ and path $P_1 \in Q_4$ under the same trie T_1, thus, clustering together their common structural restrictions (all the aforementioned paths) and their attribute restrictions. Additionally, note that the *queryInd* data structure keeps references to the last node where each path $P_i \in Q_i$ is stored, e.g. for Q_1 it keeps a set of node positions $\{\&n_2, \&n_4, \&n_5\}$ that correspond to its original paths P_1, P_2 and P_3 respectively. Finally, *edgeInd* stores all the unique edges present in the path set of Fig. 4 (b), with references to the trie roots under which they are indexed, e.g. edge $posted = (?var, pst1)$ that is present in $P_1 \in Q_1$, $P_1 \in Q_3$ and $P_1 \in Q_4$, is indexed under both tries T_1 and T_3, thus this information is stored in set $\{\&T_1, \&T_3\}$.

The time complexity of Algorithm TRIC when indexing a path P_i, where $|P_i| = M$ edges and B the branching factor of the forest, is $\mathcal{O}(M * B)$, since TRIC uses a DFS strategy, with the maximum depth bound by the number of edges. Thus, for a new query graph pattern Q_i with N covering paths, the total time complexity is $\mathcal{O}(N * M * B)$. Finally, the space complexity of Algorithm TRIC when indexing a query Q_i is $\mathcal{O}(N * M)$, where M is the number of edges in a path and N the cardinality of Q_i's covering paths.

Variable Handling. A query graph pattern Q_i contains vertices that can either be literals (specific entities in the graph) identified by their label, or variables denoted as "?var". This approach alleviates restrictions posed by naming conventions and thus leverages on the common structural constraints of paths.

However, by substituting the variable vertices with the generic "?var" requires to keep information about the joining order of each edge $e_i \in P_i$, as well as, how each $P_i \in CP(Q_i)$ intersects with the

```
Input: Update $u_i = (e_i) : e_i = (s, t)$
Output: Locate matched queries
Step 1
1  $affectedTries \leftarrow edgeInd[e_i]$; // Get affected tries
2  foreach $T_i \in affectedTries$ do
3      foreach node $n_i \in T_i$ do // Traverse $T_i$ in DFS
4          if $edge(n_c) = e_i$ then // If current node indexes $e_i$
5              $fndPos \leftarrow n$ ; // Store the position
6              break ;              // Terminate the traversal
       // Update $matV$s of $fndPos$ and its children
7      $affectedQueries \leftarrow$ Trie Traversal & Materialization ($fndPos$)
8  foreach query $Q_i \in affectedQueries$ do
9      $results \leftarrow \emptyset$;
10     foreach $P_i \in Q_i$ do // For the covering paths of $Q_i$
11         $results \leftarrow results \bowtie matV[P_i]$;
12     if $results \neq \emptyset$ then
13         $mark\_Matched(Q_i)$;
```

Figure 8: Query answering phase ($\mathcal{S}tep$ 1) of Algorithm TRIC.

rest of the paths in $CP(Q_i)$. In order to calculate the subgraphs that satisfy each covering path $P_i \in CP(Q_i)$, each $matV[e_i] : e_i \in P_i$ must be joined. Each path P_i that is indexed under a trie path $\{n_1 \rightarrow \ldots n_i \rightarrow \ldots n_k\}$ maintains the original ordering of its edges and vertices, while the order under which each edge of a node n_i is connected with its children nodes ($chn(n_i)$), is determined as follows: the target vertex $t \in e_i$ (where e_i is indexed under n_i) is connected with the source node $s \in e_{i+1} : e_{i+1} \in chn(n_i)$ of the parent node n_i. Finally, for each covering path $P_i \in CP(Q_i)$ TRIC maintains information about the vertices that intersected in the original query graph pattern Q_i, while this information is utilized during the query answering phase.

4.2 Query Answering Phase

During the evolution of the graph, a constant stream of updates $S = (u_1, u_2, \ldots, u_k)$ arrives at the system. For each update $u_i \in S$ Algorithm TRIC performs the following steps:

1. Determines which tries are affected by update u_i and proceeds in examining them.
2. While traversing the affected tries, performs the materialization and prunes sub-tries that are not affected by u_i.

In the following, we describe each step of the query answering phase of Algorithm TRIC. The pseudocode for each step is provided in Figs. 8 and 10.

$\underline{\mathcal{S}tep\ 1}$: **Locate and Traverse Affected Tries.** When an update u_i arrives at the system, Algorithm TRIC utilizes the edge $e_i \in u_i$ to locate the tries that are affected by u_i. To achieve this, TRIC uses the hash table *edgeInd* to obtain the list of tries that contain e_i in their children set. Thus, Algorithm TRIC receives a list (*affectedTries*) that contains all the tries that were affected by u_i and must be examined (Fig. 8, line 1). Subsequently, Algorithm TRIC proceeds into examining each trie $T_i \in affectedTries$ by traversing each T_i in order to locate the node n_i that indexes edge $e_i \in u_i$. When node n_i is located, the algorithm proceeds in $\mathcal{S}tep2$ of the query answering process described below (Fig. 8, lines 3 – 7).

Example 4.6. Let us consider the data structures presented in Fig. 6, the materialized views in Fig. 9, and an update $u_1 = (posted = (p2, pst1))$ that arrives into the evolving graph (Fig. 9(a)). Algorithm TRIC prompts hash table *edgeInd* and obtains list $\{\&T_1, \&T_3\}$. Subsequently, TRIC will traverse tries T_1 and T_3. When traversing trie T_1 TRIC locates node n_2 that matches update $e_1 \in u_1$ and proceeds in $\mathcal{S}tep2$ (described below). Finally, when traversing T_3 TRIC will stop the traversal at root node n_6 as its materialized view is empty $matV[hasCreator = (pst1, ?var)] = \emptyset$ (Fig. 9 (b)), thus all sub-tries will yield empty materialized views.

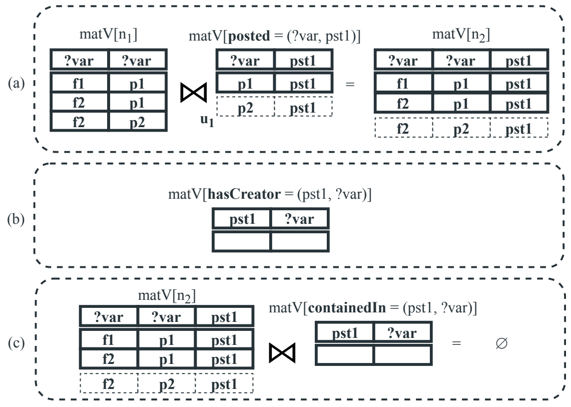


Figure 9: Updating materialized views.

```
Function: Trie Traversal & Materialization
Input: Node $n_i$
Output: Locate matched queries
// Update the current materialized view by joining the
   parent materialized view with the materialized view
   of the edge in node $n_i$
Step 2
1  $result \leftarrow matV[prnt(n_i)] \bowtie matV[edge(n_i)]$;
2  if $result = \emptyset$ then
3      return;
   // Store the query identifiers of node $n_i$
4  $affectedQueries \leftarrow affectedQueries \cup qIDs(n_i)$;
   // Recursively update the $matV$s of $n_i$'s children
5  foreach $n_c \in chn(n_i)$ do
6      Trie Traversal & Materialization ($n_c$);
7  return $affectedQueries$ ; // Return the affected $qIDs$
```

Figure 10: Query answering phase ($\mathcal{S}tep$ 2) of Algorithm TRIC.

$\underline{\mathcal{S}tep\ 2}$: **Trie Traversal and Materialization.** Intuitively, a trie path $\{n_0 \rightarrow \ldots n_i \rightarrow \ldots n_k\}$ represents a series of joined materialized views $matVs = \{matV_1, matV_2, \ldots, matV_k\}$. Each materialized view $matV_i \in matVs$ corresponds to a node n_i that stores edge e_i and the materialized view $matV_i$. The materialized view contains the results of the join operation between the $matV[e_i]$ and the materialized view of the parent node n_i ($matV(prnt(n_i))$), i.e., $matV_i = matV[prnt(n_i)] \bowtie matV[e_i]$. Thus, when an update u_i affects a node n_i in this "chain" of joins, n_i's and its children's ($chn(n_i)$) materialized views must be updated with u_i. Based on this TRIC searches for and locates node n_i inside T_i that is affected by u_i and updates n_i's sub-trie.

After locating node $n_i \in T_i$ that is affected by u_i, Algorithm TRIC continues the traversal of n_i's sub-trie and prunes the remaining sub-tries of T_i (Fig. 8, line 7). Subsequently, TRIC updates the materialized view of n_i by performing a join operation between its parent's node materialized view $matV[prnt(n_i)]$ and the update u_i, i.e., $results = matV[prnt(n_i)] \bowtie u_i$. Notice that Algorithm TRIC calculates the subgraphs formulated by the current update solely based on the update u_1 and does not perform a full join operation between $matV[prnt(n_i)]$ and $matV[edge(n_i)]$, the updated results are then stored in the corresponding $matV[n_i]$.

For each child node $n_j \in chn(n_i)$, TRIC updates its corresponding materialized view by joining its view $matV[n_j]$ that corresponds to the edge that it stores (given by $matV[edge(n_j)]$) with its parent node materialized view $matV[n_i]$ (Fig. 10, lines 1 – 7). If at any point the process of joining the materialized views returns an empty result set the specific sub-trie is pruned, while, the traversal continues in a different sub-trie of T_i (Fig. 10, lines $5-6$). Subsequently, for each trie node n_j in the trie traversal when there is a successful join operation among $matV[e_j] : e_j \in n_j$ and $matV[n_i]$, the query identifiers indexed under n_j are stored in *affectedQueries* list (Fig. 10, lines 4 and 7). Note that similarly to before, only the updated part of a materialized view is utilized as the parent's materialized view, an approach applied on database-management system [16].

Example 4.7. Let us consider the data structures presented in Fig. 6, Fig. 9, and an update $u_1 = (posted = (p2, pst1))$ that arrives into the evolving graph. After locating the affected trie node n_2 (described in Example 4.6) TRIC proceeds in updating the materialized view of n_2, i.e., $matV[n_2]$, by calculating the join operation between its parents materialized view, i.e., $matV[n_1]$ and the update u_1. Fig. 9, demonstrates the operations of joining $matV[n_2]$ with update u_1, the result of the operation is tuple $(f2, p2, pst1)$, which is added into $matV[n2]$, presented in Fig. 9(a). While the query identifiers of n_2 (i.e., Q_1) are indexed in *affectedQueries*. Finally, TRIC proceeds in updating the sub-trie of n_2, node n_3, where the updated tuple $(f2, p2, pst1)$ is joined with $matV[edge(n_3)]$ (i.e., $matV[containedIn = (pst1, ?var)]$). This operation yields an empty result (Fig. 9(c)), thus terminating the traversal.

Finally, to complete the filtering phase Algorithm TRIC iterates through the affected list of queries and performs the join operations among the paths that form a query, thus, yielding the final answer (Fig. 8, lines $8-13$).

The time complexity, of Algorithm TRIC when filtering an update u_i, is calculated as follows: The traversal complexity is $\mathcal{O}(T * (P_m * B))$, where T denotes the number of tries that contain $e_i \in u_i$, P_m denotes the size of the longest trie path, and B the branching factor. The time complexity of joining two materialized views $matV_1$ and $matV_2$, where $|matV_1| = N$ and $|matV_2| = M$, is $\mathcal{O}(N * M)$. Finally, the total time complexity is calculated as $\mathcal{O}((T * (P_m * B)) * (N * M))$.

Caching. During $\mathcal{S}tep$ 2, two materialized views are joined using a typical hash join operation with a build and a probe phase. In the build phase, a hash table for the smallest (in the number of tuples) table is constructed, while in the probe phase the largest table is scanned and the hash table is probed to perform the join. TRIC discards all the data structures and intermediate results after the join operation commences. In order to enhance this resource intensive operation, we cache and reuse the data structures generated during the build and probe phases as well as the intermediate results whenever possible. This approach constitutes an extension of our proposed solution (TRIC) and it is coined TRIC+.

4.3 Supporting richer models and languages

The proposed algorithm is easily extensible to more sophisticated data models and query languages; in this section, we briefly outline the necessary modifications to support graph deletions and updates, as well as more general types of graphs (e.g., property graphs).

Edge deletions may be handled by algorithms TRIC and TRIC+ by locating the affected paths, and traversing each path to locate the deleted edges; while visiting each edge, the materialized view that corresponds to that edge should be accessed and all affected tuples should be removed. Updates on the graph (e.g., on the label of an edge) may be modeled as an edge deletion followed by an edge addition operation. Finally, extending our solution for more general graph types, like property graphs, entails the addition of extra constraints within the nodes of the tries and the usage of a separate data structure to appropriately index these constraints. Then, query answering would include an extra phase for the determining the satisfaction of the additional constraints. Efficient execution of such extensions is an interesting topic for future research (see Section 7).

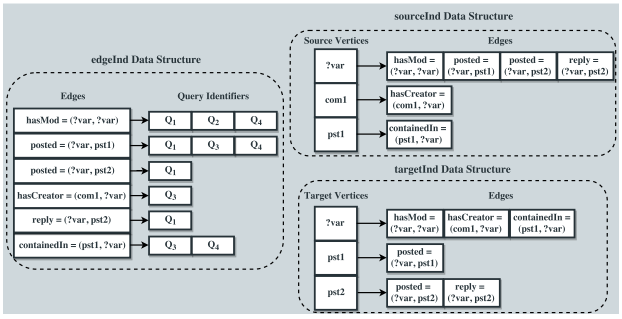


Figure 11: Index structures utilized by Algorithm INV.

5 ADVANCED BASELINES

Since no prior work in the literature considers the problem of continuous multi-query evaluation, we designed and implemented Algorithms INV and INC, two advanced baselines that utilize inverted index data structures. Finally, we provide a third baseline that is based on the well-established graph database Neo4j [43].

5.1 Algorithm INV

Algorithm INV (INVerted Index), utilizes inverted index data structures to index the query graph patterns. The inverted index data structure is able to capture and index common elements of the graph patterns at the edge level during indexing time. Subsequently, the inverted index is utilized during filtering time to determine which queries have been satisfied. In the following, we describe the query indexing and answering phase of INV.

The Query Indexing Phase of Algorithm INV, for each query graph pattern Q_i, is performed in two steps: (1) Transforming the original query graph pattern Q_i into a set of path conjuncts, that cover all vertices and edges of Q_i, and when combined can effectively re-compose Q_i, and finally, indexing those covering paths in a matrix along the unique query identifier, (2) Indexing all edges $e_i \in Q_i$ into an inverted index structure. In the following, we present each step of the query indexing phase of INV and give details about the data structures utilized.

$\underline{\mathcal{S}tep\ 1:}$ **Extracting the Covering Paths.** In the first step of the query indexing phase, Algorithm INV decomposes a query graph Q_i into a set of paths CP, a process described in detail in Section 4.1. Thus, given the query set presented in Fig. 4 (a), INV yields the same set of covering paths CP (Fig. 4 (b)). Finally, the covering path set CP is indexed into an array (*queryInd*) with the query identifier of Q_i.

$\underline{\mathcal{S}tep\ 2:}$ **Indexing the Query Graph.** Algorithm INV builds three inverted indexes, where it stores the structural and attribute constrains of the query graph pattern Q_i. Hash table *edgeInd* indexes all edges $e_i \in Q_{DB}$ (keys), and the respective query identifiers as values, hash table *sourceInd* indexes the source vertices of each edge (key), where the edges are indexed as values , and hash table *targetInd* that indexes the target vertices of each edge (key), where the edges are indexed as values. In Fig. 4(a) we present four query graph patterns, and in Fig. 11 the data structures of

INV when indexing those queries. Finally, INV applies the same techniques of handling variables as Algorithm TRIC (Section 4.1).

The Query Answering Phase of Algorithm INV, when a constant stream of updates $S = (u_1, u_2, \ldots, u_k)$ arrives at the system, is performed in three steps: (1) Determines which queries are affected by update u_i, (2) Prompts the inverted index data structure and determines which paths have been affected by update u_i, (3) Performs the materialization while querying the inverted index data structures. In the following, we describe each step of the query answering phase:

$\underline{\mathcal{S}tep\ 1}$: **Locate the Affected Queries.** When a new update u_i arrives at the system, Algorithm INV utilizes the edge $e_i \in u_i$ to locate the queries that are affected, by querying the hash table *edgeInd* to obtain the query identifier *qIDs* that contain e_i. Subsequently, the algorithm iterates through the list of *affectedQueries* and checks each query $Q_i \in qIDs$. For each query Q_i the algorithm checks $\forall e_i \in Q_i$ if $matV[e_i] \neq \emptyset$, i.e., each e_i should have a *non empty* materialized view. The check is performed by iterating through the edge list that is provided by *queryInd* and a hash table that keeps all materialized views present in the system. Intuitively, a query Q_i is candidate to match, as long as, all materialized views that correspond to its edges can be used in the query answering process.

$\underline{\mathcal{S}tep\ 2}$: **Locate the Affected Paths.** Algorithm INV proceeds to examine the inverted index structures *sourceInd* and *targetInd* by making use of $e_i \in u_i$. INV queries *sourceInd* and *targetInd* to determine which edges are affected by the update, by utilizing the source and target vertices of update u_i. INV examines each current edge e_c of the affected edge set and recursively visits all edges connected to e_c, which are determined by querying the *sourceInd* and *targetInd*. While examining the current edge e_c, INV checks if e_c is part of *affectedQueries*, if not, the examination of the specific path is pruned. For efficiency reasons, the examination is bound by the maximum length of a path present in *affectedQueries* which is calculated by utilizing the *queryInd* data structure.

$\underline{\mathcal{S}tep\ 3}$: **Path Examination and Materialization.** While INV examines the paths affected by update u_i ($\mathcal{S}tep$ 2), it performs the materialization on the currently examined path. INV searches through the paths formulated by the visits of edge sets determined by *targetInd* and *sourceInd*, and maintains a path $P_c = \{v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \ldots v_k \xrightarrow{e_k} v_{k+1}\}$ that corresponds to the edges already visited.

While, visiting each edge e_c, INV accesses the materialized view that corresponds to it (i.e., $matV[e_c]$) and updates the set of materialized views $matVs = \{matV_1, matV_2, \ldots, matV_k\}$ that correspond to the current path. For example, given an already visited path $P = \{v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} v_3\}$ its materialized view $matV[P]$ will be generated, by $matV[P] = matV[e_1] \bowtie matV[e_2]$. When visiting the next edge e_n, a new path P' is generated and its materialized view $matV[P'] = matV[P] \bowtie matV[e_n]$ will be generated. If at any point, the process of joining the materialized views yields an empty result set the examination of the edge is terminated (pruning). This allows us to prune paths that are not going to satisfy any $Q_i \in affectedQueries$. If a path P_i yields a successful series of join operations (i.e., $matV[P_i] \neq \emptyset$), it is marked as matched.

Finally, to produce the final answer subgraphs Algorithm INV iterates through the affected list of queries $qIDs \in affectedQueries$ and performs the final join operation among all the paths that comprise the query.

Caching. In the spirit of TRIC+ (Section 4.2), we developed an extension of INV, namely INV+, that caches and reuses the calculated data structures of the hash join phase.

5.2 Algorithm INC

Based on Algorithm INV we developed an algorithmic extension, namely Algorithm INC. Algorithm INC utilizes the same inverted index data structures to index the covering paths, edges, source and target vertices as Algorithm INV, while the examination of a path affected during query answering remains similar. The key difference lies in executing the joining operations between the materialized views that correspond to edges belonging to a path. More specifically, when Algorithm INV executes a series of joins between the materialized views (that formulate a path) to determine which subgraphs match a path; it utilizes all tuples of each materialized view that participate in the joining process. On the other hand, Algorithm INC makes use of only the update u_i and thus reduces the number of tuples examined through out the joining process of the paths.

Caching. In the spirit of TRIC+ (Section 4.2), we developed an extension of INC, namely INC+, that caches and reuses the calculated data structures of the hash join phase.

5.3 Neo4j

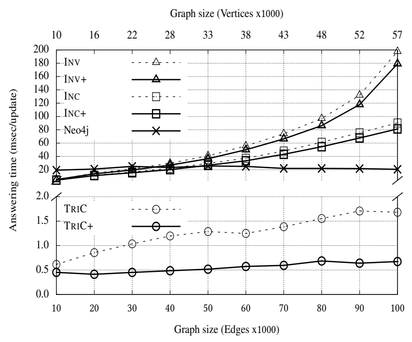
To evaluate the efficiency of the proposed algorithm against a real-world approach, we implemented a solution based on the well-established graph database Neo4j [43]. In this approach, we extend Neo4j's native functionality with auxiliary data structures to efficiently store the query set. They are used during the answering phase to located affected queries and execute them on Neo4j.

The Query Indexing Phase. To address the continuous multi-query evaluation scenario, we designed main-memory data structures to facilitate indexing of query graph patterns. Initially, in the preprocessing phase, we convert each incoming query Q_i into Neo4j's native query language Cypher[1]. Subsequently, the query indexing phase of Neo4j commences as follows: (1) indexing each Cypher query in the *queryInd* data structure, and (2) indexing all edges $e_i \in Q_i$ in the *edgeInd* data structure where e_i is used as key, and a collection of query identifiers as values. The *queryInd* structure is defined as matrix, while the *edgeInd* is an inverted index, similarly to the data structures described in Section 5.1.

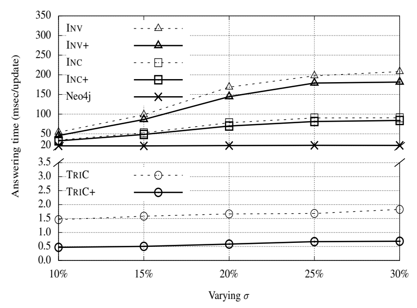
The Query Answering Phase. Each update that is received as part of an incoming stream of updates $S = (u_1, u_2, \ldots, u_k)$ is processed in the following steps: (1) an incoming update u_i is applied to Neo4j (2) the inverted index *edgeInd* is queried with $e_i \in u_i$, to determine which queries are affected, (3) all affected queries are retrieved from matrix *queryInd*, (4) the affected queries are executed.

To enhance performance, the following configurations are applied: (1) the graph database builds indexes on all labels of the schema allowing for faster look up times of nodes, (2) the execution of Cypher queries employs the *parameters syntax*[2] as it enables the execution planner of Neo4j to cache the query plans for future use, (3) the number of writes per transaction[3] in the database and the allocated memory were optimized based on the hardware configuration (see Section 6.1).

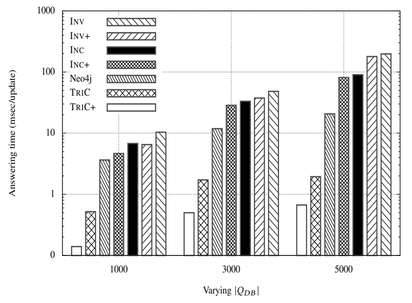
[1]https://neo4j.com/developer/cypher-query-language/
[2]https://neo4j.com/docs/cypher-manual/current/syntax/parameters/
[3]https://neo4j.com/docs/cypher-manual/current/introduction/transactions/



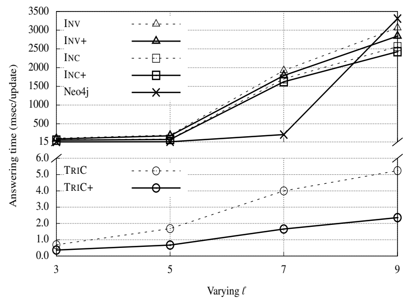
(a) Influence of graph size.



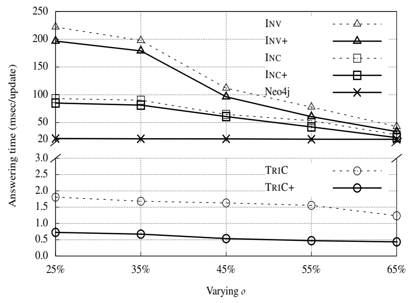
(b) Influence of σ.



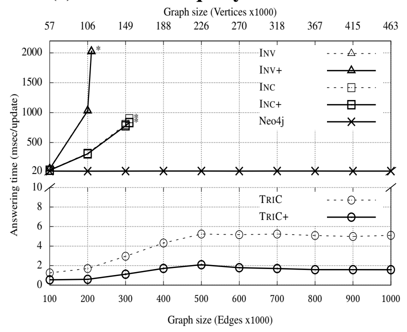
(c) Influence of query database size.



(d) Influence of ℓ.



(e) Influence of o.



(f) Influence of graph size.

Figure 12: Query answering time for the SNB dataset.

6 EXPERIMENTAL EVALUATION

In this section, we present a series of experiments that compare the performance of the presented algorithms.

6.1 Experimental Setup

Data and Query Sets. For the experimental evaluation we used a synthetic and two real-world datasets.

The SNB Dataset. The first dataset we utilized is the LDBC Social Network Benchmark (*SNB*) [12]. *SNB* is a synthetic benchmark designed to accurately simulate the evolution of a social network through time (i.e, vertex and edge sets labels, event distribution etc). This evolution is modeled using activities that occur inside a social network (i.e. user account creation, friendship linking, content creation, user interactions etc). Based on the *SNB* generator we simulated the evolution of a graph consisting of user activities over a time period of 2 years. From this dataset we derived 3 query loads and configurations: (i) a set with a graph size of $|G_E| = 100K$ edges and $|G_V| = 57K$ vertices, (ii) a set with a graph size of $|G_E| = 1M$ edges and $|G_V| = 463K$ vertices, and (iii) a set with a graph size of $|G_E| = 10M$ edges and $|G_V| = 3.5M$.

The NYC Dataset. The second dataset we utilized is a real world set of taxi rides performed in New York City[4] (*TAXI*) in 2013 utilized in *DEBS 2015 Grand Challenge* [19]. *TAXI* contains more that $160M$ entries of taxi rides with information about the license, pickup and drop-off location, the trip distance, the date and duration of the trip, and the fare. We utilized the available data to generate a stream of updates that result in a graph of $|G_E| = 1M$ edges and $|G_V| = 280K$, accompanied by a set of $5K$ query graph patterns.

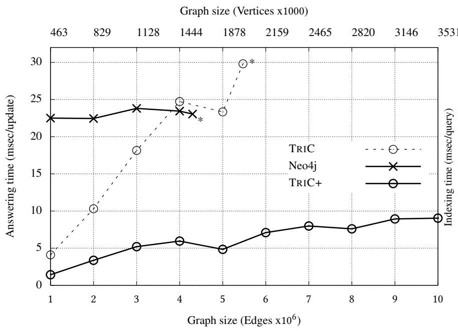
The BioGRID Dataset. The third dataset we utilized is BioGRID [35], a real world dataset that represents protein to protein interactions. This dataset is used as a stress test for our algorithms since it contains one type of edge (interacts) and vertex (protein), and thus every update affects the whole query database. We used BioGRID to generate a stream of updates that result in a graph size of $|G_E| = 1M$ edges and $|G_V| = 63K$ vertices, with a set of $5K$ query graph patterns.

Query Set Configuration. In order to construct the set of query graph patterns Q_{DB} we identified three distinct query classes that are typical in the relevant literature: chains, stars, and cycles [15, 26]. Each type of query graph pattern was chosen equiprobably during the generation of the query set. The baseline values for the query set are: (i) an average size ℓ of 5 edges/query graph pattern, a value derived from the query workloads presented in *SNB* [12], (ii) a query database $|Q_{DB}|$ size of $5K$ graph patterns, (iii) a factor that denotes the percentage of the query set Q_{DB} that will ultimately be satisfied, denoted as selectivity $\sigma = 25\%$, and (iv) a factor that denotes the percentage of overlap between the queries in the set, $o = 35\%$.

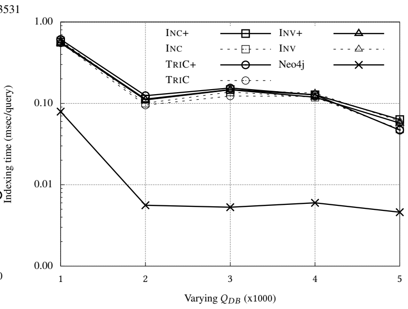
Metrics. In our evaluation, we present and discuss the filtering and indexing time of each algorithm, along with the total memory requirements.

Technical Configuration. All algorithms were implemented in Java 8 while for the materialization implementation the Stream API was employed. The Neo4j-based approach was implemented using the embedded version of Neo4j 3.4.7. Extensive experimentation evaluation concluded that a transactionsection 5.3 can perform up to $20K$ writes in the database without degrading Neo4j's performance, while in order to guarantee indexes are cached in main memory $55GB$ of main memory were allocated. A machine with Intel(R) Xeon(R) Processor E5-2650 at $2.00GHz$, $64GB$ RAM, and Ubuntu Linux 14.04 was used. The time shown in the graphs is wall-clock time and the results of each experiment are averaged over 10 runs to eliminate fluctuations in measurements.

[4]https://chriswhong.com/open-data/foil_nyc_taxi/



(a) Query answering time.



(b) Query insertion time.

Algorithm	Dataset		
	SNB	*NYC*	*BioGRID*
TRIC	201*MB*	257*MB*	233*MB*
TRIC+	248*MB*	273*MB*	262*MB*
INV	205*MB*	273*MB*	271*MB*^{50K}
INV+	228*MB*	381*MB*	301*MB*^{50K}
INC	206*MB*	273*MB*	270*MB*^{50K}
INC+	228*MB*	378*MB*	310*MB*^{60K}
Neo4j	443*MB*	590*MB*	314*MB*

(c) Memory requirements.

Figure 13: Results for (a) & (b) the *SNB* dataset, and (c) the *SNB*, *TAXI*, *BioGRID* datasets.

6.2 Results for the SNB Dataset

In this section, we present the evaluation for the *SNB* benchmark and highlight the most significant findings.

Query Answering Time. Fig. 12(a) presents the results regarding the query answering time, i.e., the average time in milliseconds needed to determine which queries are satisfied by an incoming update, against a query set of $Q_{DB} = 5K$. Please notice that the y-axis is split due to the high differences in the performance of TRIC/TRIC+and its competitors. We observe that the answering time increases for all algorithms as the graph size increases. Algorithms TRIC/TRIC+ achieve the lowest answering times, suggesting better performance. Contrary, the competitors are more sensitive in graph size changes, with Algorithm INV performing the worst (highest query answering time). When comparing Algorithm TRIC to INV, INC and Neo4j the query answering time is improved by 99.15%, 98.14% and 91.86% respectively, while the improvement between INC and INV is 54.33%. Finally, comparing Algorithm TRIC+ to INV+, INC+ and Neo4j demonstrates a performance improvement of 99.62%, 99.17% and 96.74% respectively, while the difference of INC+ and INV+ is 54.6%.

The results (Fig. 12(a)) suggest that all solutions that implement caching are faster compared to the versions without it. In more detail, Algorithms TRIC+/INV+/INC+ are consistently faster than their non-caching counterparts, by 59.95%, 9.36% and 9.91% respectively. This is attributed to the fact that Algorithms TRIC/INV/INC, have to recalculate the probe and build structures required for the joining process, in contrast to Algorithms TRIC+/INV+/INC+ that store these structures and incrementally update them, thus providing better performance.

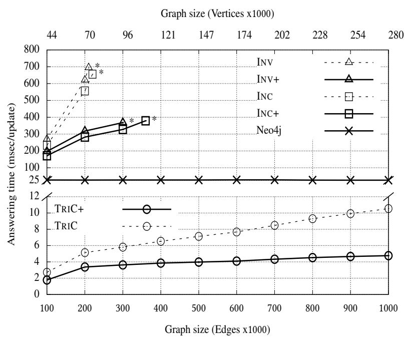
In Fig. 12(b) we present the results when varying the parameter σ, for 10%, 15%, 20%, 25% and 30% of a query set for $|Q_{DB}| = 5K$ and $|G_E| = 100K$. In this setup the algorithms are evaluated for a varying percentage of queries that match. A higher number of queries satisfied, increases the number of calculations performed by each algorithm. The results show that all algorithms behave in a similar manner as previously described. In more detail, Algorithm TRIC+ is the most efficient of all, and thus the fastest among the extensions that utilize caching, while TRIC is the most efficient solution among the solutions that do not employ a caching strategy. Finally, the percentage differences, between the algorithmic solutions remain the same as before in most cases.

Fig. 12(c) presents the results of the experimental evaluation when varying the size of the query database $|Q_{DB}|$. More specifically, we present the answering time per triple when $|Q_{DB}| = 1K$, $3K$ and $5K$, and $|G_E| = 100K$. Please notice the y-axis is in logarithmic scale. The results demonstrate that all algorithm's behavior is aligned with our previous observations. More specifically, Algorithms TRIC+ and TRIC exhibit the highest performance (i.e., lowest answering time), throughout the increase of $|Q_{DB}|$, and thus determine faster which queries of $|Q_{DB}|$ have matched given an update u_i. Similarly to the previous setups, the competitors have the lowest performance, while Algorithms INC and INC+ perform better compared to INV and INV+.

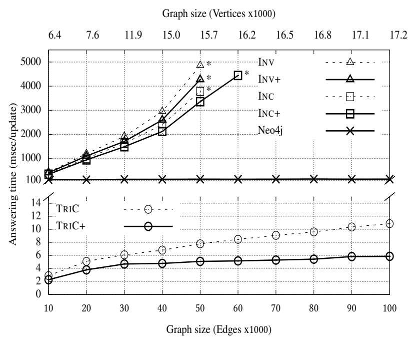
In Fig. 12(d) we give the results of the experimental evaluation when varying the average query size ℓ. More specifically, we present the answering time per triple when $\ell = 3$, 5, 7 and 9 of a query set for $|Q_{DB}| = 5K$ and $|G_E| = 100K$. We observe that the answering time increases for all algorithms as the average query length increases. More specifically, Algorithms TRIC+ and TRIC exhibit the highest performance (i.e., lowest answering time), throughout the increase of ℓs, and thus determine faster which queries have been satisfied. Similarly to the previous evaluation setups, the Algorithms INV/INV+/INC/INC+/Neo4j have the lowest performance, and increase significantly their answering time when ℓ increases, while Algorithms INC and INC+ perform better compared to INV, INV+ and Neo4j when $\ell = 9$.

Fig. 12(e) gives the results of the experimental evaluation when varying the parameter o, for 25%, 35%, 45%, 55% and 65% of a query set for $|Q_{DB}| = 5K$ and $|G_E| = 100K$. In this setup the algorithms are evaluated for varying percentage of query overlap. A higher number of query overlap, should decrease the number of calculations performed by algorithms designed to exploit commonalities among the query set. The results show that all algorithm behave in a similar manner as previously described, while AlgorithmsINV/INV+/INC/INC+ observe higher performance gains. Algorithm TRIC+ is the most efficient of all, and thus the fastest among the extensions that utilize caching techniques, while TRIC is the most efficient solution among the solutions that do not employ caching.

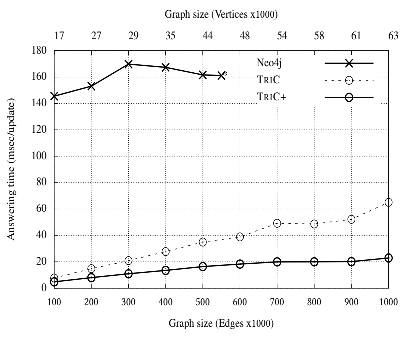
Fig. 12(f) presents the results regarding the query answering time, for all algorithms when indexing a query set of $|Q_{DB}| = 5K$ and a final graph of $|G_E| = 1M$ and $|G_V| = 463K$. Given the extremely slow performance of some algorithms we have set an *execution time threshold* of 24 hours, for all algorithms under evaluation, thus, when the threshold was crossed the evaluation was terminated. Algorithms TRIC/TRIC+ achieve the lowest answering times, suggesting better performance, while Algorithms INV/INV+/INC/INC+ are more sensitive in graph size changes and thus fail to terminate within the time threshold. More specifically, Algorithms INV/INV+ *time out* at $|G_E| = 210K$, while INC/INC+ *time out* at $|G_E| = 310K$ as denoted by the asterisks



(a) Influence of graph size.



(b) Influence of graph size.



(c) Influence of graph size.

Figure 14: Query answering time for (a) the *TAXI* dataset, and (b) & (c) the *BioGRID* dataset.

in the plot. When comparing TriC/TriC+ to Neo4j the query answering is improved by 77.01% and 92.86% respectively.

Fig. 13(a) presents the results regarding the query answering time, for Algorithms TriC, TriC+ and Neo4j when indexing a query set of $Q_{DB} = 5K$ and a final graph size of $|G_E| = 10M$ and $|G_V| = 3.5M$. Again, we have set an *execution time threshold* of 24 hours, for all the algorithms under evaluation. Algorithm TriC+ achieves the lowest answering times, suggesting better performance, while Algorithms TriC and Neo4j fail to terminate within the given time threshold. More specifically, Algorithm TriC *times out* at $|G_E| = 5.47M$, while Algorithm Neo4j times out at $|G_E| = 4.3M$ as denoted by the asterisks in the plot.

Overall, Algorithms TriC+ and TriC, the two solutions that utilize trie structures to capture and index the common structural and attribute restrictions of query graphs achieve the lowest query answering times, compared to Algorithms Inv/Inv+/Inc/Inc+ that employ no clustering techniques, as well as when compared with commercial solutions such as Neo4j. Adopting the incremental joining techniques (found in Algorithm TriC) into Algorithm Inc does not seem to significantly improve its performance when compared to Algorithm Inv. While, adopting caching techniques that store the data structures generated during the join operations, changes significantly the performance of Algorithm TriC+. Taking all the above into consideration, we conclude that the algorithms that utilize trie-based indexing achieve low query answering times compared to their competitors.

Indexing Time. Fig. 13(b) presents the indexing time in milliseconds required to insert $1K$ query graph patterns when the query database size increases. We observe that the time required to go from an empty query database to a query database of size $1K$ is higher compared to the time required for the next iterations. Please notice the y-axis is in logarithmic scale. This can be explained as follows, all algorithms utilize data structures that need to be initialized during the initial stages of query indexing phase, i.e. when inserting queries in an empty database, while as the queries share common restrictions less time is required for creating new entries in the data structures. Additionally, the time required to index a query graph pattern in the database does not vary significantly for all algorithms. Notice that query indexing time is not a critical performance parameter in the proposed paradigm, since the most important dimension is query answering time.

6.3 Results for the NYC and BioGRID Dataset

In this section, we present the evaluation for the *NYC* and *BioGRID* dataset and highlight the most significant findings.

The *NYC* Dataset. Fig. 14(a) presents the results from the evaluation of the algorithms for the *NYC* dataset. More specifically, we present the results regarding the query answering performance of all algorithms when $Q_{DB} = 5K$, $\ell = 5$, $o = 35\%$, $\sigma = 25\%$ and an execution time threshold of 24 hours. Please notice that the y-axis is split due to high differences in the performance of the algorithms. Algorithms Inv and Inv+ fail to terminate within the time threshold and *time out* at $|G_E| = 210K$ and $|G_E| = 300K$ respectively. Similarly, Algorithms Inc and Inc+ *time out* at $|G_E| = 220K$ and $360K$ respectively. When comparing Algorithms TriC and TriC+ to Neo4j the query answering is improved by 59.68% and 81.76% respectively. These results indicate that again an algorithmic solution that exploits and indexes together the common parts of query graphs (i.e., Algorithms TriC/TriC+) achieves significantly lower query answering time compared to approaches that do not apply any clustering techniques (i.e., AlgorithmsInv/Inv+Inc/Inc+/Neo4j).

The *BioGRID* Dataset. Figs. 14(b) and 14(c) present the results from the evaluation of the algorithms for the *BioGRID* dataset. In Fig. 14(b) we present the results regarding the query answering performance of the algorithms, when $Q_{DB} = 5K$, $\sigma = 25\%$ for a final graph size of $|G_E| = 100K$ and $|G_V| = 17.2K$. Additionally, we set an execution time threshold of 24 hours due to the high differences in the performance of the algorithms. The *BioGRID* dataset serves as a stress test for our algorithms, since it contains only one type of edge and vertex, thus each incoming update will affect (but not necessarily satisfy) the entire query database. To this end, Algorithms Inv/Inv+/Inc exceed the time threshold and time out at $|G_E| = 50K$, while Inc+ times out at $|G_E| = 60K$ as denoted by the asterisks in the plot. Finally, Fig. 14(c) presents the results for the *BioGRID* dataset for a final graph size of $|G_E| = 1M$ and $|G_V| = 63K$. We again observe that Algorithms TriC and TriC+ achieve the lowest answering time, while Neo4j exceeds the time threshold and times out at $|G_E| = 550K$. As it is demonstrated from the results yielded by the evaluation, Algorithms TriC and TriC+ are the most efficient of all; this is attributed to the fact that both algorithms create a combined representation of the query graph patterns that can efficiently be utilized during query answering time.

Comparing Memory Requirements. Fig. 13(c) presents the memory requirements of each algorithm, for the *SNB*, *NYC* and *BioGRID* datasets when indexing $|Q_{DB} = 5K|$ and a graph of $|G_E| = 100K$. We observe, that across all datasets, Algorithms TriC, Inv and Inc have the lowest main memory requirements, while, Algorithms TriC+, Inv+, Inc+ and Neo4j exhibit higher memory requirements. The marginally higher memory requirements of algorithms that employ a caching strategy, (i.e., Algorithms TRIC+/INV+/INC+) is attributed to the fact that all structures calculated during the materialization phase are kept in memory for future usage; this results in slightly higher memory requirements compared to algorithms that do not apply this caching technique (i.e., Algorithms TRIC/INV/INC). To this end, employing a caching strategy for all algorithms yields significant performance gains with minimal impact on main memory. Finally, Neo4j is a full fledged database management system, thus it occupies more memory to support the required specifications.

7 OUTLOOK

In this work, we proposed a new paradigm to efficiently capture the evolving nature of graphs through query graph patterns. We proposed a novel method that indexes and continuously evaluates queries over graph streams, by leveraging on the shared restrictions present in query sets. We also evaluated our solution using three different datasets from social networks, transportation and biological interactions domains, and demonstrated that our approach is up to two orders of magnitude faster when compared to typical join-and-explore inverted index solutions, and the well-established graph database Neo4j.

Our future research plans involve (i) further improving the algorithm performance by storing materializations within the trie to minimize trie traversal at query answering time and exploiting workload-driven statistics in the spirit of [23] and (ii) increasing the model expressiveness by implementing graph deletions, supporting more general graph types (e.g., property graphs), and introducing query classes that aim at clustering coefficient, shortest path, and betweenness centrality.

ACKNOWLEDGMENTS

This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement No. DFF-4093-00301B, by the Poul Due Jensen Foundation, and by project ENIRISST under grant agreement No. MIS 5027930 (co-financed by Greece and the EU through the European Regional Development Fund).

REFERENCES

[1] Paul Ammann and Jeff Offutt. 2008. *Introduction to software testing*. Cambridge University Press.

[2] D.F. Barbieri, D. Braga, S. Ceri, E.D. Valle, and M. Grossniklaus. 2010. C-SPARQL: A Continuous Query Language for RDF Data Streams. *IJSC* (2010).

[3] Yazan Boshmaf, Ildar Muslukhov, Konstantin Beznosov, and Matei Ripeanu. 2011. The socialbot network: when bots socialize for fame and money. In *ACSAC*.

[4] Horst Bunke, Thomas Glauser, and T.-H. Tran. 1990. An Efficient Implementation of Graph Grammars Based on the RETE Matching Algorithm. In *4th International Workshop on Graph-Grammars and Their Application to Computer Science*.

[5] C. Canas, E. Pacheco, B. Kemme, J. Kienzle, and H-A. Jacobsen. 2015. GraPS: A Graph Publish/Subscribe Middleware. In *Middleware*.

[6] Meeyoung Cha, Hamed Haddadi, Fabrício Benevenuto, and P. Krishna Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM*.

[7] L. Chen and C. Wang. 2010. Continuous Subgraph Pattern Search over Certain and Uncertain Graph Streams. *IEEE TKDE* (2010).

[8] Rada Chirkova and Jun Yang. 2012. Materialized Views. *Foundations and Trends in Databases* 4, 4 (2012), 295–405.

[9] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr., Khushbu Agarwal, and John Feo. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *EDBT*.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT Press.

[11] Reinhard Diestel. 2005. Graph Theory. *GTM* (2005).

[12] 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 *ACM SIGMOD*.

[13] Jun Gao, Yuqiong Liu, Chang Zhou, and Jeffrey Xu Yu. 2017. Path-based holistic detection plan for multiple patterns in distributed graph frameworks. *VLDB Journal* (2017).

[14] Jun Gao, Chang Zhou, and Jeffrey Xu Yu. 2016. Toward continuous pattern detection over evolving large graph with snapshot isolation. *VLDB Journal* (2016).

[15] Syed Gillani, Gauthier Picard, and Frédérique Laforest. 2016. Continuous graph pattern matching over knowledge graph streams. In *ACM DEBS*.

[16] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *ACM SIGMOD*.

[17] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *ACM SIGMOD*.

[18] Huahai He and A.K. Singh. 2006. Closure-Tree: An Index Structure for Graph Queries. In *ICDE*.

[19] Zbigniew Jerzak and Holger Ziekow. 2015. The DEBS 2015 grand challenge. In *ACM DEBS*.

[20] Cliff Joslyn, Sutanay Choudhury, David Haglin, Bill Howe, Bill Nickless, and Bryan Olsen. 2013. Massive scale cyber traffic analysis: a driver for graph database research. In *GRADES SIGMOD/PODS*.

[21] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. 2007. The dynamics of viral marketing. *ACM TWEB* (2007).

[22] Jure Leskovec, Ajit Singh, and Jon M. Kleinberg. 2006. Patterns of Influence in a Recommendation Network. In *PAKDD*.

[23] Amgad Madkour, Ahmed M. Aly, and Walid G. Aref. 2018. WORQ: Workload-Driven RDF Query Processing. In *ISWC*.

[24] Marios Meimaris, George Papastefanatos, Nikos Mamoulis, and Ioannis Anagnostopoulos. 2017. Extended Characteristic Sets: Graph Indexing for SPARQL Query Optimization. In *IEEE ICDE*.

[25] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. 2001. Materialized View Selection and Maintenance Using Multi-Query Optimization. In *ACM SIGMOD*.

[26] Jayanta Mondal and Amol Deshpande. 2016. CASQD: continuous detection of activity-based subgraph pattern queries on dynamic graphs. In *ACM DEBS*.

[27] Simeon C. Ntafos and S. Louis Hakimi. 1979. On Path Cover Problems in Digraphs and Applications to Program Testing. *IEEE TSE* (1979).

[28] S. Pan and X. Zhu. 2012. CGStream: continuous correlated graph query for data streams. In *CIKM*.

[29] M. Petrovic, H. Liu, and H.-A. Jacobsen. 2005. G-ToPSS - fast filtering of graph-based metadata. In *WWW*.

[30] Xuguang Ren and Junhu Wang. 2015. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *VLDB* (2015).

[31] Xuguang Ren and Junhu Wang. 2016. Multi-query optimization for subgraph isomorphism search. *PVLDB* (2016).

[32] S. Sakr, M. Wylot, R. Mutharaju, D. Le Phuoc, and I. Fundulaki. 2018. *Centralized RDF Query Processing*. Springer, Chapter 3.

[33] D. Shasha, J.T.L. Wang, and R. Giugno. 2002. Algorithmics and Applications of Tree and Graph Searching. In *PODS*.

[34] Chunyao Song, Tingjian Ge, Cindy X. Chen, and Jie Wang. 2014. Event Pattern Matching over Graph Streams. *PVLDB* (2014).

[35] Chris Stark, Bobby-Joe Breitkreutz, Teresa Reguly, Lorrie Boucher, Ashton Breitkreutz, and Mike Tyers. 2006. BioGRID: a general repository for interaction datasets. *Oxford Academic NAR* (2006).

[36] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *PVLDB* (2012).

[37] The UniProt Consortium. 2017. UniProt: the universal protein knowledgebase. *NAR* (2017).

[38] Christos Tryfonopoulos, Manolis Koubarakis, and Yannis Drougas. 2004. Filtering algorithms for information retrieval models with named attributes and proximity operators. In *ACM SIGIR*.

[39] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter A. Boncz. 2012. Heuristics-based query optimisation for SPARQL. In *EDBT*.

[40] Alex Hai Wang. 2010. Don't Follow Me - Spam Detection in Twitter. In *SECRYPT*.

[41] C. Wang and L. Chen. 2009. Continuous Subgraph Pattern Search over Graph Streams. In *ICDE*.

[42] J. Wang, B. Jin, and J. Li. 2004. An Ontology-Based Publish/Subscribe System. In *Middleware*.

[43] Jim Webber. 2012. A programmatic introduction to Neo4j. In *SPLASH*.

[44] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. 2003. Efficient RDF Storage and Retrieval in Jena2. In *SWDB*.

[45] I. Xenarios, L. Salwnski, X.J. Duan, P. Higney, S.M. Kim, and D. Eisenberg. 2002. DIP, the Database of Interacting Proteins: a research tool for studying cellular networks of protein interactions. *NAR* (2002).

[46] Chengxi Zang, Peng Cui, Chaoming Song, Christos Faloutsos, and Wenwu Zhu. 2017. Quantifying Structural Patterns of Information Cascades. In *WWW*.

[47] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. 2007. Efficient exploitation of similar subexpressions for query processing. In *ACM SIGMOD*.

Zooming Out on an Evolving Graph

Amir Aghasadeghi
New York University
amirpouya@nyu.edu

Vera Z. Moffitt
Drexel University
zaychik@drexel.edu

Sebastian Schelter
New York University
ss12727@nyu.edu

Julia Stoyanovich*
New York University
stoyanovich@nyu.edu

ABSTRACT

An evolving graph maintains the history of changes of graph topology and attribute values over time. Such a graph has a specific temporal and structural resolution. It is often useful to modify this resolution during analysis, for example, to consider communities rather than individual nodes, or to quantify changes at the level of days rather than hours.

We propose *attribute-based zoom* and *temporal window-based zoom* — two operators that support exploratory analysis of an evolving graph at different levels of resolution. We develop several alternative physical representations of an *evolving property graph* — a temporal generalization of a property graph — and detail how to implement the proposed zoom operators using dataflow operations. These different physical representations allow us to explore the trade-offs in temporal and structural locality with respect to the performance of the zoom operators. We implement the operators in Apache Spark, evaluate them on real evolving graph datasets, and demonstrate scalability to billion-edge graphs.

1 INTRODUCTION

Many social structures and systems can be represented as networks or graphs. The phenomena that are represented by these graphs can change over time, and therefore, many interesting questions about these graphs are related to their evolution rather than to their static state. Researchers study graph evolution rate and mechanisms [1, 9], the impact of specific events on further evolution [8, 39] and spatio-temporal patterns [27, 28], with most progress taking place in the last decade [24, 35, 37, 38, 40].

Our focus in this paper is on a temporal generalization of a property graph, called TGraph, which we recently introduced [37]. Figure 1 shows an example — an interaction network in which nodes represent people, and, for the students among them, include information about a school at which they are enrolled, while edges represent co-authorship. As in conventional property graphs [3], nodes and edges of a TGraph are associated with a set of key-value pairs that represent an assignment of values to attributes. In addition, TGraph associates a time interval (representing a set of discrete consecutive time points) with each state of a node or edge. For example, a `person` node *Ann* exists, and is enrolled at MIT, during the interval $T = [1, 7)$.

TGraph maintains the history of changes of graph topology and attribute values over time. It has a specific *temporal* and *structural resolution*, which users often want to modify for exploratory analysis, for example, to look at communities rather than individual nodes, or to quantify changes at the level of days rather than hours. In this paper we focus on two operators, aZoom^T and wZoom^T, that allow us to change the structural and temporal resolution of a TGraph, respectively. These operators are part of a compositional evolving graph algebra called TGA, which we presented in [37], that operates under *point semantics* [5].[1] A consequence of these semantics is that the TGraph must remain *temporally coalesced* — vertices and edges in the output of an operator must be associated with time periods of maximal length during which no change occurred.

Attribute-based zoom (aZoom^T). We may be interested in analyzing evolving graphs at different levels of *structural resolution*, to study properties and behavior of individual nodes, of communities, and of the graph as a whole. An operation that achieves this, known as *node creation*, is present in several conventional (non-temporal) graph query languages [14, 21, 32, 42]. Our focus is on a temporal generalization of this operation for graphs, called *temporal attribute-based zoom*, or aZoom^T for short.

Consider TGraph $\mathcal{G}_1$ in Figure 1, where school names are represented as values of the school property of person nodes. aZoom^T computes the TGraph in Figure 2, where schools become nodes (actors) rather than values.

aZoom^T is evaluated over a TGraph under *point semantics* and, specifically, under the principle of snapshot reducibility [5]: we evaluate the non-temporal variant of the operator over each state of the graph (also known as a "snapshot"), and then apply temporal coalescing [4] to represent each vertex or edge in the result with a single fact, corresponding to the longest interval during which no change occurred. aZoom^T is described in Section 2.2.

Temporal window-based zoom (wZoom^T). This operator changes the *temporal resolution* of a TGraph. This operation is important because it may not be known *a priori*, at the time when graph evolution is being recorded, at what time scale interesting trends can be observed. For example, changes in node centrality in a social network may be observable on the scale of weeks but not months. Understanding at what temporal resolution to consider network evolution is an integral part of exploratory analysis. Let us return to our running example in Figure 1, and assume that time points represent months of 2019. We may zoom out on $\mathcal{G}_1$ temporally, to 3-month windows, retaining nodes and edges in the result for a particular time window that were present in the input during all time points of the window. The result is presented in Figure 3, and described in more detail in Section 2.3.

Next, we explore different physical representations to answer the following questions: (i) How should we represent a TGraph to compute the result of aZoom^T and wZoom^T efficiently? Should we use a snapshot-based representation, storing graph evolution as a sequence of conventional graphs, that is easy to parallelize but lacks compactness, or should we leverage a more compact representation, as suggested by Figure 1? (ii) What representation should we use to efficiently execute *a sequence* of these operators? We address these questions, making the following contributions:

- We propose different physical representations of a TGraph and detail how to define aZoom^T and wZoom^T using dataflow operations for these representations (Section 3).

*This work was supported in part by NSF Grant No. 1916505.

[1]The focus of [37] is on defining the TGraph model and algebra, while this paper focuses on system and implementation aspects.

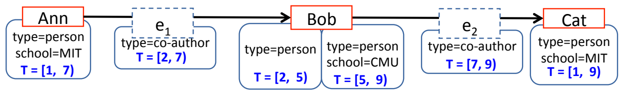


Figure 1: Evolving property graph (TGraph) $\mathcal{G}_1$.

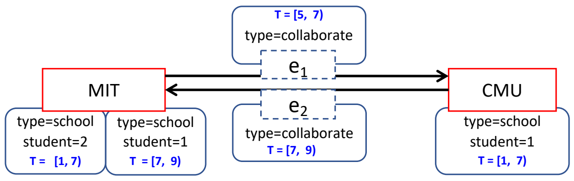


Figure 2: Result of aZoom^T over $\mathcal{G}_1$ (Figure 1). Semantically, this operation is executed over every snapshot of $\mathcal{G}_1$ to: (i) create `school` nodes for each value of the school property of `person` nodes in $\mathcal{G}_1$; (ii) count the number of persons enrolled at a school, set the value of the student property of the `school` node to that count; (iii) create edges of type `collaborate` between school nodes for which `co-author` edges were present in $\mathcal{G}_1$; and (iv) temporally coalesce the result across snapshots, due to point semantics.

- We describe how to efficiently implement aZoom^T and wZoom^T in Apache Spark (Section 4).
- We conduct an extensive experimental evaluation of aZoom^T and wZoom^T and demonstrate scalability to billion-edge graphs. We find that a physical representation that balances temporal and structural locality outperforms other representations in most cases (Section 5).

2 TGRAPH MODEL AND ZOOM OPERATORS

We provide the background on the evolving property graph model called TGraph, and define the operators aZoom^T and wZoom^T that take a valid TGraph as input, and output a TGraph.

2.1 Evolving property graphs

In [37] we proposed a logical model of an evolving graph called TGraph that represents a single graph (such as the Web, or a large collaboration network), and models the evolution of its topology, and vertex and edge properties. A TGraph is a directed multigraph: its nodes and edges have identity, and multiple edges may connect a given pair of nodes. Each entity (node and edge) has a required type label, and is associated with a (possibly empty) set of key-value pairs that represent its properties, each in the form of a property label (key) and a corresponding value. The set of properties for an entity is not fixed: it can be different among entities of the same type, and for the same entity over time.

We now recall the definition of TGraph from [37], simplifying it slightly. This definition extends the static property graph definition of Angles et al. [3] by associating periods of validity with graph nodes, edges, and their properties. Time is drawn from a linearly ordered discrete domain Ω^T.

Definition 2.1. A TGraph $\mathcal{G} = (V, E, L, \rho, \xi^T, \lambda^T)$ is a six-tuple:

- V is a finite set of *nodes* (or *vertices*), E is a finite set of *edges*, $V \cap E = \emptyset$, and L is a finite set of property labels;
- $\rho : E \longrightarrow (V \times V)$ is a total function that maps an edge to its source and destination nodes;
- $\xi^T : (V \cup E) \times \Omega^T \longrightarrow B$ is a total function that maps a node or an edge and time point to a Boolean, indicating existence of the node or edge at that time point; and
- $\lambda^T : (V \cup E) \times L \times \Omega^T \longrightarrow val$ is a partial function that maps a node or an edge, a property label, and a time point to a value of the property at that time point.

A valid TGraph conceptually corresponds to a sequence of valid conventional (non-temporal) graphs. This imposes the following conditions: (i) a condition on ξ^T that an edge can only exist at a time when both of the nodes it connects exist; and (ii) a condition on λ^T that a property can only take on a value at a time when the corresponding node or edge exists. Finally, we require that the property set of an entity **not** be empty at any time point when it exists. Practically, we require that each node and edge assign a value to a property called `type`.

Definition 2.1 associates graph nodes, edges and attribute values with *time points*. In the remainder of this paper, we will represent temporally adjacent time points by *intervals*, for syntactic compactness, as illustrated in Figure 1. Following the SQL:2011 standard, we use closed-open intervals, representing a discrete contiguous set of time points from Ω^T. This representation does not add expressiveness to a point-based representation, and is purely a syntactic device [10].

We now describe aZoom^T and wZoom^T in detail using our running example, and refer to [37] for a formal treatment.

2.2 Attribute-Based Zoom

Temporal attribute-based zoom, denoted aZoom^T, is a temporal generalization of the graph node creation operation [42]. Node creation over non-temporal graphs takes a graph pattern as input, and computes a new node for each occurrence of a match of the pattern in the input. To assign identity to new nodes, it is customary to extend this operation with a Skolem function f_s. aZoom^T will similarly create nodes in the output TGraph from disjoint groups of nodes in the input, such that nodes within a group agree on the values of all grouping attributes.

Conceptually, aZoom^T is executed over every snapshot of the input TGraph, and new nodes are assigned identity by a Skolem function f_s, which generates consistent assignments across time. In addition to creating new nodes, aZoom^T will also optionally compute values of node attributes using the aggregation function f_{agg}, including count, sum, min, max, average, and user-specified functions that are required to be commutative and associative. Next, aZoom^T computes edges as follows. Suppose that input nodes n and n' corresponds to output nodes g and g', respectively, and that edge e connects n to n'. Then, the output will contain the edge e, with g as its source and g' as its target. Essentially, the input edge is re-created in the output, and re-pointed.

Node creation, computation of node attribute values, and re-pointing of the edges, is executed over each snapshot of the input TGraph, under point semantics. As the final step, the result is then coalesced, associating a time interval of maximal length during which no change occurred with every newly-computed node and edge. We now illustrate aZoom^T with an example.

Example 2.2. Node *Ann* in Figure 1 is associated with a closed-open interval $T = [1, 7)$, signifying that the node existed in the graph for six consecutive time points with no change. *Bob* exists in the graph during $T = [2, 9)$, but with a change to its attributes at time 5, when `school=CMU` was added. School names are represented as values of the school property of person nodes.

We invoke aZoom^T to compute from $\mathcal{G}_1$ a TGraph where schools become nodes rather than values, as shown in Figure 2

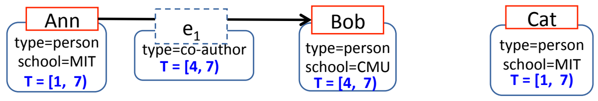


Figure 3: wZoom^T($\mathcal{G}_1$, window=3-months, nodes=all, edges=all, node.school=last(school)) over $\mathcal{G}_1$ (Figure 1).

with school nodes *MIT* and *CMU*. Note that the number of students at MIT changes over time: both *Ann* and *Cat* study there during $T = [1, 7)$, while only *Cat* studies there during $T = [7, 9)$. Note that both edge e_1 and e_2 have been redirected to newly created nodes and their validity period is updated to correct values based on when those were valid in the graph: While e_1 is valid during $T = [2, 7)$ in Figure 1, it is only valid during $T = [5, 7)$ in Figure 2, because *Bob* was not at CMU during $T = [2, 5)$.

2.3 Temporal Window-Based Zoom

The wZoom^T operator is analogous to *moving window temporal aggregation* in temporal relational algebra. This operator is inspired by stream aggregation of Li et al. [29] (adopted to graphs), and by generalized quantifiers [22].

The wZoom^T operator modifies validity periods of TGraph nodes and edges, by mapping different states of a node or an edge to a single representative state. This mapping is based on a specification of a temporal window, such as 2 months or 10 years. If the specified window is finer than the temporal resolution of the input TGraph, the operation has no effect. For example, applying wZoom^T with 1-month windows to a TGraph in which evolution is recorded across years will simply return the input TGraph. Note that, because wZoom^T is required to produce a valid TGraph as output, this operation does not support overlapping windows.

Window specification is of the form n $\{unit|\text{changes}\}$, where n is an integer, and *unit* is a time unit (e.g., 10 min, 3 years). Window specification generates a temporal relation W with the schema $(d \mid T)$, where each tuple associates a window number d with its period of validity T. We additionally require node and edge existence quantifiers {all|most|at least n|exists}, where n is a decimal representing the percentage of the time during which a node or an edge existed, relative to the duration of the window. Quantifiers are useful for observing different kinds of temporal evolution. For example, to observe strong connections over a volatile evolving graph we may include nodes that span the entire window (nodes=all), and edges that span a large portion of the window (edges=most). We refer to all and exists as universal and existential quantification, respectively.

A related point is that a given node or edge should exist at most once at any given time point, and so we must specify how conflicts in attribute values are resolved by wZoom^T. The answer to this question is determined by the window aggregation functions, which specify, for each attribute of a node or an edge, which of its values to accept as a representative for the given temporal window. We support the window aggregation functions first, last, and any (the default).

Example 2.3. Consider again TGraph $\mathcal{G}_1$ in Figure 1, and suppose that time points represent the months of 2018, and are divided into fiscal year quarters as follows: window $W1$: time points 1, 2, 3; $T = [1, 4)$, window $W2$: time points 4, 5, 6; $T = [4, 7)$, window $W3$: time points 7, 8, 9; $T = [7, 10)$. How might we quantify the state of $\mathcal{G}_1$ during each quarter, a 3-month temporal window?

Figure 3 shows the temporally coalesced results of zooming out to quarters over $\mathcal{G}_1$ with nodes=all and edges=all.

Ann is present in windows $W1$ and $W2$ in the input in Figure 1, and so is associated with $T = [1, 7)$ in the result for both universal and existential quantification. In contrast, *Bob* is present in the input for all of $W2$ but for only part of $W1$, and so is returned with $T = [4, 7)$ in the result for nodes=all, and with $T = [1, 7)$ for nodes=exists. Finally, *Cat* is present for all of $W1$ and $W2$, but for only part of $W3$ in the input (it is missing at time point 9), and so is associated with $T = [1, 7)$ in the output undernodes=all and with $T = [1, 10)$ under nodes=exists. Quantification is applied to edges analogously: e_1 is mapped to window $W2$ and e_2 is absent in the output in Figure 3, because there does not exist a quarter during which e_2 exists continuously in the input.

3 EVOLVING GRAPH REPRESENTATIONS AND DATAFLOW OPERATORS

In this section, we introduce several physical representations for a TGraph and detail how to define the zoom operations according to these representations. We express the zoom operators using general dataflow operations — directed acyclic graphs of operators resembling parallelizable second-order functions that execute user-defined first order functions. This is a popular programming model for distributed computations supported by systems such as Apache Spark [43] and Apache Flink [2].

We use the term *snapshot* to refer to a conventional (non-temporal) graph that represents the state of a TGraph during some interval in which no change occurred. Figure 4 shows the TGraph in our running example as a sequence of snapshots. When storing and accessing evolving graphs, we are concerned with preserving two kinds of locality: temporal and structural. Adopting the terminology of [19], with *structural locality*, neighboring vertices (resp. edges) of the same snapshot are laid out together, while with *temporal locality*, consecutive states of the same vertex (resp. edge) are laid out together. We develop four TGraph representations that differ in compactness and in the kind of locality (structural or temporal) they prioritize.

Representative Graphs (RG). RG represents a TGraph by a sequence of snapshots (conventional graphs), associating them with time intervals, see Figure 4 for an example. The snapshot sequence is by far the most common representation in the literature [15, 20, 24–26, 38, 40]. RG has the following schema:

```
TemporalGraph { interval: Interval,
                              snapshots: array(Snapshot) }
Snapshot { vertices: array(Vertex), edges: array(Edge) }
Interval { start: Date, end: Date }
Vertex { vid: long, type: string, attributes: dictionary }
Edge { eid: long, type: string, v1: Vertex, v2: Vertex,
       attributes: dictionary }
```

Note that vertices and edges of each snapshot store the attribute values for the interval represented by the snapshot. This representation is simple, and lends itself well to parallelizing operations in a distributed environment, as we can simply assign different snapshots to different workers. An advantage of RG is that it naturally preserves structural locality, and so is efficient for snapshot-based operations. An important drawback of RG is that it is not compact: in many real-world evolving graphs there is an 80% or larger overlap between consecutive snapshots [8].

Vertex Edge (VE). As illustrated in Figure 5, VE is a nested temporal relational representation of TGraph, with one relation storing vertices and the other edges, together with the corresponding

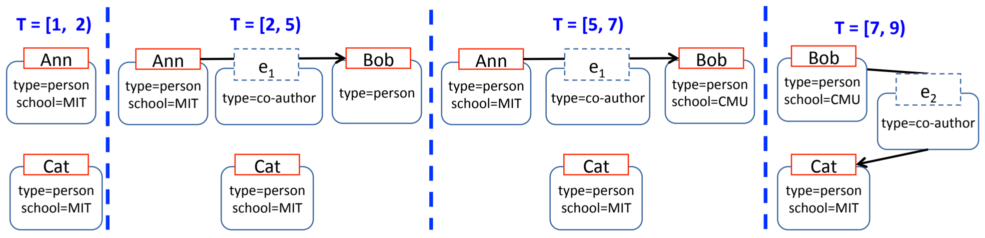


Figure 4: Representative-Graphs (RG): a "sequence of snapshots" representation of the TGraph $\mathcal{G}_1$ of Figure 1.

Vertices (V)

<u>v</u>	a	T
Ann	type=person, school=MIT	[1, 7)
Bob	type=person	[2, 5)
Bob	type=person, school=CMU	[5, 9)
Cat	type=person, school=MIT	[1, 9)

Edges (E)

<u>e</u>	v1	v2	a	T
e1	Ann	Bob	type=co-author	[2, 7)
e2	Bob	Cat	type=co-author	[7, 9)

Figure 5: Vertex-Edge (VE): nested relational representation of the TGraph $\mathcal{G}_1$ from Figure 1. The relations Vertices (V) and Edges (E) are temporally coalesced.

time intervals. Both relations are temporally coalesced, giving rise to a compact representation. VE stores all vertex properties together as a single nested attribute (and all edge properties analogously). VE has the following schema:

```
TemporalGraph { interval: Interval,
    vertices: array(Vertex),  edges: array(Edge) }
Interval { start: Date, end: Date }
Vertex { vid: long, type: string, interval: Interval,
        attributes: dictionary }
Edge { eid: long, type: string, vid1:long, vid2: long,
      interval: Interval, attributes: dictionary }
```

For edges, we store a unique edge identifier **eid**(long) to support multi-graphs, as well as the vertex identifiers **vid1**(long) and **vid2**(long) that are foreign keys referring to the vertex relation. The main advantage of VE's attribute representation is that it lends itself to schema evolution. A disadvantage is that different properties may have different evolution rates, and a change to a single property requires a new vertex or edge tuple. VE stores graph vertices and edges in unordered collections, and therefore does not maintain temporal locality by default in cases where the state of a vertex or edge changes. For example, two tuples for vertex *Bob* in Figure 5 may not be located consecutively, or even on the same worker, once the data is partitioned across a cluster. We can reconstruct temporal locality at runtime, by re-partitioning the data based on vertex or edge identifiers.

One Graph (OG), One Graph Columnar (OGC). These are two topologically compact representations. OG stores all vertices and edges once, in a single aggregated data structure, as shown in Figure 6. In OG, vertices and edges store the history of the evolution of their attributes as an array of key-value pairs, together with the corresponding validity periods. Figure 6 shows the OG representation for our example graph. Note that we have only one tuple for vertex *Bob*, which holds two sets of values for two corresponding validity periods T=[2,5) and T=[5,9). OG has the following schema:

```
TemporalGraph { interval: Interval,
    vertices: array(Vertex), edges: array(Edge) }
```

Vertices (V)

<u>v</u>	a
Ann	{ T=[1,7): type=person, school=MIT }
Bob	{ T=[2,5):type=person, T=[5,9):type=person, school=CMU }
Cat	{ T=[1,9): type=person, school=MIT }

Edges (E)

<u>e</u>	v1	v2	a
e1	Ann	Bob	{ T=[2,7): type=co-author }
e2	Bob	Cat	{ T=[7,9): type=co-author }

Figure 6: One Graph (OG): nested relational representation of the TGraph $\mathcal{G}_1$ from Figure 1. The relations Vertices (V) and Edges (E) are temporally coalesced.

```
Interval { start: Date, end: Date }
HistoryItem { interval: Interval, attributes: dictionary }
Vertex { vid: long, type: string,
                                history: array(HistoryItem) }
Edge { eid: long, type: string, v1: Vertex, v2: Vertex,
      history: array(HistoryItem) }
```

The schemas for OG and VE are similar in many ways. The main difference is that the interval and attribute dictionary in VE has been replaced with a **history** array that contains **HistoryItem**s. Each such history item stores an **interval** as the key and a dictionary of the corresponding **attributes**. The second difference is that OG contains a copy of the source and target vertex of each edge, instead of a foreign key to the vertex relation.

OGC, on the other hand, only stores the graph topology with validity periods as a graph, as shown in Figure 7. OGC has the following schema:

```
TemporalGraph { intervals: array(Interval),
             vertices: array(Vertex), edges: array(Edge) }
Interval { start: Date, end: Date }
Vertex { vid: long, type: string, intervals: Bitset }
Edge { eid: long, type: string, v1: Vertex,
      v2: Vertex, intervals: Bitset }
```

OGC is intended for topology-only attribute-less graphs, encoding the presence of a vertex or edge in each interval with a bitset. Both OG and OGC emphasize temporal locality, while also preserving structural locality, but lead to a much denser graphs than RG. This, in turn, makes parallelizing computation challenging.

In the remainder of this section, we describe how to define aZoom^T and wZoom^T in terms of dataflow operations according to our proposed representations.

3.1 Attribute-Based Zoom

We now describe aZoom^T for each TGraph representation. In the algorithms we present, V and E are overloaded to refer to the vertex and edge relations of a given snapshot (in the case of RG) or of the overall TGraph. In aZoom^T we use a Skolem function f_s to produce new vertex ids. f_s is a user-provided function that

Bitset (b): T={[1,2),[2,7),[7,9)}

Vertices (V)

v	b
Ann	[1, 1, 0]
Bob	[0, 1, 1]
Cat	[1, 1, 1]

Edges (E)

e	v1	v2	b
e1	Ann	Bob	[0, 1, 0]
e2	Bob	Cat	[0, 0, 1]

Figure 7: One Graph Column (OGC): nested relational representation of the TGraph $\mathcal{G}_1$ of Figure 1. Vertices (V) and Edges (E) are temporally coalesced. Bitsets represent validity during periods of T={[1,2),[2,7),[7,9)}.

takes the vertex id and all attributes as an input and produces a long identifier as output. We additionally apply the commutative and associative aggregation function f_{agg} to resolve cases where we have a series of vertices with identical identifiers but multiple values for the same attribute in the same snapshot. This is an important step that ensures that each snapshot in the result corresponds to a valid graph (see [36] for details).

RG. Recall thatRG maintains a collection of snapshots. We apply the same set of operations in an embarrassingly parallel manner to each snapshot, as there are no dependencies between them in this case (Algorithm 1). We iterate over each snapshot (lines 3-10) and return an RG (line 11) containing the aZoom^T result. We apply f_s to each vertex using a map (line 5) in order to compute a new identifier for each vertex. The copyWithVid function updates each vertex identifier while keeping other attributes unchanged. We then group vertices by id (line 7) and apply the aggregation function f_{agg} (line 8).

To redirect edges to the newly created vertices, we apply the function f_s to the vertices v1 and v2 of each edge in a map (line 9). The copyWithVids function updates the id of the vertices to the new identifiers. The edges contain a copy of their source and target vertices in RG, which obliviates the need for a join here.

Algorithm 1 aZoom^T over RG

```
Require: Skolem function $f_s : V \Rightarrow \mathbb{N}$; Aggregation function $f_{agg} : V \times V \Rightarrow V$
 1: $newSnapshots \leftarrow \emptyset$
 2:                                          ▷ Aggregate each snapshot
 3: for ($V$,E) in graph.snapshots do
 4:     $V' \leftarrow V$                          ▷ Update of vertex identifiers
 5:         .map{$v \Rightarrow v$.copyWithVid($f_s(v)$)}
 6:                           ▷ Vertex aggregation for identity-equivalence
 7:         .groupBy{$v \Rightarrow v.vid$}
 8:         .reduce{$(v_a, v_b) \Rightarrow f_{agg}(v_a, v_b)$}
                                      ▷ Edge redirection to new vertices
 9:     $E' \leftarrow E$.map{$e \Rightarrow e$.copyWithVids($f_s(e.v1), f_s(e.v2)$)}
10:     Add ($V'$, $E'$) to $newSnaphots$
11: return new TGraph $G(newSnapshots)$
```

VE. VE consists of two temporal relational tables for vertices and edges, which contain tuples for each vertex or edge history. Algorithm 2 details our implementation of aZoom^T for VE. We first calculate non-overlapping intervals (lines 2-5) based on the temporal splitter concept introduced in [11]. We join intervals and vertices (lines 7- 9), assign new identifiers (line 10), and enforce identity-equivalence in each interval with the aggregation function (line 12). Since VE edges only contain a foreign key to the corresponding vertices, we need to join the edges with their corresponding vertices for the edge redirection process (lines 14 and 15), before we can apply the f_s function to each corresponding vertex to redirect the edge (line 18).

Algorithm 2 aZoom^T over VE

```
Require: Skolem function $f_s : V \Rightarrow \mathbb{N}$; Aggregation function $f_{agg} : V \times V \Rightarrow V$
 1: $I \leftarrow V$       ▷ Non-overlapping intervals for each new vertex identifier
 2:     .map{$v \Rightarrow (f_s(v), v.interval)$}
 3:     .groupBy{$(vid, \_) \Rightarrow vid$}
 4:     .foldLeft(EmptyInterval)
 5:         {$(i, v) \Rightarrow$ mergeNonOverlapping($i$, $v.interval$)}
 6: $V' \leftarrow V$             ▷ Vertex aggregation for non-overlapping intervals
 7:     .join($I$).on{$(v, id) \Rightarrow v.id == i.vid$}
 8:     .flatMap{$(v, i) \Rightarrow$ verticesForIntervals($v$, $i$)}
 9:     .map{$(v, i) \Rightarrow$
10:         $v$.copyWithIdAndInterval($f_s(v)$, $i$) }
11:     .groupBy{$v \Rightarrow v.id$}
12:     .reduce{$(v_a, v_b) \Rightarrow f_{agg}(v_a, v_b)$}
13: $E' \leftarrow E$                        ▷ Edge redirection to new vertices
14:     .join($V$).on{$(e, v) \Rightarrow e.vid1 == v.id$}
15:     .join($V$).on{$((e, \_), v) \Rightarrow e.vid2 == v.id$}
16:     .map{$(e, v1, v2) \Rightarrow$
17:         $i \leftarrow$ recomputeInterval($e$, $v1$, $v2$)
18:         $e$.copyWithVidsAndInterval($f_s(v1), f_s(v2)$, $i$) }
    return new TGraph $G(V', E')$
```

OG. We implement aZoom^T for One Graph (OG) analogously to RG, with the difference that we compute over the entire TGraph rather than over each individual snapshot (Algorithm 3). We split each vertex in OG based on its history, and apply the f_s function to each element of the history array individually. We use a flatMap function on vertices combined with a map on the history elements of each vertex for this (lines 1-3). We again enforce identity-equivalence with our aggregation function (lines 4 and 5). The vertext computation portion of Algorithm 3 is illustrated in Figure fig:az-og. For edge redirection in OG, we split the edges by expanding the history of each corresponding vertex in that edge, as OG stores each edge only once. Next, we apply the Skolem function f_s to each element of the history (line 6-9).

Algorithm 3 aZoom^T over OG

```
Require: Skolem function $f_s : V \Rightarrow \mathbb{N}$; Aggregation function $f_{agg} : V \times V \Rightarrow V$
 1: $V' \leftarrow V$ .flatMap{$v \Rightarrow$
 2:     $v.history$.map{$(\_, attr) \Rightarrow$
 3:         $v$.copyWithIdAndAttributes($f_s(v.vid)$, $attr$) }}
 4: .groupBy{$v \Rightarrow v.vid$}
 5: .reduce{$(v_a, v_b) => f_{agg}(v_a, v_b)$}
 6: $E' \leftarrow E$ .map{$e \Rightarrow$
 7:     $h \leftarrow$ recompute_history($e$)
 8:     $e$.copyWithVidsAndHistory($f_s(e.v1.vid)$,
 9:         $f_s(e.v2.vid), h$)}
    return new TGraph $G(V', E')$
```

OGC does not represent attributes and so does not support aZoom^T.

3.2 Temporal Window-Based Zoom

As we did for aZoom^T, we express wZoom^T differently for each representation, with some common aspects. The first step is to

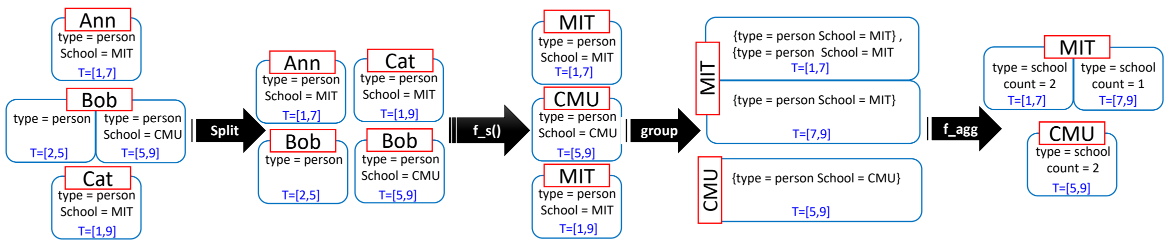


Figure 8: Illustration of the vertex computation portion of Algorithm 3, aZoom^T, over TGraph in Figure 6, with count as f_{agg}. The first two steps correspond to the call to flatMap on lines 1-3: splitting nodes based on their history array, and then calling the Skolem function f_s to generate ids for new nodes. In this example, f_s outputs the value of the school property. The next step groups vertices by id (line 4). The final step (line 5) applies the aggregation function count, storing the computed value as a vertex property.

compute the temporal window relation based on the window specification. We split the total graph lifetime temporally by applying the function computeNewIntervals to the graph. This function takes an interval as an input and returns a tuple containing the old and the recomputed interval.

A major difference to aZoom^T is that the TGraph must be coalesced *before* wZoom^T can be applied, in order to guarantee the correctness of the zoom operation. This is because aZoom^T executes over each snapshot (under snapshot reducibility), while the computation of wZoom^T is across snapshots. Consequently, if the input to wZoom^T is not coalesced, we cannot properly apply existence quantifiers and compute results of aggregation.

We additionally need to handle potential dangling edges for all representations in wZoom^T to ensure that every snapshot of the resulting TGraph is a valid graph, as specified in the condition over ξ^T in Definition 2.1. Recall that wZoom^T supports the quantifiers all, most, at least n, and exists, which can be translated to a threshold on the percentage of the time during which an entity (a vertex or an edge) existed, relative to the duration of the window: $t = 1$ for all, $t > 0.5$ for most, $t > 0$ for exists and $t > n$ for at least n. If an entity's existence meets the threshold, it will be retained in the result of the operation. A dangling edge check is only required if r_v is more restrictive than r_e, because a particular edges may pass the check, but one or more of the vertices it connects may not.

Algorithm 4 wZoom^T over RG

```
Require: resolve functions $f_v, f_e$; quantifiers $r_v, r_e$
 1:                                          ▷ Computation of new intervals
 2: $I' \leftarrow I.\mathbf{map}\{i \Rightarrow (i, \text{computeNewInterval}(i))\}$
 3:                                   ▷ Grouping of snapshots by new interval
 4: $S \leftarrow G.snapshots.\mathbf{join}(I')$
 5:     $.\mathbf{on}\{(s, interval) \Rightarrow s.i == interval.i\}$
 6:     $.\mathbf{groupBy}\{(s, interval) \Rightarrow interval.newInterval\}$
 7:                               ▷ Aggregation of vertices for new snapshots
 8: $V' \leftarrow S.\mathbf{flatMap}\{(i, snapshot) \Rightarrow$
 9:     $(i, snapshots.\text{map}\{s \Rightarrow s.vertices\})\}$
10:     $.\mathbf{groupBy}\{(i, v) \Rightarrow (i, v.id)\}$
11:     $.\mathbf{filter}\{(i, vertices) \Rightarrow \text{match\_threshold}(vertices, r_v)\}$
12:     $.\mathbf{reduceByKey}\{((v_a), (v_b)) \Rightarrow f_v(v_a, v_b)\}$
13:                                     ▷ Aggregate edges for new snapshots
14: $E' \leftarrow S.\mathbf{flatMap}\{(i, snapshot) \Rightarrow$
15:     $(i, snapshots.map\{s \Rightarrow s.edges\})\}$
16:     $.\mathbf{groupBy}\{(i, e) \Rightarrow (i, e.id)\}$
17:     $.\mathbf{filter}\{(i, edges) \Rightarrow \text{match\_threshold}(edges, r_e)\}$
18:     $.\mathbf{reduceByKey}\{((e_a), (e_b)) \Rightarrow f_e(e_a, e_b)\}$
                                              ▷ Recreate RG representation
19: $G' \leftarrow \text{merge}(I', V', E')$
```

RG implements wZoom^T as shown in Algorithm 4. We again use the computeNewInteval function to compute the new intervals based on the window specification (line 2). Next, we apply join, groupBy, and flatMap to map each vertex to one or more snapshots from the specification (lines 4-9). Then, vertices are grouped by their id within each new interval (line 10). Next, we filter vertices and edges based on the existence quantifier (line 11). We apply the math_threshold function to vertices with their respective thresholds (r) to filter vertices that do not meet the criteria of our quantifier. Finally, we apply the resolve function f_v to compute the new attribute values (line 12). We treat edges analogously (lines 14-18). At the end, we merge snapshots into a TGraph and remove dangling edges.

VE implements wZoom^T using Algorithm 5. Figure 9 illustrates this algorithm for vertex *Bob* from Figure 5. We first need to calculate the new intervals using computeNewInterval (lines 2-3). Then we join V with the intervals to align each vertex with each temporal window (lines 4-6) to split the vertices. Next, we group by interval and vertex (line 7), and filter vertices that do not pass the quantifier threshold (line 8). Finally, we resolve the vertices' final attributes (line 12). We apply the same operations to edges (lines 11-18). We remove dangling edges (given that $r_v > r_e$) with two semijoins (lines 17-19).

OG implements wZoom^T using Algorithm 6. Recall that in OG each vertex stores its interval information in a history array. We process each element of this array separately and rebuild the array afterwards (lines 1-4) for this process. We first invoke recomputeIntervals (line 2) to recompute the history array with updated intervals. Next, we leverage the aggregateAndFilterAttributes function (line 3) to group, filter and resolve vertices analogous to previous algorithms, and apply the same transformations to the edges as well (lines 5-8).

We again remove dangling edges with semijoins (lines 9-15). The only difference here is that joining edges with vertices is not enough, as we also need to update the history arrays. We achieve this with a map function which updates every edge history with the intersection of the edge history and the corresponding vertex history (lines 12 and 15) using the copyWithHistory function.

OGC implements wZoom^T similarly to OGC, but working with a bitset instead of a history array. Removing dangling edges in OGC is as simple as computing the logical and between the edge bitset and the corresponding vertex bitsets.

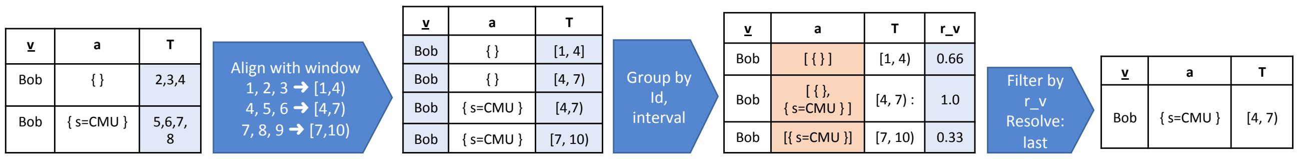


Figure 9: Illustration of Algorithm 5, wZoom^T, for vertex *Bob* in Figure 5, with window size 3 and last as f_v. The first step aligns each vertex with each temporal window (lines 4-6 of the algorithm). Next we create a single nested representation of each vertex per window and compute r_v, the fraction of the window during which the vertex was observed (line 7). Finally, we filter vertices by r_v and resolve their attribute values with f_v =last (lines 8, 9).

Algorithm 5 wZoom^T over VE

```
Require: resolve functions $f_v, f_e$; quantifiers $r_v, r_e$
1:                                          ▷ Computation of new intervals
2: $I' \leftarrow I$.map{ $i \Rightarrow (i,$ computeNewInterval($i$)) }
3:                                  ▷ Vertex aggregation for new intervals
4: $V' \leftarrow V$.join($I'$).on{ $(v, (i, n)) \Rightarrow v.n == i$ }
5:     .map { $(v, (i, newInterval)) \Rightarrow$
6:         $v$.copyWithNewInterval($newInterval$)}
7:     .groupBy{ $v \Rightarrow (v.id, v.interval)$ }
8:     .filter{$(i, vertices) \Rightarrow$ match_threshold($vertices, r_v$)}
9:     .reduceByKey{$((v_a), (v_b)) \Rightarrow f_v(v_a, v_b)$}
10:                                   ▷ Edge aggregation for new intervals
11: $E' \leftarrow E$.join($I'$).on{ $(e, (i, n)) \Rightarrow e.interval == n$ }
12:     .map { $(e, (i, newInterval)) \Rightarrow$
13:         $e$.copyWithNewInterval($newInterval$)}
14:     .groupBy{ $e \Rightarrow (e.id, e.interval)$ }
15:     .filter{$(i, edges) \Rightarrow$ match_threshold($edges, r_e$)}
16:     .reduceByKey{$((e_a), (e_b)) \Rightarrow f_e(e_a, e_b)$}
17:     if $r_v > r_e$ then                        ▷ Dangling edge removal
18:         $E'' \leftarrow E'$.semijoin($V'$)
             .on{ $(e, v) \Rightarrow e.vid1 == v.id$ and in_interval(e, v)}
19:         $E''' \leftarrow E''$.semijoin($V'$)
             .on{ $(e, v) \Rightarrow e.vid2 == v.id$ and in_interval(e, v)}
20: return new TGraph ($V', E'''$)
```

Algorithm 6 wZoom^T over OG

```
Require: resolve functions $f_v, f_e$; quantifiers $r_v, r_e$
1: $V' \leftarrow V$.map{$v \Rightarrow$
2:     $h \leftarrow$ recomputeIntervals($v.history$)
3:     $h \leftarrow$ aggregateAndFilterAttributes($h, f_v, r_v$)
4:     $v$.copyWithHistory($h$) }
5: $E' \leftarrow E$.map{$e \Rightarrow$
6:     $h \leftarrow$ recomputeIntervals($e.history$)
7:     $h \leftarrow$ aggregateAndFilterAttributes($h, f_e, r_e$)
8:     $e$.copyWithHistory($h$) }
9: if $r_v > r_e$ then                            ▷ Dangling edge removal
10:     $E'' \leftarrow E'$.semijoin($V'$)
         .on{ $(e, v) \Rightarrow e.vid1 == v.id$ and in_interval(e, v)}
11:     .map{$(e, v) \Rightarrow$
12:         $e$.copyWithHistory(intersect($e.history, v.history$)) }
13:     $E''' \leftarrow E''$.semijoin($V'$)
         .on{ $(e, v) \Rightarrow e.vid2 == v.id$ and in_interval(e, v)}
14:     .map{$(e, v) \Rightarrow$
15:         $e$.copyWithHistory(intersect($e.history, v.history$)) }
16: return new TGraph $G(V', E'')$
```

4 IMPLEMENTATION

We defined our zoom operators in Section 3 using general dataflow operations and UDFs that are implemented by a variety of popular systems. Apache Spark with GraphX [17] and Apache Flink with Gelly [7] are natural candidates for such workloads, as is Differential Dataflow [33]. We choose Apache Spark for our implementation due to its maturity and popularity.

Our implementation includes a TGraph API, several graph representations as discussed in Section 3, and several optimizations such as lazy coalescing. Our API supports chaining multiple operations together and switching between graph representations during query execution.

The VE representation is implemented directly over Spark's Resilient Distributed Datasets (RDDs) [43] while RG, OG and OGC leverage the GraphX library for static graphs [17]. We use the long datatype to represent node and edge identifiers to maintain interoperability with GraphX.

GraphX-specific implementation details. GraphX implements vertex-cut-based partitioning that reduces communication overhead [17] for certain aggregations on graphs. GraphX also provides an optimized implementation of a distributed triplet view, a concept originating from Resource Description Frameworks (RDF) [31]. The triplet view provides fast access to each edge and its corresponding source and destination vertex properties. The triplet view requires a materialized three-way join, which GraphX optimizes by implementing vertex-mirroring and a multicast join [17]. We leverage the implementation of the triplet view to efficiently access edges' vertex attributes in RG, OG and OGC. We implement RG as sequence of GraphX graphs, while OG and OGC are modeled as a single GraphX graph. GraphX mechanisms such as vertex-cut partitioning and the triplet view enabled us to implement graph operations more efficiently.

Data loading. The data is read from the Hadoop Distributed File System (HDFS). Our on-disk data layout uses Apache Parquet, a columnar data format for HDFS based on the Dremel project [34]. Apache Parquet does not have a mechanism for indexing, but it supports filter pushdown on any column by which the data is sorted on disk. We store and load vertices and edges as separate vertex and edge Parquet files. The default schema to store a graph on disk is similar to the VE schema described in Section 3. We load two of our representations (VE and RG) from this format. To apply a filter pushdown, the data on disk need to be sorted. For VE, we use the vertex/edge identifier as the first sort key, and the interval start time as the second key. Storing data in this way preserves temporal locality, and places the history of changes in a vertex or an edge together. Parquet does not support filter pushdown for datetime formats, hence we store time as UNIX timestamps (long).

We use a similar schema for RG, however, we sort vertices and edges by the interval start time first, and then by their vertex (resp. edge) identifier, to preserve structural locality. During our experiments we learned that RG can be loaded 30% faster using the structural locality instead of temporal locality (experiment omitted due to space constraints). While OG and OGC could be loaded in the same way as VE, we experimentally validated that it is significantly faster to pre-compute nested versions of the graphs with schemas described in Section 3, and then convert to OG or OGC during load time. A problem with this approach is that Parquet's filter pushdown will not work, since interval information is stored in a nested column. We resolve this issue by storing the first and last time a vertex/edge existed as a separate column on disk, and sorting on these columns.

We provide a GraphLoader utility that can initialize any of our physical representations from Apache Parquet files on HDFS or on local disk. This loader accepts a date range and filters the data through Parquet's filter pushdown. For datasets with a long evolution history, this optimization provides a substantial performance improvement (see [36]).

Coalescing. The coalesce primitive for merges adjacent and overlapping time periods for value-equivalent tuples. Several implementations have been suggested for the coalesce operation over temporal SQL relations [5]. In our implementation for VE, we use the partitioning method: grouping the vertex and the edge relation by key, then sorting by start time, and folding tuples within each group and checking pairs of adjacent tuples for value-equivalence. The effect of this operation is that a single tuple is produced for each period of maximum length during which no change occurred.

To further optimize performance, we coalesce lazily for sequences of two or more operations. Recall that aZoom^T computes in each snapshot, and so it does not require its input to be temporally coalesced to produce the correct output. In contrast, wZoom^T does require its input to be coalesced for correctness, because it computes across snapshots. This means that, in a sequence of aZoom^T and wZoom^T operators, the system does not need to temporally coalesce before invoking aZoom^T, but it must coalesce before invoking wZoom^T and at the end of the operator sequence, when the final result is produced.

5 EXPERIMENTAL EVALUATION

We conduct an experimental evaluation to study the performance of aZoom^T and wZoom^T. Our goal is to understand how different representations and their corresponding operator implementations perform for different datasets and workloads. We present three different categories of experiments: aZoom^T experiments (Section 5.1), wZoom^T experiments (Section 5.2), and experiments combining both operations (Section 5.3).

Cluster. All experiments are conducted on a 16-worker in-house Cloudera cluster, using Linux CentOS 14.04 and Spark v2.2. Each machine has 4 cores and 32 GB of RAM. Spark standalone cluster manager and Hadoop 2.6 were used. In each experiment, we report the mean runtime of three executions, each with a cold start. The runtime includes the setup time of submitting a job to the cluster manager, reading the data from disk, executing the operation, and materializing the results in memory. We set a 30-minute time-out for all experiments.

Datasets. We evaluate the performance on two real world datasets, WikiTalk and NGrams, and a family of synthetic datasets SNB, with different scale factors. All datasets are summarized in the table below, and differ in size, in the number and type of attributes, and in evolution rates, calculated as the average graph edit similarity [38] between consecutive snapshots (the edit similarity between snapshots i and j is the ratio of the number of common edges to the sum of the number edges: $2 * |E_i \cap E_j|/(|E_i| + |E_j|)$). In contrast to WikiTalk and NGrams, SNB is a growth-only graph, and so it shows a higher evolution rate.

	vertices	edges	snaps	ev. rate
WikiTalk	2.9M	10.7M	179	14.4
SNB:10	65K	1.9M	36	89
SNB:100	448K	20M	36	90
SNB:300	1.1M	59M	36	90
SNB:1000	3.3M	202M	36	91
NGrams:M	28M	606M	287	16.6
NGrams:L	48M	1.32B	328	18.2

WikiTalk is a real dataset that contains over 10 million messaging events (edges) among 3 million wiki-en users (vertices) at a 1-month resolution, from 2000 through 2016 [41]. Vertices have two attributes: `name` is a unique username for each account and `editCount` is the number of edits committed by the user (around 15K unique values). Edges have no attributes. WikiTalk is a very sparse dataset with short-lived edges and growth-only vertices: once added, a vertex persists for the lifetime of the graph and its attributes do not change.

NGrams is a real dataset that contains word co-occurrence pairs, with 88 million word vertices (3.2 unique words in all) and over 2.8 billion undirected co-occurrence edges. In our experiments we use two versions of this dataset: NGrams:L, with 328 yearly snapshots from 1520 through 1920, and NGrams:M, with 287 yearly snapshots from 1520 through 1870. NGrams is denser than WikiTalk; its vertices persist over time, while edges can appear and disappear. This dataset exhibits a linear relationship between the number of nodes and the number of edges.

The LDBC Social Network Benchmark (SNB) [12] is a synthetic graph generator that produces realistic networks with different types of entities and different attributes. We focus on SNB person entities (vertices) and on friendship relationships (edges), and generate datasets at four scale factors: SNB:10, SNB:100, SNB:300, and SNB:1000, with 36 monthly snapshots in each. SNB:1000 is the largest dataset that can be created without changing the generator source code. SNB does not generate a temporal benchmark but, since entities and relationships have timestamps, it can be viewed as a growth-only evolving graph. We use the vertices attribute `firstName` (5300 unique values in SNB:1000), edges have no attributes. The friendship network generated using SNB is growth-only graph, a graph where every vertex and edge is added once and never goes away.

5.1 Evaluation of aZoom^T

We now evaluate the performance of attribute-based zoom. Experiments are executed with RG, VE and OG, as described in Section 2, but not with OGC, which does not support aZoom^T.

Fixed number of groups, varying data size. Dataset size plays an important role in the performance of aZoom^T. We simulated different data sizes by using three datasets and varying the number of snapshots in each dataset. In this experiment and all other experiment in this section, we used aZoom^T with a hash function as the Skolem function f_S that generate new ids based on one of the attributes of the graph. In WikiTalk we group by

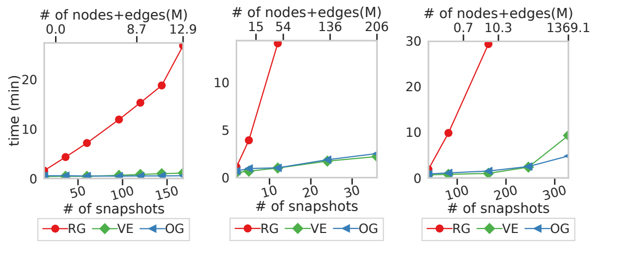


Figure 10: aZoom^T: The effect of dataset size on the runtime for each dataset. OG and VE perform on par, while RG quickly times out.

`username` (2.9M output groups), In NGrams— by `word` (3.2M output groups), and in SNB— by `firstName` (5,300 output groups).

Figure 10 shows the runtime of aZoom^T on different datasets. As expected, increasing the data size increases the execution time. OG is the best-performing representation, and VE is second-best. Both VE and OG exhibit sub-minute runtimes on WikiTalk: at most 0.54 min for OG and at most 1.09 min for VE (Figure 10a). The runtime of VE for SNB:1000 is at most 2.21 min for this graph with over 200M edges, where OG takes up to 2.53 minutes. (Figure 10b). Notably, OG scales well, even for NGrams, where OG computes in 4.8 min for 400 years worth of data and VE in 9.3, in a graph with 1.3 billion edges (Figure 10c). In contrast, RG is much slower than VE and OG, and it does not scale for the full SNB:1000 and NGrams:L dataset. It takes 26 minutes for WikiTalk, 14 minutes for 12 snapshots of SNB, timing out for anything larger and taking 29.55min to compute for 200 snapshots of NGrams, and timing out for 300 snapshots.

Fixed number of groups and graph size, varying number of snapshots. Another important factor in evolving property graphs that can impact operator performance is the number of snapshots (intervals during which no change occurred in the TGraph). We generate experimental datasets to measure this effect by merging consecutive snapshots of WikiTalk and NGrams:L, where we gradually decrease the number of intervals, while we keep the size of the dataset (in terms of the number of nodes and edges) fixed. For SNB:1000, we directly generate datasets with the desired number of snapshots. For WikiTalk, we select the last 160 months of history, and create graphs with between 2 and 160 snapshots. For NGrams, we select the last 320 years of the graph's history, and again generate datasets with between 2 and 320 snapshots. For SNB, we generate between 12 and 360 snapshots, corresponding to between 1 and 30 years worth of network evolution. Note that generating graphs in this way does neither change the number of nodes and edges, nor the group-by cardinality.

Figure 11 shows the runtime of aZoom^T as a function of the number of snapshots. OG and VE exhibit comparable performance for WikiTalk, executing in under 2 minutes, with OG being slightly more efficient. The trends are different in SNB: the runtime of aZoom^T on both OG and VE is near-constant, but VE is more efficient: 2.3 minutes for VE, compared to 2.9 minutes for OG. OG outperforms VE for NGrams; their runtime increases linearly with an increasing number of intervals.

The difference in performance across datasets is due to the nature of data evolution. WikiTalk and SNB only have one tuple per node, since attributes do not change over time, therefore an

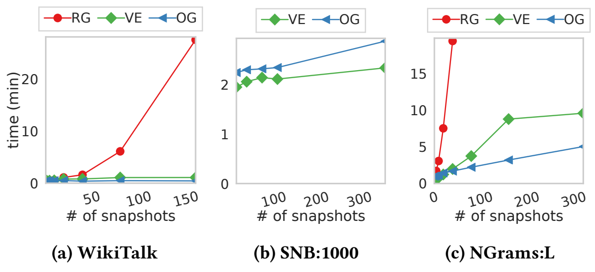


Figure 11: aZoom^T: Fixed dataset size and group-by cardinality, varying number of snapshots. The number of nodes and edges is fixed to the largest graph size, and the group-by cardinality is fixed to the natural group-by cardinality of each dataset.

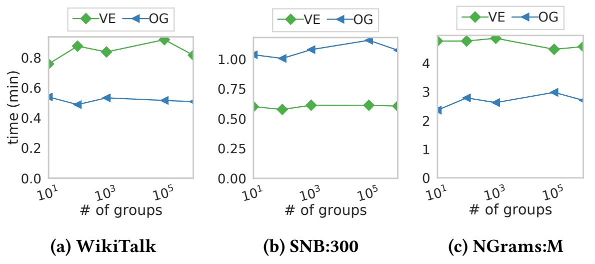


Figure 12: aZoom^T: Fixed dataset size and number of snapshots, with varying group-by cardinality.

increase in the number of intervals does not change the number of tuples (which is not the case for NGrams). We observe that RG is the least efficient representation for this operation, except for the smallest number of intervals in WikiTalk, where all representations have roughly similar performance. The running time of RG grows linearly with the number of intervals, with a high slope. We timed out this experiment at 30 minutes per execution, and RG failed to complete for SNB and NGrams at 80 intervals.

Fixed size and number of snapshots, varying group-by cardinality. In this aZoom^T experiment, we investigate the effect of group-by cardinality — the number of new nodes being created by the aZoom^T operation, on performance. We work with the WikiTalk, SNB:300 and NGrams:300 datasets at their original temporal resolution. We vary the number of groups in the output by assigning a group identifier to each node in the input. Group identifiers are drawn uniformly at random from a given integer range. We varied the range to control group-by cardinality, setting it to 10, 100, 1,000, 100,000, and 1,000,000. Figure 12 shows the results of this experiment. We observe that the runtime of aZoom^T over OG, VE and RG is not affected by group-by cardinality. For visibility purpose, we did not include RG in Figure 12. On WikiTalk, RG showed an execution time of about 29 minutes for all the group-by cardinality values.

Frequency of change. In our final aZoom^T experiment, we study the effect of the frequency of change on performance. Therefore, we synthetically change vertex attributes values with a fixed frequency. While this intervention does not change the size of the graph in terms of the number of nodes and edges, it does change the storage requirements (e.g., the number of tuples for VE, or the length of the array in OG) for each vertex.

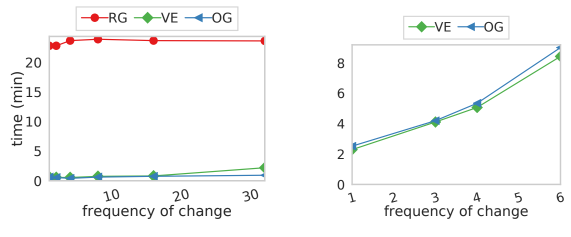

(a) WikiTalk (b) SNB:1000

Figure 13: aZoom^T: Fixed dataset size and number of snapshots, varying frequency of change of vertex attributes

Figure 13 shows the effect of the frequency of change on the performance for WikiTalk (Figure 13a) and SNB:1000 (Figure 13b). The size of each graph and the contained number of snapshots is fixed to the full dataset size. While the group-by cardinality does vary, the number of new groups is of the same order of magnitude as in the original graphs. We observe that the frequency of change has no effect on the performance of RG. This is because RG stores each vertex once per snapshot, irrespective of whether there was a change between consecutive snapshots. The runtime of aZoom^T on OG is higher when more changes occur. This is expected: Recall that OG stores attributes with their corresponding validity intervals in an array, and so a higher frequency of change results in longer arrays, which slows down operations on OG. VE stores each change as a new tuple,and a higher frequency of change results in more tuples, slowing down VE as well.

Summary. We studied the effects of data size, the number of snapshots, the frequency of attribute change, and the group-by cardinality (e.g., the number of newly-computed nodes) on the performance. We observed that OG is the best representation for aZoom^T, followed by VE. For our largest experimental datasets, with over 1.3 billion edges, aZoom^T can be executed in less than 5 minutes with OG. The dataset size (the total number of nodes and edges) affects operator performance on all datasets. The number of representative graphs (snapshots) has a small effect on VE and OG, and a significant effect on RG. We did not observe an effect of the group-by cardinality on the runtime in any representation. The frequency of change has a small effect on RG, but it affects VE and OG significantly.

5.2 Evaluation of wZoom^T

We now investigate the performance of wZoom^T. In all experiments, we load RG from disk enforcing structural locality, and VE enforcing temporal locality. For OG and OGC we load data from nested format described in Section 4.

Fixed time window, changing data size. In this experiment, we fix the zoom window size to 3 snapshots for WikiTalk (grouping into up to 60 temporal windows) and SNB:1000 (grouping into up to 12 temporal windows), and 25 snapshots for NGrams:L (grouping into up to 16 temporal windows). We load different temporal slices of each graph and measure the execution time of wZoom^T. Figure 14 shows the results of this experiment. We applied "exists" quantifiers for both nodes and edges. We observed similar results for "all" quantifiers (except that they make wZoom^T slightly faster as fewer nodes and edges have to be kept in the result), which we omit for space reasons.

As expected, increasing the size of the graph increases the runtime on all representations. Our implementation based on

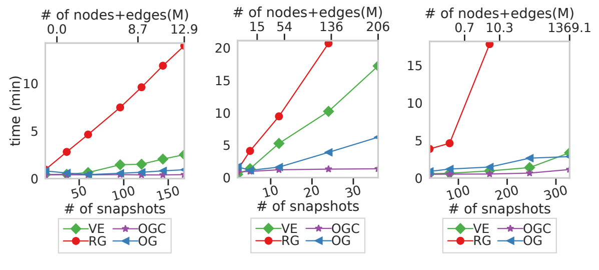

(a) WikiTalk (b) SNB:1000 (c) NGrams:L

Figure 14: wZoom^T: Fixed window size, changing data size, nodes=exists, edges=exists

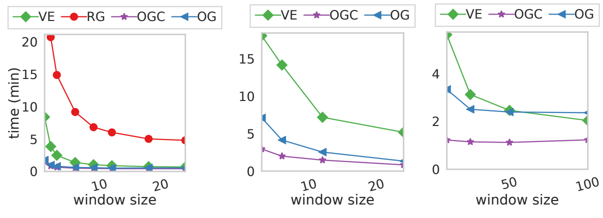

(a) WikiTalk (b) SNB:1000 (c) NGrams:L

Figure 15: wZoom^T: Fixed data size and number of intervals with varying window size, nodes=all, edges=all. OG and OGC outperform other representations.

OGC is the clear winner for all datasets, taking 0.41 minutes for WikiTalk, 1.25 minutes for SNB and 1.12 minutes for NGrams. For WikiTalk and SNB, OG is the second winner while VE performs better for NGrams:L, particularly for larger TGraph sizes. Finally, RG performs worst for all datasets. The reason for VE's significant performance drop on SNB is window size. We look at the impact of window size on wZoom^T in the next section.

Fixed data size, varying temporal window size. In the previous experiment, we used a fixed zoom window size and increased the size of the graph. In this experiment, the size of the graph is fixed and we vary the size of the temporal window. Figure 15 shows the corresponding results. RG does not scale for temporal window-based zoom on large datasets, therefor we only report performance numbers for RG on WikiTalk. We observe that the performance of OG and OGC does not depend on the window size, while the operations on VE take longer to execute for smaller temporal windows. OGC is the winner among all datasets followed by OG; VE exhibits longer runtimes for smaller window sizes especially. This is because VE creates copies of each tuple in order to align them with the computed time windows. The smaller the window, the more tuples are created in the intermediate stage. This effect is more visible for WikiTalk and SNB because of their growth-only nature. In SNB, each vertex or edge exists from its start date to the life time of the graph, therefore VE needs to create a large amount of copies as each of those long intervals is split into intervals corresponding to the window size.

Summary. We studied the effect of data size and of temporal window size on performance. Our experiments showed that OGC performs best, followed by OG and VE. RG was the slowest representation in all cases. We also observed that smaller temporal window sizes (and thus more windows to compute) lead to longer execution time for RG and VE.

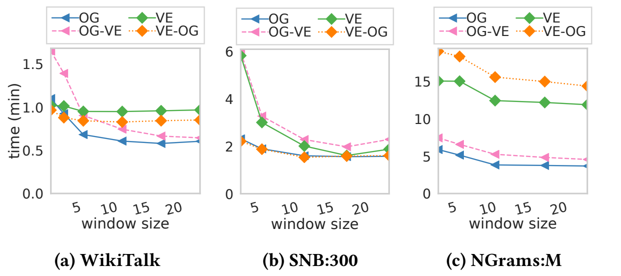


Figure 16: aZoom^T - wZoom^T combination and switching between memory representations. Fixed data size, group-by cardinality and number of intervals, varying the size of windows, node quantifier 'all', edge quantifier 'all'.

5.3 Operation Chaining

In this section we chain together aZoom^T to a wZoom^T and investigate whether switching between representations improves performance. Since OGC does not support attribute-based operation and due to the high memory usage and scalability issues of RG, we only run our experiments on VE and OG.

In the first experiment we run aZoom^T then wZoom^T with different windows sizes on WikiTalk, SNB:300 and NGrams:M. For aZoom^T on WikiTalk, we use edit count as the zoom attribute, for SNB we use first name, and for NGrams we use the word attribute. Figure 16 shows the results of this experiment. The x-axis lists window sizes for wZoom^T (in months for WikiTalk and SNB, and in years for NGrams), while the y-axis denotes the running time in minutes. Each line shows which representation is used. On WikiTalk, OG is the winner while OG-VE, VE-OG and VE are slightly slower. On SNB:300, VE-OG, and OG are fastest, and OG-VE is slowest, followed by VE.

In the previous section we saw that VE performs slightly better for aZoom^T on SNB, and OG performs significantly better for wZoom^T, so it makes sense for VE-OG and OG to show the best performance and for VE and OG-VE to show the worst. For NGrams, OG is the clear winner followed by OG-VE. The worst-performing combination here is VE-OG, followed by VE. On NGrams, OG performs significantly better for both aZoom^T and wZoom^T, and this can explain the results we are observing here.

In the next experiment we change the order of aZoom^T and wZoom^T. While this reordering does not always produce the same result, we can safely reorder the operations for WikiTalk and SNB, since no attributes change in these datasets, and so applying wZoom^T or aZoom^T first produces the same result with the "exist" quantifier for both vertices and edges.

Figure 17 shows the effect of group-by cardinality on wZoom^T - aZoom^T and aZoom^T - wZoom^T. In this experiments, we load the full graph for each dataset, project each node attribute to a random value based on group-by cardinality, and then perform the operations, with window size set to 6 months for WikiTalk and SNB, and 10 years for NGrams. We vary group-by cardinality from 10 to 1 million. We observe an increase in the execution time as the group-by cardinality increases, which we attribute to the fact that aZoom^T produces a larger intermediate graph for cases where we perform aZoom^T first. In contrast, we see no significant change in the execution time when wZoom^T is executed first. Interestingly, performing wZoom^T first in NGrams yields faster running time. Unlike in WikiTalk and SNB, vertices in NGrams are not growth-only, and they also span over a longer

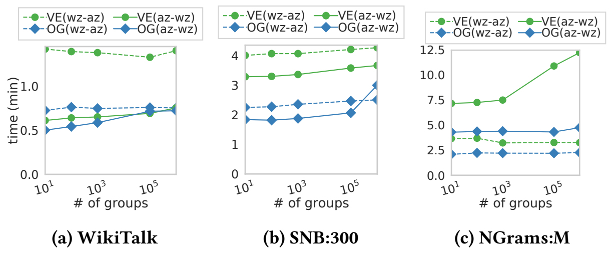


Figure 17: aZoom^T and wZoom^T performance for different group-by cardinalities with different zoom orders. Fixed data size and number of intervals. OG-based implementations perform best in most cases.

period of time. wZoom^T will reduce the number of snapshots and vertex tuples, which explains why wZoom^T - aZoom^T is faster than aZoom^T - wZoom^T.

Summary. We studied combining aZoom^T and wZoom^T for different combinations of parameters. Our experiments show that, while OG alone performs best in most cases, switching between representations does not significantly affect the running time. We also found that running aZoom^T before wZoom^T is fastest for growth-only datasets.

5.4 Summary

In this section, we first studied the effects of data size, the number of snapshots, the frequency of attribute change, and the group-by cardinality on the running time of aZoom^T. We observed that OG is the best performing representation for aZoom^T, followed by VE. We showed that representing the TGraph as a sequence of independent snapshots in RG results in the by far worst performance. The second part of this section focused on wZoom^T. We varied graph size and window size, and observed that OGC is the best-performing representation, followed by OG and VE. RG again exhibited the worst performance for wZoom^T. The last part of this section focused on combining aZoom^T and wZoom^T.

Overall, we found that OG, which balances temporal and structural locality, outperforms other representations in most cases.

6 RELATED WORK

Temporal models and languages in the relational literature are very mature (see, e.g., [10, 16, 23]). However, the same cannot be said for evolving graphs, where models differ in what time representation they adopt (point or interval), what top-level entities they model (graphs or sets of nodes and edges), whether they represent topology only or attributes or weights as well, and what types of evolution they support. Harary and Gupta [20] were, to the best of our knowledge, the first to informally propose to model graph evolution as a sequence of static graphs. This model has been predominant in the literature [15, 24–26, 38, 40], with various restrictions on the kinds of changes that can take place during graph evolution. In contrast to existing work, TGraph assigns periods of validity to nodes, edges and their properties, capturing evolution of graph topology and of node and edge attributes, and supports point-based semantics [37].

The attribute-based zoom operator is a temporal generalization of the node creation operator that is present in several conventional (non-temporal) graph query languages [42]. For example, StruQL outputs new nodes in a create clause, corresponding to the node creation operation with a Skolem function to create the object ids [13], while GOOD provides an abstraction operator that allows to create new nodes to represent multiple nodes based on shared properties [18]. To the best of our knowledge, a temporal generalization of this operator has not been considered except in our own prior work [37], and also has not been implemented in systems.

That said, the G* system supports SQL-style aggregation using the AggregateOperator per graph snapshot, supporting a limited version of summarization [26]. G* ingests evolving graph data one snapshot at a time, replicated across all machine without any compression, and is most similar to our RG. Our experiments showed that G* is not capable of loading graphs with a large number of intervals and does not scale for large size graphs [36]. We were not able to fully ingest any of our datasets used in Section 5 into G*. Chronograph, a system designed for temporal graph traversal [6], implements a version of temporal aggregation for the purpose of converting point-based to period-based semantics for edges, but not for nodes.

Temporal aggregation operators over relational data can be found in the literature, typically as an extension of non-temporal relational aggregation (see, e.g., example 10 in [11]). Li et al. proposed a general window aggregate for data streams [29] that can be applied to temporal relational data. Window aggregate semantics is based on a sliding window specification — a range and a slide — based on the desired data attribute that has a domain with a total order. The range specifies the width of the window e.g., 100 seconds or 100 rows, and the slide specifies how windows are formed. We are not aware of any proposal for an operator capable of changing the temporal resolution of evolving graphs, besides our own, introduced in [37], and no systems work on such an operator.

In our work we implement aZoom^T and wZoom^T operators in a dataflow system, and instantiate our ideas over Apache Spark [43], using the GraphX [17] library. We leverage the graph-specific optimizations provided by GraphX, as described in our implementation section, and incorporate temporal semantics into data representations and operators.

7 CONCLUSION

In this paper we proposed an implementation of two zoom operators — aZoom^T and wZoom^T — on evolving graphs. We detailed four physical representations — RG, VE, OG, and OGC, and described how to define the zoom operators using distributed dataflow operations, tailored to the corresponding data representation. We discussed how to efficiently implement the operators in Apache Spark with its GraphX library, and explained that our operator definitions could easily be implemented in other dataflow systems such as Apache Flink. In an extensive experimental evaluation on several real datasets with up to 1.3 billion edges, we explored the trade-offs in terms of temporal and structural locality with respect to zoom operator performance. We find that OG, which balances temporal and structural locality, outperforms the other representations in most cases.

In our future work we will extend our system to support additional operations on evolving graphs, such as Pregel-style analytics [30]. We will propose query optimization techinques for our workloads. Finally, we will design a query language with support for the proposed temporal zoom operators, among others.

REFERENCES

[1] Charu C. Aggarwal and Karthik Subbian. 2014. Evolutionary Network Analysis. *ACM Comput. Surv.* 47, 1 (2014), 10:1–10:36.
[2] Alexander Alexandrov et al. 2014. The stratosphere platform for big data analytics. *VLDB* 23, 6 (2014), 939–964.
[3] Renzo Angles et al. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40.
[4] Michael H. Böhlen. 2009. Temporal Coalescing. In *Encyclopedia of Database Systems.* 2932–2936.
[5] Michael H Böhlen et al. 2009. Temporal Compatibility. In *Encyclopedia of Database Systems.* 2936–2939.
[6] Jaewook Byun et al. 2019. ChronoGraph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *IEEE TKDE* (2019).
[7] Paris Carbone et al. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.
[8] Jeffrey Chan et al. 2008. Discovering correlated spatio-temporal changes in evolving graphs. *Knowl. Inf. Syst.* 16, 1 (2008), 53–96.
[9] Junghoo Cho and H Garcia-Molina. 2000. The evolution of the web and implications for an incremental crawler. *VLDB* (2000), 200–209.
[10] Jan Chomicki. 1994. Temporal Query Languages: A Survey. In *ICTL.*
[11] Dignös et al. 2012. Temporal Alignment. In *SIGMOD.*
[12] Orri Erling et al. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD.* 619–630.
[13] Mary F. Fernández et al. 1997. A Query Language for a Web-Site Management System. *SIGMOD Record* 26, 3 (1997), 4–11.
[14] Mary F. Fernández et al. 2000. Declarative Specification of Web Sites with Strudel. *VLDB J.* 9, 1 (2000), 38–55.
[15] Afonso Ferreira. 2004. Building a reference combinatorial model for MANETs. *IEEE Network* 18, 5 (2004), 24–29. https://doi.org/10.1109/MNET.2004.1337732
[16] Shashi K Gadia and Chuen-Sing Yeung. 1988. A generalized model for a relational temporal database. In *ACM SIGMOD Record*, Vol. 17. ACM, 251–259.
[17] Joseph E. Gonzalez et al. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX.* 599–613.
[18] Marc Gyssens et al. 1994. Decomposing Constraint Satisfaction Problems Using Database Techniques. *Artif. Intell.* 66, 1 (1994), 57–89.
[19] Wentao Han et al. 2014. Chronos : A Graph Engine for Temporal Graph Analysis. In *EuroSys.*
[20] F. Harary and G. Gupta. 1997. Dynamic graph models. *Mathematical and Computer Modelling* 25, 7 (1997).
[21] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the SIGMOD.* 405–418.
[22] Ping-yu Hsu and D Stott Parker. 1995. Improving SQL with Generalized Quantifiers. In *ICDE.*
[23] Christian S. Jensen and Richard T. Snodgrass. 2009. Temporal Data Models. In *Encyclopedia of Database Systems.* 2952–2957.
[24] Andrey Kan et al. 2009. A Query Based Approach for Mining Evolving Graphs. In *AusDM 2009*, Vol. 101.
[25] Udayan Khurana and Amol Deshpande. 2016. Storing and Analyzing Historical Graph Data at Scale. In *EDBT.*
[26] Alan G. Labouseur et al. 2014. The G* graph database: efficiently managing large distributed dynamic graphs. *Distrib. and Parallel Databases* 33, 4 (2014).
[27] M. Lahiri and Berger-Wolf. 2008. Mining Periodic Behavior in Dynamic Social Networks. In *2008 Eighth IEEE ICDM.* 373–382.
[28] Timothy LaRock et al. 2019. Detecting Path Anomalies in Time Series Data on Networks. *arXiv preprint arXiv:1905.10580* (2019).
[29] Jin Li et al. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD.*
[30] Grzegorz Malewicz et al. 2010. Pregel: a system for large-scale graph processing. In *ACM SIGMOD.* 135–146.
[31] F Manola et al. 2013. RDF primer. W3C Recommendation 10, 1–107 (2004).
[32] Mauro San Martín et al. 2011. SNQL: A Social Networks Query and Transformation Language. In *AMW.*
[33] Frank McSherry et al. 2013. Differential Dataflow. In *CIDR 2013,.*
[34] Sergey Melnik et al. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. In *VLDB.*
[35] Youshan Miao et al. 2015. ImmortalGraph: A System for Storage and Analysis of Temporal Graphs. *ACM Transactions on Storage* 11, 3 (2015), 14–34.
[36] Vera Z. Moffitt. 2017. *Framework for Querying and Analysis of Evolving Graphs.* Ph.D. Dissertation. Drexel University.
[37] Vera Zaychik Moffitt and Julia Stoyanovich. 2017. Temporal graph algebra. In *Proceedings of DBPL 2017.* 10:1–10:12.
[38] Chenghui Ren et al. 2011. On Querying Historical Evolving Graph Sequences. *Proceedings of the VLDB Endowment* 4, 11 (2011), 726–737.
[39] Ingo Scholtes et al. 2016. Higher-order aggregate networks in the analysis of temporal networks: path structures and centralities. *The European Physical Journal B* 89, 3 (2016), 61.
[40] Konstantinos Semertzidis et al. 2015. TimeReach: Historical Reachability Queries on Evolving Graphs. In *EDBT.*
[41] Jun Sun and Jérôme Kunegis. 2016. Wiki-talk Datasets.
[42] Peter T. Wood. 2012. Query languages for graph databases. *SIGMOD Record* 41, 1 (2012), 50–60.
[43] Matei Zaharia et al. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

Efficient Search for Multi-Scale Time Delay Correlations in Big Time Series Data

Nguyen Ho[†], Torben Bach Pedersen[†], Van Long Ho[†], Mai Vu[‡]
[†]Aalborg University, [‡] Tufts University

ABSTRACT

Very large time series are increasingly available from an ever wider range of IoT-enabled sensors deployed in different environments. Significant insights and values can be obtained from these time series through performing cross-domain analyses, one of which is analyzing time delay temporal correlations across different datasets. Most existing works in this area are either limited in the type of detected relations, e.g., linear relations alone, only working with a fixed temporal scale, or not considering time delay between time series. This paper presents our Time delaY COrrelation Search (TYCOS) approach which provides a powerful and robust solution with the following features: (1) TYCOS is based on the concept of mutual information (MI) from information theory, giving it a strong theoretical foundation to detect all types of relations including non-linear ones, (2) TYCOS is able to discover time delay correlations at multiple temporal scales, (3) TYCOS works in an efficient, bottom-up fashion, pruning non-interesting time intervals from the search by employing a novel MI-based noise theory, and (4) TYCOS is designed to efficiently minimize computational redundancy. A comprehensive experimental evaluation using synthetic and real-world datasets from the energy and smart city domains shows that TYCOS is able to find significant time delay correlations across different time intervals among big time series. The performance evaluation shows that TYCOS can scale to large datasets, and achieve an average speedup of 2 to 3 orders of magnitude compared to the baselines by using the proposed optimizations.

1 INTRODUCTION

Rapid advancements in IoT technology have enabled the collection of enormous amounts of time series data at unprecedented scale and speed. For example, a modern wind turbine has hundreds of sensors sampled at a high frequency, a smart building contains thousands of sensors sensing the surrounding environment, and an autonomic vehicle carries numerous vision sensors. All of them are collecting terabytes of data everyday. Analyzing these massive, heterogeneous and rich datasets can help uncover hidden patterns and extract new insights to support evidence-based decision making.

While time series analysis has been studied extensively in the past, its importance and value only continue to grow. One of the first steps to harness the enormous potential from modern big time series is to discover correlations among heterogeneous and cross-domain datasets. Consider for example the NYC Open Data [2] with more than 1,500 published datasets containing quantitative data from different domains, including weather and transportation, energy and environment etc. Cross-domain analysis among these datasets can reveal new insights about the city and its citizens, and thus aid policy makers in decision making. For instance, finding correlations between weather and transportation data can lead to the identification of individual weather events, such as the occurrence of a storm or a hurricane, which then helps explain an abnormal increase in the number of accidents. Data correlation is also useful in behavioral prediction and future planning. For example, illustrating that weather data (e.g., wind speed) is well-correlated with energy production can provide accurate prediction of the city's energy capacity at a specific time, thus allowing better resource planning. In the financial domain, data correlation can help forecast the price movement of related stocks, or predict the purchasing behavior of consumers, and thus assist investors in making real-time investment decisions. Not only is it useful in reasoning and predicting, data correlation can also be considered as one of the three building blocks to establish a causal relation [3], and thus can serve as a basis for constructing inference and learning models.

Despite its potential use, finding correlations in big time series is challenging. Not only does the very large volume of data raises significant challenges in terms of performance and scalability, their complex and noisy nature also presents difficulties in finding different types of correlation relations, or in the ability to deal with adaptive temporal scales. For example, stock prices or weather data exhibit non-linear relations, which cannot be captured by traditional correlation metrics such as Pearson Correlation Coefficient [23]. Besides, there is often a misconception that finding correlations and finding similarities in time series are the same task, where in fact, they are two different problems. Finding correlations is to look for statistical relationships in the data, whereas finding similarities means to find the optimal matching and/or alignment between time series sequences. Unlike the correlation-based approach, similarity metrics (which have positive values only) cannot distinguish between un-correlated and negatively correlated time series, both of which may have values close to 0. For example, consider a pair of time series (X, Y) generated by a sine function $y = sin(x)$. Here, $X \in (-\infty, \infty)$ represents a linearly increasing time series, while Y follows a sine function of X. In this example, X and Y do not exhibit any similarities among their values, but they do have an underlying relation. Such non-linear relations are common in areas such as signal processing, but cannot be detected using similarity measures. Thus, methods such as those used in Dynamic Time Warping [28] or MatrixProfile [31] have significant limitations in analyzing modern time series.

To make the problem even more challenging, cross-domain correlations might appear at different temporal scales. For example, correlations involving weather data might span over multiple temporal durations ranging from hours (e.g., during rain showers), to days, or even weeks (e.g., during a storm) depending on the weather events. Likewise, interactions between events might not always occur simultaneously. In practice, it is common to see events of one phenomenon influence other phenomena only after some delay of time. For instance, an increase of incidents caused by heavy rain can only be observed minutes or hours after the rain starts; or the impact of one rising stock on other stocks is visible only a few hours later. Given a heterogeneous set of time series data, there is a need to identify not only which datasets are correlated, but also when the correlations occur, and at what time delay.

Although correlation analysis has been researched extensively, current techniques are limited either in the type of detected relations, i.e., only linear ones, or the temporal scale and time delay in which they deal with. Most of the correlation works do not consider adaptive temporal scales as they assume correlations only exist for a fixed time period, e.g., [29], or ignore the time delay between variables of interest. There is no existing work that offers a holistic solution for searching window-based correlations, considering both multiple temporal scales and time delay, in modern big time series data.

This paper aims to address those challenges and limitations by introducing the Time delaY COrrelation Search (TYCOS) approach, making the following novel contributions: (1) We propose the first, to our knowledge, comprehensive solution for the multi-scale time delay correlation search problem that extracts significant correlations from big time series. (2) TYCOS is based on the concept of mutual information from information theory, giving it a strong theoretical foundation and the ability to discover various types of correlation relations, including linear and non-linear, monotonic and non-monotonic, functional and non-functional. (3) By combining the well-known Late Acceptance Hill Climbing (LAHC) search method with a window-based approach, TYCOS can automatically discover time delay correlations at multiple temporal scales, without requiring user inputs to specify the window sizes or the delay. (4) Based on mutual information properties, we propose a novel theory to identify noise in the data, enabling efficient pruning of non-interesting time intervals from the search, thus significantly reducing the search space and improving the search speed. (5) TYCOS is designed with efficient data structures to reuse the MI computation across a large number of windows, thus minimizing the computation redundancy. Moreover, TYCOS is scalable since it is designed in an efficient bottom-up fashion, making the method memory efficient and suitable for big datasets. And finally (6), we perform a comprehensive experimental evaluation using synthetic and real-world datasets from the energy and smart city domains, which shows that TYCOS is able to find interesting and important correlations among time series with high accuracy, can scale to large datasets, and achieves an average speedup of 2 to 3 orders of magnitude compared to the baselines.

2 RELATED WORK

Finding correlations among datasets is a fundamental step in data exploration. In the past, correlation analyses relied on traditional statistical metrics such as covariance or correlation coefficients to measure correlations [13, 15, 18, 19, 32]. However, these metrics are usually best for linear and/or monotonic dependencies. Recent studies had attempted to approach the problem from a high level. Sarma et al. [10] use the concept of *relatedness*, Pochampally et al. [24] use *joint precision* and *joint recall*, Alawini et al. [4] rely on the history and schema of datasets, Roy et al. [26] use the concept of *intervention*, to identify relations between datasets or data tables. Middelfart et al. [21] propose a bitmap-based approach to measure change relationships in a data cube. Chirigati et al. [7] propose a topology-based framework to identify spatio-temporal relationships in heterogeneous data corpuses. These studies, however, only focus on overall correlations. None of them consider correlations in time windows.

Surprisingly, very little effort has been made to design efficient solutions for time delay window-based correlations. Among them, Rakthanmanon et al. [25] design a Dynamic Time Warping-based technique (MASS) to quickly find the most similar subsequences in time series. Although considered to be the state of the art for subsequences matching, the technique does not have a mechanism to automatically search for correlated windows, but rather relies on a provided query. To improve MASS, Yeh et al. [31] designed the MatrixProfile framework to perform similarity joins between time series. However, as will be shown in Section 8.3, MASS and MatrixProfile cannot detect complex relations such as non-linear and non-functional ones. Other works, e.g., [8, 29] propose sliding window-based procedures to detect hidden correlations. However, they only focus on fixed size windows, not considering time delay, or using correlation coefficients as correlation measures, and thus, cannot find multi-scale time delay correlations and are limited in the types of relations they can detect. Our work in this paper overcomes those limitations. Since TYCOS uses MI as a correlation metric, it can discover all types of relationships. Furthermore, TYCOS works in a bottom-up fashion, and can thus automatically discover time delay correlations at multiple temporal scales.

Prior to this work, we investigated the use of MI in correlation discovery, and proposed AMIC [16, 17], a top-down approach to search for multi-scale correlations in big data. However, AMIC does not consider time delay correlations. Recently, we examine the power of LAHC in correlation search in a short paper [14]. The present paper significantly extends [14] by considering time delay correlations, and proposes a novel noise theory and MI computation technique to achieve better performance.

3 BACKGROUND

3.1 Mutual Information

MI is a statistical measure to quantify the shared information between two probability distributions. Given two discrete random variables X, Y with the corresponding probability mass functions (p.m.fs) $p(x)$, $p(y)$, and the joint distribution $p(x, y)$, the MI between X and Y is defined as

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(x, y) \log \frac{p(x, y)}{p(x)p(y)} \tag{1}$$

Intuitively, $I(X;Y)$ represents the reduction of uncertainty of one variable (e.g., X) given the knowledge of another variable (e.g., Y) [9]. The larger $I(X;Y)$, the more information is shared between X and Y, and thus, the less uncertainty about one variable when knowing the other. The property that MI is equal to zero if and only if the considered variables are statistically independent, otherwise always positive if there exists any kind of dependency (e.g., linear and non-linear) [11], makes MI a versatile measure to capture correlations in noisy datasets which often exhibit a high degree of bias and abnormality, causing their relationships to often be arbitrary and non-linear.

Estimating mutual information: Eq. (1) is the theoretical definition of MI but is usually not used for computing MI, as it requires having the distributions of the underlying data which are often unknown in practice. To estimate MI from collected samples, we choose an estimation method proposed by Kraskov et al. [20] (hereafter called the *KSG* method) for several reasons: (1) The *KSG* method outperforms other estimators (e.g., histogram, kernel density estimation) in terms of computational efficiency and accuracy [22]; (2) The method uses $k-$nearest neighbor approximation and thus is data efficient, adaptive and has minimal bias [20]. These reasons make the *KSG* method particularly suitable for discovering temporal correlations in big and heterogeneous time series where the dependencies between different time series can be complex and might occur at multiple time scales.

The main idea of the *KSG* estimator is, instead of directly computing the joint and marginal probability distributions of the considered variables, it approximates the distributions by computing the densities of data points in nearby neighborhoods [20]. Specifically, *KSG* computes the probability distribution for the distance between each data point and its k^{th} nearest neighbor. For each data point, it searches for k nearest neighbor clusters (k is a pre-defined parameter) and computes the distance d_k to the k^{th}-neighbor. Then, the population density is estimated by counting the number of data points that fall within d_k. This leads to the computation of MI between two variables X and Y as [20]:

$$I(X;Y) = \psi(k) - 1/k - \langle \psi(n_x) + \psi(n_y) \rangle + \psi(n) \quad (2)$$

where ψ is the digamma function, k is the number of nearest neighbors, (n_x, n_y) are the number of marginal data points in each dimension falling within the distance d_k, n is the total number of data points and $\langle \cdot \rangle$ is the average function. The digamma function ψ is a monotonically increasing function. Thus, the larger n_x and n_y (i.e., more data points fall within the distance d_k), the lower $I(X;Y)$, and vice versa.

3.2 Late Acceptance Hill Climbing

Our correlation search algorithm is built based on LAHC [6] which we briefly introduce next. LAHC is an optimization technique attempting to find local optimal solutions for a given problem through iterative improvement. Given a target function f and a current solution S of f, LAHC tries to improve S by exploring potential candidates in the nearby neighborhood. If a better solution for f is found (according to some criteria), the current solution S is replaced by this new solution S_{new}, and the process is repeated until no further improvement can be made. LAHC is an extension of the classic Hill Climbing (HC) [27], but it differs from HC in its acceptance rule: a solution S_{new} is accepted if S_{new} is better than either the current solution S or a solution S_{old} found in the history. To do that, LAHC uses a fixed length array L_h to maintain a history of the most recently accepted solutions, and use L_h to justify the goodness of a candidate solution.

4 PROBLEM FORMULATION

Definition 4.1 *(Time series)* A time series $X_T = \{x_1, x_2, ..., x_n\}$ is a sequence of data values that measures the same phenomenon during an (observation) time period T, and is sorted in time order.

Note that the time period $T = [t_1, t_n]$ contains n time steps where each time step t_i has a recorded value $x_i \in X_T$, and t_1 and t_n denote the first, and the last time step of T. We say X_T has length n if X_T contains n data samples.

Definition 4.2 *(Time window)* A time window $w = [t_s, t_e]$ is a temporal sub-interval of T that records the events of X_T from time step t_s to time step t_e, and forms a (sub) time series $X_w = \{x_{t_s}, ..., x_{t_e}\} \subseteq X_T$.

We say w has size m, denoted as $|w| = m$, if w contains m time steps, and is equivalent to X_w containing m data samples.

Definition 4.3 *(Pair of time series)* A pair of two time series $(X_T, Y_T) = (\{x_1, x_2, ..., x_n\}, \{y_1, y_2, ..., y_n\})$ contains data collected from X_T and Y_T that measure two separate phenomena during the same observation period T. A tuple $(x_i, y_i) \in (X_T, Y_T)$ records the data values on X_T and Y_T at the same time step t_i.

Definition 4.4 *(Pair of time windows)* Let $w_X = [t_{x_s}, t_{x_e}]$, $w_Y = [t_{y_s}, t_{y_e}]$ be time windows of X_T and Y_T, respectively. Assume w_X and w_Y have the same length, i.e., $|w_X| = |w_Y|$. The pair of time windows $(w_X, w_Y) = ([t_{x_s}, t_{x_e}], [t_{y_s}, t_{y_e}])$ records the events of X_T from $[t_{x_s}, t_{x_e}]$, and of Y_T from $[t_{y_s}, t_{y_e}]$, and forms a pair of (sub) time series $(X_w, Y_w) = (\{x_{t_{x_s}}, ..., x_{t_{x_e}}\}, \{y_{t_{y_s}}, ..., y_{t_{y_e}}\}) \subseteq (X_T, Y_T)$.

Definition 4.5 *(Time delay window of a time series pair)* Let $(w_X, w_Y) = ([t_{x_s}, t_{x_e}], [t_{y_s}, t_{y_e}])$ be a pair of time windows like in Definition 4.4, and τ be an integer. The pair (w_X, w_Y) is called a time delay window of (X_T, Y_T) with the delay τ if $t_{y_s} - t_{x_s} = \tau$, and is denoted as $w_{X,Y+\tau} = ([t_s, t_e], \tau)$, where $t_s = t_{x_s}$ and $t_e = t_{x_e}$ are the start time and the end time of $w_{X,Y+\tau}$ on X_T, and τ is the time delay of w_Y w.r.t. w_X.

The window $w_{X,Y+\tau} = ([t_s, t_e], \tau)$ in Definition 4.5 defines a one-to-one mapping $f\colon w_X \mapsto_\tau w_Y$ that maps each event in w_X to the corresponding event in w_Y. The mapping is time correspondence, i.e., the event at the i^{th} time step of X_T in w_X is mapped to the event at the $(i+\tau)^{th}$ time step of Y_T in w_Y. Each window $w_{X,Y+\tau}$ is characterized by three parameters: the start time t_s, the end time t_e, and the time delay τ. The size of $w_{X,Y+\tau}$ equals to the size of w_X and w_Y, i.e., $|w_{X,Y}| = |w_X| = |w_Y|$.

A time delay window represents a shift (also called a "delay" or "lag") in time between two time series X_T and Y_T, and the value of τ indicates the shifted time units. Since τ can be equal to 0, or positive, or negative, the window $w_{X,Y+\tau} = ([t_s, t_e], \tau)$ is generalized for all time shifting scenarios. Semantically, if $\tau = 0$, then $w_{X,Y+\tau}$ does not have a time delay (or events of X_T in w_X and events of Y_T in w_Y occur at the same time). Whereas if $\tau > 0$, then w_Y is delayed τ time units from w_X (or events in w_Y occur τ time units after events in w_X). Similarly, if $\tau < 0$, w_X is delayed τ time units from w_Y. For brevity, in this paper, the two terms *time delay window* and *window* are used interchangeably.

Example 1. Consider a pair of time series (*Rain Precipitation (RP)*, *Injured Pedestrian (IP)*), and a time window $w_{RP,IP+30} = ([9.00\ am, 10.00\ am], 30\ mins])$. The window $w_{RP,IP+30}$ contains events of *RP* during $[9.00\ am, 10.00\ am]$, and maps them to events of *IP* occurring 30 minutes later, i.e., during the interval $[9.30\ am, 10.30\ am]$.

Fig. 1 illustrates 3 different scenarios of time window on (X_T, Y_T). Here, $w_1 = ([t_{s_1}, t_{e_1}], \tau_1 = 0])$ has no time delay, thus starts and ends at the same time on X_T and Y_T. Instead, the window $w_2 = ([t_{s_2}, t_{e_2}], \tau_2 > 0])$ has a time delay $\tau_2 > 0$, thus Y_T is shifted from X_T. The window $w_3 = ([t_{s_3}, t_{e_3}], \tau_3 < 0])$ has $\tau_3 < 0$, thus X_T is shifted from Y_T, similarly for w_4.

Definition 4.6 *(Mutual information of a window)* Let (X_T, Y_T) be a pair of time series, and $w_{X,Y+\tau}$ be a *time delay window* of (X_T, Y_T). The MI between X_T and Y_T within $w_{X,Y+\tau}$ is estimated using the *KSG* estimator as:

$$I_{w_{X,Y+\tau}} = I(X_w; Y_w) = \psi(k) - \frac{1}{k} - \frac{1}{m} \sum_{\substack{x_i \in X_w \\ y_j \in Y_w}} [\psi(n_{x_i}) + \psi(n_{y_j})] + \psi(m) \quad (3)$$

where m is the size of $w_{X,Y+\tau}$, and n_{x_i} and n_{y_j} are the number of data points falling within the k^{th}-nearest distances in each dimension d_x and d_y of point $(x_i, y_j) \in (X_w, Y_w)$.

Fig. 2 illustrates how to compute the MI of a window using *KSG* estimation. Consider a window $w_{X,Y+\tau}$ contains 7 data points

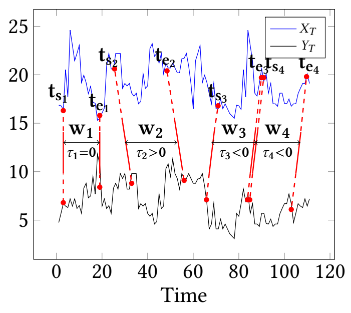


Figure 1: Time windows

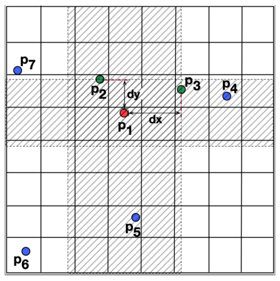


Figure 2: MI computation

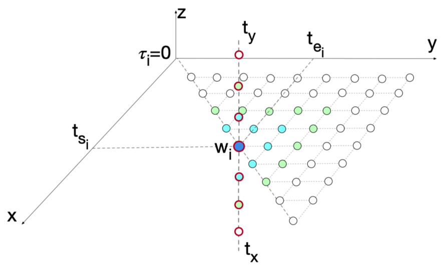


Figure 3: Search space of TYCOS

$\{p_1=(x_1,y_1),...,p_7=(x_7,y_7)\}$, with their positions projected into a two dimensional grid as in Fig. 2. Without loss of generality, we assume $\tau=0$ (events in X_w and Y_w occur at the same time step), the nearest neighbor parameter $k=2$, and the distance metric between neighbors is the *maximum norm*[1]. Under this setting, the 2 nearest neighbors of p_1 are p_2 and p_3 (in green), and the nearest distances from p_1 to its nearest neighbors in each dimension are dx and dy. The nearest distances allow the *KSG* estimator to form the marginal regions (in gray shade), from which the marginal counts are computed. In this case for point p_1, the marginal counts are n_{x_1}=3 (including p_2,p_3,p_5), and n_{y_1}=3 (including p_2,p_3,p_4). Similar steps are applied to other data points from p_2 to p_7. Finally, the marginal counts n_{x_i}, n_{y_i} are inserted into Eq. 3 to compute the MI of $w_{X,Y+\tau}$.

Definition 4.7 *(Correlated time delay window)* Let $w_{X,Y+\tau}$ be a *time delay window* of (X_T,Y_T), and $I_{w_{X,Y+\tau}}$ be the MI of $w_{X,Y+\tau}$. The two time series X_T and Y_T are said to be correlated within $w_{X,Y+\tau}$ iff $I_{w_{X,Y+\tau}} \geq \sigma$ where $\sigma > 0$ is a pre-defined correlation threshold.

Problem Statement: *Time delaY COrrelation Search (TYCOS). Let (X_T,Y_T) be a pair of time series measured during the time interval T, and $w_{X,Y+\tau}$ be a time delay window of (X_T,Y_T). Then TYCOS aims to find a set S of $w_{X,Y+\tau}$ such that $s_{\min} \leq |w_{X,Y+\tau}| \leq s_{\max} \wedge \tau \leq td_{\max} \wedge I_{w_{X,Y+\tau}} \geq \sigma \wedge \forall w_i, w_j \in S : w_i \nsubseteq w_j \wedge w_j \nsubseteq w_i$, where $|w_{X,Y+\tau}|$ denotes the size of $w_{X,Y+\tau}$, $s_{\min}$ and $s_{\max}$ are the minimum and maximum sizes that a window can have, $td_{\max}$ is the maximum time delay, and σ is the pre-defined correlation threshold.*

The goal of TYCOS is to find a set S of non-overlapping time delay windows that respect size and time delay constraints, and have their MI satisfying the pre-defined correlation threshold. As the size of each window is restricted in the range $[s_{\min}, s_{\max}]$, this implies that if correlations exist in the pair (X_T,Y_T), they will last at least for length $s_{\min}$, and at most for length $s_{\max}$. This assumption is meaningful especially when working with real datasets. For example, when searching for weather-related correlations, one could assume that correlations can only last for at most *several* weeks which correspond to the longest duration of a weather event. Furthermore, the time delay of a window is also assumed to be bounded by a maximum value $td_{\max}$ that represents the longest shifting duration between two time series. The value of $td_{\max}$ can be used to prevent spurious correlations. For example, a heavy rain cannot have an impact on the number of injured pedestrians one year later. Setting $td_{\max}$ value, for now, will rely on the expert's domain knowledge.

[1]L_∞: $d(p_i,p_j) = \| (d_x, d_y) \|_{\max} = \max(\| x_i - x_j \|, \| y_i - y_j \|)$

5 TYCOS: TIME DELAY CORRELATION SEARCH

5.1 Search Space and Time Complexity

The search space of TYCOS is represented by the number of *feasible* windows (*feasible* windows are those that respect the size and time delay constraints), illustrated in Fig. 3. Here, the x−axis represents the start time t_s, the y−axis represents the end time t_e, and the z−axis represents the time delay τ of a window. Each point in this 3−dimensional grid represents a window w_i identified by its start time index t_{s_i}, end time index t_{e_i}, time delay τ_i, and its MI I_{w_i}. Since the start time index t_{s_i} always has to be smaller than the end time index t_{e_i}, the *feasible* windows will reside only in half of the grid (Fig. 3).

LEMMA 1. *Let (X_T,Y_T) be a pair of two time series of length n, and $s_{\min}$, $s_{\max}$ be the minimum and maximum sizes of a window, $td_{\max}$ be the maximum time delay between X_T and Y_T. Then the size of TYCOS search space is $O(n^3)$.*

PROOF. To find all feasible windows, initially, a *Brute Force* search can start with a window $w_0 = ([t_{s_0}, t_{e_0}], \tau_0)$ at the minimum size $s_{\min}$ and the initial time delay $\tau_0 = 0$. For each start index t_{s_i}, it extends the end index t_{e_i}, creating a new and larger window $w_i^{'}$ until it reaches the maximum size $s_{\max}$. With one start index t_{s_i}, the number of windows created by extending the end index is: $s_{\max} - s_{\min} + 1$.

Furthermore, each window w_i has the possibility to shift $(2 * td_{\max})$ times (corresponding to *negative* and *positive* values of τ), creating $(2 * td_{\max})$ possible time delay windows. Finally, there are $(n - s_{\min} + 1)$ possible start indices t_{s_i}. Thus, the total number of feasible windows of TYCOS is:

$$(n - s_{\min} + 1) * (s_{\max} - s_{\min} + 1) * 2 * td_{\max} \sim n^3 \qquad (4)$$

if $s_{\max} \to n \wedge td_{\max} \to n \wedge s_{\min} \ll n$. □

LEMMA 2. *Let n be the length of (X_T,Y_T), and m be the average size of a window, then the worst-case time complexity of a Brute Force search for TYCOS on (X_T,Y_T) is $O(n^3m^2)$.*

PROOF. The complexity of TYCOS depends on the number of windows it needs to evaluate, and the time required to compute the MI of each window. The number of windows to be evaluated for TYCOS is $O(n^3)$, according to Lemma 1.

On the other hand, the MI is computed using the *KSG* estimator, in which the most expensive operator is the k-nearest neighbor (kNN) search. Therefore, the complexity of MI computation depends on the complexity of kNN search. Consider a window w_i of size m. A basic kNN algorithm applied to w_i will require $O(kdm)$ to find k nearest neighbors for one sample (d is the data dimensionality), and thus $O(kdm^2) \sim O(m^2)$ (if k and d are significantly smaller than m) for all samples in w_i [12]. Hence, the worst-case time complexity of a Brute Force search for TYCOS is $O(n^3m^2)$. However, if a more efficient data structure is used, such as k-d tree [5] or grid-based structure (for low dimensional data) [30], the expected-case *kNN* complexity is $O(kdm \log m) \sim O(m \log m)$, and thus, the expected-case Brute Force complexity is $O(n^3 m \log m)$. □

5.2 TYCOS$_{\textbf{LAHC}}$: A LAHC Based Approach

The time complexity of a Brute Force approach for TYCOS is computationally prohibitive in practice. For example, a pair of time series with *n=9,000* data points, $s_{\max} = 400$, $s_{\min} = 20$, and $td_{\max} = 20$ will create *136,870,440* windows. Our Brute Force search implemented in *C++* and run on a standard PC will take more than 12 hours to process all generated windows. In the next section, we propose a heuristic search method using LAHC to speed up the TYCOS process.

To improve the TYCOS process, we look at two angles for improvements: (1) reducing the search space, and (2) optimizing the MI computation. To reduce the search space, we adopt LAHC, and propose a novel MI-based theory to prune unpromising windows. To optimize the MI computation, we design efficient data structures so that we can reuse the computation across windows. The following sections discuss the intuition behind our approach and detail how LAHC can be applied to TYCOS. The MI-based theory and its applicability to TYCOS are introduced in Section 6. The efficient MI computation is described in Section 7.

5.2.1 The choice of LAHC. To explain the intuition behind the LAHC-based method, consider Fig. 4 that illustrates the MI value fluctuation across windows. Here, the $y-$axis represents the MI values of corresponding time windows on the $x-$axis. Given the correlation threshold σ (red line), the three windows which correspond to the three locally maximal points (in red) indicate highly correlated areas, and can be found by identifying the three peak (red) points in the search space. Since LAHC guarantees to achieve local optimal solutions, it becomes an ideal foundation for solving the TYCOS problem.

5.2.2 Apply LAHC to TYCOS. Indeed, finding correlations in time series means to find windows that maximize the MI. Thus, we consider the problem of searching for time delay correlations using LAHC, namely TYCOS$_{\textbf{LAHC}}$ (or TYCOS$_{\textbf{L}}$ in short), as a *maximization* problem. Specifically, the target function of TYCOS$_{\textbf{L}}$ is a *maximize* function, and our goal is to find windows where their MIs are locally maximal values that satisfy σ.

a) Search space navigation. We first illustrate how LAHC navigates through the search space of TYCOS in Fig. 5, with the three axes being the start time ($x-$axis), the end time ($y-$axis) and the time delay ($z-$axis) of a window. Assume $w_i = ([t_{s_i}, t_{e_i}], \tau_i)$ is the window where the search is currently at. Starting from w_i, if TYCOS$_{\textbf{L}}$ follows a rightwards trajectory on the $y-$axis, it moves the end time t_{e_i} of w_i forward in time, thus enlarging the window size. If it follows a leftwards trajectory on the $y-$axis, it moves the end time t_{e_i} backward in time, thus reducing the window size. Similarly, moving along the $x-$axis, TYCOS$_{\textbf{L}}$ can reduce or increase the start time t_{s_i}, therefore, extending or narrowing the size of w_i accordingly. On the $z-$axis, following the t_x direction, TYCOS$_{\textbf{L}}$ increases the shifting time of X_T w.r.t. Y_T. Following the t_y direction, TYCOS$_{\textbf{L}}$ will shift Y_T further from X_T. In both cases, it increases the time delay but keeps the same window size.

While exploring the search space in multiple directions, TYCOS$_{\textbf{L}}$ creates different windows by adjusting the indices of the current window. The generated windows are grouped into the same neighborhood if they share similar indices. The *neighborhood* concept is defined below.

Definition 5.1 *(δ-neighbor)* Let $w = ([t_s, t_e], \tau)$ be a window of (X_T, Y_T), and assume (X_T, Y_T) has length *n*. A window $w' = ([t_s', t_e'], \tau')$ is a *δ-neighbor* of w if $t_s' = t_s \pm \delta \lor t_e' = t_e \pm \delta \lor \tau' = \tau \pm \delta$, where δ is a pre-defined moving step such that $1 \le \delta \le n \land s_{\min} \le \left|w'\right| \le s_{\max} \land \tau' \le td_{\max}$.

A δ-neighbor window w' has at least one of its indices (i.e., $t_{s'}$, or $t_{e'}$, or τ') differing a δ step from the indices of w.

Definition 5.2 *(δ-neighborhood)* Let $w = ([t_s, t_e], \tau)$ be a window of (X_T, Y_T). A *δ-neighborhood* of w, denoted as N_δ, is formed by all *δ-neighbors* $w' = ([t_s', t_e'], \tau')$ of w.

The *neighborhood* concept is illustrated in Fig. 5. Consider the window w_i (in blue). The nearest $\delta-$neighborhood of w_i, called the $1-$neighborhood N_1, is the area formed by the 26 windows in blue color w_i^1 where $i = 1, ..., 26$. Each window in this neighborhood differs from w_i by *one* δ step, either by its start index, or its end index, or its time delay, or the combinations of them, or all. Going further, another neighborhood of w_i, called the $2-$neighborhood N_2, is the 50 windows in green color area. Each $\delta-$*neighborhood* forms an area where TYCOS$_{\textbf{L}}$ will iteratively look for potential candidates to improve w_i.

b) TYCOS$_{\textbf{L}}$ algorithm. We provide the outline of TYCOS$_{\textbf{L}}$ in Algorithm 1, and explain it in the following.

Consider a time series pair (X_T, Y_T), and let I_w be the target function to be maximized. To improve I_w, TYCOS$_{\textbf{L}}$ will start with an initial feasible solution, and explores its neighborhood to look for better solutions. Let $w = w_0$ where$|w_0| = s_{\min} \land \tau_0 = 0$ be an initial solution (Alg. 1, line 2). The goodness of w_0 is evaluated by computing $I(w_0)$ (line 3). Starting from w_0, TYCOS$_{\textbf{L}}$ will first explore its nearest neighborhood N_1, and search for a better solution than w_0 in this area. To do that, it creates all δ_1-neighbors of w_0 to form N_1. Then for each $w' \in N_1$, it computes $I(w')$ and selects the best neighbor *bestnb* which has the highest MI (lines 5 – 8). Next, it determines whether *bestnb* is a better solution than the current one w using the following policies:

- (*Policy 1*) If: $I_{bestnb} > I_w$ or $I_{bestnb} > I_{w_h}$ where $w_h \in L_h$, then: *bestnb* is a better solution than w and thus, w is replaced by *bestnb* (lines 10 – 12).
- (*Policy 2*) If: $I_{bestnb} \le I_w$ and $I_{bestnb} \le I_{w_h}$, then there is no better solution in the considered neighborhood, thus, no improvement can be made (lines 14 – 15).

In *Policy 1*, a better solution is found, the search moves to this new solution $w = bestnb$, and repeats the neighborhood exploration process on the new w. Note that since LAHC also uses a historical value w_h to justify a potential candidate solution, the newly selected solution *bestnb* might be better than w_h, but not necessarily better than the current solution w. This type of approximation creates some "randomness" in the search, which is helpful, for example, when the search needs to escape from plateau situations, i.e., when the search space is flat. In *Policy 2*, no better solution is found, then the *stopping conditions* are checked. If the *stopping conditions* are not yet satisfied, the search continues exploring larger neighborhoods. Otherwise, it stops and the value I_w at the stopping point is the locally maximal value. Finally, w is accepted and inserted into the result set S if $I_w \ge \sigma$ (lines 19 – 20).

When the *stopping conditions* are satisfied and TYCOS$_{\textbf{L}}$ stops, the time series pair might not be scanned entirely. In that case, TYCOS$_{\textbf{L}}$ restarts again on the remaining part of the data, looking

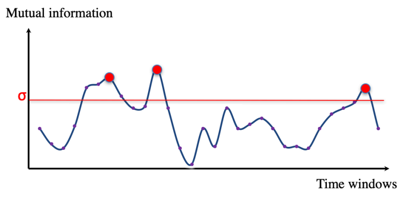


Figure 4: MI fluctuation

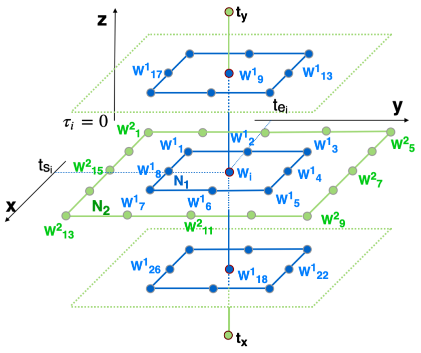


Figure 5: Explore neighborhood

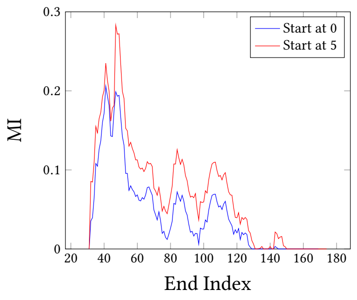


Figure 6: Changes of MI

for new local optimal solutions, until the entire time series are searched (line 21).

Stopping conditions: Ideally, TYCOS$_L$ will stop immediately when no better solution can be found in the considered neighborhood. However, to avoid situations where the occurrence of a temporary setback stops the search too early, an idle period is used to measure the number of non-improvements observed. The search will stop when the pre-defined max idle period $T_{maxIdle}$ is reached (line 4).

Initial solution: The initial window w_0 can be at the beginning, or at an arbitrary position in the time series. A good initial solution can help reach satisfying solutions faster, and vice versa. In Section 6, we rely on an MI-based theory to select a good initial solution, leading to a more promising exploration for the search.

The history list L_h: TYCOS$_L$ maintains a history L_h of the most recently accepted solutions and uses it to justify the goodness of a potential candidate. In our implementation, TYCOS$_L$ follows the *random* policy when selecting and updating an item in the history (line 9 and 16 – 18).

Algorithm 1 TYCOS$_L$: LAHC for TYCOS

```
    Input: $(X_T, Y_T)$: pair of time series
    Params: $\sigma, \varepsilon, s_{\min}, s_{\max}, td_{\max}$
    Output: $S$: a set of non-overlapping windows whose MI $\geq \sigma$
 1: while $(X_T, Y_T)$ is not scanned entirely do
 2:     Initial solution $w := w_0$ with $|w_0| = s_{\min} \wedge \tau_0 = 0$
 3:     Compute $I(w_0)$                        ▹ Evaluate the goodness of $w_0$
 4:     while $t_{idle} < T_{maxIdle}$ do
 5:         $N := Neighbors(w)$                 ▹ Identify the neighbors of $w$
 6:         for $w' \in N$ do
 7:             Compute $I(w')$                 ▹ Evaluate the goodness of $w'$
 8:         $bestnb := BestNeighbor(N)$         ▹ Select best neighbor in $N$
 9:         $w_h := random.get(L_h)$            ▹ Randomly select from $L_h$
10:         if $I_{bestnb} > I_{w_h}$ or $I_{bestnb} > I_w$ then
11:             $w := bestnb$                   ▹ Accept the candidate
12:             $t_{idle} := 0$                 ▹ Reset the idle time
13:         else
14:             $w := w$                        ▹ Reject the candidate
15:             $t_{idle} := t_{idle} + 1$      ▹ Increase the idle time
16:         if $I_w > I_{w_h}$ then             ▹ Update the history list
17:             $w_h := w$
18:             $I_{w_h} := I_w$
19:     if $I_w \geq \sigma$ then
20:         Insert $w$ to $S$
21:     $TYCOS_L(X'_T, Y'_T)$                   ▹ Restart TYCOS$_L$
22: return $S$
```

6 NOVEL NOISE THEORY TO IMPROVE TYCOS

6.1 Noise Identification

When TYCOS$_L$ performs the neighborhood exploration, conceptually, it is performing a depth-first search. Each neighbor window is considered as an expansion to a deeper level of the search tree, and the expansion only stops when the stopping conditions are met. During the expansion, some part of the data might be revisited multiple times, which can lead to redundant computation. To reduce potential redundancy, we explore several MI properties to establish principles that can help narrow the search space. Specifically, we seek the answer for the following research question: *"When should a certain part of data be completely removed from the search?".*

This research question concerns the removal of a data partition from the search without affecting its final outcomes. This is due to the fact that by repeatedly expanding the neighborhood, TYCOS$_L$ revisits a data partition multiple times, and in some cases, a particular data partition might be irrelevant to the search's objectives, i.e., including this particular data partition in the search process does not lead to promising results. If that data partition can be identified, it should not be included in future explorations of the search. The following example demonstrates this situation.

Consider the window w_i (blue point), and its neighborhood N_1 and N_2 in Fig. 5. In N_1 and N_2, neighbors that belong to the same exploration direction might contain overlapping data. For instance, $w^1_4 \in N_1$ is expanded from w_i by extending its end index by a δ_1 step, while $w^2_7 \in N_2$ is an extension of w^1_4 by enlarging w_i's end index a δ_2 step ($\delta_2 > \delta_1$). The process of extending one window to another window results in overlapping data that will be revisited multiple times in different exploration iterations.

On the other hand, consider Fig. 6 that plots the MI values of a time series pair with different start indices: the blue line starts at index 0, the red line starts at index 5, i.e., the data from 0 to 5 are not considered in the red line. From Fig. 6, it can be seen that by excluding the data range $[0-5]$ from the search, the MI values of subsequent windows increase and are larger than when including the considered range. This implies that the data range $[0-5]$ provides no information about the dependency between the times series pair, and thus can be considered as "noise" and eliminated from future exploration.

The above research question thus can be answered by establishing a "noise" identification principle. To do that, we rely on the following theorem to understand when a data partition can be considered as "noise" and should be eliminated.

Definition 6.1 (*Mixture distribution*) Let X and U be discrete random variables with the corresponding p.m.fs $p_X(x)$, $p_U(u)$. Let Z be a new random variable which is drawn from the same distribution as X with probability θ and from the same distribution as U with probability $1-\theta$ for a given $\theta \in [0,1]$. Then Z is said to have a mixture distribution between $p_X(x)$ and $p_U(u)$ with probability θ and is written as $Z = X \odot_\theta U$.

Theorem 6.1. *Let X, Y, U, V be discrete random variables and $p_X(x), p_Y(y), p_U(u)$, and $p_V(v)$ be their corresponding p.m.fs. Let*

$Z = X \odot_\theta U$ and $W = Y \odot_\eta V$ where $\odot$ denotes the mixture of two variables. Assume that, except for X and Y, other variables are pair-wise independent, i.e., $(U \perp V) \wedge (X \perp U) \wedge (X \perp V) \wedge (Y \perp U) \wedge (Y \perp V)$. Then $I(X;Y) \geq I(Z;W)$.

Proof. Z and W are the two mixed variables: $Z = X \odot_\theta U$ and $W = Y \odot_\eta V$. Then, for a value of x drawn according to $p_X(x)$ and a value of u drawn according to $p_U(u)$, we can write the probabilities for Z as follows:

$$p_Z(x) = P(Z = X)p_X(x) = \theta p_X(x) \tag{5}$$

$$p_Z(u) = P(Z = U)p_U(u) = (1-\theta)p_U(u) \tag{6}$$

Similarly, for $y \sim p_Y(y)$ and $v \sim p_V(v)$, we have:

$$p_W(y) = P(W = Y)p_Y(y) = \eta p_Y(y) \tag{7}$$

$$p_W(v) = P(W = V)p_V(v) = (1-\eta)p_V(v) \tag{8}$$

Then, we can write the following joint probabilities:

$$p_{Z,W}(x,y) = \theta\eta p_{X,Y}(x,y) \tag{9}$$

$$p_{Z,W}(x,v) = \theta(1-\eta)p_{X,V}(x,v) \tag{10}$$

$$p_{Z,W}(u,y) = (1-\theta)\eta p_{U,Y}(u,y) \tag{11}$$

$$p_{Z,W}(u,v) = (1-\theta)(1-\eta)p_{U,V}(u,v) \tag{12}$$

We have the MI between X and Y as

$$I(X;Y) = \sum_y \sum_x p_{X,Y}(x,y) \log \frac{p_{X,Y}(x,y)}{p_X(x)p_Y(y)} \tag{13}$$

And the MI between Z and W as

$$I(Z;W) = \sum_w \sum_z p_{Z,W}(z,w) \log \frac{p_{Z,W}(z,w)}{p_Z(z)p_W(w)} \tag{14}$$

Since Z can take the values in $\mathcal{R}_X$ if z is drawn from X, and in $\mathcal{R}_U$ if z is drawn from U (similarly for W), then from Eq. (14), it follows that:

$$\begin{aligned}
I(Z;W) = &\sum_{w\in\mathcal{R}_Y}\sum_{z\in\mathcal{R}_X} p_{Z,W}(x,y)\log\frac{p_{Z,W}(x,y)}{p_Z(x)p_W(y)} \\
&+ \sum_{w\in\mathcal{R}_Y}\sum_{z\in\mathcal{R}_U} p_{Z,W}(u,y)\log\frac{p_{Z,W}(u,y)}{p_Z(u)p_W(y)} \\
&+ \sum_{w\in\mathcal{R}_V}\sum_{z\in\mathcal{R}_X} p_{Z,W}(x,v)\log\frac{p_{Z,W}(x,v)}{p_Z(x)p_W(v)} \\
&+ \sum_{w\in\mathcal{R}_V}\sum_{z\in\mathcal{R}_U} p_{Z,W}(u,v)\log\frac{p_{Z,W}(u,v)}{p_Z(u)p_W(v)} \\
= &\sum_{y\in\mathcal{R}_Y}\sum_{x\in\mathcal{R}_X} \theta\eta p_{X,Y}(x,y)\log\frac{\theta\eta p_{X,Y}(x,y)}{\theta p_X(x)\eta p_Y(y)} \\
&+ \sum_{y\in\mathcal{R}_Y}\sum_{u\in\mathcal{R}_U} (1-\theta)\eta p_{U,Y}(u,y)\log\frac{(1-\theta)\eta p_{U,Y}(u,y)}{(1-\theta)p_U(u)\eta p_Y(y)} \\
&+ \sum_{v\in\mathcal{R}_V}\sum_{x\in\mathcal{R}_X} \theta(1-\eta) p_{X,V}(x,v)\log\frac{\theta(1-\eta) p_{X,V}(x,v)}{\theta p_X(x)(1-\eta) p_V(v)} \\
&+ \sum_{v\in\mathcal{R}_V}\sum_{u\in\mathcal{R}_U} (1-\theta)(1-\eta) p_{U,V}(u,v)\log\frac{(1-\theta)(1-\eta) p_{U,V}(u,v)}{(1-\theta)p_U(u)(1-\eta) p_V(v)}
\end{aligned} \tag{15}$$

Eq. (15) can be rewritten as

$$\begin{aligned}
I(Z;W) = &\ \theta\eta I(X;Y) + (1-\theta)\eta I(U;Y) \\
&+ \theta(1-\eta)I(X;V) + (1-\theta)(1-\eta)I(U;V)
\end{aligned} \tag{16}$$

Since we assume

$$(U \perp V) \wedge (X \perp U) \wedge (X \perp V) \wedge (Y \perp U) \wedge (Y \perp V)$$

This leads to

$$I(U;Y) = 0 \wedge I(X;V) = 0 \wedge I(U;V) = 0$$

Thus, Eq. (16) becomes

$$I(Z;W) = \theta\eta I(X;Y) \tag{17}$$

where $\theta \leq 1$ and $\eta \leq 1$, which leads to

$$I(X;Y) \geq I(Z;W)$$

□

Theorem 6.1 says that, if U and V are independent from each other and from X and Y, then adding them to X and Y will bring more uncertainty to (X, Y), in other words, they reduce the shared information $I(X;Y)$.

Definition 6.2 *(Consecutive windows)* Let $w_{X,Y+\tau} = ([t_s, t_e], \tau)$ and $w'_{X,Y+\tau'} = ([t_{s'}, t_{e'}], \tau')$ be the two time delay windows of (X_T, Y_T). Then $w_{X,Y+\tau}$ and $w'_{X,Y+\tau'}$ are consecutive iff $t_{s'} = t_e + 1 \wedge \tau = \tau'$.

From Definition 6.2, $w_{X,Y+\tau}$ and $w'_{X,Y+\tau'}$ are *consecutive* if they are next to each other and have the same shifting time, i.e., $w'_{X,Y+\tau'}$ starts right after the end time of $w_{X,Y+\tau}$. Since $w'_{X,Y+\tau'}$ follows $w_{X,Y+\tau}$, terminologically, we call $w_{X,Y+\tau}$ the *followed* window, and $w'_{X,Y+\tau'}$ the *following* window. Examples of *consecutive* windows are w_3 and w_4 in Fig. 1.

Definition 6.3 *(Concatenation operation $\odot$ of consecutive windows)* Let $w_{X,Y+\tau} = ([t_s, t_e], \tau)$ and $w'_{X,Y+\tau'} = ([t_{s'}, t_{e'}], \tau')$ be two consecutive windows of (X_T, Y_T). The concatenation between $w_{X,Y+\tau}$ and $w'_{X,Y+\tau'}$ is defined as: $w''_{X,Y+\tau} = w_{X,Y+\tau} \odot w'_{X,Y+\tau'} = ([t_s, t'_e], \tau)$. The concatenation operation joins two consecutive windows $w_{X,Y+\tau}$ and $w'_{X,Y+\tau'}$ into one bigger window $w''_{X,Y+\tau}$ which has its start time being the start time of the *followed* window, and its end time being the end time of the *following* window.

Based on the result of Theorem 6.1 and Definitions 6.2, 6.3, we define *noise* as follows.

Definition 6.4 *(Noise)* Let $w_{X,Y+\tau}$, $w'_{X,Y+\tau'}$ be two consecutive windows of (X_T, Y_T), $w''_{X,Y+\tau} = w_{X,Y+\tau} \odot w'_{X,Y+\tau'}$ be their concatenating window, and ε ($0 \leq \varepsilon < \sigma$) be a real number representing the *noise threshold*. Assume that $I_{w_{X,Y+\tau}} > 0$. Then $w'_{X,Y+\tau'}$ is called *noise* w.r.t. $w_{X,Y+\tau}$ iff $I_{w'_{X,Y+\tau'}} < \varepsilon \wedge I_{w''_{X,Y+\tau}} < I_{w_{X,Y+\tau}}$.

The noise principle says that if the MI of the *following* window $w'_{X,Y+\tau'}$ is less than the noise threshold, and the MI of the *followed* window $w_{X,Y+\tau}$ decreases after the concatenation, then the *following* window is *noise* w.r.t. the *followed* window.

6.2 Applying Noise Theory to Prune the Search Space

Based on the noise identification principle, we propose two improvements to be made in TYCOS$_\mathbf{L}$. We name TYCOS$_\mathbf{L}$ with noise theory applied as TYCOS$_\mathbf{LN}$.

6.2.1 *Initial noise pruning.*

Previously, we said that TYCOS$_\mathbf{L}$ can start out at the beginning of, or at an arbitrary location in the time series. This, however, can lead the search to an unpromising exploration area. For example, if the search starts out at the valleys in Fig. 4, it might take longer time to reach the top of the hill than if it starts somewhere on the edges. To avoid the

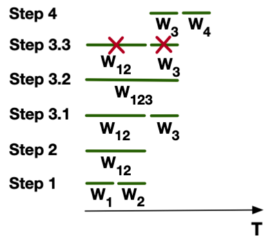


Figure 7: Initial window

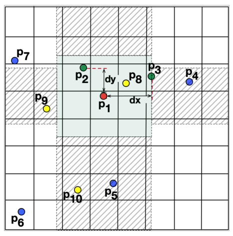


Figure 8: Efficient MI computation

"valley-trapped" situations, we use the noise theory to find a good starting point. The search is at a good starting point if the initial solution w_0 has $I_{w_0} \geq \varepsilon$ (the *noise threshold*). To find such a point, we first divide the time series into non-overlapping windows of the minimal size $s_{\min}$ with no time delay ($\tau = 0$), and then hierarchically combine them to form larger, and hopefully better windows. The combination stops when it finds a window w that has $I_w \geq \varepsilon$. Fig. 7 demonstrates this procedure.

In *Step 1*, initially the search starts with two minimal consecutive and non-overlapping windows w_1, w_2, and evaluates their goodness by computing I_{w_1}, I_{w_2}. In *Step 2*, it combines the *two* windows into a bigger one w_{12}, and computes $I_{w_{12}}$. Next, it compares the goodness of the 3 windows, and select the one that has the highest MI. Assuming that $\{I_{w_1}, I_{w_2}\} \leq I_{w_{12}} < \varepsilon$, then w_{12} is the one selected among the *three*. Since $I_{w_{12}}$ is still less than ε, it moves to *Step 3.1*, where a next minimal window w_3 is evaluated both separately (by computing I_{w_3}), and together with w_{12} (by computing $I_{w_{123}}$).

Assume that $I_{w_3} < \varepsilon$, and that by combining w_3 to w_{12}, it reduces the MI $I_{w_{12}}$, i.e., $I_{w_{123}} < I_{w_{12}} < \varepsilon$. According to Theorem 6.1, we can conclude that w_3 is noise w.r.t. w_{12}. Thus, the combination w_{123} does not lead to a promising result. The next window to be considered is w_4. However, w_{12} cannot be combined with w_4 without the presence of w_3, which we know is noise of w_{12}. Thus, the combination w_{1234} should not be formed, and w_{12} should also be eliminated from future consideration (*Step 3.3*). Next, in *Step 4*, w_3 is evaluated again in combination with w_4, and the procedure is repeated until it can find a window that has MI > ε. Once the starting point is determined, TYCOS**LN** begins its neighborhood exploration as described in Section 5.2.

6.2.2 Subsequent noise detection. The noise identification principle is also beneficial during the neighborhood exploration. We explain its applicability in Fig. 5. Assume w_i is the current window and w_4^1, w_7^2 are its neighbors when moving along the $y-$axis. In the first exploration, the neighbor w_4^1 is considered. Since w_4^1 is created by extending the end index of w_i by a δ_1-step, we have: $w_4^1 = w_i \odot w_{\delta_1}$ where w_{δ_1} is the extension to be concatenated with w_i. Assume that by applying our noise theory to w_i, w_{δ_1}, and w_4^1, we conclude that w_{δ_1} is noise w.r.t. w_i. In this case, it is not promising to further explore the neighborhoods of w_i along the $y-$axis in that direction. In the next exploration, TYCOS**LN** will omit w_7^2, as well as the entire forward direction along the $y-$axis.

Ensuring the completeness of TYCOS$_{\mathbf{LN}}$*:* When TYCOS**LN** stops at a locally optimal solution, it has followed the best path and explored to the deepest level of the current tree. This, however, does not guarantee that is the only path. In fact, we want to find the set of all windows that are above the correlation threshold. Thus, to ensure the completeness of the search, TYCOS**LN** is designed recursively so that once it stops at the locally optimal solution, it goes back to the previously found starting point and continues exploring other paths to find all feasible solutions.

Algorithm 2 reflects on how the noise theory is applied in TYCOS. In line 2, the noise theory is applied to find a good starting point. During the neighborhood exploration, the theory is applied again to prune the search space (line 5).

Algorithm 2 TYCOS**LN**: Apply noise theory to TYCOS**L**

```
    Input: $(X_T, Y_T)$: pair of time series
    Params: $\sigma$, $\varepsilon$, $s_{\min}$, $s_{\max}$, $td_{\max}$
    Output: $S$: a set of non-overlapping windows whose MI $\geq \sigma$
 1: while $(X_T, Y_T)$ is not scanned entirely do
 2:     Initial solution $w := InitialNoisePruning((X_T, Y_T), \varepsilon)$
 3:     Compute $I(w)$    ▷ Evaluate the goodness of the initial solution
 4:     while $t_{idle} < T_{maxIdle}$ do
 5:         $N := SubsequentNoiseDetection(w, \tau)$  ▷ Apply Theorem 6.1 to identify promising neighbors of $w$
 6:         $w := EvaluateCandidateSolution(w, N)$  ▷ Follow the steps 8-18 in Algorithm 1 to improve $w$
 7:     if $I_w \geq \sigma$ then
 8:         Insert $w$ to $S$
 9:     $TYCOS_{\mathbf{LN}}(X_T', Y_T')$                ▷ Restart TYCOS$_{\mathbf{LN}}$
10: return $S$
```

6.3 Setting the Correlation Threshold

6.3.1 Using normalized MI. Since MI is a measure of total dependence between variables, its magnitude represents the strength of the correlation. As the MI value is always non-negative, its lower bound is 0. However, the MI's upper bound varies and thus, it is difficult to set an appropriate correlation threshold using MI magnitude when data characteristics and their relationships are unknown. To overcome this challenge, we propose a robust method to set the correlation threshold based on the *normalized MI*:

$$0 \leq \tilde{I}_w = \frac{I_w}{H_w} \leq 1 \tag{18}$$

where I_w is the MI and H_w is the entropy of the window w.

In Eq. (18), the window entropy H_w represents the amount of uncertainty contained in the window w. Thus, $\tilde{I}_w$ represents the fraction of the window's uncertainty reduced by the shared information I_w. The larger $\tilde{I}_w$, the more information is shared between the window's variables, and thus the stronger correlation. The normalized MI $\tilde{I}_w$ is always scaled between [0, 1], and thus provides an easier way for users to set the threshold σ.

6.3.2 Using top-K filtering. Top-K maintains a list of K (K is a predefined parameter) windows that have the highest MI up to the current point. The top-K list represents the top correlated time-series windows, and can be used to set the value of σ. In this top-K filtering approach, σ starts with the MI value of the initial window w_0. As the search proceeds, the top-K list is filled, and σ gets updated by the minimum MI value in the list. Once the top-K list is full, it will get updated if there is a new window that has MI greater than the current value of σ. The element with the least MI value in the top-K list will be replaced by this new window, and σ is updated accordingly.

7 EFFICIENT MI COMPUTATION

In this section, we discuss the efficient MI computation (based on Eq. (2)) in TYCOS. Due to space limitations, the discussion will be brief and touch only important points.

Recall that while exploring its neighborhood, TYCOS might visit the same data partition multiple times. For example, while evaluating w_4^1 and w_7^2 in Fig. 5, TYCOS will repeatedly revisit w_i because w_4^1 and w_7^2 are extended from w_i. To minimize the redundancy, we design an efficient MI computation method so that computation of overlapping data can be reused across windows.

We observe that neighboring windows in each neighborhood N_i can differ from the current window w_i by only a small data partition w_{δ_i}, where w_{δ_i} is either removed from or added to w_i. For instance, in Fig. 5, w_8^1 differs from w_i by removing a w_{δ_1} data partition from w_i, whereas w_4^1 differs from w_i by adding a w_{δ_1} data partition to w_i. The removal of old data and the addition of new data can introduce different types of changes to the previous computation of w_i. These changes can be either changing the k-nearest neighbors or changing the marginal counts n_x, n_y of existing points. To track those changes, we introduce the *influenced region* and *influenced marginal region* concepts for each data point.

Definition 7.1 *(Influenced region (IR))* An *IR* of point $p_i = (x_i, y_i)$ is a square bounding box $R_i = (l_i, r_i, b_i, t_i)$, where l_i, r_i, b_i, t_i are its left-, right-, bottom-, and top-most indices, respectively, and are computed as $l_i = x_i - d$, $r_i = x_i + d$, $b_i = y_i - d$, $t_i = y_i + d$ where $d = \max(d_x, d_y)$.

Definition 7.2 *(Influenced marginal region (IMR))* The *IMRs* of point p_i are the marginal regions located within the nearest distance d_i in each dimension.

Fig. 8 illustrates these concepts. The *influenced region* of p_0 is the square colored in green, and the *influenced marginal regions* are those with gray shade in either dimension.

Lemma 3. *Given a window w_i and a data point $p \in w_i$, a new point o inserted into w_i will become the new k^{th}-neighbor of p iff o is within IR of p.*

Lemma 4. *Given a window w_i and a data point $p \in w_i$, an existing point o deleted from w_i will change the k nearest points of p iff o is within IR of p.*

Lemma 5. *Given a window w_i and a data point $p \in w_i$, a new point o inserted into w_i will increase the marginal count n_x (or n_y) of p iff o is within IMR_x (or IMR_y) of p.*

Lemma 6. *Given a window w_i and a data point $p \in w_i$, an existing point o deleted from w_i will reduce the marginal count n_x (or n_y) of p iff o is within IMR_x (or IMR_y) of p.*

Proof. Proofs of Lemmas 3, 4, 5, 6 are straightforward, thus omitted. □

Lemmas 3, 4, 5, 6 display unique properties of *IRs* and *IMRs*. An *IR* maintains an area where any point p_j either falling into or being removed from this region will change the k nearest points of p_i. In this case, a new k-nearest neighbors search is required for p_i. Instead, an *IMR* maintains an area where any point p_j either falling into or being removed from it will change the marginal counts of p_i. In this case, the marginalized neighbors of p_i have to be recounted.

Fig. 8 illustrates how changes are introduced and managed. For simplicity, we only discuss cases when new points are added into the previous computation. Changes introduced by removing points can be handled in a similar way. Assume that at time t_1, a new point p_8 is added to the current window and falls into the *IR* of p_1. The addition of p_8 changes the k^{th}-nearest neighbor of p_1, thus, triggers a new nearest neighbor search for p_1. At time t_2, a new point p_9 arrives and falls into the y-marginal influenced region of p_1, for which it will alter the marginal count n_y (but no new k-nearest neighbor search is required in this case). Similarly, a new point p_{10} will increase the marginal count n_x. In these cases, only a recount of n_x or n_y is performed.

As the result of our *efficient MI computation*, for each window, only a minimum search region (containing new points) and a minimum update region (containing points affected by added and removed points) require additional computation. The rest is reused, and thus minimizing the computational cost.

8 EXPERIMENTAL EVALUATION

We evaluate the effectiveness and efficiency of TYCOS using both synthetic and real-world datasets. Effectiveness measures the method qualitatively by assessing the quality of extracted windows, while efficiency measures the method quantitatively in terms of its performance and accuracy.

8.1 Baseline methods

Effectiveness evaluation: TYCOS is compared against four baseline methods. The first baseline is a traditional correlation metric: Pearson Correlation Coefficient (PCC) [23]. The second is the Fast Subsequence Search (MASS) algorithm [25], often used for subsequences matching in time series. The third is MatrixProfile [31], considered to be the state of the art method for similarity join between time series. The final baseline is the Adaptive Mutual Information-based Correlation (AMIC) [17] framework that follows a top-down approach to search for multi-scale temporal correlations in big time series.

Efficiency evaluation: TYCOS runtime is compared against the Brute Force and MatrixProfile (which uses different window lengths) methods. In addition, different variants of TYCOS, including LAHC-based TYCOS ($\text{TYCOS}_\mathbf{L}$), $\text{TYCOS}_\mathbf{L}$ with noise theory applied ($\text{TYCOS}_\mathbf{LN}$), $\text{TYCOS}_\mathbf{L}$ with the proposed efficient MI computation ($\text{TYCOS}_\mathbf{LM}$), and $\text{TYCOS}_\mathbf{L}$ with both noise theory and efficient MI computation ($\text{TYCOS}_\mathbf{LMN}$), are compared against each other to illustrate the effectiveness of the proposed noise theory and MI computation technique. We do not compare AMIC against TYCOS quantitatively, however, as AMIC does not consider time delay correlations, and thus, has a different search space. PCC and MASS are also not considered for efficiency evaluation because they lack mechanisms to automatically search for correlated windows.

8.2 Parameter setting for TYCOS

TYCOS requires setting 5 parameters: correlation threshold σ, noise threshold ε, minimum window size $s_{\min}$, maximum window size $s_{\max}$, and maximum time delay $td_{\max}$. Among these, σ, $s_{\min}$, $s_{\max}$, and $td_{\max}$ are user parameters, while ε is a hyper parameter.

The value of σ determines the strength of extracted correlations. The larger the σ, the stronger the correlations. In our experiments, we set the value of σ using the normalized MI (scaled between $[0, 1]$) introduced in Section 6.3. On the other hand, the values of $s_{\min}$, $s_{\max}$ and $td_{\max}$ are context dependent and is set based on domain knowledge. That is, given an application domain, it is usually intuitive how small/large a window could be and how long a time shift is possible. For example, when a user analyzes weather related data, he/she might decide that the longest duration of a weather event is *two weeks*, and thus set the size of $s_{\max}$ to *two weeks*. Similarly, a user can set $td_{\max}$ to *24 hours* by assuming that weather events have impacts on other events only within *a day* duration. Table 2 lists the values of σ, $s_{\min}$, $s_{\max}$ and $td_{\max}$ we use in each dataset.

For the hyper parameter ε, we set $\varepsilon = \frac{1}{4}\sigma$ in all experiments. This means that a window whose MI is less than 25% of the correlation threshold is considered unpromising to explore. The ratio $\varepsilon/\sigma = 0.25$ is chosen based on empirical studies we conduct

Table 1: **Identifying different types of correlation relations ($N(\mu, \sigma)$: normal distribution, $u \sim U(0, 1)$: uniform distribution)**

		$td = 0$ (No time delay)					$td = 150$ (With time delay)				
Relation	$y = f(x)$	PCC	MASS	MatrixProfile	AMIC	TYCOS	PCC	MASS	MatrixProfile	AMIC	TYCOS
Independent	$y \sim N(0, 1), x \sim N(3, 5)$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Linear	$y = 2x + u, x \in [0, 10]$	✓	✓	✓	✓	✓	×	×	✓	×	✓
Exp.	$y = 0.01^{x+u}, x \in [-10, 10]$	✓	✓	×	✓	✓	×	×	×	×	✓
Quad.	$y = x^2 + u, x \in [-4, 4]$	×	✓	×	✓	✓	×	×	×	×	✓
Circle	$y = \pm\sqrt{3^2 - x^2 + u}, x \in [-3, 3]$	×	×	×	✓	✓	×	×	×	×	✓
Sine	$y = 2 * sin(x) + u, x \in [0, 10]$	×	×	×	✓	✓	×	×	×	×	✓
Cross	$y_1 = x + u, y_2 = -x + u, x \in [-5, 5]$	×	×	×	✓	✓	×	×	×	×	✓
Quartic	$y = x^4 - 4x^3 + 4x^2 + x + u, x \in [-1, 3]$	×	✓	×	✓	✓	×	×	×	×	✓
Square root	$y = \sqrt{x}, x \in [0, 25]$	×	✓	×	✓	✓	×	×	×	×	✓

Table 2: Parameters setting

Parameter	Energy datasets	Smart city datasets
Correlation threshold σ	0.3	0.2
Minimum window size $s_{\min}$	3 samples $\simeq$ 3 mins	3 samples $\simeq$ 15 mins
Maximum window size $s_{\max}$	10080 samples $\simeq$ 7 days	4032 samples $\simeq$ 14 days
Maximum time delay $td_{\max}$	2880 samples $\simeq$ 2 days	288 samples $\simeq$ 1 day

on different datasets, which consistently show that $\varepsilon/\sigma \simeq 0.25$ yields the best trade-off between accuracy and runtime gain. Section 8.5 shows this trade-off analysis, together with an analysis of the effects of σ, $s_{\max}$ and $td_{\max}$ on the performance of TYCOS.

8.3 Effectiveness evaluation

A) Evaluation on synthetic datasets: We generate synthetic datasets containing different types of relations, including both linear and non-linear, monotonic and non-monotonic, functional and non-functional functions. Then, we combine the generated relations into the same time series pair (the first time series is the values of x, the second time series is the values of $y = f(x)$). The individual relations are separated by independent data, and the time delays, td={0, 50, 100, 150} (samples), are added between x and y. Next, we apply TYCOS, and the baselines PCC, MASS, MatrixProfile and AMIC to the time series to verify whether the methods can detect the generated relations. A method detects a relation in a given pair of time series if it can locate a window w where (X_w, Y_w) corresponds to that relation. Table 1 shows the relations ($y = f(x)$ and u is added noise) recognized by the tested methods (the ✓ sign denotes an identified relation, and the × sign denotes an unidentified relation). The plots of the generated relations can be found in [17].

We see that when there is no time delay ($td = 0$), TYCOS and AMIC can detect all types of relations, while PCC, MASS, and MatrixProfile cannot detect non-linear and non-functional relations, e.g., a circle relation. When there is time delay ($td \neq 0$), PCC, MASS and AMIC cannot detect any relations, while MatrixProfile can detect only linear relations, unlike TYCOS which can detect all the tested relations.

B) Evaluation on real-world datasets: We evaluate TYCOS on two real-world data collections: smart energy [1] and smart city [2]. Using real-world applications, our goal is to make sense of extracted windows and learn insights from them. We describe the datasets, and the findings in the following.

The energy datasets [1]: measure energy usage from electrical devices in residential households in Maryland, USA during 07/2013-07/2014, and 02/2015-02/2016. There are 72 electrical plugs in total, and their consumptions are reported in minute and hour interval. We create pairwise time series from 72 plugs, and apply TYCOS and AMIC on each time series pair.

The smart city datasets: The NYC Open Data [2] contains more than 1,500 spatio-temporal datasets, providing rich information about NYC. For evaluation purposes, we consider two collections of data related to *weather* and *transportation.* Within *transportation*, we focus on the *Collision* dataset reporting the number of accidents in the city. The *Weather* dataset has 30 variables, recording weather condition in 5-minute and hour resolutions. The *Collision* dataset has 29 variables, recording incidents happened in minute resolution.

Summary of the results: On the energy datasets, TYCOS can extract correlations from more than 50 different time series pairs, while AMIC extracts fewer windows than TYCOS, and omits any correlations that have time delay. On smart city datasets, TYCOS is able to find correlations that could not be confirmed in [17] by AMIC. Due to space limitations, we cannot discuss all of them, but instead just show a few extracted correlations in Table 3 to illustrate our observations. In each column, the first number is the number of extracted windows, the second number is the time delay range, and the × sign denotes no windows can be extracted.

Table 3: Extracted correlations (h: hour, m: minute)

Correlations	TYCOS	AMIC
(C1) *Kitchen* vs. *Dish Washer*	80, [0-4h]	25, 0h
(C2) *Kitchen* vs. *Microwave*	21, [0-1h]	5, 0h
(C3) *Clothes Washer* vs. *Dryer*	39, [10-30m]	×
(C4) *Bathroom Light* vs. *Kitchen Light*	14, [1-5m]	×
(C5) *Kitchen Light* vs. *Microwave*	11, [0-2m]	4, 0m
(C6) *Children Room Light* vs. *Living Room Light*	8, [15-40m]	×
(C7) *Precipitation* vs. *Collisions*	28, [0.5-2h]	×
(C8) *Wind Speed* vs. *Collisions*	23, [0.25-1h]	×
(C9) *Precipitation* vs. *Pedestrian Injured*	16, [0.5-2h]	×
(C10) *Wind Speed* vs. *Motorist Killed*	12, [0.25-1h]	×

Interpretation of extracted windows: We interpret some of the correlations in Table 3 by comparing with the findings of [7, 17], and/or by plotting the data of extracted windows. Here, C1 presents a correlation between the energy usage of the *kitchen* and of the *dish washer*, with the time shift ranging from 0 to 4 hours. The extracted windows indicate frequent activities of kitchen from 16.00 to 19.00, and of dish washer from 21.00 to 23.00. C4 presents a correlation between the light upstairs in the bathroom, and the light downstairs in the kitchen, with an average time shift from 1 to 5 minutes. The correlation occurs frequently from 6.00 to 7.00. This pattern might hint that, either more than one person are living together so that when one is in the bathroom, the other goes to the kitchen; or that the same person wakes up in the early morning, goes to the bathroom and then comes to the kitchen. Interestingly, C5 can help provide extra information for C4. A correlation between the kitchen light and the microwave is identified, with a time shift between two devices is from 0 to 2 minutes, indicating the person might come to the kitchen to prepare breakfast. On smart city datasets, C7 and C8 present correlations between the increase of precipitation/ wind speed, and the number of collisions, with a time shift from 0.25 to 2 hours. In [17], AMIC could not confirm C7 and C8, because it does not consider the time delay between time series, and thus fail to capture correlations that are shifted in time. Furthermore, we found that precipitation has stronger impact on pedestrians

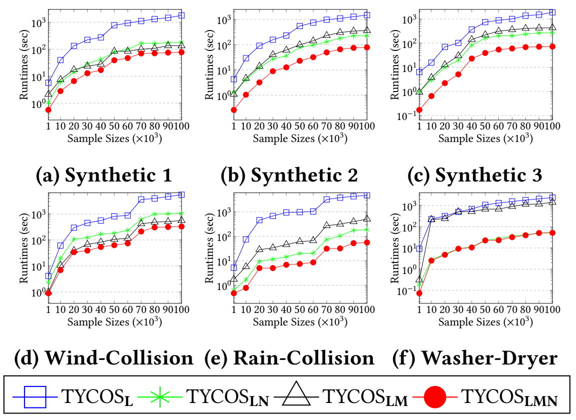


Figure 9: Runtime evaluation of TYCOS

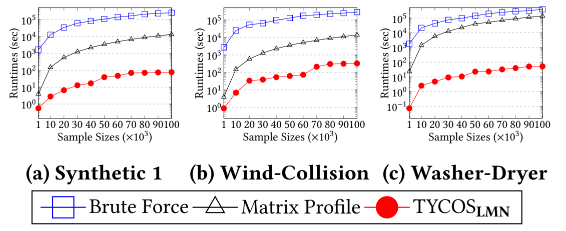


Figure 10: Brute Force, Matrix Profile, and TYCOS$_{\mathbf{LMN}}$

than on motorists or cyclists, while contrarily, wind has more impact on motorists and cyclists than pedestrians (C9, C10).

8.4 Efficiency evaluation

TYCOS performance is evaluated in terms of its runtime and accuracy. TYCOS is implemented in C++, and the experiments are run on a standard PC that has 2.7 GHz processor, 16 GB of RAM, and 512 GB of SSD.

A) Runtime evaluation: TYCOS runtime is evaluated by comparing its 4 different versions: TYCOS$_{\mathbf{L}}$, TYCOS$_{\mathbf{LN}}$, TYCOS$_{\mathbf{LM}}$, TYCOS$_{\mathbf{LMN}}$, and the Brute Force and MatrixProfile baselines. First, different TYCOS versions are compared against each other. The results on both synthetic and real-world data are shown in Fig. 9. The synthetic datasets, *Synthetic 1*, *Synthetic 2*, and *Synthetic 3*, are created by combining multiple relations from Table 1 into one time series pair. From Fig. 9 where the y-axis is in log scale, it can be seen that TYCOS$_{\mathbf{LMN}}$ achieves the best performance among all versions. Its speedup w.r.t. TYCOS$_{\mathbf{L}}$ ranges from 10 to 150 depending on data sizes. The average speedup is 20 on synthetic data, and 60 on real-world data. Furthermore, the noise theory and the efficient MI computation technique result in different speedups depending on data (there are situations where the noise theory is more efficient, and vice versa). The average speedup is 39 for the noise theory, and 32 for the efficient MI computation. However, applying both always yields better speedups than applying either of them.

Next, TYCOS with the best performance, TYCOS$_{\mathbf{LMN}}$, is compared against Brute Force and MatrixProfile. The results are shown in Fig. 10 (note log scale in the y-axis). We can see that TYCOS$_{\mathbf{LMN}}$ can achieve an average speedup of more than 3 orders of magnitude over Brute Force, and of more than 2 orders of magnitude over MatrixProfile, both of which are, however, exact.

B) Accuracy evaluation: To evaluate the accuracy of TYCOS, we compare the similarity of windows extracted from 3 versions: Brute Force, TYCOS$_{\mathbf{L}}$ and TYCOS$_{\mathbf{LN}}$. Note that the efficient MI computation technique does not change the accuracy of TYCOS$_{\mathbf{L}}$, thus, TYCOS$_{\mathbf{LM}}$ and TYCOS$_{\mathbf{LMN}}$ are not considered in this evaluation. Moreover, two windows are considered to be similar if they cover a similar range of indices. The comparison between

Table 4: Accuracy evaluation

	TYCOS$_{\mathbf{L}}$ vs. Brute Force		TYCOS$_{\mathbf{LN}}$ vs. TYCOS$_{\mathbf{L}}$	
Data Size	Synthetic Data	Real Data	Synthetic Data	Real Data
1K	96.2	95	100	100
10K	97.52	95.1	97.91	95.05
20K	94.08	91.7	98.19	97.78
30K	92.4	89.5	96.4	95.19
40K	97.85	95.1	98.17	97.01
50K	93.69	94.7	96.12	93.91
60K	95.49	94.8	97.1	97.78
70K	90.6	94.3	94.5	95.15
80K	88.75	91.02	96.21	95.8
90K	92.8	89.3	93.01	94.7
100K	93.1	94.7	95.8	94.94

Brute Force and TYCOS$_{\mathbf{L}}$ evaluates how accurate the LAHC approach on the TYCOS problem is, while the comparison between TYCOS$_{\mathbf{L}}$ and TYCOS$_{\mathbf{LN}}$ validates the accuracy of the noise theory. Since Brute Force generates overlapped windows, the generated windows are aggregated and the overlapped windows are combined together. The same synthetic and real-world datasets as when evaluating the runtime are used in this experiments.

Table 4 shows the average accuracy of TYCOS$_{\mathbf{L}}$ w.r.t. Brute Force, and of TYCOS$_{\mathbf{LN}}$ w.r.t. TYCOS$_{\mathbf{L}}$. Depending on the data sizes, TYCOS$_{\mathbf{L}}$ extracts from 88% to 98% similar windows compared to Brute Force, while TYCOS$_{\mathbf{LN}}$ extracts windows that are from 90% to 100% similar to TYCOS$_{\mathbf{L}}$.

The quantitative evaluation proves that our proposed theory and technique are very effective in improving the search performance. They help achieve an average speedup of more than 3 orders of magnitude compared to the Brute Force method, while maintaining highly accurate results.

8.5 Effects of Parameters

We examine how the major parameters: ε, σ, $s_{\max}$, and $td_{\max}$, affect the performance of TYCOS. We do not consider $s_{\min}$ in this experiment because $s_{\min}$ has minimal impact on TYCOS results.

A) Noise threshold ε**:** First, we examine how different values of ε affect the accuracy and runtime, using both synthetic and real-world data in Fig. 11. We can see, as the ratio ε/σ increases, the runtime gain increases (Fig. 11b), but the error rate also increases (Fig. 11a, error rate is measured by the number of missing windows). This result is intuitive because as the ratio ε/σ increases, more of the TYCOS search space is pruned, leading to higher speedup and larger errors. Next, we perform a trade-off analysis between accuracy and runtime gain as a means for choosing a proper value of the noise threshold ε. In Fig. 12, the accuracy and the runtime gain of each tested dataset are plotted together, with the ratio ε/σ on the x-axis. On the two tested datasets, i.e., energy and smart city datasets, we found that, when $\varepsilon/\sigma \in [0.05, 0.3]$, TYCOS$_{\mathbf{LN}}$ maintains an error rate less than 5%, while reducing the runtime up to 50%, compared to TYCOS$_{\mathbf{L}}$. Thus, our experimental setting $\varepsilon = \frac{1}{4}\sigma$ proved to be effective and robust. This threshold can be adjusted according to user's preference for accuracy.

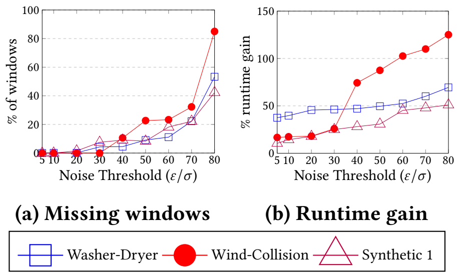
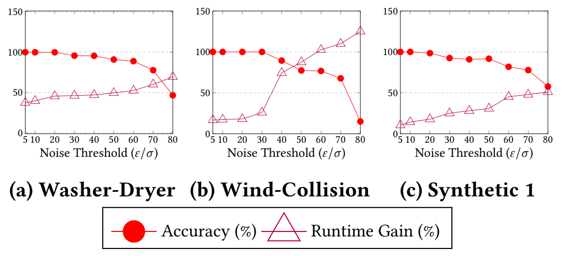
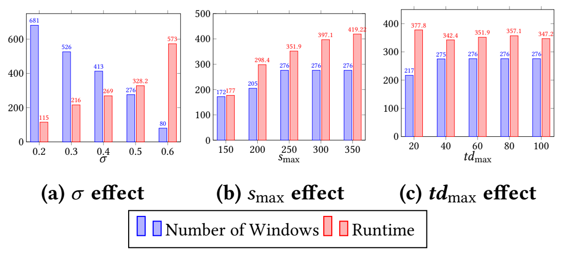


Figure 11: Effect of noise threshold ε



(a) Washer-Dryer **(b) Wind-Collision** **(c) Synthetic 1**

Figure 12: Trade-off analysis



(a) σ effect **(b) $s_{\max}$ effect** **(c) $td_{\max}$ effect**

Figure 13: Effect of σ, $s_{\max}$ and $td_{\max}$

B) Correlation threshold σ: We vary the values of σ to examine its effect, shown in Fig. 13a. We observe that, the correlations are stronger as σ increases, and thus, fewer windows are extracted. However, the runtime also increases because larger neighborhoods need to be explored to find strong correlations. For example, only 80 windows are extracted compared to 681 windows when σ increases from 0.2 to 0.6, while the runtime increases from 115 to 573 seconds.

C) Window size $s_{\max}$ and time delay $td_{\max}$: We examine how $s_{\max}$ and $td_{\max}$ affect TYCOS. We found that, although $s_{\max}$ and $td_{\max}$ are context dependent, the algorithm will converge after the two parameters reach certain values. When the convergence occurs, TYCOS extracts the same set of windows, while maintaining a similar runtime for $td_{\max}$, but an increasing runtime for $s_{\max}$. Fig. 13b and Fig. 13c illustrate this evaluation. Here, using the *(Snow, Collision)* datasets, TYCOS converges at $s_{\max} = 250$ and $td_{\max} = 60$, with 276 windows extracted when the convergence occurs. After the convergence, the runtime continues increasing as $s_{\max}$ goes beyond the value 250, while keeping similar values as $td_{\max}$ goes more than 60.

9 CONCLUSION AND FUTURE WORK

To our knowledge, TYCOS is the first comprehensive solution for the multi-scale time delay correlations search problem. TYCOS has the ability to extract all types of correlation relations, including both linear and non-linear, monotonic and non-monotonic, functional and non-functional ones. Our major contributions are: (1) integration of TYCOS and LAHC for multi-scale time delay correlations search, (2) the novel MI-based theory for noise identification, (3) the efficient MI computation technique to reduce computational redundancy. We perform an extensive evaluation on the effectiveness and efficiency of TYCOS, using both synthetic and real-world datasets. The evaluation shows that TYCOS can detect various types of relations in synthetic data, and find significant and interesting correlations in real-world data. The proposed noise theory and MI computation technique are also proved to be effective and improve the search performance by 2 to 3 orders of magnitude compared to the baselines. In future work, TYCOS can be extended to capture correlations across spatial dimensions. The result of this work can also provide a foundation for deeper data analysis, such as perform mining or infer causal effects from the extracted correlations.

ACKNOWLEDGMENT

This work has been partially supported by the DICYPS project funded by Innovation Fund Denmark, the GoFLEX project funded by the Horizon 2020 program, the SEMIOTIC project funded by Independent Research Fund Denmark, and the AAU International Postdoc Program.

REFERENCES

[1] *Net-Zero Energy Residential Test Facility.* https://pages.nist.gov/netzero/data.html.
[2] *NYC Open Data.* https://opendata.cityofnewyork.us.
[3] A. Agresti and B. Finlay. 2014. *Statistical Methods for the Social Sciences.* Pearson Education Limited.
[4] A. Alawini, D. Maier, K. Tufte, and B. Howe. 2014. Helping scientists reconnect their datasets. In *SSDBM.*
[5] J.L. Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975).
[6] E.K. Burke and Y. Bykov. 2017. The late acceptance hill-climbing heuristic. *European Journal of Operational Research* 258, 1 (2017).
[7] F. Chirigati, H. Doraiswamy, T. Damoulas, and J. Freire. 2016. Data polygamy: the many-many relationships among urban spatio-temporal data sets. In *SIGMOD.*
[8] R. Cole, D. Shasha, and X. Zhao. 2005. Fast window correlations over uncooperative time series. In *ACM SIGKDD Proc.*
[9] T.M. Cover and J.A. Thomas. 2012. *Elements of information theory.* John Wiley & Sons.
[10] A. Das Sarma, L. Fang, N. Gupta, A. Halevy, H. Lee, F. Wu, R. Xin, and C. Yu. 2012. Finding related tables. In *SIGMOD.*
[11] S. de Siqueira Santos, D.Y. Takahashi, A. Nakata, and A. Fujita. 2013. A comparative study of statistical methods used to identify dependencies between gene expression signals. *Briefings in bioinformatics* 15, 6 (2013).
[12] J.H Friedman, J.L. Bentley, and R.A. Finkel. 1976. An algorithm for finding best matches in logarithmic time. *ACM Trans. Math. Software* 3, SLAC-PUB-1549-REV. 2 (1976).
[13] M. Gribaudo, T.T.N. Ho, B. Pernici, and G. Serazzi. 2014. Analysis of the influence of application deployment on energy consumption. In *E2DC.*
[14] N. Ho, T.B. Pedersen, M. Vu, V.L. Ho, and C.A.N. Biscio. 2019. Efficient Bottom-Up Discovery of Multi-Scale Time Series Correlations Using Mutual Information. In *ICDE.*
[15] N. Ho and B. Pernici. 2015. A data-value-driven adaptation framework for energy efficiency for data intensive applications in clouds. In *IEEE SusTech.*
[16] N. Ho, H. Vo, and M. Vu. 2016. An adaptive information-theoretic approach for identifying temporal correlations in big data sets. In *IEEE BigData Proc.*
[17] N. Ho, H. Vo, M. Vu, and T.B. Pedersen. 2019. AMIC: An Adaptive Information Theoretic Method to Identify Multi-Scale Temporal Correlations in Big Time Series Data. *IEEE Trans. Big Data* (2019), 1–18.
[18] T.T.N. Ho. 2013. Activity recognition using smartphone based sensors. In *Master thesis.* Politecnico di Milano, Italy. http://hdl.handle.net/10589/85064.
[19] T.T.N. Ho. 2017. Towards sustainable solutions for applications in cloud computing and big data. In *Doctoral thesis.* Politecnico di Milano, Italy. http://hdl.handle.net/10589/131740.
[20] A. Kraskov, H. Stögbauer, and P. Grassberger. 2004. Estimating mutual information. *Physical review E* 69, 6 (2004), 066138.
[21] M. Middelfart, T.B. Pedersen, and J. Krogsgaard. 2013. Efficient Sentinel Mining Using Bitmaps on Modern Processors. *IEEE TKDE* 25, 10 (2013).
[22] A. Papana and D. Kugiumtzis. 2009. Evaluation of mutual information estimators for time series. *International Journal of Bifurcation and Chaos* 19, 12 (2009).
[23] K Pearson. 1895. Notes on Regression and Inheritance in the Case of Two Parents. (1895).
[24] R. Pochampally, A. Das Sarma, X.L. Dong, A. Meliou, and D. Srivastava. 2014. Fusing data with correlations. In *SIGMOD.*
[25] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. 2012. Searching and mining trillions of time series subsequences under dynamic time warping. In *ACM SIGKDD Proc.*
[26] S. Roy and D. Suciu. 2014. A formal approach to finding explanations for database queries. In *SIGMOD.*
[27] S.J Russell and P. Norvig. 2016. *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,.
[28] S. Salvador and P. Chan. 2007. Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis* 11, 5 (2007).
[29] S. Shakil, J.C Billings, S.D Keilholz, and C.-H. Lee. 2018. Parametric dependencies of sliding window correlation. *IEEE TBE* 65, 2 (2018).
[30] M. Vejmelka and K. Hlaváčková-Schindler. 2007. Mutual information estimation in higher dimensions: A speed-up of a k-nearest neighbor based estimator. In *ICANNGA Proc.*
[31] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H.A. Dau, D.F. Silva, A. Mueen, and E. Keogh. Matrix profile I: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets. In *2016 ICDM.*
[32] P. Zhang, Y. Huang, S. Shekhar, and V. Kumar. 2003. Correlation analysis of spatial time series datasets: A filter-and-refine approach. In *PAKDD Proc.*

Dynamic Query Refinement for Interactive Data Exploration[*]

Alexander Kalinin[†]
Vertica, a Micro Focus Company
Cambridge, Massachusetts
allex2001@gmail.com

Ugur Cetintemel
Brown University
Providence, Rhode Island
ugur@cs.brown.edu

Zheguang Zhao
Brown University
Providence, Rhode Island
zheguang_zhao@brown.edu

Stanley Zdonik
Brown University
Providence, Rhode Island
sbz@cs.brown.edu

ABSTRACT

Queries that can navigate large search spaces to identify complex objects of interest cannot be efficiently supported by traditional DBMSs. Searchlight is a recent system that aims to address this fundamental shortcoming by deeply yet transparently integrating Constraint Programming (CP) logic into the query engine of an array DBMS. This hybrid model enables exploration of large multi-dimensional data sets progressively and quickly.

Fast query execution is only one of the requirements of effective data-exploration support. Finding the right questions to ask is another notoriously challenging problem, given the users' lack of familiarity with the structure and contents of the underlying data sets, as well as the inherently fuzzy goals in many exploration-oriented tasks. To this end, in the context of Searchlight, we study the modification of initial query parameters at run-time. We describe how to dynamically refine (i.e., relax or tighten) the parameters of a query, based on the result cardinality desired by the user and the live query progress. This feature allows users to iterate over the datasets faster and without having to make accurate guesses on what parameters to use. Our experimental results show that the proposed techniques introduce little or no overhead while yielding considerable time savings compared to user-driven, manual query refinements. The result is a system that not only optimizes machine resource usage but also reduces user effort.

1 INTRODUCTION

Consider a researcher working with the MIMIC II dataset [1], which contains medical information for a number of ICU patients over a large period of time. Assume that she is studying historical ABP (Arterial Blood Pressure) signal readings and wants to identify time intervals that satisfy the following constraints:

- The length of the time interval can be from 8 to 16 seconds, and it can start at any point in time.
- The average signal amplitude must be within [150, 200] range for the interval.
- The maximum amplitude of the signal over the interval must exceed the maximum amplitude of the signal over its left and right neighborhoods by at least 80. The left (right) neighborhood is defined as an 8-second interval to the left (right) of the main interval.

This seemingly simple *search query* is difficult to express and even harder to optimize using traditional query languages and DBMSs [10]. Recent systems, such as Searchlight [10, 11], extends SciDB [2] with constraint-based search and optimization support, effectively integrating two mature technologies (DBMS and CP-based solver) to address queries that operate on large search spaces and over large data sets.

Now imagine that the user gets *zero* results after running the initial query, as illustrated in the top part of Figure 1. It turns out that the query was over-constrained! She then tries to guess the correct constraint parameters manually by *relaxing* the average amplitude constraint: now the average ABP amplitude must be within the [150, 250] range. When she runs this new query, she gets flooded with a large number of intervals, many of which overlap, as shown in the middle band of Figure 1. Since such a result can be overwhelming for her to parse and study, expecting that there might be more focused and "better" initial results she could work with, she then tries to *tighten* the interval to [150, 220]. After running the query again, she gets a reasonable number of results that she can explore in detail, as illustrated at the bottom of Figure 1.

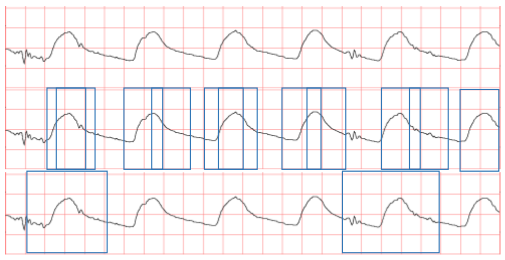


Figure 1: Exploring the ABP waveform data. *Top*: original query result. *Middle*: an over-relaxed query result. *Bottom*: final query result.

This scenario illustrates a number of problems:

- The user has to go through a series of guesses to identify a query that outputs a desirable number of results. This is often a frustrating and cumbersome process, especially if the user has limited knowledge about the data, which is often the case when exploring new data sets.
- The process of manual refinements (i.e., relaxation and tightening) might be quite complex even for seemingly

[*]This work was funded in part by NSF IIS-1526639.
[†]This work was done while the author was a PhD student at Brown University.

simple queries. In the earlier example, the user can modify the average amplitude constraint, or the two neighborhood constraints, or any combination of those, which collectively create an exponential number of possibilities.
- With manual refinements, no guarantees can be given for the final result. In general, users prefer to get results that are as close as possible to the original query parameters.

We address these problems in the context of Searchlight, which gives us an open DBMS platform to develop and test our solutions. At the same time, we emphasize that these problems would arise in any system supporting constraint-based queries and traditional solving technology. We also note that these issues are amplified when dealing with large datasets, as the query times can be larger and it becomes even more critical to perform such query iterations quickly and efficiently.

1.1 Motivation

A common initial step in data exploration is the identification of a small number of interesting cases that the user dives into for deeper study. In such cases, query refinements are applied to assure a target result cardinality to meet a "budget" constraint typically on user time or money. For example, a medical researcher may have the budget to run a survey on a specific number of patients with certain characteristics. A business may have a discount budget that is sufficient only to a specific number of customers within a given campaign. An advancement officer of a university may have a limited time to call a certain number of alumni on a given day. An astronomy researcher may have the time to study a limited number of celestial objects for a research project. In all these cases, the ability to have the system automatically tweak the query to produce a result set of a desirable size is very helpful.

Beyond these generic use scenarios, we have anecdotal evidence from a demonstration of waveform data exploration [11]. In this demo, the users were given the ability to fill in the parameters for template queries, such as signal amplitude, interval length, etc. However, since they had very limited knowledge about the presented dataset, their queries often output either too many results (sometimes thousands) or nothing at all. Both of the outcomes were frustrating, and the users had to go through a number of trial-and-error runs to identify a small set of results that is amenable to manual deeper inspection.

1.2 Query Refinement Approaches

Ideally, any system should perform any refinements automatically by detecting if the query needs to be modified during its evaluation without specific directives from the user. If there is a need for relaxing the query, it should choose the constraints and the degree of modification such that the final results are *closest* to the original constraint parameters according to the provided distance function. In the case of tightening, it should rank the results and output the top ones according to the specified ranking function. For performance optimization, it should reuse as much computation as possible to minimize the overhead from multiple searches. At the same time, if there is no need to modify the query, the automatic approach should not incur any significant overhead.

Query refinements have been studied in the context of relational DBMSs [4, 6–8, 12–18]. Similar approaches can be extended to array DBMSs (such as SciDB) as well. However, the query described in the example belongs to a different kind of *search* queries, which is the reason why an engine such as Searchlight that supported CP was used in the first place instead of the original array DBMS. As we argue in Section 6, existing methods cannot be easily applied to constraint-based search queries. These methods generally assume that either the result cardinality can be easily estimated for the query, or appropriate range-based indexes (e.g., B-trees or R-trees) exist over the objects of interest (which are time intervals for the example above), so that the search space of all possible objects could be traversed efficiently. Due to the ad-hoc nature of search queries, however, the result cardinality is hard to estimate. At the same time, since objects of interest are defined by the query itself, as part of the constraint specification, indexing becomes infeasible [10].

Another approach, possibly applicable to some simple search queries, would be to rewrite such queries in SQL and execute them on a relational (or array) engine, opening the possibility of using the existing query refinement techniques. However, previous research [9, 10] suggests that executing such search queries in a traditional engine, while possible, incurs significant performance penalties. More importantly, such conversion results in very complex SQL queries, for which the applicability of existing refinement methods would be quite limited.

1.3 Overview of Dynamic Query Refinement

In a nutshell, Searchlight uses Constraint Programming (CP) to perform the search over an in-memory *synopsis* of the original data. The solver dynamically builds the search tree, possibly pruning parts of the tree that cannot satisfy the constraints. We observe that the main reason the original query fails and, thus, produces an empty result is search pruning. If the query needs relaxation, only the previously pruned parts of the search space need to be revisited. Thus, we track these parts and later "replay" the search over them, if needed. The replaying is guided by a user-provided distance function — more promising replays are explored first. It is important to mention that this replay happens automatically, when we detect the absence of the desired number of results. At the same time, replaying is performed as part of the original query, thus removing the need to re-explore previously finished parts of the search space, resulting in considerable time savings. From the logical perspective, this approach can be seen as introducing an objective function and searching for results that minimize it. This results in a very natural extension of the traditional CP model that already supports objective function-based search. The only piece of information required from the user is the desired cardinality of the result k and, possibly, the distance function $d()$ [1]. Our approach guarantees to produce top-k results closest to the original constraints (i.e., minimizing $d()$).

Query tightening can be seen as a dual problem for query relaxation. Thus, the two work closely together. If the number of results at some point during the search exceeds the desired result cardinality, we switch the approach to query tightening, which essentially ranks all results according to the specified ranking function $r()$ [2]. The main idea behind our approach involves introducing a *dynamic* ranking constraint at that particular moment during the search that restricts the final result to top-k objects maximizing $r()$. As in the case of relaxation, it can logically be seen as introducing an objective function with the maximization

[1]We provide built-in distance functions by default.
[2]Some built-in ranking functions are provided by default as well.

goal. We call such a constraint dynamic since its parameters are constantly updating depending on the current result.

1.4 Contributions

Our main contributions are as follows:

- We introduce a novel distributed relaxation and tightening framework for search queries, which automatically detects the need for modifying the query at run-time and performs query refinements without the need for any user intervention. The user just needs to specify the target result cardinality and, optionally, the distance and ranking functions.
- The framework is general and applicable to different types of constraints. It works for any constraint of form $a \leq f_c() \leq b$, where $f_c()$ is an arbitrary expression, including User-Defined Functions (UDFs). It can be extended to other types of constraints, with the introduction of a meaningful distance and/or ranking measure. If desired, it can be even applied to other engines beyond Searchlight, provided they follow the CP execution model.
- Our implementation of this framework operates at the DBMS engine level and extends it in a natural way that is compatible with a generic CP model. Due to such a seamless integration, it does not interfere with the existing query processing, does not impact performance of queries that do not require relaxation/tightening, and leverages existing DBMS engine features.

We performed an extensive experimental evaluation over synthetic and real (MIMIC II) data, which we discuss in Section 5. Our results reveal tremendous time savings compared to manual refinements and very low overhead.

The rest of the paper is organized as follows. In Section 2 we describe Searchlight as the DBMS/CP platform we use to explore our solutions. Section 3 presents the formal model behind our relaxation/tightening framework. Section 4 describes design and implementation of our solutions. Section 5 presents the experimental evaluation Section 6 discusses the related work and Section 7 concludes the paper.

2 SEARCHLIGHT BACKGROUND

Searchlight is an extension of a traditional query processing engine for efficient execution of search queries. It introduces Constraint Programming (CP) methods to the query processing and operates *inside* the engine. The user submits a search query in form of a constraint program (decision variables and constraints over them).

Searchlight processes the query by creating a number of CP *Solvers* that perform the search on an in-memory *synopsis* of the original data. Since synopsis is a lossy compression of the original data, the CP-provided results (solutions) might contain false positives. Thus, the results need to be verified over the original data. This is done by another Searchlight component called *Validator*. The two components work concurrently to facilitate online answering. When Validator confirms a solution, it is immediately output, while false positives are filtered out. While Searchlight does not have the notion of a query plan, each execution can be imagined as a pipeline between two "operators": Solver and Validator. The Solver receives the query, outputs a stream of solutions (tuples) to the Validator, while the latter filters out false positives and outputs the final solutions to the user. Searchlight is implemented as part of the SciDB query engine and uses its infrastructure to distribute both search space and data across the cluster. Thus, Solvers and Validators exhibit both multi-node and multi-core parallelism. The details of query processing are out of scope of this paper, and the in-depth discussion can be found in the Searchlight paper [10].

There are no specific limitations on types of queries Searchlight can work with, since CP solvers are quite general. Users can use any of the constraints generally found in CP solvers. At the same time, most useful types of queries tend to use aggregate functions to assess regions in the data. For example, the query presented in the introduction section can be represented in Searchlight as follows:

- Decision variable x defines the start of the resulting interval. Its domain is the entire length of the recorded data, since the interval can start anywhere.
- Variable lx defines the length of the interval, $lx \in [8, 16]$.
- The amplitude constraint: $avg(x, x + lx, ABP) \in [150, 200]$, where $avg()$ is a built-in Searchlight aggregate. We will denote it c_1 for future reference.
- The left neighborhood constraint: $|max(x, x + lx, ABP) - max(x - 8, x, ABP)| \geq 80$. The right neighborhood is similar. We will denote them as c_2 and c_3 respectively.

When the Solver receives a CP specification similar to the above, it dynamically builds a *search tree* for the query. An example of such a tree for the query above can be seen in the left part of Figure 2. The nodes represent the current variable domains (search states), while edges represent Solver decisions that lead to the corresponding search states. Children's variable domains are subsets of the parent's ones, and leaves have the variables *bound* (i.e., the domains are scalars). The decision process (search heuristic) is tunable, can be selected and modified by the user [10]. The search process itself is performed in a traditional backtracking way. If, while building the tree, the Solver establishes a violation of query constraints (by relying on estimations from the synopsis), the entire corresponding sub-tree is pruned from the search (marked with the red "no" symbols in the figure), and is never built or visited. When a leaf of the tree is reached (i.e., the variables are bound to scalar values), the corresponding variable assignment (solution) is passed to the Validator.

3 QUERY REFINEMENT MODEL

In this section we discuss our relaxation/tightening framework. While described in the context of Searchlight, the model does not depend on its implementation details and is primarily driven by well established CP concepts. Throughout the paper, we use query "constraining" as a synonym for "tightening".

Each CP search query consists of a number of decision variables X and constraints C. Given such a query the search framework either outputs all results or proves there is none. However, if the user specifies the desired *cardinality* of the result k, the framework behaves differently depending on the outcome of the original query:

- The query outputs exactly k results. In this case the behavior stays the same.
- The query outputs a set of results $R = r, |R| < k$. In this case it is automatically modified to produce additional $k - |R|$ results. The additional results are guaranteed to minimize the specified $RD(r)$ function (discussed below). This is query relaxation.

- The query outputs $|R| > k$ results. In this case the framework effectively ranks the results based on the $RK(r)$ function (discussed below), and the query returns top-k results according to $RK(r)$. This is query constraining.

When relaxing or constraining a query, we consider only range-based constraints of form $a \le f(X) \le b$. The nature of $f()$ is not important. It might be an arbitrary algebraic expression containing User-Defined Functions (UDFs), built-in functions and variables from X. In our running example $f()$ is the $avg()$ function for constraint c_1 and the $max(x, x + lx) - max(x - 8, x)$ expression for c_2. We generally treat $f()$ as black boxes, but we assume the knowledge of values a', b' at every node of the search tree, such that $a' \le f(X) \le b'$ for all possible values of X at that node. This information is readily available as part of the search process and is used to make search decisions (for building the tree) and prune parts of the tree. In Searchlight a', b' are derived from the each node's variable domains by using the synopses to estimate constraint functions [10]. Thus, no modification to the query engine should be required in this regard. By default, we consider all range-based constraints for relaxation/constraining, but the user can exclude any of them from the process. We denote constraints considered for relaxation (constraining) as C^r (C^c). C^r does not necessarily equal C^c.

The relaxation and constraining processes are based on result ranking via separate relaxation and constraining ranking functions. In the two following sections we describe the default functions we use. Then we discuss the custom ranking functions requirements.

3.1 Query Relaxation Model

Assume a constraint $c \in C^r : a \le f_c(X) \le b$ and a result r such that $f_c(r) = t$. We define the relaxation distance $RD_c(r)$ as follows:

$$RD_c(r) = \begin{cases} 0 & \text{if } a \le t \le b \\ \frac{t-b}{max(f_c(X))-b} & \text{if } t > b \\ \frac{a-t}{a-min(f_c(X))} & \text{if } t < a \end{cases}$$

Then, the total relaxation distance $RD(r)$ is:

$$RD(r) = \max_{c \in C^r} w_c RD_c(r),$$

where $w_c \in [0, 1]$ are constraint weights, which can be defined by the user. By default, $w_c = 1$.

We selected the $RD()$ definition above just as a suitable *default*, aiming at providing reasonable out-of-box experience. As we discuss in Section 3.3, users can choose their own functions, provided they respect certain requirements. In general, p-norm is a logical choice when some distance between query points needs to be measured [4]. We chose max ($p = \infty$) to penalize results where some outlier constraints have large $RD_c()$ values. This allows us to limit the distances of final results. A weighted sum ($p = 1$) or Euclidean distance ($p = 2$) are other viable choices.

The denominators in $RD_c(r)$ require further explanation. In general, $f_c()$ from different constraints might have different scales. For example, one constraint might deal with ages (e.g., $f_{c_1}() \in [0, 150]$), while another with similarities (e.g., $f_{c_2}() \in [0, 1]$). That is why we perform [0, 1]-normalization of each $RD_c(r)$ by dividing it by the maximum possible difference in values. Min/max values for $f_c()$ can be usually derived from the obvious domain restrictions (e.g., age cannot exceed 150). We additionally allow users to specify the max/min values with the query, giving them more control over relaxation: we will not relax the corresponding constraints beyond the specified min/max values. We will use $RD_c(r)$ to denote the normalized distances, as well as the original ones, where appropriate.

In addition to $RD(r)$ for each result r we define $VC(r)$ as the number of constraints from C^r violated by r divided by $|C^r|$. Thus, $VC(r) \in [0, 1]$ is the normalized number of violated constraints. Then, we define the total relaxation penalty for r as:

$$RP(r) = \alpha RD(r) + (1 - \alpha)VC(r)$$

The $RP()$ allows users to prefer results with smaller number of violated constraints, which is especially important in cases when $RD(r)$ "looses" information about individual constraints (e.g., as in our choice of max). α controls the degree of the preference (the default is 0.5). We picked weighted sum as a suitable default, giving the user the choice between two criteria, $RD()$ and $VC()$. However, this default is not essential for the proposed framework. $RP()$ can be changed, and other criteria can be incorporated in the formulation, if required.

The model provides the following *relaxation guarantee*: if the user submits query Q with the cardinality requirement k, the query outputs at least k results r with the lowest $RP(r)$ values possible.

The definition above naturally incorporates queries in no need of relaxation, with $\ge k$ results. If r satisfies the original constraints, $RP(r) = 0$. Thus, the guarantee is automatically fulfilled. If the user chose to specify tight min/max bounds for some $f_c()$, we might not be able to find k results in case there are not enough results satisfying even *maximally* relaxed constraints (i.e., $\min f_c() \le f_c() \le \max f_c()$). This is because we effectively treat such $f_c()$ specification as a "hard" constraint.

Let us revisit the running MIMIC example. Assume the user wants $k = 3$ results and $C^r = \{c_1, c_2, c_3\}$. Additionally, $avg()$ and $max()$ values for the ABP signal lie within [50, 250]. Then a possible search might progress as follows:

(1) A result $r_1 = (180, 85, 85)$ is found (we write a result as a tuple of $f_i()$ values). Since it satisfies the constraints, $RP(r_1) = 0$ and it is output to the user.
(2) $r_2 = (190, 80, 90)$. Since, $RP(r_2) = 0$, it is output.
(3) Searchlight cannot find any more results, so it starts relaxing the query.
(4) $r_3 = (160, 70, 60)$ is found. It violates c_2 and c_3. For r_3: $RD_{c_1} = 0, RD_{c_2} = \frac{10}{80} = 0.125, RD_{c_3} = \frac{20}{80} = 0.25$. Thus, $RD(r_3) = 0.25$, and $RP(r_3) = \frac{1}{2}(0.25 + \frac{2}{3}) = 0.458$.
(5) $r_4 = (130, 80, 80)$ is found. For r_4: $RD_{c_1} = \frac{20}{100} = 0.2, RD_{c_2} = 0, RD_{c_3} = 0$. Thus, $RD(r_4) = 0.2$, and $RP(r_4) = \frac{1}{2}(0.2 + \frac{1}{3}) = 0.267$. Since $RP(r_4) < RP(r_3)$, r_3 is discarded, and r_4 is put into the result.

3.2 Query Constraining Model

The query performs constraining only when the number of results exceeds the k required by the user. That means during constraining each result r_i satisfies all constraints in C^c. For each function $f_c(X)$ from constraints in C^c the user can specify her preference in form of maximization or minimization of the function. For example, if some constraint's $f_c(X)$ is a property like the amplitude of a signal, the user might prefer large values of $f_c(X)$. On the other hand, if $f_c(x)$ is some spatial distance, the user might prefer smaller values. For each constraint $c \in C^c : c = a \le f_c(X) \le b$ and result $r : f_c(r) = t$ we define the ranking function $RK_c(r)$ as follows:

$$RK_c(r) = \begin{cases} \frac{b-t}{b-a} & \text{if } c \text{ is being maximized} \\ \frac{a-t}{b-a} & \text{if } c \text{ is being minimized} \end{cases}$$

Since r satisfies the constraints, $a \le t \le b$. As in the case of query relaxation, we normalize $RK_c()$ to $[0, 1]$ to account for the possibility of different scales for $f_c()$. If the interval is half-open, i.e., a or b is not specified, a suitable domain boundary for $f_c()$ can be used instead.

We define the full rank of result r as:

$$RK(r) = 1 - \sum_{c \in C^c} w_c RK_c(r),$$

where $w_c, 0 \le w_c \le 1 \wedge \sum_c w_c = 1$ represent the constraint weights to prioritize some constraints over others. By default, $w_c = \frac{1}{|C^c|}$. Note, $RK(r)$ assigns higher ranks to better results, which is more natural to the user. As in the case of $RD()$, any p-norm could be a reasonable choice for $RK()$. We saw the weighted sum providing meaningful results in practice. In addition, it allows us to demonstrate that the approach is flexible to the choice of distance functions.

Assuming these definitions, the model provides the following *constraining guarantee*: when the user submits query Q with the cardinality requirement k, if Q has at least k results r, the query outputs at most k results with highest $RK(r)$ possible. Note, if the query does not have at least k results, the relaxation will be performed instead.

Revisiting the running MIMIC example, let $C^c = \{c_1, c_2, c_3\}$, $w_{c_i} = \frac{1}{3}$. Let us assume the user prefers maximization for all constraints and wants a single result. Note, that f_{c_1} and f_{c_2} has the maximum value, 200, derived from the domain. The search might progress as follows:

(1) A result $r_1 = (160, 100, 100)$ is found. Its rank is $RK(r_1) = 1 - \frac{1}{3}(\frac{40}{50} + \frac{100}{120} + \frac{100}{120}) = 0.178$.
(2) A result $r_2 = (150, 80, 85)$ is found. Its $RK(r_2) = 0.014$. Since $RK(r_2) < RK(r_1)$, r_2 is discarded.
(3) The next result is $r_3 = (190, 120, 120)$. Its $RK(r_3) = 0.289$. Since $RK(r_3) > RK(r_1)$, r_1 is discarded, and r_3 becomes the new top-1.

In addition to the scalar ranking approach just described we support another popular vector-based ranking called *skyline* [3]. In that case the query simply outputs non-dominated results, where a result is a vector of values of $f_c(), c \in C^c$ (as in the example above). By definition, V dominates W, iff $\forall i : v_i \ge w_i \wedge \exists i : v_i > w_i$. The meaning of $>$ for each $f_c()$ is defined by the user's minimization/maximization preference, as in the scalar case. For skyline, however, we cannot guarantee that the number of results will not exceed k, since non-dominated vectors are not comparable.

3.3 Approach Customization

In addition to the default penalty and ranking functions discussed above, the user can add their own custom functions. The functions may be called by the query engine during the search at any search tree node, where some variables may still be unbound. This means the custom $RP()/RK()$ functions must be able to output the penalty/rank interval for all possible solutions contained in the corresponding sub-tree. Our implementation provides the user with the current variable domains and synopsis-based intervals for all constraint functions at any node. This information should be enough to compute the required bounds.

The functions must conform to the certain requirements to ensure the relaxation/constraining guarantees and proper performance. For a custom RP returning ranges $[lp, hp]$, the requirements are:

- $RP() \ge 0$, with larger values corresponding to worse relaxation. All results satisfying the original query must have $RP() = 0$.
- $lp = hp$ at solutions (leaves) of the tree, since the variables are bound there.
- $RP()$ cannot underestimate lp, which means it cannot be greater than the minimum of penalties for all possible solutions at the corresponding sub-tree.

The requirements for the RK functions are similar, with the only differences that $RK()$ assigns larger values to better candidates and, thus, should not underestimate hp. The user can define her own dominance measure for the skyline ranking as well. The corresponding function is periodically called over the current top-k results to determine if any of the current sub-tree solutions might enter the top-k.

In principle, customization can go beyond static ranking function. *Dynamic* functions is an interesting extension, where the ranking functions may change depending on the results already found, e.g., the user might want to prefer "diversity" among the relaxed results so that their relaxation distances differ by at least the specified amount. This can be accomplished by introducing new constraints depending on the results already discovered. Other dynamic modifications might be possible as well, depending on the user's preference. Studying such richer functionality is left for future work.

4 QUERY RELAXATION AND CONSTRAINING

In this section we describe the implementation of the relaxation/constraining model. Our approach is general and can be applied to other CP-based engines.

In general, a CP solver dynamically builds and traverses the search tree. The dynamic nature of the search process allows for modification of existing constraints and the addition of new ones. Thus, if a query does not produce enough results satisfying the original constraints, we can revisit some parts of the search tree with modified (relaxed) constraints. If a query produces too many results, we can introduce new dynamic constraints to prune results having smaller ranks than the already found ones.

The efficiency of the relaxation and constraining heavily relies on effective pruning. We exploit the following cases:

- During the main search at the Solver. This is the most effective point. It allows us to prune parts of the search tree with possibly large number of candidates and avoid unnecessary validations later.
- Just before validating the candidate at the Validator. This is effective in case Validators lag behind the Solvers, and their candidate queues grow large. In that case new ranking/penalty information about the current result might allow us to perform additional pruning and avoid expensive I/O.
- After validating the candidate at the Validator. Even if the candidate passes the validation over the real data, it is necessary to check them again with the up-to-date penalty/ranking values. However, this is is done only for correctness, with no performance benefits. These checks do not require any additional I/O.

4.1 Query Relaxation

A CP solver dynamically builds the search tree and validates query constraints at every node of the tree. A node either satisfies the constraints or *fails*. Successful nodes eventually lead to leaves, which produce candidate solutions. When the search is finished, if we have not found k results (the user's desired cardinality), we start revisiting parts of the search tree with modified, relaxed constraints. We do not have to revisit successful search nodes, since these nodes satisfied the *original* constraints and cannot yield any new solutions. Previously failed search nodes, on the other hand, could lead to new candidates satisfying the *relaxed* constraints. We call this process *fail replaying*.

When we encounter a failed search node, we prune the node as usual. However, we also record the current search state of the node:

- Current decision variable domains. This information is crucial when the fail is replayed later to resume the search from this exact point without revisiting any extraneous search nodes.
- The ranges $[a', b']$ for every function $f_c(), c \in C^r$. As we discussed in Section 3, these are available as part of the normal search process.

After this information has been obtained, we compute the best (*BRP*) and worst (*WRP*) relaxation penalties possible for the saved failed node (i.e., for the solutions in its sub-tree). For the built-in *RP* function this is straightforward from the definition. The custom function, as discussed in Section 3.3, must compute these values itself. After the relaxation penalties are computed, the fail is inserted in the priority queue ranked by the *BRP*.

If during the search at least k results are found, we just stop recording the fails. At the same time the constraining mechanism turns on, which we discuss further in Section 4.3. If the main search completes with less than k results, the relaxation is needed. We start replaying fails from the priority queue (i.e., minimizing *BRP*). To replay a fail, we do the following:

(1) A new CP search is initiated with the decision variables assigned the domains recorded at the fail. This can be seen as traveling back in time to the moment just before the fail.
(2) Each violated constraint is modified: $a \leq f_c() \leq b \rightarrow a' \leq f_c() \leq b'$, where $[a', b']$ is the recorded interval. This guarantees the search will not fail again when resumed.

The new search is handled exactly the same as the original one. Thus, it might fail again, at other search nodes further in the search tree. One example is when a previously valid constraint becomes violated due to more accurate synopsis estimations (those tend to become better closer to leaves). Such *repeated* fails are caught as the original ones and might be replayed later. During replays we explore only previously untouched parts of the search tree. No search nodes are ever revisited, which improves performance and guarantees the absence of duplicate results.

The discussed replay mechanism can be seen as naïve, since it does not allow any additional pruning at the search level. When we replay a fail, we relax previously failed constraints maximally, so they cannot fail again. Thus, we just relax constraints until the search reaches the leaves. The new candidates, however, do not necessarily belong to the best-k results. To improve the efficacy of pruning we take into account the Maximum Relaxation Penalty (*MRP*) among the already found results.

$MRP \in [0, 1]$ effectively defines the worst penalty a result can have to belong to the best-k, and it constantly changes during the execution. Let us first assume the built-in *RP* function. While the number of results found is less than k, the $MRP = 1$. When at least k results have been found, *MRP* might decrease. We modify the process as follows:

- When a fail is recorded, its *BRP* is compared with *MRP*. If $BRP > MRP$, the fail is discarded completely, since its search sub-tree cannot provide any useful candidates (i.e., with $RP(r) \leq MRP$).
- When a fail is selected for replaying, its recorded intervals $[a', b']$ are tightened according to the current *MRP* to improve pruning, since the constraints will be relaxed as minimally as possible. In addition, we repeat the *BRP* and *MRP* comparison, since *MRP* might have changed.

Let us discuss the interval tightening in more detail. Assume the built-in *RP*() function discussed in Section 3.1 with $\alpha \neq 0$. If $\alpha = 0$, the relaxation distance does not influence *RP*(), so no tightening is possible. Otherwise, to qualify for the result, all candidate solutions r must have $RP(r) \leq MRP$. Since $RP(r) = \alpha RD(r) + (1 - \alpha)VC(r)$,

$$RD(r) \leq \frac{MRP - (1 - \alpha)VC(r)}{\alpha}.$$

Let us revisit the example from Section 3.1 and illustrate the above algorithm with Figure 2. In this figure the search nodes are rectangles with the synopsis values for the c_1 and c_2 functions (we do not show c_3). The search encounters two fails. The first one in the search order is the lower one, for which both c_1 and c_2 are violated, since $110 < 150$ and $60 < 80$. Its $BRP = \frac{1}{2}(max(\frac{40}{100}, \frac{20}{80})) + \frac{1}{2}\frac{2}{3} = 0.53$. The fail is recorded into the table. Then the search encounters the upper fail, for which only c_2 is violated. Its $BRP = \frac{1}{2}(\frac{20}{80}) + \frac{1}{2}\frac{1}{3} = 0.29$. Assume $MRP = 0.5$ at some point during the relaxation, and the next fail is being taken from the table. According to the formula above, $RD(r) \leq 0.33$. Thus, the $[10, 60]$ is tightened to $[80 - 0.33 * 80, 60] = [53, 60]$, which is used as the relaxed c_2.

For the custom *RP*() function we cannot apply the same logic for tightening intervals, since the custom *RP*() is effectively a black box. In this case the constraints are relaxed to the $[a', b']$ intervals. However, at each search node we call the custom *RP*() function to check against the *MRP*. If the search node does not pass the check, we fail the node and prune the sub-tree completely.

After the search tree with relaxed constraints reaches a leaf, the corresponding solution is submitted to the Validator in the same way as in the original search. Validator is aware of the relaxation and validates the relaxed candidate accordingly, by taking into account the current *MRP* value. As we discussed in the beginning of Section 4, it performs two checks: one before and one after the validation. At the first one it compares the candidate's *BRP*(r) value (supplied by the Solver) with the current *MRP* and discards the candidate if $BRP(r) > MRP$. Otherwise, it performs the validation as usual with *all* constraints relaxed maximally with respect to *MRP*. Relaxing all constraints here is required for correctness, since even if some constraints fail during validation, the solution's penalty might still be below the *MRP*, and it will qualify for the result. That is why the second check is required after the validation.

It is the responsibility of the Validator to update the *MRP* value, since it produces the final result. If a new result decreases *MRP*, the Validator broadcasts the change to all instances in the cluster, so *MRP* is (asynchronously) updated for all Solvers/Validators

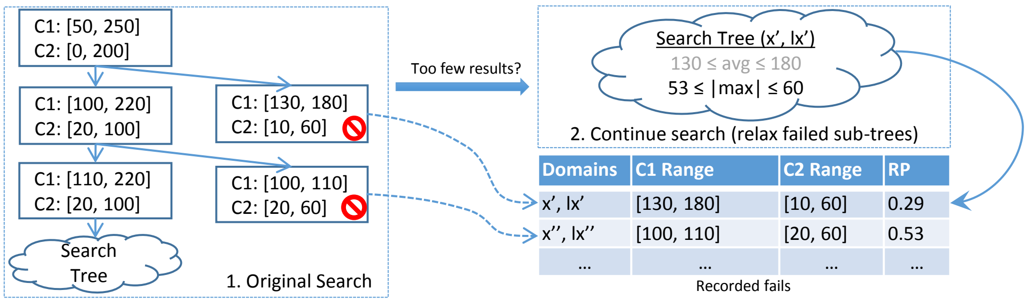


Figure 2: Example of fail recording and replaying for the running MIMIC query.

participating in the query. No special changes to the distributed query processing is required beyond that.

The correctness of the the model's relaxation guarantee is due to the correctness of the CP search itself. While the search process itself uses heuristics to guide the process, the pruning is performed in the provable way, without eliminating any valid results. Constraints modification during the search process is guided by the *MRP*, which is maintained based on the currently discovered results, so subsequent relaxation results will not be eliminated. As far as the complexity of the approach goes, it is equivalent to the complexity of the underlying CP search problem, since the relaxation is performed as in terms of CP.

4.2 Query Relaxation Optimizations

We now discuss a number of useful optimizations for query relaxation. These do not modify the main algorithm, but offer performance gains in common situations.

Computing functions at fails. When we catch a fail, we save the $[a', b']$ intervals for functions $f_c(), c \in C^r$. However, if the search fails at a search node, it does not necessarily mean all constraints have been verified yet. The fail might happen at the first violated constraint, in which case the subsequent constraints are not touched at all. This implies some $f_c()$ values might be unknown. For example, in our running MIMIC example, if c_1 fails, c_2, c_3 are not verified, and min/max ABP values are not computed.

In such cases we can force the computation via the Searchlight API to obtain $[a', b']$ ranges for the remaining constraints. However, $f_c()$ might be relatively expensive, and the cumulative overhead of the fail recording might become quite large. We perform the computation in a lazy way instead, when the fail is replayed. There are compelling reasons behind the lazy evaluation. First of all, if the query does not require any relaxation (i.e., it discovers at least k results) or the fail does not pass the *MRP* check later, the fail and its intervals will not be needed at all. Thus, the total completion time might improve. Second, delays for interactive results might decrease, as we do not pay the full price of computing $f_c()$ immediately, but later, when really needed.

This does not require any significant changes to the engine. Lazy evaluated functions' values are recorded as unknown, and the constraints are considered as non-violated. During the replay, Searchlight automatically estimates unknown values and checks the constraints.

Partial relaxation at replays. When we replay fails, we relax the violated constraints according to the saved $[a', b']$ intervals (tightened with respect to the *MRP* value). However, even the tightened intervals might be quite wide, resulting in poor pruning. This is especially true for fails happening closer to the root of the tree. To avoid over-relaxation, we do not relax the violated constraints all the way, but rather use a percentage of the relaxation interval — a parameter called Replay Relaxation Distance ($0 \le RRD \le 1$). If, for example, a constraint $f_c() \le 10$ needs to be relaxed to $f_c() \le 20$, and $RRD = 0.3$, we relax the constraint to $f_c() \le 10 + (20 - 10) \times 0.3 = 13$. The parameter exposes a trade-off. On the one hand, the relaxation becomes more conservative, avoiding potential performance drops. Too much tightening, however, might result in an increased number of fails and the cost of maintenance. On the other hand, loosing the relaxation decreases the number of replays, but might result in increased cost of the search itself. Our experiments showed that the cost of the search usually considerably outweighs the cost of the maintenance, and decreasing the value of *RRD* might considerably speed-up some queries without slowing down the others. This parameter does not change the result and is purely performance-related.

Saving function states at fails. When recording a fail, we store enough information to replay the fail later. However, some $f_c()$ functions might have additional information computed. For example, *max*() might store support coordinates for its $[a', b']$ range, which might allow us to avoid recomputing the function at other nodes of the tree. We extended the Searchlight API with the ability to serialize such information and save it when recording a fail. During the replay this information is restored. The optimization provides significant performance benefits in the presence of a large number of fails with expensive functions.

Sorting the Validator queue on ***BRP*****.** When candidate solutions are received by a Validator, they are put into its FIFO queue, which does not take candidate *BRP* values into account. However, candidates with better *BRP* values might have a better chance of belonging to the final result. They also might help to decrease the *MRP* faster, which improves pruning at Solvers. Thus, we decided to use a priority queue ranked by *BRP* at the validators instead. While a priority queue is generally more expensive than FIFO, in practice performance benefits resulting from better pruning outweigh the queue maintenance costs, as supported by our experiments.

Speculative Relaxation. Before relaxing a query, we executed the *original* query until completion. Only then the recorded fails are replayed if needed. If the user does not mind intermediate results, relaxation can start when first fails are encountered. We call this *speculative relaxation* and provide it as an option.

Speculative relaxation is done by additional CP solvers replaying fails concurrently with the main execution. This is done only when the Validators are idle, so that relaxed candidates do not interfere with the main search. Speculative Solvers still consume

CPU resources, which might slow down the main Solvers and increase query result latency for the user. At the same time, they might provide *relaxed* results much faster. The best use-case for speculation is probably when the user has little insight about the data, and expects an original query to fail.

4.3 Query Constraining

Query constraining deals with the problem of many results. That means it does not need to examine fails, since it is only interested in the results satisfying the *original* constraints, which failed sub-trees cannot contain. After the query begins execution, we consider only the query relaxation and tracks the fails. If the query produces at least k results, it turns off the relaxation, stops tracking fails and starts constraining the query to prune inferior results. The pruning is based on ranking supplied by the built-in or custom function, as described in Section 3.2. Effective pruning allows us to avoid discovering and ranking each result of the original query by eliminating entire parts of the search tree that cannot contain better results than already discovered.

Recall that we explore three possible points of pruning. The first one is Solver-based, at the search tree. When at least k results are found, Searchlight computes the Minimum result RanK (*MRK*) — the minimum rank $RK()$ among all the k results found so far. This is similar to the query relaxation's *MRP*. For a new result to belong to the top-k results its rank *RK* must exceed *MRK*. Similar to query relaxation, for each search sub-tree we compute *BRK* — the Best possible RanK, which is the maximum *RK* among all solutions that might be found in the sub-tree. This can be easily done by using Searchlight-provided synopsis estimations for constraint functions. If the *BRK* value for the sub-tree falls below *MRK*, it is pruned, since no results from that sub-tree can enter the top-k. The check is done by introducing a new *dynamic* constraint into the search: $BRK(r) \geq MRK$. The constraint is dynamic, since *MRK* is updated during the search, with new information coming from both local and remote Searchlight instances. It results in progressively better values for *BRK*, and, thus, better results. At the same time the pruning becomes progressively tighter, resulting in better performance. The *MRK* updates are performed by Validators, as in the case of *MRP*, since Validators produce final results. The cost of checking the dynamic constraint is negligent. We ensure it is done after the original constraints have been checked at the node, so all $f_c()$ functions have been already computed. Computing *BRK* itself just involves a small number of arithmetic operations.

Let us revisit the example from Section 3.2 with the same C^c and parameters. Let us assume at some search node $c_1 \in [100, 190], c_2, c_3 \in [100, 200]$. Then, for the sub-tree $BRK = 1 - \frac{1}{3}(\frac{10}{50} + 0) = 0.933$. If, for example, $MRK = 0.8$, the search will continue for the sub-tree. However, if at some node in the sub-tree $c_1 \in [100, 180], c_2, c_3 \in [100, 150]$, the *BRK* becomes $\frac{1}{3}(\frac{20}{50} + 2\frac{50}{120}) = 0.589 < MRK$. Thus, the sub-tree is pruned.

When a leaf of the search tree is reached, the corresponding candidate is sent to the Validator, which does checks similar to the query relaxation, but without relaxing any constraints. It takes into account the $BRK(r)$ value of the candidate and updates the global *MRk* (if r enters the top-k).

Skyline computation (see Section 3.2) is done similarly to the scalar query constraining. It is implemented by introducing another dynamic constraint for the result. However, instead of checking the scalar *MRK* value at every node, the constraint compares the estimated $f_c()$ intervals with the current skyline. If the sub-tree is dominated by the skyline, it is pruned. Otherwise, we keep traversing the tree and passes the candidates to the Validator.

The same correctness and complexity argument as for the relaxation applies to the constraining as well, since it is performed as CP search.

5 EXPERIMENTAL RESULTS

We performed an extensive experimental evaluation of the proposed techniques. The main part of the evaluation consisted of measuring the benefits of using our approach against the only alternative available to the user — manual relaxation/constraining. Additionally, we wanted to make sure these features do not bring any significant overhead to the existing query processing. Another important part of the evaluation was to measure the benefits of our optimizations from Section 4.2. Note that existing query refinement solutions are designed for traditional database systems running SQL over relational data, and are not readily applicable to constraint-based search queries over multidimensional data, which we study here. Finally, a head-to-head comparison of Searchlight with a pure DBMS approach is available elsewhere [10] for both synthetic and a real-world data sets. The same paper [10] also argues about the prohibitive complexity of formulating CP queries in relational terms.

We measure the benefit of our approach in terms of query latency. While the quality of the final results might be another measure, such a measure would be governed by the user via the ranking/penalty functions and, thus, can be considered as immutable for our purposes.

Performing a user study would be important to measure the usability of our approach. Such a study would allow us to measure user satisfaction with the quality of the refined results and to more accurately account for time savings for the user. This works was primarily directed at the design and implementation part of the framework. It also lacks a GUI component, as well as a realistic workload to perform a meaningful user study. We leave a user study for future work.

All experiments were performed on a four-instance Searchlight Amazon AWS cluster. The cluster consisted of c4.xlarge machines running Debian 8.6 (kernel 3.16) with 7.5GB of memory. We used two data sets. The first one was a synthetic data set from the original Searchlight paper, 100GB total size. The second data set was a part of the MIMIC II [1] waveform data for the Arterial Blood Pressure (ABP) signal. The data set size was 100GB as well. These data set are representative of general Searchlight workloads: the synthetic one introduces areas of varying function amplitudes, while the MIMIC provides real-world distribution. On a side note, the data sets choice plays secondary importance to the queries, since our approach relies on the efficacy of the *general* Searchlight search process and should work for all data sets that can be handled efficiently by Searchlight.

We used a variety of queries to perform the experimental evaluation. However, to provide concise and meaningful presentation, we discuss different aspects of our approach with the help of two characteristic queries for each data set. By default, we assumed the user's cardinality requirement of 10 results. The queries were as follows:

- **S-SEL** (from Synthetic SELective) is an empty-result query for the synthetic data set. Being maximally relaxed it becomes a non-empty, but very selective query.

- **S-LOS** (from Synthetic LOoSe) outputs empty result initially. However, the maximally relaxed version is very loose, outputs a very large number of results, and does not allow the search process to perform much pruning.
- **M-SEL/LOS** (from Mimic SELective/LOoSe). This are the MIMIC versions of the queries above

Semantically, the M-SEL/LOS correspond to the running example from Section 2, They contain exactly the same variables and constraints, but different parameters (domains and thresholds). Thus, we do not repeat the query constraints here. S-SEL/LOS have the same constraints (function amplitude and neighborhoods), but with synthetic attributes from the generated data. So queries basically look for certain "spikes" in the data, where a spike is determined by comparing the resulting intervals with the neighborhood.

The main idea behind choosing selective and loose types of queries is the observation that a selective query allows the user to perform manual relaxation without significant performance penalties. The user just have to relax the constraints maximally, and the query still finishes in a reasonable amount of time. The user then can choose the best 10 results. A loose query, however, being maximally relaxed outputs an avalanche of results, which results in significant latency. Such over-relaxation might be quite costly in practice. Since the user cannot easily predict the selectivity beforehand, the system should be able to handle both types of queries automatically.

We used the maximally relaxed versions of the queries above to measure the performance of the query constraining, since they output more than 10 results. As in the case of relaxation, the selective queries' results can be ranked manually. For the loose query this is infeasible.

5.1 Query Relaxation

We measured the benefits of the automatic relaxation over the manual approach, in which the user would be forced to guess the correct query, possibly in several iterations. This manual relaxation scenario is the only alternative available to the users and exactly the case we want to avoid in practice, hence it was important to compare it with our solution. The manual approach was performed using Searchlight as well, so the search engine remained the same. We will often refer to the automatic approach as just Searchlight. We studied the following scenarios:

- **USER-3**. This is a common user scenario. The original query gives an empty answer. Then the user relaxes it in a cautious way, several times, and gets the required number of results on the second try. Thus, she comes through 3 iterations (hence, the name). In practice the number of iterations might be much larger, due to a large number of relaxation possibilities, but three iterations was enough to demonstrate our point.
- **USER-2**. In this scenario the user guesses the query correctly from the first try, for the total of 2 iterations. Note, this scenario is quite infeasible in practice, and can be seen as an oracle-based approach, in which the user immediately knows the correct relaxation. This approach establishes an important baseline.
- **USER-MAX**. This is the scenario in which the user just relaxes the query maximally after the original query fails. Depending on the query, this might perform like the oracle approach above (e.g., for selective queries) or just start outputting a large stream of results without any means of pruning (for loose ones). In the latter case the user would have to stop the query and guess further, since such queries might easily take hours to finish.

First, we provide query completion times for the selective queries S-SEL and M-SEL under different scenarios described above. The results are illustrated in Table 1, where the "SL" column corresponds to the automatic Searchlight relaxation approach discussed in the paper. For the USER-2 scenario in the parenthesis we specify the completion time of the second, correctly relaxed, query. For these queries the USER-2 and -MAX scenarios are basically equivalent since there is no much penalty in relaxing the query maximally.

Table 1: S/M-SEL query completion times (secs) for query relaxation.

Query	SL	USER-3	USER-2	USER-MAX
S-SEL	97	327	210 (120)	216
M-SEL	150	544	380 (240)	380

As can be seen, even comparing with the USER-2 approach Searchlight provided considerable performance gains. They come from two sources:

- Searchlight does not need to re-explore the already traversed parts of the search tree. After the main search is finished, it can concentrate only on the previously unexplored (i.e., failed) parts. This is in contrast with any of the manual approaches, which have to start every query iteration from scratch.
- When relaxing the query, Searchlight is able to provide additional pruning based on the best results found so far. While for selective queries it is not necessarily the game changer, it has a much more pronounceable effect for loose queries, which we show later.

We also measured the time it took Searchlight to obtain the first result. For the S-SEL queries it took Searchlight 42 seconds in the SL approach versus 91s seconds for the USER-2 (the best) approach. The corresponding times for the M-SEL query were 45 and 198 seconds. We saw similar trends across all queries we ran. The results come as no surprise, since Searchlight is able to start the relaxation right away without restarting queries with new parameters.

When it came to the overhead of the query relaxation approach itself, it did not exceed 5 seconds for the synthetic and 3 seconds for the MIMIC queries. This overhead mainly came from assessing and recording the fails.

Table 2 provides the corresponding results for the loose queries. For the "Max" case, we stopped the query after 1 hour (hence the > symbol in the table), since this was enough to demonstrate our point.

Table 2: S/M-LOS query completion times (secs) for query relaxation.

Query	SL	USER-3	USER-2	USER-MAX
S-LOS	105	314	208 (106)	>3600
M-LOS	91	177	118 (83)	>3600

This experiment shows the same trend as for the selective queries with a single exception: the USER-MAX and USER-2 behaved differently. When the user relaxed the query maximally, it ran for a very long period of time (we stopped it after 1 hour) due to the very large number of results. In practice, the user cannot just stop the query and rank the currently found results, since she is not guaranteed to find the top-k among them. However, Searchlight guarantees correct top-k results. Since the maximal relaxation is out of the question, without Searchlight the user would have to continue the guessing game of scenario USER-3 but with potentially more iterations and longer times.

Searchlight outputs the first result in 92 and 45 seconds for S-LOS and M-LOS, respectively. The corresponding times for the USER-2 were 108 and 77 seconds. This is the same trend as we discussed for the selective queries. The auto relaxation overhead remained at comparably low levels: 15 seconds for S-LOS and 1 second for M-LOS.

The next experiment measured the overhead of the auto relaxation for the queries that do not need it. We wanted to explore the possibility of keeping the relaxation always on, without the user turning the knob. For this experiment we ran the second query from USER-2 with the relaxation turned on. The results are given in Table 3.

Table 3: Query completion times (secs) for queries not needing relaxations.

Relax	S-LOS	M-LOS	S-SEL	M-SEL
Off	106	83	120	240
On	116	98	127	290

As can be seen, turning on the auto-relaxation does not have any significant impact on the query completion times. The notable exception is M-LOS query, for which Searchlight submitted a lot of relaxed candidates to the Validator before 10 results were actually found. However, this overhead was the largest we saw for a large number of queries we ran. Even with such an overhead the auto-method would be quite helpful for the user, since the relaxed candidates are output to her as useful feedback. At the same time, the time to first result did not change significantly for all queries, which means the interactivity was not hampered, and the overhead was limited to the total completion time.

5.2 Query Constraining

In the absence of automatic query constraining, the only option is to run the query until completion and then rank results at the client. While this might work for queries with small number of results, it is very inefficient for queries returning a lot of them. In addition, the manual approach misses significant pruning opportunities. Our main results are shown in Table 4. "Off" means no constraining, which is equivalent to the manual approach; "Rank" means scalar ranking automatic constraining, and "Skyline" — vector domination constraining. Both approaches were described in Section 3.2. By default we specify times in seconds, and we use 'h' and 'm' symbols to denote hours and minutes.

Table 4: Query completion times (secs) for query constraining.

Method	S-LOS	M-LOS	S-SEL	M-SEL	M-SEL'
Off	2h 8m	2h 24m	120	240	263
Rank	60	154	29	139	135
Skyline	314	13m	93	269	218

The loose S/M-LOS queries could not even finish in a reasonable time. These queries actually were outputting results with very low latency during the execution. However, since constraints were loose, they created an avalanche of such results without the ability to prune. To guarantee the top-10 results the user would have to stop the query and constrain it by hand, possibly in several iterations.

At the same time, for the same queries Searchlight provided considerable performance gains, coming from pruning both at Solvers and Validators, as we described in Section 4.3. This was especially evident for the rank-based constraining. The skyline constraining was less effective, with respect to the query completion time. However, comparing with the manual "Off" approach, the performance benefits were considerable. The reduced efficacy can be attributed to the nature of skyline — it is harder to prune interval-based search nodes at Solvers and candidates at Validators.

The selective queries S-SEL and M-SEL allowed us to measure the constraining benefits for the queries for which the client-based filtering is a viable alternative due to their reasonable completion time. In most cases our approach resulted in considerable gains. The M-SEL query is somewhat of an exception, for which the skyline approach performed worse than the "Off" approach. This can be attributed mostly to some overhead from the skyline based checks during pruning (without any benefits) and slightly different rebalancing of the candidates between Validators. The last column of the table (M-SEL') provides results for another selective MIMIC query. It can be seen that both rank and skyline auto approaches provided significant improvements for query completion times, so the M-SEL case should not be considered a trend for selective queries.

When it comes to the overhead of the automatic approach, it is kept at the minimum. Actually, it is smaller than that for the query relaxation since it does not need any maintenance similar to tracking of failed search nodes. As for the Solver- and Validator-level checks, they are quite cheap for the rank-based constraining, being in-memory algebraic comparisons. For skyline-based constraining the checks are somewhat more expensive, and they must be active all the time, from the beginning of the query. However, the checks can be done quite efficiently using the variety of existing methods for skyline computation. This problem is well-researched, for example, for relational skylines. The overhead in this case is basically the cost of computation, which cannot be avoided for such a non-trivial constraint.

5.3 Query Relaxation Optimizations

In this section we describe experiments to measure the performance of the relaxation optimizations from Section 4.2.

Computing functions at fails. In this experiment we measured the difference between the two different strategies to compute functions $f_c()$ when catching fails. The first strategy, "Full", corresponds to fully evaluating all functions at the failed node. The "Lazy" corresponds to the lazy evaluation. Both strategies were described in Section 4.2.

The results are presented in Table 5 where the parenthesis times specify the times to first result, which is a reasonable measure of interactivity. While times to the first result might seem large, they include the completion time of the *original* query, which found no results at all. The optimization provided benefits for more expensive synthetic queries. At the same time it did not result in any overhead for all queries. We also ran additional experiments for more expensive MIMIC queries, for which we took the same M-SEL/LOS queries and increased their cardinality requirements from 10 to 200 results. We saw the significant benefits at the fail recording stage: for some queries the fail tracking overhead decreased from 30 to 15 seconds.

Table 5: Query completion and first result times (secs) for fail recording methods.

Method	S-LOS	M-LOS	S-SEL	M-SEL
Full	120(100)	81(45)	112(46)	149(45)
Lazy	105(90)	91(45)	97(42)	150(45)

Saving UDF states at fail recording. In this experiment we measured the impact of saving additional UDF information when recording fails. In contrast with the previous optimization, which just lazily postpones computation of some UDFs, this optimization allows us to avoid re-computation of some UDFs completely. The results are presented in Table 6 for query completion and first-result times (the latter is given in parenthesis).

Table 6: Query completion and first-result times (secs) for the UDF saving optimization.

UDF saving	S-LOS	M-LOS	S-SEL	M-SEL
On	105(90)	91(45)	97(42)	150(45)
Off	113(111)	104(70)	97(40)	154(46)

The optimization was especially beneficial for the loose queries. For the selective queries the benefits were not pronounced due to the structure of those queries. The replays were relatively cheap, involving less re-computation. As in the previous case, this optimization did not result in any overhead as well, at the same time allowing better performance in many cases. The memory footprint for the saved states depends on the functions. Standard aggregate functions use about 80 bytes per save for the two-dimensional data set (16 bytes for the range itself plus 64 bytes for the support coordinates for the min and max values).

Speculative execution. As can be seen from the experiments, the time to the first result is often quite large. This is a logical result for empty-result queries, since Searchlight first finishes the main, non-relaxed, query and only then tries to relax it. We support speculative relaxation as a means to start the relaxation sooner. The corresponding query completion and first result times are presented in Table 7.

Table 7: Query completion and first result times (secs) for speculative relaxation.

Speculation	S-LOS	M-LOS	S-SEL	M-SEL
On	128(7)	90(45)	115 (2)	152(47)
Off	105(90)	91(45)	97(42)	150(45)

As can be seen from the results in many cases speculative execution significantly improved times to the first result. We could not find suitable queries to demonstrate the same trend for the MIMIC queries. While the speculative Solver for those queries replayed some of the fails, they resulted in a small number of non-perspective candidates. As we discussed in Section 4.2, speculative Solvers are restricted to fails found by main Solvers so far.

As expected, the speculative relaxation has its own overhead coming from the consumption of CPU resources by the speculative Solver. For some queries the increase in the completion time was significant. We decided to run an additional experiment (not shown here), with one additional CPU thread available. As expected, the times for the speculative relaxation turned on and off were the same, which suggests the overhead is CPU related and cannot be trivially extinguished. Basically, the decision of trading off some completion time to faster interactive results is up to the user.

Partial relaxation during replays. This optimization addresses the issue of over-relaxing the query at a fail replay, when early fails might be relaxed in a very loose way because of loose estimations. As we discussed in Section 4.2, the *RRD* parameter ($0 \leq RRD \leq 1$), allows us to perform the relaxation in a more controlled way by enforcing tighter relaxation. The results of changing this parameter for the loose queries are presented in Table 8. We did not see any significant effect of the parameter to first result times.

Table 8: Query completion times (secs) for different *RRD* values.

Query *RRD*	0.1	0.3	0.5	0.7	1.0
S-LOS	106	105	106	106	106
M-LOS	87	91	112	145	54m

As can be seen from the table, the optimization resulted in gains only for some queries (we saw gains for other MIMIC queries as well). M-LOS query, for instance, fails almost immediately, and replaying its early fails with maximal relaxation effectively results in traversing most of the search tree. For S-LOS, on the other hand, search fails are relatively deep in the search tree, so maximal relaxation does not cause significant increase in the number of visited search nodes. In general, the benefits of the optimization depend on the nature of the search tree, which in turn depends on the query. At the same time, it does not introduce any overhead, which allows us to keep it always on. We saw a slightly elevated number of fail recordings and replays, but not significant enough to cause any drop in performance.

Additional experiments with the fail and candidate queues. We extended the Validator to sort the candidates on the *BRP* value. This in general might allow the Validator to identify better relaxed results faster, which in turn results in better *MRP* values and more effective pruning. We performed the experiment over a number of queries with different cardinality requirements to vary the number of candidates at runtime. For some queries, we saw 8-12% improvement in total completion times and no major impact on the first results.

We also measured the benefits of our fail-based approach presented in the paper against simply continuing the search "through" the fail. The latter would still involve relaxing constraints, but no fail recording (and later replaying) would be made — the search would be immediately resumed from the point of failing. One reason to do that would be to simplify the approach and decrease the memory overhead. However, we claim it would result in a sub-optimal approach, where the *utility* of the fails is not taken into consideration. Our experiments supported this claim. When we replayed the fails in the order they were encountered, simulating the immediate search resume, we did not see any improvements in the completion or first result times. Moreover, for some queries the completion times increased up to several orders of magnitude. For example, for S-LOS the time increased from 105 seconds to 56 minutes. We believe these results emphasize the necessity of a utility-based approach.

6 RELATED WORK

Query relaxation [12, 14–16] deals with the empty-answer and too-few-answers query problems by relaxing the original query constraints. The past work on query refinement can be studied under two broad categories. The first includes relaxation based on some statistics readily available in the database. For example, the Stretch-and-Shrink (SnS) framework [14] uses query cardinality estimations via precomputed samples and then uses the estimations to find relaxed ranges for each range-based query constraint independently. The framework heavily relies on fast cardinality estimations. Multiple estimations might have to be made at every step of the interactive refinement. Another framework [4] uses histograms to produce cardinality estimations and derive proper constraint ranges for the query. The results presented to the user are ranked based on the distance (e.g., Euclidean) from the original constraint ranges. There is also the possibility of using probabilistic [15] and machine learning [16] frameworks to produce relaxations. These methods, however, still rely on statistics to provide probabilistic estimations for the relaxation decision or the learning stage to understand the rules hidden inside the data. On the other hand, Searchlight targets queries for which the cardinality is not known beforehand. It would be also hard to estimate properly due to a large search space and possible complexity of query constraints. The results are also not known, and might be expensive to find, which makes the learning stage or probabilistic estimations infeasible.

The other category includes methods that use indexes to relax constraints. One such approach [12] is to relax join and selection predicates, and obtain the relaxation *skyline*. These methods have limited applicability for Searchlight, since results cannot be indexed beforehand. Also Searchlight generally works with regions (subsets) instead of single tuples, and the search space itself depends on the query constraints. Additionally, query constraints might be more complex than ranges, potentially referencing data outside of regions (e.g., the neighborhood constraints in the query from the Introduction). This makes R-trees (or other traditional index trees) generally ineffective for traversal and pruning.

The too-many-results problem creates the dual problem of *contracting* the query. These methods [4, 14] generally use precomputed statistics to make fast cardinality estimations and find suitable ranges for query constraints. The corresponding frameworks usually handle both relaxation and contraction at the same time. Another approach is to get rid of excessive answers by ranking and outputting only the best few. The "best" can be based on a scalar ranking function (top-k queries) or vector domination (skyline [5] queries). These methods commonly rely on traditional precomputed structures, such as views [6, 8] and R-trees [17, 18]. The view-based approaches require advance knowledge of at least a part of the workload to materialize proper views. The R-tree approaches traverse the tree and perform MBR-based pruning. If such structures are not readily available, the only option is to perform a sequential scan and either build the required structures or do the processing during the scan (e.g., sorting [3, 7], batch computation [3] or building structures optimized for particular queries [17]). For Searchlight, no indexes are available beforehand, thus we performed a comprehensive search. This might seem similar to the sequential scan-based approaches, however, as we argued, the nature of the constraint-based queries is different and, thus, requires novel approaches.

7 CONCLUSION

Fast query execution is necessary but not sufficient for effective interactive data exploration. Users often go through multiple query iterations to identify a reasonable set of results that matches their goals. It is thus critical for the underlying system to aid the users, optimizing for human labor, time and attention, to maximize user productivity.

We introduce dynamic and automatic refinement of constraint-based search queries, based on user-specified target result cardinalities. When relaxing a query, we guarantee optimal results according to user-specified distance functions. When constraining, we output the top results according to a ranking function. Unlike previous solutions, our approach does not require any pre-computed indexes nor does it require result cardinality estimations, which might be extremely hard to obtain accurately for queries with complex constraints. Our approach instead alters the constraints during the run-time. Our techniques naturally fit in and can effectively leverage the common features of CP platforms and DBMSs. Furthermore, our approach can explore more promising parts of the search space first, which considerably improves pruning and, consequently, provides better interactivity and query completion times.

Our approach provides significant performance benefits in comparison with the tedious and inefficient manual approach. At the same time, it incurs negligible performance overhead, even for queries that end up not needing any refinements.

REFERENCES

[1] [n. d.]. MIMIC II Dataset. https://mimic.physionet.org/. ([n. d.]).
[2] [n. d.]. SciDB. https://www.paradigm4.com/. ([n. d.]).
[3] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. 2001. The Skyline Operator. In *ICDE*. 421–430.
[4] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2002. Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. *ACM Trans. Database Syst.* 27, 2 (June 2002), 153–187.
[5] Michael J. Carey and Donald Kossmann. 1997. On saying "Enough already!" in SQL. *SIGMOD Rec.* 26, 2 (1997), 219–230.
[6] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. 2006. Answering Top-k Queries Using Views. In *VLDB*. 451–462.
[7] Ronald Fagin, Amnon Lotem, and Moni Naor. 2001. Optimal Aggregation Algorithms for Middleware. In *PODS*. 102–113.
[8] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. 2001. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In *SIGMOD*. 259–270.
[9] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. 2014. Interactive Data Exploration Using Semantic Windows. In *SIGMOD*. 505–516.
[10] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. 2015. Searchlight: Enabling Integrated Search and Exploration over Large Multidimensional Data. In *VLDB*. 1094–1105.
[11] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. 2016. Interactive Search and Exploration of Waveform Data with Searchlight. In *SIGMOD*. 2105–2108.
[12] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. 2006. Relaxing Join and Selection Queries. In *VLDB*. 199–210.
[13] Gang Luo. 2006. Efficient Detection of Empty-result Queries. In *VLDB*. 1015–1025.
[14] Chaitanya Mishra and Nick Koudas. 2009. Interactive Query Refinement. In *EDBT*. 862–873.
[15] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. 2013. A Probabilistic Optimization Framework for the Empty-answer Problem. *VLDB* 6, 14 (2013), 1762–1773.
[16] Ion Muslea. 2004. Machine Learning for Online Query Relaxation. In *SIGKDD*. 246–255.
[17] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2005. Progressive Skyline Computation in Database Systems. *ACM Trans. Database Syst.* 30, 1 (March 2005), 41–82.
[18] Man Lung Yiu and Nikos Mamoulis. 2009. Multi-dimensional Top-k Dominating Queries. *The VLDB Journal* 18, 3 (2009), 695–718.

Micro Analysis to Enable Energy-Efficient Database Systems

Chen Yang[1], Yongjie Du[1], Zhihui Du[2], Xiaofeng Meng[1]
[1]School of Information, Renmin University, China
[2]Department of Computer Science and Technology, Tsinghua University, China

ABSTRACT

CPU has been identified as the energy bottleneck for database systems and existing approaches only allow database systems to trade the performance for energy. However, our work show that cutting down the energy cost of database systems without losing the CPU performance is feasible. We first develop a measurement methodology to accurately evaluate the energy cost of different CPU micro-operations. Then three popular database systems with different setups and data sizes are used as benchmarks to explore the energy distribution on CPU micro-operations. Our experimental results show that L1 data cache (L1D cache) consumes 39%-67% of total CPU energy and it is definitely the energy bottleneck of database systems. This finding inspires us a novel idea on building energy-efficient database systems with customized CPU architecture that features low L1D cache energy cost. A proof-of-concept system is developed to evaluate this idea and the experimental results show that our solution can not only achieve 60% of peak energy saving but also gain further performance improvement.

1 INTRODUCTION

As the infrastructure of the data center, database system has been limited by the energy wall. The energy cost of powering the database server is not only rapidly approaching the machine acquisition cost[24], but the energy wall also limits the scalability of database server. CPU[13, 19, 25], main memory[23] and disk[24] may become the energy bottleneck due to different computer architectures and database types. In this paper, we focus on the energy profiling of typical relational database systems on the x86_64 architecture with local disk, and many evidences have confirmed that CPU consumes more power than other major components in our scenario[13, 19, 25].

According to whether the workload is running or not, the CPU energy cost may be classified into Busy-CPU energy cost and Idle-CPU energy cost. The Idle-CPU energy cost has been reported to reduce from 50% to 18%[19]. Undoubtedly, the Busy-CPU energy cost as the dominant part attracts a lot of research work. For example, both academia and industry have expended a great deal of effort in energy-oriented query optimization[21, 28, 29] and the external energy knobs based approaches[13, 19, 21, 23]. The basic idea is building the cost model based on the Busy-CPU energy cost to choose the energy-optimized query plan or set the appropriate CPU voltage and frequency according to the database load status. These methods consider CPU as a black box and the energy cost is reduced by trading the performance, such as a 43%-80% performance loss[21, 28].

Actually, there are many different micro-operations inside the CPU and they expect different energy costs. Existing black-box optimization methods cannot take advantage of micro energy cost characteristics of database workloads so they often lead to significant performance loss to meet the energy saving demand.

Energy-efficient database systems expect that the CPU architecture can significantly cut down the Busy-CPU energy cost. To achieve this object, an in-depth breakdown of Busy-CPU energy cost is indispensable. It cannot only help us identify the energy bottleneck on CPU, but it is also the basic work to design a novel customized CPU architecture for energy-efficient database machine. Breakdown analysis of Busy-CPU energy cost will help to answer some important questions such as what is the microscopic distribution of Busy-CPU energy cost and how different database implementations and settings affect this distribution.

In this paper, we design micro-benchmarks to breakdown the Busy-CPU energy cost of typical read query workloads, and enable the energy-efficient database system on the customized CPU architecture. Unless otherwise specified, the query will refer to read query. It is very difficult to achieve an accurate breakdown of energy cost on a real database system. The micro-operations which can be monitored are so many e.g., about 514 events inside our CPU that we cannot evaluate the energy cost for every micro-operation. To isolate the energy cost of an individual micro-operation, we need to overcome many mutual related factors, such as the compiler optimization and architectural features. In addition, to enable energy-efficient database systems, we have investigated many CPU architectures and updated the kernels of both operator system and database system to make them support the customized CPU architecture well. Finally, we present a clear energy cost distribution pattern of database systems and give a proof-of-concept system to cut down the energy cost without losing the performance. The major contributions are as follows.

- A micro analysis based accurate energy breakdown method is proposed for the Busy-CPU energy cost with typical query workloads.
- The energy bottleneck L1D cache is identified based on extensive experiments on three typical database systems.
- The L1D energy-efficient CPU architecture design is proposed and the experimental results show that 60% of peak energy saving can be achieved with further performance improvement.

The rest of the paper is organized as follows. Section 2 presents our evaluation approach and the energy cost of micro-operations. Section 3 profiles the energy cost of database systems along with a detailed analysis. Section 4 presents our proof-of-concept system design, optimization approach and evaluation results. Section 5 analyzes the energy cost preference of some typical scenarios. Section 6 describes the related work. Section 7 summaries our work and presents directions for future work.

2 MICRO ANALYSIS METHOD FOR BUSY-CPU ENERGY

In this section, we will present our methodology on how to break down the Busy-CPU energy cost into the energy cost of different micro-operations for queries in relational database systems.

Corresponding author is Xiaofeng Meng.

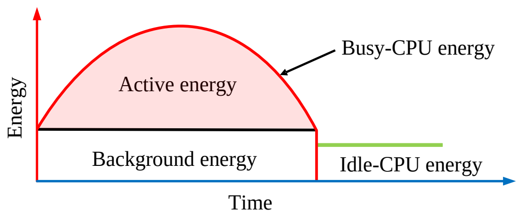


Figure 1: Example of energy cost along the workload running.

2.1 Key Idea

Depending on whether the CPU is on operational states or idle states, we name the total energy cost as Busy-CPU energy cost and Idle-CPU energy cost, shown in Figure 1. When fixing the CPU frequency and voltage, we further divide Busy-CPU energy cost into the Active energy cost and the Background energy cost, defined as

$$Busy\text{-}CPU\ \ energy = Active\ \ energy + Background\ \ energy.$$

When running a workload, the Active energy is the real cost used for the calculation and data movement and the Background energy is the fixed cost to activate the hardware. Obviously, the Active energy cost can reveal the power usage of a workload so that it is our profiling target.

We first formalize the Active energy cost as the sum of the energy cost of micro-operations. Next, we mainly solve three issues. (1) We have to identify which micro-operations to be profiled. The query workloads are typically data-intensive. This motivates us to pay an attention to the data movement related micro-operations, such as the cache load. When we know the executed counts of these micro-operations and the energy cost driving them once, we can actually evaluate their energy cost. (2) How to quantify the energy cost of an individual micro-operation. We construct many micro-benchmarks, each of which shows a simple performance behavior issued by the specific micro-operations. Then, we also build energy models to map the energy cost of micro-benchmarks into the energy cost of an individual micro-operation. (3) How to verify the accuracy of the energy cost of an individual micro-operation. We have to construct the other micro-benchmarks which have the clear and complex performance behaviors. We can identify the difference between the estimated energy cost and the measurement value to take the verification. We will give a detailed introduction focusing on the above three issues.

2.2 Problem Formalization

We denote the analyzed micro-operation set as MS and then formalize the Active energy cost E_{active} for the workload w as

$$E_{active}(w) = E_{other}(w) + \sum_{m \in MS} E_m(w), \tag{1}$$

where $E_m(w) = N_m(w) \times \Delta E_m$ is the energy cost of the micro-operation m. ΔE_m is the energy cost driving the micro-operation m once and N_m is the count executing the micro-operation m and E_{other} is the unisolated energy cost, including the calculation, L1I cache and TLB, etc. According to Eq. (1), we have to solve the ΔE_m and N_m. Noting that our energy breakdown model is based on the stable voltage and CPU frequency. The dynamic voltage and frequency scaling (DVFS) will cause fluctuations on ΔE_m, so we disable DVFS during the testing. However, we can also evaluate the impact of different voltages and CPU frequencies. To solve Eq. (1), we must construct the set MS and evaluate the ΔE_m of every micro-operation in MS.

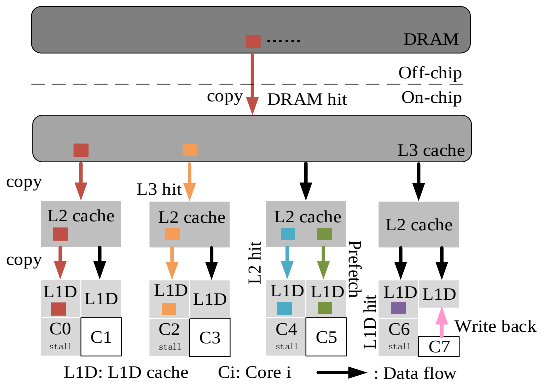


Figure 2: Example of data movement on the modern CPU architecture. The CPU core in the white box is busy and it in the gray box is stall.

2.3 Energy Breakdown Representation

Generally speaking, the query workload is usually data-intensive, so that we tend to choose data movement related micro-operations to characterize the energy cost. For the modern CPU architecture, the data is frequently moved between main memory and registers. In order to cross the memory wall, the CPU (e.g., Intel i7-4790 in our experiment) contains a three-level cache memory sub-system, shown in Figure 2. The closer the cache memory is to the CPU core, the smaller, faster and more energy-efficient it will be. We describe three ways of data movement that have the great impact on query workloads.

Regular data fetching. If the data is not in registers, the load instruction will fetch the data of the cache line size (e.g., 64 Bytes) and the CPU core could stall to wait for the data return. Noting that if instructions are uncorrelated with each other, speculation and out-of-order execution can disable the pipeline bubble. In addition, data fetching follows a step-by-step replication strategy. The CPU will first fetch data from L1D cache. If L1D cache hits (L1D hit), the data will be loaded into the register, like Core 6 in Figure 2. Otherwise, the CPU will go to the next level cache to search for data, called as L1D cache miss (L1D miss). Specifically, when L1D cache misses, L2 cache starts to search for data. There are also two cases: hit and miss. If L2 cache hits, the data will be copied to L1D cache first, and then copied to the register, like Core 4 in Figure 2. Similarly, L3 hit and DRAM hit are like Core 2 and Core 0 in Figure 2. Although the step-by-step replication strategy can provide the good data locality, the data movement leads to much energy cost.

Data prefetching. In order to improve the CPU performance, the data prefetching technique is used to predict the data usage under the background and fetch it without the pipeline bubble, like Core 5 in Figure 2. So, the data prefetching will also cause the data movement behavior. Four prefetching techniques are provided in Intel i7-4790[3]. Two are implemented through L1D hardware prefetcher to replicate the data into L1D cache in advance. Unfortunately, they cannot support the performance counter in Intel i7-4790. The two other types of prefetches that can be generated by the L2 hardware prefetcher – prefetches into the L2 cache (L2 prefetching) or prefetches into the L3 cache (L3 prefetching). Their behaviors can be monitored by the performance counter. In our paper, the data prefetching only means the L2 hardware prefetcher.

Write-back. Although the query is the read-only workload, lots of temporary data, e.g., local variables, have to be created and updated. The store operation always updates the data into L1D cache within 1 cycle. Because the local variables do not need to be eventually persisted so that they are rarely written to the lower level memory due to the write-back strategy. For example, L1D cache store hit rate is 99.86% in our experiment. So, we also evaluate the store operation which writes the data into L1D cache.

Based on the above analysis, we define the micro-operation set MS as

$$MS = \{L1D, Reg2L1D, L2, L3, mem, pf, stall\}$$

for the query workload. $\forall m \in \{L1D, L2, L3, mem\}$ is that a load operation reads the data from m to the next higher level memory. For example, $m = L2$ means to load the data from L2 cache to L1D cache. $Reg2L1D$ means that the store operation writes the data from the register to L1D cache. We combine two different types of prefetches generated by the L2 hardware prefetcher together as a micro-operation pf meaning to prefetch data. The micro-operation $stall$ is the stall event due to memory access. Recalling the Eq. (1), we next evaluate N_m and ΔE_m.

The energy breakdown of update/write queries is a totally different problem from the read queries. We need to know how to write the data into main memory. It may involve more micro-operations about writing. We do not discuss it in depth in this paper.

2.4 Micro-Operation Counting

The modern processors have built-in performance monitor unit (PMU) to record the performance related events. We can use Linux Perf[5] or ocperf[6] to get them from PMU to evaluate the micro-operation count N_m. For $\forall m \in \{L1D, L2, L3, pf\}$, it is worth noting that N_m is the sum of both the hit count and the miss count due to the step-by-step replication strategy. N_{pf} involves the L2 prefetching count N_{pf}^{L2} and the L3 prefetching count N_{pf}^{L3}. Especially, N_{mem} is the miss count of L3 cache. $N_{Reg2L1D}$ is the hit count when writing the data into L1D cache. N_{stall} is the stall cycle count due to the data load.

2.5 Energy Evaluation of Micro-Operation

To quantify ΔE_m, we design a set of micro-benchmarks which can achieve the specific performance behavior, such as only accessing L1D cache, etc. Finally, we can use the Active energy cost of micro-benchmarks to evaluate the ΔE_m of different micro-operations.

2.5.1 Micro-Benchmark Design. Isolating memory access to only follow a specific performance behavior is a challenging task in modern processors. The design of the micro-benchmark methodology is inspired by the recent work[22]. The out-of-order execution, speculation execution and data prefetching have worked well in hiding memory latencies but at the same time make the energy cost benchmarking for an individual instruction difficult. In addition, the treading switching, DVFS and the compiler optimization could affect the evaluation accuracy. In order to accurately evaluate ΔE_m, we make a lot of effort on it. First, our micro-benchmarks should minimize the effect of CPU architectural optimization. Second, we must configure the appropriate runtime environment to reduce the error, shown in Section 2.5.3. Third, we must review the assembly code which is generated by the compiler to disable some compiler flags who will change the performance behavior. Our micro-benchmarks follow two design frameworks to skip out of the architectural optimization.

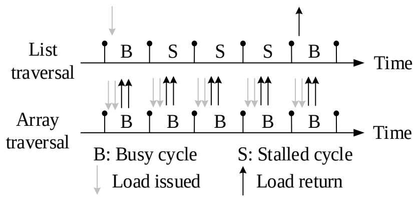


Figure 3: CPU execution behaviors when list traversal and array traversal only load data from L1D cache.

List traversal. We allocate a size of memory as an array Arr of pointers. The size of each item is 64 Bytes (i.e., cache line size) which can be processed by a load operation. We link every item as the list. Then, we traverse the list many times. Through the correct implementation, we can ensure that micro-benchmarks only load data from the specific memory layer. In this way, the energy cost of each micro-benchmark can be broken down into the energy cost of the specific micro-operations and stall cycle.

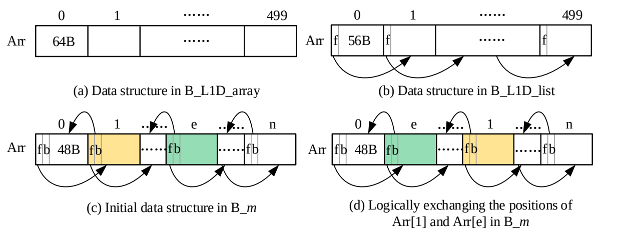
The list structure can make sure the data items have the back-and-forth dependency. Due to the access dependency, in the premise of disabling the perfetching the CPU does not know the address of the next data item until the previous item is finished. An example is shown in Figure 3. We assume the data in L1D cache and a L1D load requires 4 cycles from issue to return. Due to the unknown address of next data item, the pipeline is forced to break, leading to 1 load cycle and 3 stall cycles, i.e., L1D load energy and stall energy. The link traversal can disable the out-of-order execution and the pipeline to minimize the energy cost measurement error incurred by CPU architectural optimization.

Array traversal. The energy cost of stall cycle cannot be separated by list traversal. So, we design the array traversal framework. In it, the micro-benchmark allocates a size of memory as an array Arr (also 64 Bytes per data item), and then sequentially traverses it many times. Under some special conditions, CPU architectural optimization can make micro-benchmarks only load data from the specific memory layer without stall cycles.

As well known, the data item in array is completely independent and supports random access because the CPU knows the address of all data items before traversing. The load of next data does not have to wait for the finish of the previous data item. As shown in Figure 3, although a L1D load still requires 4 cycles, 3 stall cycles can be hidden due to the continuous pipeline. In addition, the Dual-Issue technique of Intel i7-4790 can issue two load instructions per cycle without stall.

2.5.2 Micro-Benchmark Set. For evaluating ΔE_m ($\forall m \in MS$), we build a micro-benchmark set MBS including 6 micro-benchmarks to show the specific performance behaviors as follows.

B_L1D_array. This micro-benchmark is used to evaluate ΔE_{L1D}, only accessing L1D cache without stall cycles. As shown in Algorithm 1, it follows the array traversal framework and allocates memory size smaller than or equal to the size of L1D cache. As shown in Figure 4a, if $S_{mem} = 32KB$ (i.e., L1D cache size in i7-4790), the Arr size is 500 and the size of each item is 64 Bytes. T is usually set into a huge value, such as 1 billion, to maintain



Arr
64B
(a) Data structure in B_L1D_array
56B
(b) Data structure in B_L1D_list
48B
(c) Initial data structure in B_m
(d) Logically exchanging the positions of Arr[1] and Arr[e] in B_m

Figure 4: Examples of data structures in different micro-benchmarks. B_m represents B_L2, B_L3 and B_mem. Pointer f points the next data item and pointer b points the previous data item.

a stable memory access pattern. A large T also applies to Algorithm 2-4. For balancing the performance, it can also be reduced moderately. So, all the data easily fit the L1D cache, so there will not be any miss after initial set of loads. Noting that we unroll the *Arr* and traverse it instead of loop traversal, it will avoid the effects of loop control statements as much as possible. By this programming optimization, 98.6% instructions are the desired load instructions.

Algorithm 1 B_L1D_array

```
Input: $S_{mem}$: allocated memory size; $T$: loop times;
1: allocate $S_{mem}$ memory size as an array $Arr$ with $\frac{S_{mem}}{64}$ items;
2: for iter $i$ in range $T$ do
3:     traverse $Arr$ through unrolling $\frac{S_{mem}}{64}$ times;
4: end for
```

B_L1D_list. The energy cost of this micro-benchmark mainly involves the load operation from L1D cache and the CPU stall. As shown in Algorithm 2, it follows the list traversal framework and allocates memory size smaller than or equal to the size of L1D cache, such as still 32KB. We also give an example to show such data structure in Figure 4b.

Algorithm 2 B_L1D_list

```
Input: $S_{mem}$: allocated memory size; $T$: loop times;
1: allocate $S_{mem}$ memory size as an array $Arr$ with $\frac{S_{mem}}{64}$ items;
2: partition each item into a pointer f (the first 8 Bytes) and the
   last 56 Bytes of data;
3: for iter $j$ in range $\frac{S_{mem}}{64} - 1$ do
4:     $Arr[j].f = \&Arr[j+1]$;
5: end for
6: for iter $i$ in range $T$ do
7:     use pointer f to traverse $Arr$ and unroll $\frac{S_{mem}}{64}$ times;
8: end for
```

B_L2, B_L3 and B_mem. They will be used to evaluate ΔE_m ($\forall m \in \{L2, L3, mem\}$). As shown in Algorithm 3, they follow the list traversal framework and only access the specific memory layers. When only accessing the memory layer m, the allocated memory size should be as close as possible to the sum of the m size and sizes of its higher caches to ensure that the majority of data is in m. Using B_L2 as an example, the allocated memory size should be close to 288KB (32KB L1D cache and 256KB L2 cache). Similar setup methods can be extended to B_L3 and B_mem. Noting that we do not use the linked list structure similar to B_L1D_list, the sequential load along physical position means the simple access pattern, which is easily employed by the modern CPU to improve the performance. However, it has the serious impact on profiling. For example, when setting $S_{mem} = 260KB$ for Algorithm 2, the L1D hit rate is 55%, so that there is no guarantee that only L2 cache is accessed. Thus, we randomize the access order (logical position) and generate jump access on a large span to break the data locality, as shown in Figure 4d. Because the low memory layer is far larger than the high memory layer, the data will usually miss in the high memory layer.

Algorithm 3 B_L2, B_L3 and B_mem

```
Input: $S_{mem}$: allocated memory size; $T$: loop times; $\varepsilon_{span}$: span
   threshold of two given data items;
1: allocate $S_{mem}$ memory size as an array $Arr$ with $\frac{S_{mem}}{64}$ items;
2: partition each item into a pointer f (the first 8 Bytes), and
   pointer b (the second 8 Bytes) and the last 48 Bytes of data;
3: for iter $j$ in range $\frac{S_{mem}}{64} - 1$ do
4:     $Arr[j].f = \&Arr[j+1]$;
5:     $Arr[j].b = \&Arr[j-1]$ $(j-1>0)$;
6: end for
7: for iter $z$ in range $\frac{S_{mem}}{64} - 1$ do
8: //avoid frequent exchange of logical neighbors when $e$ is
   always the same value.
9:     randomly pick $e \in [1, \frac{S_{mem}}{64} - 2]$ to satisfy $|z-e| > \varepsilon_{span}$
   and $Arr[e]$ is not the logical neighbor of $Arr[z]$;
10:    exchange the logical positions of $Arr[z]$ and $Arr[e]$;
11: end for
12: for iter $i$ in range $T$ do
13:    use pointer f to traverse $Arr$ and unroll $\frac{S_{mem}}{64}$ times;
14: end for
```

B_Reg2L1D. The energy cost of this micro-benchmark mainly involves the store operation from registers and the L1D cache. As shown in Algorithm 4, it only accesses the same variable repeatedly, but it is effective to ensure that the CPU only execute the store operation. This benchmark always access the same variable, the CPU can find it in registers, instead of reading it from L1D cache every time. In addition, the allocated memory size is large enough, so that the CPU has to perform multiple store operations to complete the assignment. Due to temporary variable assignment, the vast majority of store operations only involve L1D cache.

Algorithm 4 B_Reg2L1D

```
Input: $T$: loop times; $ut$: unrolling times;
1: allocate 64 Bytes of memory size as a variable $A$;
2: for iter $i$ in range $T$ do
3:     execute $p = A$ through unrolling $ut$ times; //variable
   assignment
4: end for
```

Our micro-benchmark set can achieve the specific performance behavior. Noting that it may be not the only way, but it has been enough accurate to help us profile the energy cost of database systems.

2.5.3 Runtime Configuration. The accurate execution of our micro-benchmarks depends on some runtime configurations to overcome the measurement error.

Compiler optimization. In order to minimize the impact of unnecessary instructions, our micro-benchmark set is compiled with an optimization level of -O3. The necessary temporary variables are added with a *volatile* modifier, such as a temporary pointer variable for linked list pointer tracking, which can cancel the compiler's active optimization to avoid the microscopic behavior changing.

Thread switching. In order to prevent the thread attached by the micro-benchmark from being switched between different idle CPUs during execution, the micro-benchmark will be fixed on a specific logic core to run.

DVFS knobs. EIST (Enhanced Intel SpeedStep Technology), an Intel DVFS implementation, changes CPU frequency and voltage to balance energy cost and performance. The energy cost of micro-components will increase with the improvement of CPU frequency and voltage. Because our micro-benchmarks are effective under the stable frequency and voltage, EIST technique will incur the measurement error of energy cost. We turn off these options and execute our micro-benchmarks under the given frequency and voltage according to our experimental requirement.

Prefetcher. The data prefetching could lead to the unexpected load instructions into our micro-benchmarks. So, the hardware prefetcher will be turned off by modifying MSR registers when running our micro-benchmarks. It will be turned on when evaluating the energy cost of query workloads.

Through running our micro-benchmarks, we can isolate the specific micro-operations and reduce most of measuring errors. It will lead to an easy solution to ΔE_m in next section.

2.5.4 Energy Model to Evaluate ΔE_m. We construct a series of energy models to map the energy cost of the micro-benchmark into ΔE_m. For any micro-benchmark $mb \in MBS$, we define the Active energy cost of mb as $E(mb)$ which will be given in next section. Here, we assume that $E(mb)$ is known to give the solution of ΔE_m.

Through running B_L1D_array which only loads data from L1D cache, we can solve the ΔE_{L1D} as

$$\Delta E_{L1D} = \frac{E(B\_L1D\_array)}{N_{L1D}}.$$

Recalling that N_{L1D} is the count loading data from L1D cache in Section 2.4. Similarly, when running B_L1D_list, we can solve the ΔE_{stall} as

$$\Delta E_{stall} = \frac{E(B\_L1D\_list) - E_{L1D}}{N_{stall}}.$$

For the micro-operations $x, y \in C = \{L1D, L2, L3, mem\}$, if the micro-operation x loads data from the higher memory layer than y, we denote $x > y$. When running the micro-benchmark B_L2, B_L3 and B_mem, respectively, we can solve the ΔE_m as

$$\Delta E_m = \frac{E(B\_m) - \sum_{i>m}^{i \in C} \Delta E_i N_i - E_{stall}}{N_m}. \tag{2}$$

Due to the step-by-step replication strategy, loading data from the low memory layer must also lead to a load operation from the higher memory layer. So, the load energy cost of the higher memory layer needs to be eliminated in Eq. (2). When running B_Reg2L1D, we can solve the $\Delta E_{Reg2L1D}$ as

$$\Delta E_{Reg2L1D} = \frac{E(B\_Reg2L1D)}{N_{Reg2L1D}}.$$

Prefetching energy. In term of the energy cost, the data prefetching and the regular data fetching are similar. we follow the assumption that the energy is mainly consumed in moving data from a specific memory layer to a higher layer [18] and set $\Delta E_{pf}^{L2} = \Delta E_{L3}$ and $\Delta E_{pf}^{L3} = \Delta E_{mem}$. ΔE_{pf}^{L2} is the energy cost of an individual L2 prefetching from L3 cache and ΔE_{pf}^{L3} is the energy cost of an individual L3 prefetching from main memory[4].

2.5.5 Verification Method of ΔE_m. For verifying the accuracy of ΔE_m, we propose a verification micro-benchmark set $VMBS$ derived from MBS, where each micro-benchmark shows a more complex performance behavior.

Micro-benchmarks in $VMBS$ essentially perform a series of data movement operations and data calculation operations to simulate the real workload. We first construct two micro-benchmarks B_add and B_nop. They only loop the known number of add and nop instructions to evaluate the ΔE_{add} and ΔE_{nop}. Next, we add the *add* and *nop* instructions into the micro-benchmarks in MBS to finally construct $VMBS$ including 7 micro-benchmarks shown in Table 3. For example, B_L1D_list_nop is to add the *nop* instruction into B_L1D_list.

When running a micro-benchmark $v \in VMBS$, Eq. (1) is used to solve the estimated Active energy cost $\overline{E}_{active}(v)$ where we set the $E_{other}(v) = \Delta E_{add} N_{add}(v) + \Delta E_{nop} N_{nop}(v)$. We measure the real $E_{active}(v)$ and define the accuracy as

$$acc(v) = 1 - \frac{|\overline{E}_{active}(v) - E_{active}(v)|}{E_{active}(v)},$$

where $acc < 0$, set $acc = 0$. If the acc is closer to 1, the estimated energy cost is closer to the real energy cost, showing that the ΔE_m got by our approach is accurate.

Through the above effort, we have got all of ΔE_m ($m \in MS$). Next, we introduce how to measure the Active energy.

2.6 Active Energy Evaluation

Our experiments run on the server with an Intel i7-4790 processor (L1D cache size is 32KB, L2 cache size is 256KB and L3 cache size is 8MB), 32GB DDR3-1600 main memory and a 500GB SATA hard drive. Our processor is popular to enable that our results are representative. The operator system is Ubuntu 14.04 including Linux Perf, ocperf and RAPL (Running Average Power Limit)[10].

We leverage RAPL to measure the energy cost. RAPL counters are highly accurate on x86_64 and allow us to separately measure the energy cost of the core domain $E(core)$, the package domain $E(package)$ (including the core, L3 cache and memory controller) and the main memory domain $E(memory)$. We can run an only-blocked program (e.g., sleep 1) to use its RAPL's measurement values as the Background energy cost of three different domains per second when disabling idle states (i.e., C-states[7]). For workloads which do not access L3 cache and main memory, we observe the $E(core)$ as the Busy-CPU energy cost, such as B_L1D_list and B_L1D_array. For the workloads which do not access main memory, we observe the $E(package)$ as the Busy-CPU energy cost. For other workloads, we observe $E(package) + E(memory)$ as the Busy-CPU energy cost. E_{active} is Busy-CPU energy cost minus the corresponding Background energy cost.

2.7 Selection of CPU Frequency and Voltage

Our micro-benchmarks set is usually used to evaluate ΔE_m under the fixed CPU frequency and voltage. However, the real workloads widely run with the EIST knob turned on to balance the performance and energy. So, in this section we will profile the performance of TPCH query workloads when turning on EIST

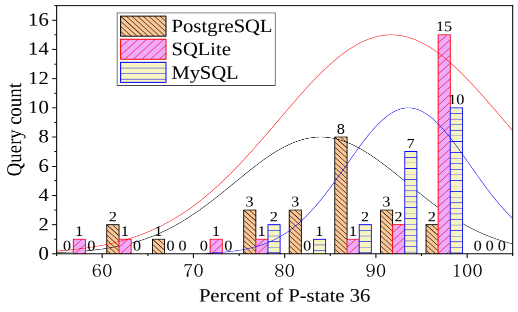


Figure 5: Query count distribution bars and the fitting distribution curves based on the percent of P-state 36.

Table 1: Runtime behaviors of micro-benchmarks

Micro-benchmarks	BLI	L1D miss%	L2 miss%	L3 miss%	IPC
B_L1D_list	98.9	0.01	-	-	0.26
B_L1D_array	99.5	0.01	-	-	2.02
B_L2	98.5	99.93	0.02	-	0.09
B_L3	98.6	98.68	99.98	0.01	0.03
B_mem	97.8	98.86	99.88	97.45	0.005
B_Reg2L1D	99.98	0.02	-	-	1.01
B_add	98.4	-	-	-	2.01
B_nop	99.87	-	-	-	3.99

Table 2: Energy cost of micro-operations at different CPU frequencies and voltages

P-state	36 (3.6GHz)	24 (2.4GHz)	12 (1.2GHz)
Micro-operations	Energy cost (nJ)		
ΔE_{L1D}	1.30	0.90 ↓	0.60 ↓
ΔE_{L2}	4.37	3.25 ↓	1.64 ↓
ΔE_{L3}, ΔE_{pf}^{L2}	6.64	5.91 ↓	5.33 ↓
ΔE_{mem}, ΔE_{pf}^{L3}	103.1	99.1 ↓	99.04 ↓
$\Delta E_{Reg2L1D}$	2.42	1.60 ↓	1.10 ↓
ΔE_{stall}	1.72	1.07 ↓	0.80 ↓
ΔE_{add}	1.03	-	-
ΔE_{nop}	0.65	-	-

to identify their preference for CPU frequency and voltage. That can help us evaluate a more reasonable ΔE_m.

EIST usually sets the CPU into different states to save energy[1], including C-states and P-states[7]. C-states are idle states {C0, C1, C2, ...}. C0 means the CPU non-idle and others mean that the CPU enters an idle state with different energy-saving levels. C0 can be further subdivided into different P-states. So, P-states can be called as operational states. Each P-state also has a different energy-saving level. We focus on P-states due to the profiling of Active energy cost. In truth, a P-state is both a frequency and voltage operating point. For the high CPU load, a high-performance P-state might be set, and vice versa. Intel i7-4790 includes 29 candidate P-states. CPU frequency of each P-state differs by 100MHz.

[1]The modern processor frequency can be separated into (1) core frequency involving ALU, L1 cache, L2 cache and etc, and (2) uncore frequency involving L3 cache, memory controller and etc. In this paper, the CPU frequency means core frequency. The uncore frequency in Intel i7-4790 will dynamically match the CPU frequencies.

Table 3: Energy cost of verification micro-benchmarks and the accuracy

Verification micro-benchmarks	$\bar{E}_{active}$ (J)	E_{active} (J)	*acc*%
B_L1D_list_nop	129.34	122.04	94.36
B_L1D_array_add	169.85	150.71	88.73
B_L2_nop	122.01	125.57	97.08
B_L3_add	215.37	224.16	96.07
B_mem_nop	396	345.37	87.22
B_L1D_list_L2	168.29	158.26	94.01
B_L1D_list_nop_add	193.06	186.94	96.83

The highest P-state is 36 (3.6GHz CPU frequency) and the lowest is 8 (800MHz CPU frequency).

In this experiment, we turn on the EIST knob and set the P-state range from 8 to 36. Our purpose is to analyze the P-state preference of query workloads. We use 22 TPCH queries [8] to benchmark PostgreSQL, SQLite and MySQL with baseline configuration and baseline data size (more detailed instructions in Section 3) and sample the runtime P-state per 100 milliseconds. According to the percent of P-state 36, Figure 5 shows query count distributions of three database systems. We find that most of queries tend to run at P-state 36, due to the high CPU load (average 96% CPU usage). So, we will fix the CPU at P-state 36 in the following trunk experiment. We also evaluate the impact of other P-states on our results in Section 3.5.

2.8 Results of Micro-Benchmarks

We turn off these knobs which will lead to measurement errors shown in Section 2.5.3, fix the CPU at P-state 36 and specify all workloads to run on the core CPU0. For B_L1D_list, B_L1D_array and B_Reg2L1D, we allocate the memory size as 31KB; for B_L2, allocate 260KB; for B_L3, allocate 6MB and for B_mem allocate 60MB. These setups ensure that our micro-benchmarks only fetch data from the single memory layer under an acceptable latency.

Performance behaviors of micro-benchmarks. As shown in Table 1, BLI (Body-Loop Instruction%) is the percentage of the desired instructions in the main loop and IPC (Instruction Per Clock) is the number of instructions per cycle. For BLI, 98.9% instructions of B_L1D_list is to load data from L1D cache. For other micro-benchmarks, this metric also has a good performance, showing that our micro-benchmarks have little noise instructions. In addition, our micro-benchmarks can provide the specific performance behavior. For B_L1D_list, L1D miss rate is only 0.01%, showing that it always only accesses the L1D cache. Even for B_mem, it can still skip out of cache memory and load data from main memory with a hit rate 97.45%. Especially, IPC shows the CPU stall status. For B_L1D_array, IPC is 2.02 showing that CPU is always busy and no stall. However, IPC=0.26 for B_L1D_list shows that 4 cycles are required to execute a load operation. For B_Reg2L1D, IPC is 1.01 and the number of L1D store instructions are 98.37% of all instructions, showing that CPU always executes 1 store instruction per cycle. These results reveal that our benchmarks can work properly with specific performance behaviors.

Evaluation of ΔE_m. According to E_{active} of micro benchmarks and energy models in Section 2.5.4, we give the energy cost of micro-operations in Table 2. The unit of energy cost is Nanojoule. For the load operation the data is closer to the CPU causing the energy cost to be lower. Especially for ΔE_{L1D}, it is

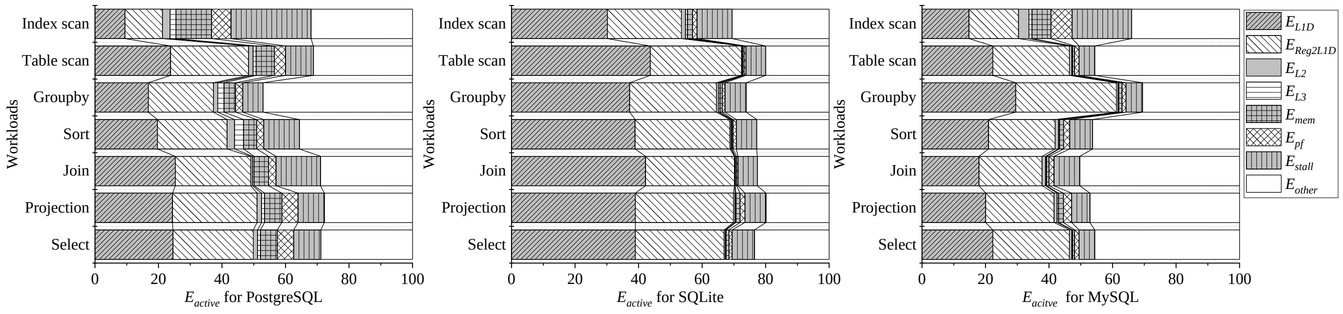


Figure 6: Active energy cost breakdown of the basic query operations for three different database systems.

the lowest than other load operations. Oppositely, loading from the main memory will get a high energy cost penalty. Actually, it explains that why improving cache hit rate can improve both the performance and energy-efficiency.

Verification of ΔE_m. We use the verification method in Section 2.5.5 to evaluate the accuracy of ΔE_m. In Table 3, we give the real Active energy cost and the estimated Active energy cost and the accuracy. The unit of energy cost is Joule. The average accuracy is 93.47%. Even for the most complex B_mem_nop, the accuracy is still high. It shows that ΔE_m proposed by us is accurate enough to break down the energy cost of real workloads.

3 ENERGY COST DISTRIBUTION OF QUERIES

In this section, we use ΔE_m to break down the energy cost of query workloads implemented on the real database systems. We use the PostgreSQL-9.5.2, SQLite-3.14.2 and MySQL-8.0.13 as our analysis targets. We will deeply analyze the energy cost of 7 basic query operations and 22 queries in TPCH[8] with different data sizes and knob settings. We will compare the energy cost breakdown of queries with the typical CPU-bound workloads. In addition, we also show the impact of different P-states on the energy cost breakdown.

To avoid the random error, we disable the result display by updating the database kernels and run the workloads 100 times (10 times for long-running workloads). Finally, the average energy cost is got. We turn on the hardware prefetchers, specify all workloads to run on the core CPU0 and fix the CPU into P-state 36. In addition, the percent of the Background energy cost is 47.2%-51.7% of Busy-CPU energy cost in our experiments. Our main findings are summarized as follows.

- 77.7%-89.2% of Busy-CPU energy cost can be broken down into the energy cost of data movement and the Background energy cost. The energy cost of data movement (7 micro-operations in *MS*) is 55%-76.4% of Active energy cost.
- $E_{L1D} + E_{Reg2L1D}$ is 39%-67% of Active energy cost, identified as the energy bottleneck. This phenomenon does not appear in the typical CPU-bound workloads. In addition, it is little affected by the data size, the database setting and CPU frequency and voltage.
- The sequential scanning in query workloads is the major reason that leads to this energy cost pattern.

3.1 Experimental setup

We take our experiments with 100MB (baseline), 500MB and 1GB data. In addition, each database system has many configurable knobs. We investigate them and tune two important knobs that also have similar roles in three database systems. In Table 4, we give three kinds of database settings to limit the memory usage. The resource size provided to three database systems at each setting is approximate. The `small` setting looks stringent and the `large` setting is relaxed.

Table 4: Knob settings for three database systems

Database systems	Knobs	Small	Baseline	Large
PostgreSQL	Shared_buffers	8MB	128MB	1024MB
	Work_mem	4MB	64MB	512MB
SQLite	Cache_size	2000	16000	65000
	Page_size	4KB	8KB	16KB
MySQL[1]	Inbuffer_size	8MB	128MB	1024MB
	Inpage_size	4KB	8KB	16KB

3.2 Energy Cost of Basic Query Operations

We profile the energy cost of 7 basic query operations with the baseline data size on the baseline setting. 77.7%-89.2% Busy-CPU energy cost can be broken down into the energy cost of data movement and the Background energy cost, showing that our energy breakdown approach can work well on database systems. Figure 6 gives the breakdown of E_{active}. The energy cost of data movement is 68.1% for PostgreSQL, 76.4% for SQLite and 56.8% for MySQL, becoming dominant. For the three database systems, the energy cost distribution is similar, and much energy is consumed in L1D cache load/store. $E_{L1D} + E_{Reg2L1D}$ is 41.6% for PostgreSQL, 66.6% for SQLite and 43.4% for MySQL. So, we can think that this phenomenon could be general for most of relational database systems.

L1D cache load. Actually, we can easily explain why the percent of E_{L1D} is high. In the database systems, almost all of the query operations is based on sequential scan. Even if segmentation or paging strategies are used to manage data, each data block is big enough to fit the CPU cache to ensure a good data locality. So, the modern CPU architecture can ensure most of data to be loaded at L1D cache when sequentially scanning. For example, L1D hit rate of 7 basic query operations is 97.74% and IPC of the complex `join` operator is 1.85, showing good data locality.

L1D cache store. In addition, the reason why the energy cost of L1D cache store is high is because the query workload will generate many temporary data, such as temporary storage of intermediate data and output stream. These temporary data are

[1]For MySQL knobs, `inbuffer_size` is short for `innodb_buffer_pool_size` and `inpage_size` is short for `innodb_page_size`.

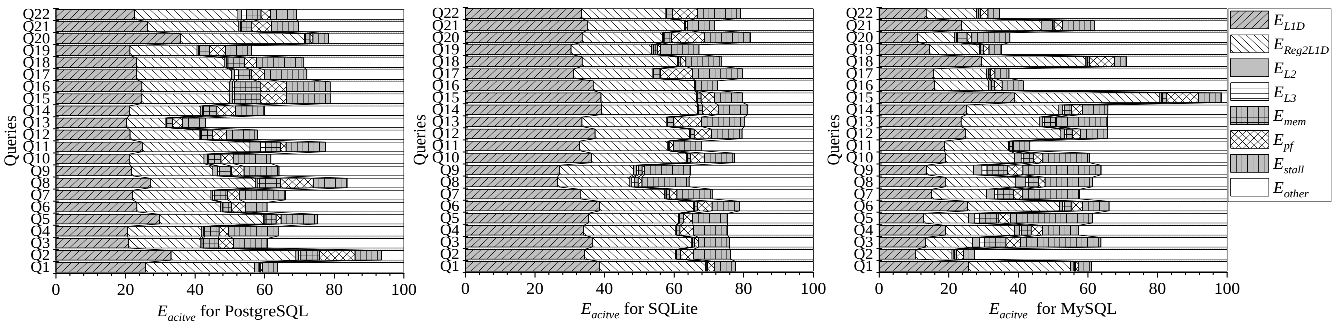


Figure 7: Active energy cost breakdown of TPCH for three different database systems.

written in L1D cache but they little have to be persisted due to the write-back strategy. In our experiment, the store operations are frequently issued by 7 basic query operations, being about 66% of the load operations but 99.86% of them occur at L1D cache.

In summary, although the implementation of three database systems has impact on the energy cost distribution, L1D cache load/store is still their energy bottleneck. In addition, the query workload is read-only operation, but the energy cost of temporary data write is still high.

3.3 Energy Cost of TPCH

We profile the energy cost of TPCH with the baseline data size on the baseline setting. 79.2%-88.7% Busy-CPU energy cost can be broken down. As shown in Figure 7, the energy cost of data movement is 65% for PostgreSQL, 75% for SQLite and 55% for MySQL. In addition, the energy cost distributions of three database systems are similar. The percent of E_{L1D} + $E_{Reg2L1D}$ is so attractive, 46.8% for PostgreSQL, 60% for SQLite and 38.6% for MySQL. The phenomenon is like it in basic query operations because complex queries are the combination of basic operations.

Sequential scanning. For SQLite, the percent of E_{L1D} + $E_{Reg2L1D}$ is higher than that of two other database systems either in the TPCH or the basic query operations because SQLite tends to the sequential scanning. Actually, the energy cost of sequential scanning prefers to L1D cache. We take an example to illustrate the relationship between sequential scanning and the energy cost of L1D cache. As shown in Figure 6, the difference of both index scan and table scan is scan table using the index (B tree) or not. Obviously, index scan utilizes the pointer chasing to reorganize data causing the relatively weak data locality. Table scan tends to the sequential scanning. Without exception, the percent of E_{L1D} + $E_{Reg2L1D}$ reduces and the percent of E_{stall} increases for index scan compared with table scan.

Similarly, SQlite as the mobile database is usually used to manage the small-scale data so that it does not involve many complex optimization strategies, such as the hash join. The main data access method is sequential scanning. It is reasonable because the hardware optimization is more important than software optimization for the small-scale data. The sequential scanning is easily sped up by the hardware optimization, such as speculation and out-of-order execution. It will lead to the less stall cycles. For SQLite, the present of E_{stall} is 12% lower than two other database systems, showing the good performance of sequential scanning. For optimizing performance on large-scale data, we think that both PostgreSQL and MySQL may construct the complex data structure and reorganize the data, such as the compact buffer management. These optimizations can improve the performance, e.g., average 3.31× faster than SQlite in our experiment. However, they will introduce the extra calculations, and they also hinder hardware optimization due to the weak data locality, leading to the low percent of E_{L1D} + $E_{Reg2L1D}$.

Impact of data size. We evaluate the energy cost distributions of three database systems with different data sizes (100MB, 500MB and 1GB) on the baseline setup. As shown Figure 8, we only illustrate the average energy cost result of 22 queries in TPCH as a vector due to the space limitation and PG is short for PostgreSQL. As the data size increases, the energy cost distributions of three database systems have not changed significantly. We also analyze the energy cost change of every query and find that E_{stall} of 14% queries is improved by 2× when increasing the data size. The large data size may lead to frequent swapping in and out of data pages and CPU stall. In general, the L1D cache load/store is still the energy bottleneck which is hardly affected by the data size.

Impact of different settings. As shown in Figure 9, we also compare the impact of different database settings being from Table 4. For MySQL, E_{stall} is reduced when the *large* setting provides more memory to fit data pages but PostgreSQL and SQLite are not sensitive to different settings. It still suggests that different settings have little impact on the energy cost distribution.

In summary, sequential scanning is the basic behavior of query workloads. For different database system implementations, the dependency of sequential scanning affects the energy allocation for L1D cache. Data size and database settings do not lead to substantial changes in this energy cost distribution. So, our findings could be general for query workloads of database systems.

3.4 Comparison to CPU-Bound Workloads

With the optimization of database systems and the performance improvement of the disk, the database system has tended to be CPU-bound from disk-bound. For example, CPU usage is 96% and IPC=1.9 showing a busy CPU, when running TPCH in our experiment. In this section, we compare the energy cost distributions of query workloads with the energy cost distributions of typical CPU-bound workloads. As shown in Figure 10, we evaluate the energy cost of 9 workloads in the classic CPU benchmark CPU2006-v99[2], involving the compression, compiling and simulation workloads, etc. Not like the query workloads, the energy cost distributions of typical CPU-bound workloads are not similar to each other. For three database systems, the percent of E_{L1D} + $E_{Reg2L1D}$ of 76% queries is greater than 40%, but the percent is only 11% for CPU2006. For some extreme CPU2006 workloads (Mcf and Libquantum), E_{L1D} + $E_{Reg2L1D}$ is so low at only 5.6%, but this behavior does not occur in query workloads.

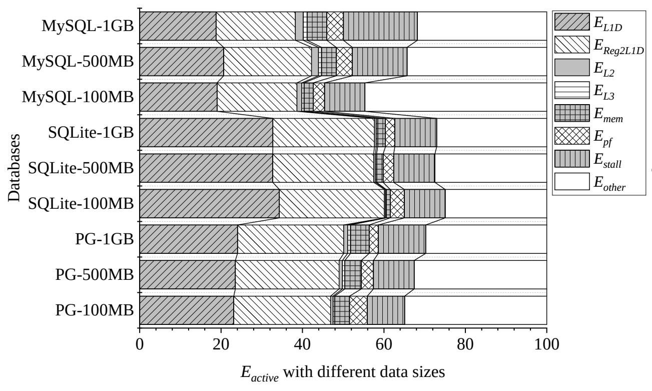


Figure 8: Impact of data size.

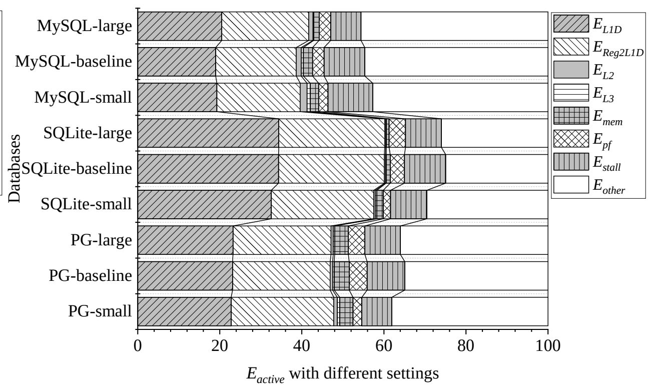


Figure 9: Impact of database setting.

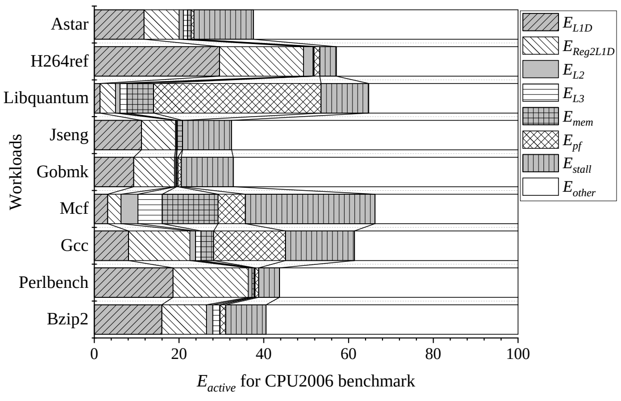


Figure 10: Energy cost breakdown of CPU2006.

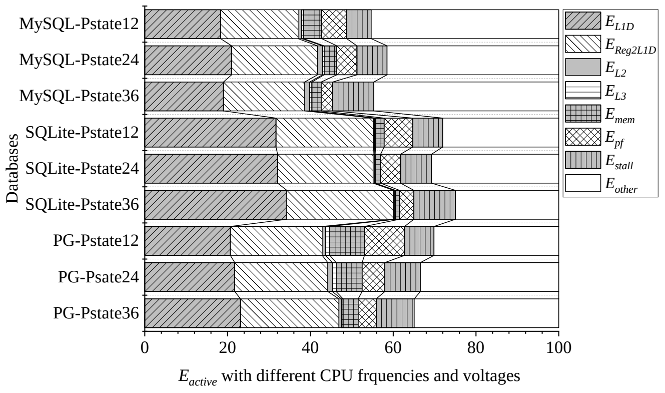


Figure 11: Impact of CPU frequencies and voltages.

In summary, we think that the energy cost pattern of query workloads is totally different from the typical CPU-bound workloads and it could be unique to query workloads.

3.5 Impact of CPU Frequency and Voltage

In the real scenario, the EIST knob is usually turned on, so that we in this section explore the impact of different CPU frequencies and voltages on the energy cost breakdown. We select two other P-states: P-state 24 and P-state 12 to first evaluate energy cost of micro-operations and then break down energy cost of three databases under baseline knob settings and baseline data size.

As shown in Table 2, the energy cost of micro-operations under low P-states will definitely reduce. The closer the micro-operation is to the CPU core, the more significantly the energy cost decreases. For example, ΔE_{L1D} reduces by 53.8% from P-state 36 to P-state 12, but 3.9% for ΔE_{mem}.

As shown in Figure 11, we compare the energy cost breakdown of three databases at different P-states. In our experiment, E_{active} decreases by 32%±2% at P-state 24 and 51%±1% at P-state 12, but the energy cost breakdown has little impact due to the lower energy cost of micro-operations. In detail, because ΔE_{mem} has little change at different P-states, the percent of both E_{mem} and E_{pf} (involving main memory accessing) have a significant improvement at P-state 12, about 2× and 2.2 × compared with P-state 36. However, the absolute impact is still little. Actually, different P-states cannot change the query runtime characteristics, so that the L1D cache load/store hit rate is still high, only leading to the slight reduction of the percent of $E_{L1D} + E_{Reg2L1D}$ at low P-states. For example, the percent of $E_{L1D} + E_{Reg2L1D}$ at P-state 12 only decreases by 4%-8.6% for three databases, compared with it at P-state 36. Our result shows that the L1D cache load/store operations are still the energy cost bottleneck at different CPU frequencies and voltages.

In essence, this experiment reveals the energy cost profiling for typical query runtime characteristics at different CPU frequencies and voltages. For other query scenarios, the CPU could not always be at the high P-state when turning on the EIST knob, such as real-time query workloads. However, their runtime characteristics should be similar to those shown in this paper. So, we think that this energy cost bottleneck could be general for many query scenarios even if turning on the EIST knob.

In summary, the low CPU frequency and voltage will lead to the low energy cost of micro-operations, but the L1D cache load/store operations are still the energy cost bottleneck.

4 PROOF-OF-CONCEPT SYSTEM

In the section, we will discuss the customized CPU architecture design which can enable the energy-efficient database systems. The optimization and evaluation on SQLite will be given.

4.1 L1D Energy-Efficient CPU Architecture

L1D cache load/store is the energy bottleneck, but their optimization is difficult on typical x86_64 architecture because L1D cache has the lowest energy cost than other memory layers.

For database systems, a good energy-efficient CPU architecture should provide the lower energy cost L1D cache (i.e., Arch

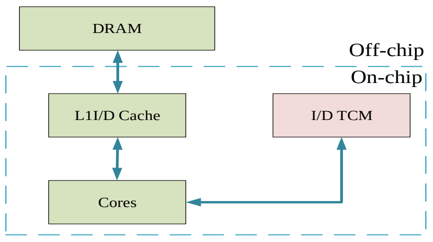


Figure 12: ARM1176JZF-S architecture as an L1D energy-efficient CPU architecture.

1) or provide a piece of the low energy cost memory at the same speed as L1D cache (i.e., Arch 2). For Arch 1, users can transparently migrate the database systems on it. For Arch 2, users need to update the kernel of database systems to decide what data to put into the low energy cost memory. We investigate many CPU architectures and only find the architecture similar to Arch 2. As shown in Figure 12, ARM1176JZF-S[1] supports 16KB L1D cache, 32KB DTCM (Data Tightly Coupled Memory) and 256MB main memory. TCM is the programmable on-chip memory which is as fast as L1 cache but its energy cost is lower than L1 cache. So, ARM1176JZF-S could be as an L1D energy-efficient CPU architecture. In this section, we will use the DTCM load instead of the L1D cache load to reduce Busy-CPU energy cost.

4.2 System-Level Co-Design

We will optimize SQLite because, as a mobile database it can work well on ARM architecture, but the optimization is still difficult. First, although Linux supports many hardware environments, it cannot be directly compiled in this ARM environment. The ARM hardware environments are so diverse that Linux can only identify some mainstream architectures and it does not support ARM1176JZF-S well. We have to modify and update Linux kernel to make it support TCM, Linux perf and cross compiling. Second, we have to implement the TCM driver and API enabling that TCM can be accessed in user space. It took us about 2 months to build an available runtime environment. For SQLite, our optimization strategies are as follows.

Database buffer. We allocate 16KB DTCM for database buffer, which will be dynamically managed by SQLite.

Special variables. We use the Linux perf to profile the SQLite's runtime and find that about 70% L1D cache load operations are issued by the `sqlite3VdbeExec()` function to execute the query plan. This phenomenon in x86_64 architecture is similar to it in ARM architecture. We allocate 4KB DTCM and put some key structures of `sqlite3VdbeExec()` in it, such as query plan (`Vdbe`), meta data (`Vdbe->db`), cursor (`Vdbe->apCsr` and `Vdbe->apCsr->aOffset`), head address of heap space (`Vdbe->aOp` and `Vdbe->aMem`), etc.

B tree. Every table in SQLite is organized as a B tree. The primary key or row ID will be the key of B tree. So, the top layers of B tree will be frequently read. Based on this, we allocate 12KB DTCM to put the root and first few layers of B-tree of current tables into DTCM. We divide DTCM memory evenly according to the number of tables being queried. In this way, we can ensure that more B tree data of small tables are loaded into DTCM.

Noting that our strategies are for L1D cache load operation. The energy cost optimization of L1D cache store operation is more difficult. We do not discuss it in this paper.

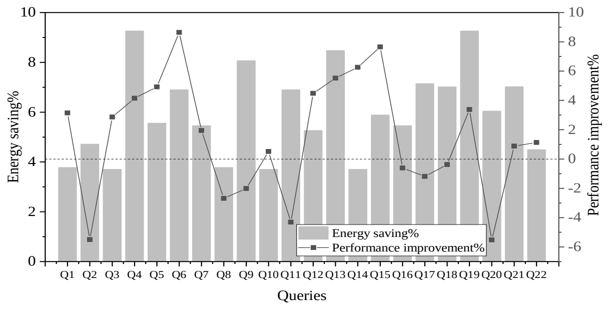


Figure 13: Energy saving and performance improvement for SQLite using DTCM or not on ARM1176JZF-S.

4.3 Evaluation Results

ARM does not support RAPL, so that we use the external power meter to measure the energy cost. We first design a micro benchmark B_DTCM_array, which only loads data from DTCM. It is similar to Algorithm 1 but allocates memory from DTCM, instead of main memory. Compared with B_L1D_array, the energy cost of B_DTCM_array can reduce by 10% with no performance loss. Therefore, 10% will be as the peak energy saving of DTCM in our experimental environment.

In Figure 13, we show the energy saving and performance improvement of the optimized SQLite. Noting that we compare whether SQLite uses DTCM on ARM, not SQLite on Intel CPU or ARM CPU. We use 10MB TPCH data and the `small` setting to take our experiment. Our optimization makes SQLite save average 6% energy and improve average 1.5% performance. It means that our approach can achieve 60% of the peak energy saving. The dominant factor is that accessing the hot data in DTCM can bypass the L1D cache leading to L1D cache energy saving. In addition, the performance of 64% queries can be further improved due to the avoidance of hot data misses. For the non-optimized SQLite, 25% L1D miss rate in ARM1176JZF-S could cause the hot data to swap in and out the L1D cache from main memory. However, DTCM has the fixed physical address, so that the hot data in DTCM is not loaded from main memory.

Advantages. Limited to the hardware implementation, 6% energy saving in our experiments may seem to be less, but our approach has three advantages as follows. First, compared with the existing approaches, our approaches can save energy with no performance reduction. It is advantageous for energy-strict real-time tasks, such as UPS-powered data centers and databases in smart phones. Second, our approach is orthogonal to existing approaches. Our approach tends to save energy from the view of CPU architecture, so that it can work together with application-level energy optimization approaches, such as energy-oriented query optimization and DVFS-based approaches. Third, our approach is depend on the implementation of TCM. So, these results in our paper only suggest that our approach can achieve 60% of the peak energy saving, and do not mean the final energy-saving potential. The existing work shows that the optimized TCM has got 40% energy saving compared with L1D cache[9]. If integrating such an optimized TCM into ARM1176JZF-S, our approach should get a maximum 24% energy saving.

In summary, our optimized SQLite can achieve 60% of the peak energy saving. It can also achieve 1.5% performance improvement due to the avoidance of hot data misses.

Table 5: Energy cost bottleneck of B_mem at different CPU frequencies and voltages

P-state	36 (3.6GHz)	24 (2.4GHz)	12 (1.2GHz)
Micro-operations	Energy cost (J) / Percent%		
E_{mem}	295.4 (16.6%)	284.2 (29.8%)	284.6 (47.4%)
E_{stall}	**1416.1 (79.8%)**	**630.6 (66.2%)**	**310.1 (51.6%)**
E_{active}	1772.5 (100%)	952.9 (100%)	600.5 (100%)

5 POTENTIAL OPTIMIZATIONS

Based on the Busy-CPU energy breakdown results, we also find some additional interesting energy cost phenomena in Figure 6 and 7 that suggests other optimization approaches.

In index-intensive scenario, especially for index scan of PostgreSQL and MySQL, the percent of both E_{mem} and E_{pf} becomes prominent, also causing a high E_{stall}. Similar phenomenon can also be seen in PostgreSQL's basic query operations compared with them of two other database systems. In addition, as the data size increases, the main memory access is also more frequent, especially for MySQL. They imply a memory-bound tendency. In this section, we will focus on the energy cost optimization of memory-bound workloads. Actually, we think that improving main memory performance or radically lowering CPU frequency and voltage are efficient ways to save energy.

To explain our idea, we first profile the energy cost of a typical memory-bound workload under different CPU frequencies and voltages. The micro-benchmark B_mem is a typical memory-bound workload and we break down its energy cost shown in Table 5. Interestingly, such a slight change to ΔE_{mem} does not imply that the low P-state will cause the energy cost to increase for memory-bound workloads, although the elapsed time may be increased. Actually, B_mem's performance bottleneck is main memory, the energy cost bottleneck is the CPU (E_{stall}, not E_{mem}). This result suggests that the energy cost bottleneck is in the CPU, even if for non-CPU bound workloads. So, the ultra-linear decrease in E_{stall} causes a reduction in E_{active} with slight performance loss. For example, B_mem only trades 7% performance loss for 46% E_{active} saving when lowering P-state from 36 to 24. The energy-efficiency ($\frac{Perf}{Energy}$)[14] is improved by 70%. In addition, EIST cannot work well on memory-bound workloads. When turning on the EIST knob, the percent of P-state 36 is 98.6% due to the high CPU load (99.8% CPU usage), implying failure of dynamic energy saving.

Actually, for memory-bound query scenarios, an energy-saving chance is to reduce E_{stall}. Improving main memory performance tends to reduce N_{stall} or radically lowering CPU frequency and voltage tends to reduce ΔE_{stall}. We take a preliminary experiment on PostgreSQL's index scan to confirm the second approach. When lowering P-state from 36 to 24, PostgreSQL's index scan only trades 20% performance loss for 27% E_{active} saving, showing that the energy-efficiency is improved by 10%. However, our strategy is not trivial. PostgreSQL's table scan, a CPU-bound workload, has to trade 30% performance loss for 28% E_{active} saving, i.e., the energy-efficiency is reduced by 3%. So, a customized DVFS approach is expected for memory-bound query scenarios. It should analyze the query plan, such as index-intensive or not, and monitor the main memory access to employ a more radical DVFS strategy.

The percent of MySQL's E_{other} is higher than that of two other databases, especially for basic query operations, so that energy-efficient calculation components or instruction-related components, e.g., instruction TCM (ITCM), should be considered.

6 RELATED WORK

Energy characterization and optimization in database systems is the basic work to design an energy-efficient database systems. There are extensive researches on this topic from different aspects, including (1) macro energy cost breakdown, (2) trade-off based energy optimization and (3) employing TCM. However, we take a micro analysis of the Busy-CPU energy cost of database systems and then provide a customized CPU architecture to enable the energy-efficient database systems.

Macro energy cost breakdown. The energy-efficient database design is systematically reported in [14]. From then on, the researchers had made much effort to make sense of its energy bottleneck. Research work focuses on the breakdown of the energy cost of the major resources, i.e., the CPU, main memory and disk, on various of system architectures. For the classic x86_64 architecture with local disk, the CPU is identified as the energy cost for disk-based database systems[19, 25] and in-memory database systems[13]. For the architecture of ARM+RDRAM (Rambus DRAM), the main memory is identified as the bottleneck[23]. For the system with the remote disk array, the disk is the energy bottleneck[24]. These above conclusions are difficult to explain whether the power is consumed by the database system or by the hardware itself because the measurement of the total energy cost contains the Background energy cost and Idle-CPU energy. Our work divides the Busy-CPU energy cost into the Active energy cost and the Background energy cost, and only profile the Active energy cost which is consumed by database systems. In addition, the existing work focuses on the macro energy breakdown of major resources. However, the main drawback is that they cannot give the clear optimization suggestions on hardware architecture due to the lack of the fine-grained information. We study the micro energy analysis inside CPU and have the ability to make sense of microscopic energy cost distribution. It is helpful to design the CPU architecture for energy-efficient database systems.

Trade-off based energy optimization. For the x86_64 architecture, the CPU has been identified as the energy bottleneck for database systems. Because the energy cost distribution inside the CPU is unknown, the existing work sees the CPU as an inseparable whole to design the energy-oriented query optimizer or tune the external DVFS knobs. Inspired by the performance-oriented query optimizer, PET [28, 29] as an energy-aware query optimization constructs a cost model to choose the low-energy query plans under a DBA-specified energy/performance trade-off level. QED [21] uses query aggregation to leverage common components of queries to reduce energy cost. These query optimization techniques are used to gracefully trade response time for energy. DVFS knobs can be configured by users to trade the voltage and CPU frequency for energy. It provides the chance to optimize the energy. PVC[21] and sweet spots[13] attempt all of combinations between the voltage and CPU frequency for a specific workload to choose one combination which can minimize the energy cost. Their drawback is hard to apply to all queries. Other approaches leverage feedback-control loops to dynamically set DVFS knobs using the load profile and can obey a query latency limit as a soft constraint. Based on this idea, lots of work has achieved the adaptive energy control for various of databases, such as the disk-based transaction-oriented DBMS [20, 26, 30] and data-oriented in-memory DBMS [19]. In order to get a good energy-saving effect, these techniques expect a relaxed run-time constraint, so that they in essence trade the performance for the energy. Through our micro analysis of the Busy-CPU energy cost, we have found the micro-operation level energy bottleneck. With the help of the L1D cache energy-efficient CPU architecture, we can cut down the energy cost of database systems with slight performance improvement.

Employing TCM. TCM as on-chip memory is usually used for performance improvement by combining the micro performance feature of applications, such as digital signal processing[12], MapReduce framework[16] and embedded multi-media[15]. However, we attempt to use TCM to reduce the energy cost, combining the micro energy feature of database systems. The main idea of our optimization is to put the hot data into TCM to save the energy. The similar idea for TCM has appeared. They at compile-time analyze a given piece of program, identify the hot data and put them into TCM at runtime to reduce the energy cost of data movement, such as a SPM management framework for nest-loop[17], a data program relationship graph for global and stack variables[27] and heap data[11]. Facing the complex database systems, these program-level optimization methods under specific premises may not work. Actually, our optimization is system-level. We profile the runtime behavior of database system, review its source code and elaborately identify the hot data. Although the optimized SQLite is only a proof-of-concept system, it confirms that our method is feasible. Providing the customized CPU architecture is a possible way to enable energy-efficient database systems.

7 SUMMARY & FUTURE WORK

In this paper, we propose a novel idea to reduce the energy cost based on profiling the energy cost of CPU micro-operations for databases systems. Our approach can break down the majority of Busy-CPU energy cost and isolate the accurate energy cost of data movement. The CPU energy breakdown method exposes that L1D cache load/store is the energy bottleneck of database systems. The finding supposes that we may achieve energy efficiency by adopting a customized CPU architecture with lower L1D energy cost. TCM can meet this requirement well and an optimized system-level co-design solution for SQLite is implemented to evaluate the proposed idea. The experimental result of the proof-of-concept system shows that our method can achieve 60% of the peak energy saving with further performance improvement.

In future, we will try to profile the energy cost of other typical database systems, such as NoSQL systems to identify their energy distribution feature on CPU and check if our method can be employed into more type of database systems.

8 ACKNOWLEDGEMENT

This research was partially supported by the grants from the Natural Science Foundation of China (No. 61532016, 91646203, 91846204, 61532010, 61762082).

REFERENCES

[1] 2019. ARM11. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf.

[2] 2019. CPU2006. http://www.spec.org/cpu2006/.

[3] 2019. Intel prefetcher. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.

[4] 2019. L2 prefetcher behavior. https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/703019.

[5] 2019. Linux Perf. http://www.brendangregg.com/perf.html.

[6] 2019. ocperf. https://github.com/andikleen/pmu-tools.

[7] 2019. P-state and C-state. https://software.intel.com/en-us/blogs/2008/03/12/c-states-and-p-states-are-very-different/.

[8] 2019. TPCH. http://www.tpc.org/tpch/.

[9] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. 2002. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 2002 International Symposium on Hardware/Software Codesign*. 73–78.

[10] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design*. 189–194.

[11] Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. 2005. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing* 1, 4 (2005), 521–540.

[12] Syed Z Gilani, Nam Sung Kim, and Michael Schulte. 2011. Scratchpad memory optimizations for digital signal processing applications. In *Proceedings of 2011 Design, Automation & Test in Europe*. 1–6.

[13] Sebastian Götz, Thomas Ilsche, Jorge Cardoso, Josef Spillner, Thomas Kissinger, Uwe Aßmann, Wolfgang Lehner, Wolfgang E Nagel, and Alexander Schill. 2014. Energy-efficient databases using sweet spot frequencies. In *Proceedings of the 2014 IEEE/ACM International Conference on Utility and Cloud Computing*. 871–876.

[14] Stavros Harizopoulos, Mehul Shah, Justin Meza, and Parthasarathy Ranganathan. 2009. Energy efficiency: The new holy grail of data management systems research. In *Proceedings of the 2009 biennial Conference on innovative data systems research*. 81–90.

[15] Wen Shu Hong, Hui Juan Cui, and Kun Tang. 2005. Scratchpad Memory Assignment in Embedded Multimedia Application. *Acta Electronica Sinica* 33, 11 (2005), 1937–1940.

[16] Christoforos Kachris, Georgios Ch. Sirakoulis, and Dimitrios Soudris. 2015. A MapReduce scratchpad memory for multi-core cloud computing applications. *Microprocessors & Microsystems* 39, 8 (2015), 599–608.

[17] Mahmut Kandemir, Jagannathan Ramanujam, Mary Jane Irwin, Narayanan Vijaykrishnan, Ismail Kadayif, and Amisha Parikh. 2001. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th Design Automation Conference*. 690–695.

[18] Gokcen Kestor, Roberto Gioiosa, Darren J Kerbyson, and Adolfy Hoisie. 2013. Quantifying the energy cost of data movement in scientific applications. In *2013 IEEE international symposium on workload characterization*. 56–65.

[19] Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. 2018. Adaptive energy-control for in-memory database systems. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of data*. 351–364.

[20] Mustafa Korkmaz, Alexey Karyakin, Martin Karsten, and Kenneth Salem. 2015. Towards Dynamic Green-Sizing for Database Servers.. In *Proceedings of the 2015 International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*. 25–36.

[21] Willis Lang and Jignesh M Patel. 2009. Towards Eco-friendly Database Management Systems. *Proceedings of the 2009 biennial Conference on innovative data systems research* (2009).

[22] Dhinakaran Pandiyan and Carole Jean Wu. 2014. Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms. In *IEEE International Symposium on Workload Characterization*. 171–180.

[23] Jayaprakash Pisharath, Alok Choudhary, and Mahmut Kandemir. 2004. Reducing energy consumption of queries in memory-resident database systems. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*. 35–45.

[24] Meikel Poess and Raghunath Othayoth Nambiar. 2008. Energy cost, the key challenge of today's data centers: a power consumption analysis of TPC-C results. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1229–1240.

[25] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A Shah. 2010. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 231–242.

[26] Yi-Cheng Tu, Xiaorui Wang, Bo Zeng, and Zichen Xu. 2014. A system for energy-efficient data management. *ACM SIGMOD Record* 43, 1 (2014), 21–26.

[27] Sumesh Udayakumaran and Rajeev Barua. 2003. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*. 276–286.

[28] Zichen Xu, Yi-Cheng Tu, and Xiaorui Wang. 2010. Exploring power-performance tradeoffs in database systems. In *Proceedings of the 2010 IEEE International Conference on Data Engineering*. 485–496.

[29] Zichen Xu, Yi-Cheng Tu, and Xiaorui Wang. 2012. PET: reducing database energy cost via query optimization. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1954–1957.

[30] Zichen Xu, Xiaorui Wang, and Yi-cheng Tu. 2013. Power-aware throughput control for database management systems. In *Proceedings of the 2013 International Conference on Autonomic Computing*. 315–324.

Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication

Thamir M. Qadah[1†], Suyash Gupta[2], Mohammad Sadoghi[2]
Exploratory Systems Lab
[1]Purdue University, West Lafayette
[2]University of California, Davis
[†] Umm Al-Qura University, Makkah
tqadah@purdue.edu, sugupta@ucdavis.edu, msadoghi@ucdavis.edu

ABSTRACT

Distributed database systems partition the data across multiple nodes to improve the concurrency, which leads to higher throughput performance. Traditional concurrency control algorithms aim at producing an execution history equivalent to any serial history of transaction execution. Hence an agreement on the final serial history is required for concurrent transaction execution. Traditional agreement protocols such as Two-Phase-Commit (2PC) are typically used but act as a significant bottleneck when processing distributed transactions that access many partitions. 2PC requires extensive coordination among the participating nodes to commit a transaction.

Unlike traditional techniques, deterministic concurrency control techniques aim for producing an execution history that obeys a pre-determined transaction ordering. Recent proposals for deterministic transaction processing demonstrate high potential for improving the system throughput, which had led to their successful commercial adoption. However, these proposals do not efficiently utilize and exploit modern computing resources and are limited by design to conservative execution.

In this paper, we propose a novel distributed queue-oriented transaction processing paradigm that fundamentally re-thinks how deterministic transaction processing is performed. The proposed paradigm supports multiple execution paradigms, multiple isolation levels, and is amenable to efficient resource utilization. We employ the principles of our proposed paradigm to build Q-Store, which is the first to support speculative execution and exploits intra-transaction parallelism efficiently among proposed deterministic and distributed transaction processing systems. We perform extensive evaluation against both deterministic and non-deterministic transaction processing protocols and demonstrate up to two orders of magnitude of improved performance.

1 INTRODUCTION

Distributed transaction processing is challenging due to the inherent overheads of costly commit protocols like *2-Phase-Commit* (2PC) [13]. Even for use cases such as in-memory databases and stored-procedure-based transactions, 2PC is either used (e.g., [24, 25, 46]) or avoided by eliminating the processing of multi-partitioned transactions (e.g., [27, 28]). Note that 2PC by itself does not ensure serializable transaction processing, and it requires a distributed concurrency control protocol to guarantee serializability. Traditional concurrency control protocols may abort active distributed transactions non-deterministically to

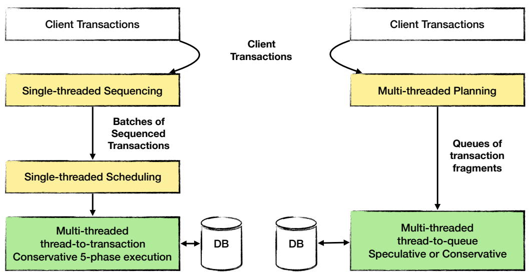

Client Transactions
Client Transactions
Client Transactions
Single-threaded Sequencing
Multi-threaded Planning
Batches of Sequenced Transactions
Queues of transaction fragments
Single-threaded Scheduling
Multi-threaded thread-to-transaction Conservative 5-phase execution
DB
DB
Multi-threaded thread-to-queue Speculative or Conservative

Figure 1: Overview of transaction processing in Calvin (left) and Q-Store (right)

ensure serializable transaction processing. When such abort decisions are coupled with 2PC, the cost of the distributed transaction processing is further increased because of the overhead of rollbacks and restarts.

Deterministic databases [12, 40] reduce the cost of committing distributed transactions by imposing a single order on executing a batch of transactions prior to actual execution. By ensuring the same pre-execution ordering, deterministic database systems eliminate the need to abort transactions for violating serializability guarantees (in optimistic concurrency control), avoiding deadlocks (in pessimistic concurrency control), or node crash failures.

Unfortunately, the state-of-the-art designs of distributed deterministic databases suffer from other inefficiencies. We identify three of these inefficiencies that limit their performance and scalability. First, they rely on single-threaded pre-execution sequencing and scheduling mechanisms which cannot exploit multi-core computing architectures and limit vertical throughput scalability [20]. Second, they mostly support a conservative (non-speculative) form of transaction execution. One exception is the work by Jones et al. [24], which performs speculative-execution only for multi-partition transactions but limits the concurrency for single-partition transactions (a property inherited from H-Store's design). Third, they follow a thread-to-transaction assignment which limits intra-transaction parallelism [34, 35].

In this paper, we propose a novel transaction processing paradigm of *queuing-oriented processing*, and describe Q-Store. Q-Store is built on the principles of queue-oriented paradigm, which provides a unified abstraction for processing distributed transactions deterministically and does not suffer from the inefficiencies such as lower utilization of cores. Furthermore, it admits multiple execution paradigms (i.e., *speculative* or *conservative*) and multiple isolation levels (i.e., serializable isolation or read-committed isolation) seamlessly, unlike existing proposals of the deterministic database. It is important to note that several existing non-deterministic database systems already support multiple forms of isolation levels (e.g., [22, 30, 33, 37, 38]).

Our *queue-oriented* transaction processing paradigm can efficiently utilize the parallelism available with commodity multi-core machines by maximizing the number of threads doing useful work. Q-Store processes batches of transactions in two multi-threaded yet deterministic phases of *planning and execution*, as shown in the right side of Figure 1. Each phase utilizes all available computing resources efficiently, which improves the system's throughput significantly. The planning phase is carried out by multiple *planning-threads*, delivering maximum CPU utilization. Planning-threads generate queues of transaction operations that require minimal coordination among *execution-threads*. These queues are executed by execution-threads that are assigned to different cores to maximize cache efficiency. Each execution-thread is assigned one or more queues for execution. In other words, these queues constitute a schedule for executing transnational operations of a batch of transactions. Any coordination among execution-threads is performed via efficient and distributed lock-free data structures. In particular, we make the following contribution, in this paper.

- We propose a novel queue-oriented transaction processing paradigm that facilitates distributed transaction processing and unifies local and remote transaction processing in a single paradigm based on pre-determined priorities of queues. Our proposed paradigm supports multiple execution paradigms and multiple isolation levels and leads to implementation of efficient transaction processing protocol (Section 3).
- We present a formalization of our proposed paradigm and prove that it produces serializable histories to guarantee serializable isolation. We also formally show how our paradigm can support read-committed isolation seamlessly (Section 4).
- We design and build Q-Store, which is a distributed transaction processing system that relies on the principles of our proposed queue-oriented paradigm (Section 5).
- We present the results of an extensive evaluation of Q-Store. In our evaluation, we compare Q-Store against non-deterministic and deterministic transaction processing protocols using workloads from standard macro-benchmarks such as YCSB and TPC-C. We perform our evaluation using a single code-base, which allows us to conduct an apple-to-apple comparison against 5 transaction processing protocols. Our experiments demonstrate that Q-Store out-performs state-of-the-art deterministic distributed transaction processing protocols by up to 22.1×. Against non-deterministic distributed transaction processing protocols, Q-Store achieves up to two orders of magnitude better throughput (Section 6).

2 BACKGROUND

In this section, we give an overview of Calvin [40] as a representative for deterministic databases. As far as we know, Calvin is regarded as the state-of-the-art distributed deterministic transaction processing protocol, and has been commercialized [12]. Other deterministic transaction processing protocols are either designed for non-distributed environments (e.g., [9, 11, 35]) or a variation that improves parts of Calvin's protocol while re-using the remaining parts as-is (e.g., [44]). These proposals are covered in Section 7 in more details. We also briefly describe the transaction model used by Q-Store which adopts the same transaction model used by [35], which is in sharp contrast from Calvin's transaction model.

2.1 Transaction Processing in Calvin

This section gives a brief description of how Calvin works based on [40]. The basic processing flow requires 3 phases: a sequencing phase, a scheduling phase, and an execution phase with 5 sub-phases. Figure 1 (left), illustrate these phases.

Each node, in Calvin, runs a single sequencer thread, a single scheduler thread, and one or more worker threads. The sequencer thread forms batches of sequenced transactions. It uses a time-based demarcation of batches. Batches formed by different nodes are processed by scheduler threads in strict round-robin fashion. Scheduler threads use deterministic locking to schedule transactions that require the full knowledge of the read/write sets of transactions, which is similar to *Conservative 2PL* [8]. Unlike Conservative 2PL, Calvin ensures that conflicting transactions are deterministically processed according to their sequence number in the sequencing batch. For example, let t_a and t_b denote two conflicting transactions (i.e., cannot be scheduled to execute concurrently), and $seq(t)$ denote the sequence number of transaction t as determined by the sequencer thread. If $seq(t_a) < seq(t_b)$, then Calvin ensures that t_a is scheduled before t_b. Once locks on all the records are acquired by the scheduler thread, the transaction is ready for execution, and it is given to a worker thread for execution.

As Calvin is a distributed database system, each worker thread executes an assigned transaction in the following phases:

Phase 1 - Read/write set analysis: This phase is used to determine the set of nodes that are participating in the transaction. For this set, nodes that are executing at least one write operation are marked as active participants.

Phase 2 - Perform local read operations: This phase is performed by all participants if records are available locally.

Phase 3 - Serve remote read operations: Multicast records to active participants. This phase is the last phase performed by non-active participants. At this point, they can declare the transaction as completed and move to the next transaction.

Phase 4 - Collect remote read operations: This phase is performed by active participants only, and they need to wait for remote records before moving to the next phase. Hence, worker threads can postpone the active transaction (while waiting) and resume another transaction that is ready for execution.

Phase 5 - Execute transaction logic and perform local write operations: This phase is also performed only by active participants.

Discussion. The original Calvin paper by Thomson et al. [40] does not clearly describe how a transaction is committed (or aborted). However, by looking into the code-base of one of the implementations of Calvin from [20], which we ported to our test-bed, we discovered that the basic idea goes as follows.

The sequencer determines the participant nodes of every sequenced transaction by performing **Phase 1** from above. When a participant node completes its work on a transaction, it sends a one-way acknowledgment (ACK) message to the sequencer of the transaction. When the sequencer collects all ACK messages from all participants, it commits the transaction and sends a response message to the client of the transaction. Worker threads (that execute transactions) can re-use read/write set analysis performed by the sequencer thread to avoid needless computation.

2.2 Q-Store's Transaction Model

We adopt the same transaction model used by [35]. In this model, a transaction is broken into fragments. A fragment can perform

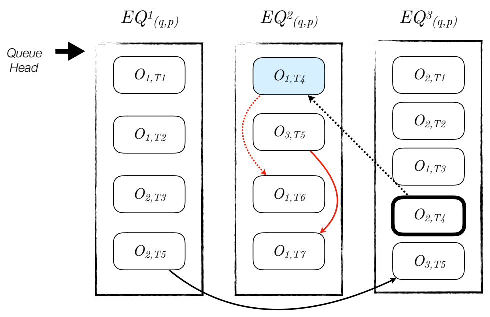


Figure 2: An example illustrating transaction dependencies in Q-Store. Execution-queues (EQs) are planned by planning-thread $PT_{(q,p)}$

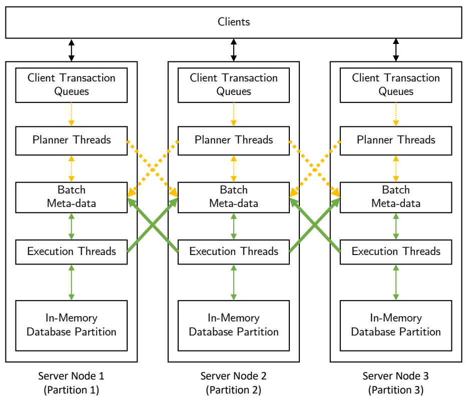

Clients
Client Transaction Queues
Planner Threads
Batch Meta-data
Execution Threads
In-Memory Database Partition
Server Node 1 (Partition 1)
Server Node 2 (Partition 2)
Server Node 3 (Partition 3)

Figure 3: System Architecture

multiple operations on the same record, such as read, modify, and write operations. A fragment can cause the transaction to abort, and in this case, we refer to such fragments as abortable fragments.

Furthermore, there are can be dependencies among fragments. In Figure 2, we illustrate these dependencies. There are 7 planned transactions in 3 execution-queues. Fragments are denoted as O_{i,T_j} where i denotes the fragment index in transaction T_j. We describe the notations in detail in Section 4.

Data dependencies exist when an operation in a fragment requires a value that is read by another fragment of the same transaction (solid black arrow between O_{2,T_5} and O_{3,T_5}).

Conflict dependencies exist between fragments from different transactions that access the same record, and the dependee fragment performs a write operation (solid red arrow between O_{3,T_5} and O_{1,T_7}).

Two kinds of commit dependencies exist between fragments. The first kind is concerned with fragments of the same transaction. In this case, a commit dependency exists between two fragments of the same transaction if the dependee is an *abortable* fragment (dotted black arrow between O_{2,T_4} and O_{1,T_4}). In this example, O_{2,T_4} is an abortable fragment. The second kind of commit dependencies, which we refer to as *speculation dependencies*, exist between fragments of different transactions. Tracking them is required when using the speculative execution paradigm. A speculation dependency exists between the two fragments when the dependent fragment reads speculatively uncommitted data written by the dependee fragment (dotted red arrow between O_{1,T_4} and O_{1,T_6}).

Discussion. It is worth noting that speculation dependencies are a realization of conflict dependencies. Tracking speculation dependencies is needed to ensure correct transaction execution with speculative execution. Note that, in Q-Store, conflict dependencies are not explicitly tracked during planning. It is possible to capture these during planning, but that would introduce additional overhead to the planning phase, which is undesirable.

3 TRANSACTION PROCESSING IN Q-STORE

In this section, we describe the novel and unique features of Q-Store. As far as we know, Q-Store is the first distributed deterministic transaction processing system to provide following features.

Efficient two-phase distributed processing model. In Figure 1, we show the critical differences between Calvin's processing model and Q-Store' processing model. On the left side (Calvin) of Figure 1, the total number of phases required to process a batch of transactions is 3 with the execution phase requiring 5 sub-phases. Note that the sequencing and the scheduling phases in Calvin are single-threaded. On the right side, Q-Store processes a batch of transactions in two multi-threaded phases of planning and execution. The execution phase does not include any sub-phases. Q-Store reduces the number of phases compared to Calvin (See Figure 1). Furthermore, Q-Store uses all available cores efficiently. All available threads work on the planning of a batch, then all of them work on execution.

Multi-paradigm execution. The design of Q-Store admits multiple execution paradigm. The processing of a batch of transactions can be speculative or conservative. Transaction isolation can be serializable or read-committed.

Queue-oriented processing. The planning phase in Q-Store abstracts the logical semantics of a transaction into prioritized queues of transaction fragments. Queues provide ordering for conflict fragments that seamlessly resolve conflict dependencies among fragments of different transactions. Therefore, threads during execution only deal with the other dependencies. Furthermore, queues can be implemented efficiently to ensure efficient execution and communication.

3.1 Queue-oriented Architecture

In Figure 3, we illustrate an example architecture of Q-Store, which consists of three server nodes. A client may send transactions that require access to multiple partitions, which we call multi-partition transactions. A client selects one of the server nodes for a given transaction and sends the transaction to the selected server. The role of the selected server is to coordinate the execution of the received transaction. Note that a server can be selected during the client session establishment, which allows mechanisms for load-balancing. Mechanisms for load-balancing include client-side libraries and middle-ware-based mechanisms. These details are beyond the scope of this paper.

Also in Figure 3, each node maintains a set of local *client transaction queues.* There is one client transaction queue per planner-thread to avoid contention. Planner-threads create fragments from transactions and capture dependencies, and create queues of fragments for each execution thread. Each planner-thread also updates the *Batch Meta-data* distributed data structure. The Batch Meta-data stores information about fragment dependencies, and execution-queues progress status. It is a globally shared lock-free distributed data structure that is used to facilitate minimal coordination among execution threads. In Figure 3, yellow arrows depict communication patterns during the planning-phase

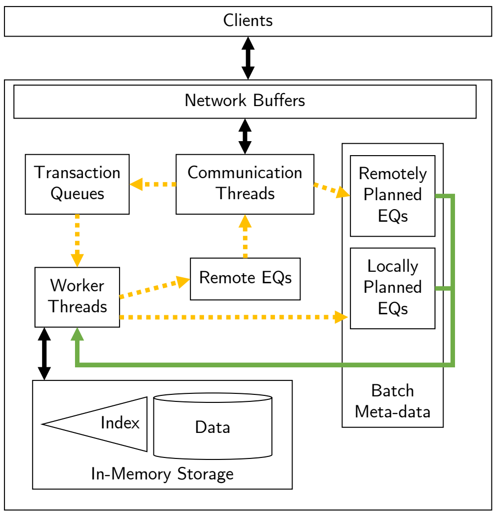


Figure 4: Server Node Architecture

while green arrows depict communication patterns during the execution-phase.

Zooming into a single node, Figure 4, we illustrate the major components of a server node. Similar to Figure 3, yellow and green arrows, depict communication during planning-phase and execution-phase, respectively.

Each server employs a set of threads to complete various tasks. We can broadly categorize threads into two sets: (i) *communication threads*, and (ii) *worker threads*. Q-Store employs communication threads to handle message transmission and reception among the servers and clients. They are also responsible for handling messages between server components and network buffers. They store client transactions in transaction queues, send and receive remote execution-queues (EQs), and apply updates to the batch meta-data.

Each worker thread may participate in either one or two phases: *planning* and *execution* (e.g., we can have dedicated worker threads for each phase). Hence depending on the phase, we refer to these worker threads as either *planning-threads* or *execution-threads*. We use $\mathbb{P}$ to represent the set of planning-threads and $\mathbb{E}$ to represent the set of execution-threads. The planning-threads take a set of transactions and generate *plans* to execute these transactions. The execution-threads execute transactions according to these plans.

3.2 Priorities in Q-Store

In Q-Store, we use the notion of priorities to impose order at various levels granularity. The concept of priority captures the ordering of queues and transaction fragments elegantly. Execution-threads need to respect these priorities to ensure the correct ordering of conflicting transactions. We have three different levels of granularity from the perspective of execution.

We formalize the notion of priorities by representing our distributed system as a set $\mathbb{S}$, which is the set of server nodes. We assign each server S_q a priority q, that is,

$$\mathbb{S} := \{S_1,\ S_2, ..., S_q\},\ \text{where}\ q \geq 1$$

Q-Store requires each server to associate a priority p with each of its planning-threads. Note that the planning-thread priority p differs from the server priority q. As each planning-thread also inherits the priority of its server, so each planning-thread has two associated priorities. Hence, we use the $P_{q,p}$ representation for a planning thread with priority p.

$$\mathbb{P}_q := \{P_{q,1},\ P_{q,2}, ...,\ P_{q,p}\},\ \text{where}\ q, p \geq 1 \quad (1)$$

Planning-threads create execution-queues for transactions and tag them with their priorities. The execution-queues created by a planning-thread constitute schedules of transaction fragments of the set of transactions processed by the planning-thread. Execution-threads execute fragments according to planned schedules while respecting the priorities of execution-queues in addition to checking and resolving dependencies among fragments.

3.3 Logging and Recovery

Q-Store like other deterministic transaction processing systems (e.g., [25, 40]) assumes a deterministic stored procedure based transaction model [1]. Within this model, all inputs of a transaction are available before this transaction can start execution. Therefore, the input of a batch of transactions is logged before they are delivered to execution-threads. Periodic check-pointing of the database state is used to reduce the time required for recovery in case of a failure. In this paper, we mainly focus on transaction execution as we can rely on the same techniques for logging and recovery as [25, 40].

4 FORMALIZING Q-STORE

We now formalize the planning and execution phases of Q-Store. Later in this section, we also prove that Q-Store transaction processing protocol produces serializable histories.

4.1 Planning Transactions

As stated earlier in the previous section, the set of planning-threads $\mathbb{P}_q$ at a server inherit its priority q, and each planning-thread $P_{q,p}$ in the set $\mathbb{P}_q$ has another priority p to prioritize planning-threads of the same server. In general, planning-threads may use any mechanism to create execution-queues as long as they ensure that conflicting operations are placed in the same queue. For example, a *range*-based partitioning of the *record-identifiers* can be used, which ensures that operations accessing the same record are placed in the same execution queue. However, a placement strategy that minimizes the dependencies among execution-queues can yield better performance. More sophisticated approaches based on some cost model are also possible as long as the planning times are minimized and do not introduce significant overhead to the processing latency. The study of such strategies is out of the scope of this paper.

We denote the set of these execution-queues as $\mathbb{Q}_{q,p}$ and individual execution-queue as $Q^i_{q,p}$.

$$\mathbb{Q}_{q,p} := \{Q^1_{q,p},\ Q^2_{q,p}, ..., Q^i_{q,p}\},\ \text{where}\ i \geq 1 \quad (2)$$

In Q-Store, each planning-thread processes a batch of transactions and places its fragments in its respective execution-queues. Hence, each i^{th} execution-queue $Q^i_{q,p}$ contains a set of operations that access the records belonging to that sub-partition, which implies that fragments from two execution-queues created by the same planning-thread have operations, access records in different sub-partitions, and any conflicting fragments (i.e., access the same records) are placed in the same execution-queue.

Q-Store's planning-threads try to balance the load and create execution-queues equal to the number of execution-threads in the system. Such planning is done to keep execution-threads from being idle. More formally:

$$\forall\, P_{q,p}, \in \mathbb{P}_q,\ |\mathbb{Q}_{q,p}| \geq |\mathbb{E}| \tag{3}$$

However, it is undesirable in practice to have the number of execution-queues much larger than the number of execution-threads because it can lead to performance degradation due to low-level issues (e.g., cache-locality).

Each transaction can perform multiple operations. These operations can be grouped into fragments if they are accessing the same record. Otherwise, a fragment has a single operation. We denote the set of fragments in a transaction T as O_T. For presentation simplicity, let us assume that each fragment $O_{k,T}$ in set O_T can be either a read (R) or a write (W).

$$O_T := \{O_{1,T},\ O_{2,T}, ..., O_{k,T}\}$$

A planning-thread may distribute the fragments of a given transaction across multiple execution-queues. Q-Store needs to impose an order to the transactions that are being planned. This order can be as simple as the order imposed by the *client-transaction-queue*. A transaction and its fragments inherit the priorities of its planning-thread.

Hence, we can identify the order of a transaction T using a triple (i, p, q), where q is the priority of the server, p is the priority of the planning thread, and i can be the order imposed by the client-transaction-queue.

$$\forall\, i, j,\ i < j \rightarrow T_{(j,p,q)} \xrightarrow{\text{follows}} T_{(i,p,q)} \tag{4}$$

Equation 4 shows that as $T_{(i,p,q)}$ has a smaller identifier (i) than $T_{(j,p,q)}$, so it must have been placed in the client-transaction-queue before $T_{(j,p,q)}$.

Since transactions may have operations accessing remote partitions, planning-threads similarly create remote execution-queues to be executed at remote nodes. Note that our notation of an execution-queue $Q^i_{q,p}$ identifies the priority q of a remote server, which guides the execution phase. Therefore, queue execution at remote nodes is also deterministic.

When the planning-threads have collectively processed a set of transactions, they mark the resulting batch of execution-queues (local and remote) as ready for execution and deliver them to (local and remote) execution threads.

4.2 Speculatively Executing Transactions

The execution phase is performed by a set of execution-threads. Each server consists of a set of execution-threads $\mathbb{E}$.

$$\mathbb{E} := \{E_1, E_2, ..., E_j\},\ \text{where } j \geq 1$$

We require all the execution-threads to adhere to the following condition strictly:
`Condition`: *For each record, operations belonging to higher priority execution-queues must always be executed before executing any lower priority operations.*

$$\begin{gathered}\forall\, Q_m \in \mathbb{Q}_{q,p},\ \forall\, Q_n \in \mathbb{Q}_{s,t},\ \forall\, O_i \in Q_m,\ \forall\, O_j \in Q_n \\ \mid (q > s)\ \vee\ ((q = s)\ \wedge\ (p > t)) \rightarrow O_j \xrightarrow{\text{follows}} O_i\end{gathered} \tag{5}$$

This condition ensures that the order of executed operations follows a single order within and across servers. In other words, Q-Store requires execution-threads $\mathbb{E}$ to process the operations from those execution-queues, which have the highest priority among all the servers and planning-threads. However, Q-Store does allow the execution-queues produced by a single planner thread to be executed in parallel because they have the same priority.

Execution-threads process fragments from the execution-queues speculatively such that fragments are allowed to read uncommitted data (speculating that it would commit at a later time). Q-Store tracks these speculative actions and captures corresponding speculation dependencies (Section 2.2).

When a violation of an integrity constraint causes a transaction to abort, other fragments of the same transaction that have updated records must rollback as well. The other fragments may have uncommitted updates that have been read by fragments belonging to other transactions. In this case, dependent fragments and their respective transactions must rollback, causing a cascade of aborts through the batch.

4.3 Conservatively Executing Transactions

Q-Store also seamlessly supports a conservative execution, which introduces stalls when processing queues, but has the advantage of avoiding cascading aborts. In Q-Store, a transaction is aborted when the transaction logic induces an abort (e.g., for violating an integrity constraint). By design, non-deterministic aborts (e.g., for ensuring deadlock-free execution) do not exist in Q-Store.

Looking back at our example illustrating transaction dependencies from Section 2.2, Fragment O_{1,T_4} depends on O_{2,T_4} which is abortable. In conservative execution, the execution-thread executing $EQ^2_{q,p}$ stalls until the dependency is resolved. The event of resolving the dependency indicates that O_{2,T_4} is not going to abort. Therefore, any records updated by fragment O_{1,T_4} are safe for any read operations by subsequent fragments in the execution-queue.

Fragments are marked by planning-threads to ensure that execution-threads know when to wait and stall the processing of an execution-queue. When execution-threads encounter a marked fragment, they stall waiting for its commit dependencies to be resolved. Execution-threads can work on other execution-queues if they need to stall due to unresolved commit dependencies. Therefore, we are still exploiting parallelism by allowing other fragments to execute. If an integrity constraint violation happens, then, only one transaction is aborted and rollbacked.

4.4 Serializability

We now prove the serializability guarantees of Q-Store's transaction processing model.

THEOREM 4.1. Q-Store*'s distributed transaction processing is serializable.*

PROOF. One principle of our queue-oriented paradigm is to treat local and remote execution-queues in the same way. Therefore, the fact that an execution-queue is remote or local is an orthogonal concept.

Let us assume that Q-Store produces a non-serializable history, which means that there exist 4 transaction fragments that are executed in an incorrect order. Let these fragments be as follows: $O_{i,T_n}, O_{j,T_n}, O_{k,T_m}$ and O_{l,T_m}. Here O_{i,T_n} conflicts with O_{k,T_m} and O_{j,T_n} conflicts with O_{l,T_m}. Further, let $n < m$ in the client transaction queue, which means that a planner plans T_n before T_m. A non-serializable history means that at one execution-queue O_{i,T_n} is executed before O_{k,T_m} while O_{k,T_m} is executed before O_{i,T_n} at another execution node. More formally,

$$\exists Q_a, Q_b \text{ s.t. } \{O_{i,T_n}, O_{k,T_m}\} \in Q_a \wedge \{O_{j,T_n}, O_{l,T_m}\} \in Q_b \tag{6}$$

Furthermore, the following constraint captures one possible non-serializable history for the transaction fragments.

$$O_{k,T_m} \xrightarrow{\text{follows}} O_{i,T_n} \land O_{j,T_n} \xrightarrow{\text{follows}} O_{l,T_m} \tag{7}$$

The other non-serializable history is captured by:

$$O_{i,T_n} \xrightarrow{\text{follows}} O_{k,T_m} \land O_{l,T_m} \xrightarrow{\text{follows}} O_{j,T_n} \tag{8}$$

Either Q_a or Q_b has fragments from T_m ordered before T_n, which contradicts the fact that the planner of Q_a and Q_b planned T_n before T_m.

□

4.5 Read-committed Isolation

Not only that, Q-Store supports multiple execution paradigms but also multiple isolation levels seamlessly using the queue-oriented paradigm. Supporting read-committed isolation requires planning-threads to produce an additional set of execution-queues $Q'^{j}_{q,p}$ such that they only contain read-only transaction fragments as shown in Eq. 9. Read-only transaction fragments do not perform any write operations.

$$\begin{aligned} &\forall\, P_{q,p}, \in \mathbb{P}_q, \mathbb{Q}_{q,p} := \{Q^1_{q,p},\ Q^2_{q,p}, ..., Q^i_{q,p}\} \\ &\cup \{Q'^1_{q,p},\ Q'^2_{q,p}, ..., Q'^j_{q,p}\},\ \text{where } i \geq 1, j \geq 1 \end{aligned} \tag{9}$$

Furthermore, Q-Store employs a copy-on-write technique that creates a private copy of the updated records. Using these two simple techniques, Q-Store can support read-committed isolation seamlessly.

4.6 Discussion

The performance of speculative execution is dependent on the workload. Two properties of the workload can degrade the performance of speculative execution. The activation of logic-induced aborts which leads to the cascading aborts phenomena. The conservative execution solves this issue at the cost of more coordination among execution-threads.

For either of the execution paradigms, there is another workload property that impacts their performance negatively. The existence of a large number of data dependencies among fragments (see Section 2.2) in the planned workload limits the concurrency because it forces additional coordination among threads to resolve these data dependencies.

In Q-Store, mitigating the impact of data dependencies require more intelligent planning. Planning-threads can minimize the data dependencies among execution-queues. However, solving the minimization problem cannot introduce significant latency. Furthermore, because the database is partitioned, this can only work for local execution-queues. Planning-threads can intelligently move read-only fragments to a special set of execution-queues that allow resolving data-dependencies before executing dependent fragments. The implementation of these optimization remains as future work.

Limitations Advantages and disadvantages of deterministic transaction processing are discussed in the literature [36]. The key limitation of deterministic transaction processing is that the knowledge of the full read/write sets is required. One approach is to run the transaction without committing its write-set to compute the full read/write sets [40]. In general, this approach does not guarantee the finality of the read/write set when running the transaction. Another approach is to partially execute the transaction over multiple batches instead of a single batch. The study of these approach is beyond the scope of this paper.

5 IMPLEMENTATION

We now present some key details for our implementation of Q-Store. In our implementation of Q-Store, we model various components of Q-Store as a set of producers and consumers. As stated in Section 3, Q-Store includes a set of communication-threads. These threads perform two tasks: (i) consuming messages from the network and storing them in respective queues, and (ii) consuming messages from the worker-threads and pushing those on to the network. The task of consuming messages from the network involves reconstructing the raw buffers into appropriate message types so that other threads can interpret them.

In Q-Store, we partition the database using a range-partitioning scheme. At each server, we allocate an equal number of worker threads that assume the roles of both the planning-threads and execution-threads but only one role at a time. This scheme simplifies both the planning and execution phases as computing the number of sub-partitions across the whole cluster requires no additional communication.

When an input thread receives a client transaction, it places the transaction into a client-transaction-queue associated with one of the planning-threads, in a round-robin fashion. We allocate one client-transaction-queue for each planning-thread. This approach eliminates contention among the planning-threads to fetch the next transaction.

Q-Store employs a *count-based batch demarcation* mechanism which requires Planning-threads to create batches of transactions containing a specific number of transactions. However, *time-based implementations* for defining batches are also possible (e.g., a batch is created every 5 milliseconds).

Our Q-Store's implementation requires minimal low-level synchronization among all the threads in the system. Communication threads and worker threads utilize lock-free data structures to interact. For instance, if a worker thread is currently acting as a planning-thread, then as soon as it has processed the required number of transactions for the next batch and created its execution-queues, it starts acting as an execution-thread and checks for any available execution-queue to process. When it has executed all the required execution-queues, then it resumes the role of a planning-thread.

Batch Meta-data Q-Store requires execution-threads to process both the local execution-queues and remote execution-queues. This requirement implies there is a need to store locally generated execution-queues and incoming remote execution-queues. We employ a distributed lock-free data-structure, which we refer to as the *Batch meta-data* (illustrated in Figures 3 and 4), to store these execution-queues as well as any relevant meta-data needed to fulfill transactions dependencies. The implementation of dependencies uses a count to represent the number of dependencies to be resolved. When a dependency is resolved, we use atomic operations to decrement the dependency count. The communication-threads push the incoming remote execution-queues directly to the batch meta-data, which makes these queues available for execution. In this case, communication-threads are acting as virtual planning-threads. Execution-threads access this batch meta-data to fetch any available remote execution-queues. Moreover, the batch meta-data also stores the incoming acknowledgment messages (ACK), which an execution-thread transmits after processing a remote execution-queue, and the commit protocol uses them.

Commitment Protocol Q-Store's design allows us to support two light-weight commitment protocols. We can commit a transaction as soon as its last operation has been processed when using conservative execution. Alternatively, we can defer the commitment of all the transactions to the end of the batch when using speculative execution.

Note that the former approach requires additional implementation complexity to ensure that committed transactions do not read uncommitted updated from aborted transactions. The latter approach could cause a non-trivial increase in the latency at the client because all transactions are committed at the end. However, the latter approach also helps the system to amortize the cost of the commit protocol over a batch of transactions [7].

One of the key advantages of employing deterministic transaction processing protocols is that non-deterministic aborts are no longer possible (e.g., aborts induced by concurrency control algorithms). Therefore, there no need to rely on *costly commit protocols*, such as 2PC.

For speculative execution in Q-Store, the commit protocol commits the whole batch after all the execution-queues are processed. On completing the execution of an execution-queue, the worker thread sends an ACK message notifying the planner's node about it. When the planner's node receives the ACK message, it updates the batch meta-data associated with the remote execution-queue. Further, Q-Store requires the local execution-threads to directly update the batch meta-data. When all the local execution-queues are executed and remote execution-queues are acknowledged, the planner node starts the commit stage for the planned transactions.

To commit a particular transaction, we check if all of its fragments' dependencies are resolved. If so, the transaction is committed. Otherwise, the transaction needs to be aborted, and the rollback process is started. During rollback, the speculative dependency path is walked, and dependent transactions are aborted. Note that, in the conservative execution, there are no speculative dependencies, and there are no cascading aborts.

6 EVALUATION

In this section, we present an extensive evaluation of Q-Store. We implement our techniques in ExpoDB [18, 19, 35]. We compare the performance of Q-Store's speculative execution with the following concurrency control techniques. The conservative execution's performance evaluation and analysis remain future work.

- NO-WAIT: A representative of pessimistic protocols. A two-phase locking (2PL) variant that aborts a transaction if a lock cannot be acquired [3].
- TIMESTAMP: A basic time-ordering protocol [3] that is a representative of time-ordering concurrency control protocols.
- MVCC: An optimistic concurrency control protocol that relies on maintaining multiple versions of the accessed records. We select MVCC as representative of multi-version concurrency control protocols.
- MaaT: An optimistic concurrency control protocol [29] that is a representative of optimistic concurrency control protocols.
- Calvin: A deterministic transaction processing protocol [40].

We use a range-based partitioning instead of the original hash-based partitioning used by [20].

Cluster Setup We use a total of 32 Amazon EC2 instances for all experiments (16 server nodes and 16 client nodes). The instance type *c5.2xlarge*, which has 16GB of RAM and 8 vCPUs.

Table 1: Workload configurations parameters. Default values are in parenthesis.

Parameter Name	Possible Parameter Values
Common parameters:	
% of multi-partition txns.	1%, 5%, 10%, 20%, (50%), 80%, 100%
YCSB Workloads:	
Zipfian's theta	(0.0), 0.4, 0.8, 0.9, 0.99
% of write operations	0%, 5%, 20%, (50%), 80%, 95%
Operations/txn.	2, 4, 8, 12, (16)
Partitions accessed/txn.	2, 4, (8), 12, 16
Server nodes counts	2, 4, 8, (16)
Batch sizes	$5K$, $10K$, $20K$, $40K$, ($80K$), $160K$, $320K$
TPC-C Workloads:	
% of Payment txn.	0%, 50%, 100%

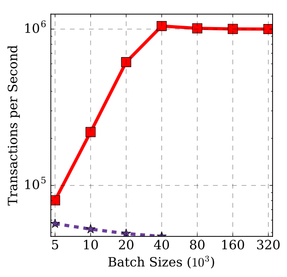
We use Ubuntu 16.04 (xenial), GCC 5.4, Jemalloc 4.5.0 [2, 23] and compile our code with -O2 compiler optimization flag. We pin threads to cores to reduce the variance from the operating system scheduling and the effect of the caching system. Each dedicates 4 threads as worker threads, and 4 as communication threads. For Calvin, 2 out of the 4 worker threads are dedicated to sequencing and scheduling tasks. Each client node maintains a load of $10K$ active concurrent transactions.

Workloads We use two common macro-benchmarks for our evaluation. The first one is YCSB [5]. YCSB is representative of web applications used by YAHOO. The YCSB benchmark is modified to have transactional capabilities by including multiple operations per transaction. Each operation can be either a `READ` or a `READ-MODIFY-WRITE` operation. The benchmark consists of a single table that is partitioned across server nodes, and each node hosts 16 million records. The benchmark can be configured to capture various workload characteristics.

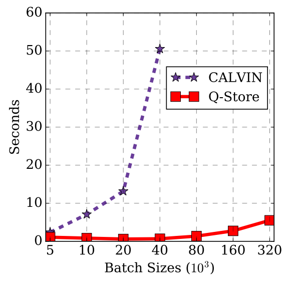
We also experiment with workloads based on the industry-standard TPC-C [41]. The TPC-C benchmark simulates a wholesale order processing system. There are 9 tables and 5 transaction types in this benchmark. All tables are partitioned across server nodes, where a partition can host one or more warehouses. Similar to previous studies in the literature[20, 45], we focus on the two main transaction profiles (`NewOrder` and `Payment`) out of the five transaction profiles, which correspond to 88% of the default TPC-C workload mix [41].

We report the average of 3 trials where each experiment trial runs for 120 seconds, and we ignore the measurements of the first 60 seconds, as it is used as a warm-up period. All reported measurements are observed by the client-side; thus, they are reflective of practical settings. Table 1, shows the various configuration parameters we used in our evaluation. Unless mentioned otherwise, we employ the default values.

Our experimental evaluation focuses on answering the following questions: (1) How does batch size affects the performance of batch-based distributed transaction processing systems (e.g., Calvin and Q-Store)? How do these systems handle high-volume workloads with large batches of concurrent multi-partition transactions? How do the following workload characteristics impact the performance of distributed transaction processing protocols: (a) the contention induced by data access skew; the percentage of multi-partition transactions in the workload; (b) the percentage of update operations in each transaction; (c) the transaction size (i.e., the number of operation per transaction); (d) the number of partitions accessed per transaction, and; (e) the transaction profiles? (3) How do these transaction protocols scale with respect to the number of nodes in the cluster?



(a) Throughput



(b) 99th Percentile Latency

Figure 5: Impact of varying batch sizes on the system throughput and 99th percentile latency of deterministic systems.

6.1 YCSB Experiments

The YCSB benchmark is versatile, and we use it to answer many of the questions related to sensitivity factors. We start by studying the impact of batch sizes for protocols that rely on batching.

Impact of Batch Sizes Using the YCSB benchmark, we first study the impact of batch size on protocols that rely on batching, such as Calvin and Q-Store. The current implementation of Q-Store uses a count-based batch demarcation mechanism. On the other hand, the original Calvin implementation uses a time-based mechanism. For this set of experiments to be meaningful, we modified Calvin to use a count-based batch demarcation mechanism and make it stand on the same ground as Q-Store. We use the default parameters and varying the batch size from 5K to 320K. The results are shown in Figure 5. Compared to Calvin, Q-Store scales very well as we increase the batch size up to $80K$.

Moreover, Calvin's throughput is very low because both the sequencing layer and the scheduling layer are single-threaded per node. With a large number of transactions per batch, those layers act as a bottleneck for the system. These results also show that Q-Store's architecture can utilize computing and network resources more efficiently. Beyond $80K$, the throughput of Q-Store plateaus as transaction processing becomes CPU-bound, and the latency starts to increase because worker threads take more time to process large batches. Calvin cannot handle large batches as transaction latency values exceed the experiment period. Remarkably, at $40K$ batches, Q-Store demonstrates an improvement of 22.1× the throughput of Calvin and an order of magnitude lower latency.

The most significant insight for Q-Store is that for large deployments (e.g., here, we have a total of 64 worker threads distributed over 16 server nodes), we need more work per thread to ensure efficient transaction processing and to hide the latency. Q-Store can handle large batches of concurrent transactions while keeping the latency low.

The presented results indicate that Q-Store is efficient in terms of performing useful work locally. The bottleneck is in the communication protocol, which is expected because the network is slower than local communication.

In the remaining experiments, we use the original time-based batch demarcation mechanism for Calvin and use their reported parameter of $5ms$ [20]. We observe that with $5ms$ time-based batch demarcation, Calvin produces batches of size 160 per node approximately.

Variable Contention In this set of experiments (Figure 6), we vary the Zipfian skew factor θ from 0.0 (uniform) to 0.99 (extremely skewed). As θ approaches 1.0, the data access becomes more skewed within a partition, but the partitions are chosen uniformly per transaction. In other words, each partition receives uniform access, but the record access within the partition is skewed. It is possible to use a Zipfian distribution for partitions as well, but that would not measure the performance of how each node is dealing with skewness. Further, in such a case, mostly one node is active while the remaining nodes are idle most of the time. We use a 50% multi-partition workload such that the 16 operations in a transaction randomly access exactly 8 partitions.

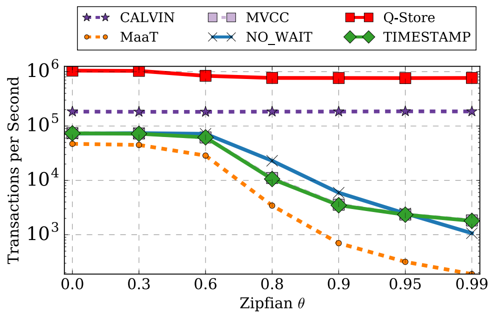


Figure 6: Impact of varying the data access skewness parameter θ of the Zipfian distribution on systems throughput (log scale).

Both Calvin and Q-Store perform better because they both avoid the cost of the two-phase commit protocol (2PC). However, Q-Store achieves up to 6× better throughput. The first reason for that is queue-oriented execution and communication. Q-Store sends a queue of ordered operations that belong to several concurrent transactions to remote nodes. Thus, Q-Store ensures a more efficient communication.

Since different threads execute queues in parallel, Q-Store exploits intra-transaction parallelism (both within a node and across nodes) better than Calvin. For Calvin, the level of contention does not affect its performance because the bottleneck is in the sequencing and scheduling layer. Note that Q-Store's throughput degrades slightly under high-contention (i.e., beyond $\theta = 0.6$) due to the imbalance in the size of execution queues.

The throughputs for non-deterministic protocols are low because they require a costly 2PC protocol for committing each transaction. As the contention increases, the abort rates also increases, which lowers their performance even more. When transactions abort, they are retried using a random back-off period. Under high-contention, transactions may abort multiple times, which effectively increases the latency per transaction, which lowers the throughput. Remarkably, Q-Store achieves nearly two orders of magnitude better system throughput under high-contention in comparison to non-deterministic protocols.

Varying multi-partition transactions rate Now, we focus on the impact of multi-partition transactions in the workload. We vary the percentage of multi-partition transactions in the workload from 0% (single-partition transactions only) to 100% (multi-partition transactions). We fix the values of other parameters to the default values. The results shown in Figure 7 are for low contention (i.e., $\theta = 0.0$). Note that in comparison to Figure 6, there is no noticeable difference in the throughputs of the protocols with single-partition transaction workloads, except for Calvin.

Non-deterministic protocols do not need to perform 2PC, which allows them to avoid 2PC's cost. When the rate of multi-partition transactions increases, non-deterministic protocols incur the overhead of 2PC to ensure serializable execution, and

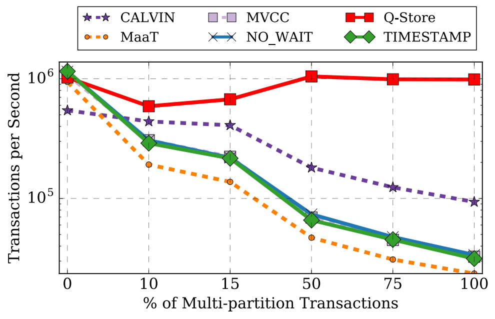


Figure 7: Impact of varying the percentage of multi-partitions transactions in the workload on the system's throughput.

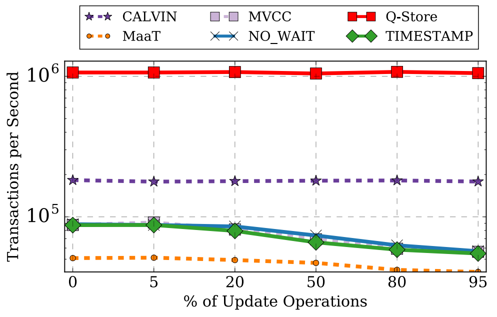


Figure 8: Impact of varying the percentage of update operations in the workload on the system's throughput.

thus, their throughput decreases. Thus, our results validate previously published results (e.g., [20]), which illustrate the poor performance of non-deterministic protocols.

Despite the deterministic nature of Calvin, its throughput also decreases as the rate of multi-partition transaction increases. Calvin needs to send a given transaction to all participants and waits for their responses before scheduling the next conflicting transaction. This approach increases the communication overhead per transaction and negatively affects the performance of Calvin. Unlike Calvin, Q-Store is not sensitive to multi-partition transactions. In addition to avoiding 2PC overhead, it has minimal communication overhead. Q-Store communicates only a minimal number of execution queues between partitions, which contain scheduled operations of several transactions. Thus, it effectively reduces the communication overhead per transaction. Q-Store outperforms Calvin's throughput by up to 10.6×.

Vary the percentage of update operations In the following experiments, we study the impact of the percentage of the update operations on the transaction processing performance. In previous experiments, we used a value of 50%, which means that 8 out of 16 operations are updating the database in each transaction. To study this factor, we vary the percentage of update operations from 0% (read-only operations) to 95%. We fix the remaining parameters to their default values. Note that increasing the rate of update operations increases the contention on records (e.g., exclusive locks induce record contention).

Figure 8 shows the result of varying the percentage of update operations. The results show that neither Q-Store nor Calvin are sensitive to this factor. Calvin employs deterministic locking to avoid aborting transactions unnecessarily while Q-Store executes operations according to their order in a given queue.

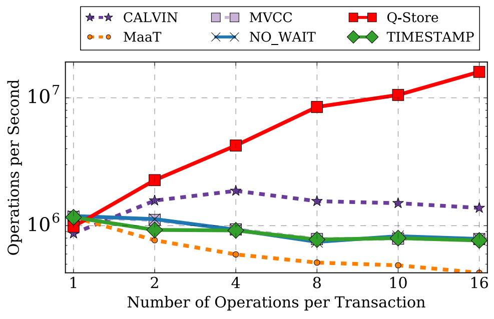


Figure 9: The impact of varying the number of operation per transaction on system's throughput. We force each operation access a different partition. This results is for low contention $\theta = 0.0$.

In other words, for Q-Store there is no difference between the *read* or *update* operations as Q-Store executes each operation in-order, which eliminates any sensitivity to this factor. With non-deterministic protocols, we observe that the abort-rate increases as the contention increases due to more update operations in the workload. For NO-WAIT, MaaT, TIMESTAMP, and MVCC, the abort-rates are up to 41%, 19%, 7%, and 6%, respectively, at 95% update rate.

When a transaction is read-only, there is no need to perform 2PC, but participants still need to communicate messages to finalize the running transaction. As the transaction involves more update operations, the overhead of 2PC protocol becomes more substantial, which negatively affects the performance of non-deterministic protocols that rely on 2PC as their atomic commitment protocol. Notably, Q-Store shows an improvement in its system's throughput by up to 5.9× and 17.1× over Calvin and MVCC (the next best non-deterministic protocol), respectively.

Vary the number of operations per transaction Now, we experiment with varying the number of operations per transaction. We set the percentage of multi-partition transactions to 50%, and force each transaction to access the same number of partitions as its number of operations. For example, if a transaction has 4 operations, the number of partitioned accessed by that transaction is also 4. However, each partition has the same probability of access by any operation, and we do not force operations to be remote.

This experiment aims to capture execution and communication overheads as transactions become larger. For non-deterministic protocols, as the number of operations increases, the cost of 2PC increases because it is more likely that more nodes need to participate in the commitment protocol. Calvin performs better than other non-deterministic protocols, but its performance does not scale with larger transactions. Q-Store, on the other hand, scales well as the number of operations per transaction increases. With 16 operations per transaction, Q-Store's performance reaches a remarkable throughput of nearly 16 million operations per second. These numbers are 12× and 20× better than those for Calvin and NO-WAIT, respectively, as shown in Figure 9. These gains are due to the proposed efficient queue-oriented execution and communication. For Q-Store, the number of queues communicated is constant (but their sizes may vary) while the other protocols exchange messages for remote operations, which increases the overall communication overhead.

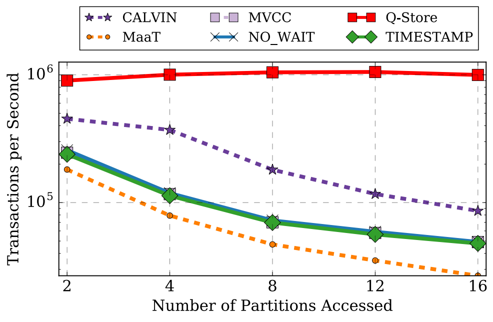


Figure 10: The impact of varying the number of partitions accessed by each transaction on the system's throughput.

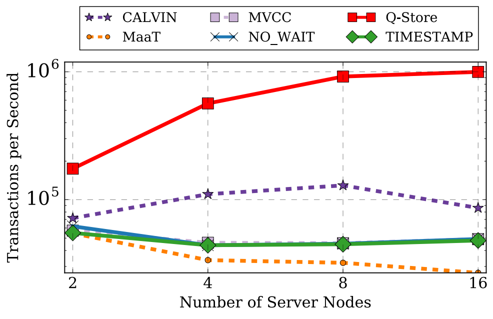


Figure 11: Throughput scalability results while varying the number of server nodes.

Vary partitions per transaction In Figure 10, we show the results for varying the number of partitions accessed by transactions having 16 operations. We use uniform data access, which leads to a low contention workload. By having uniform data access, the effect of contention is negligible, which can help us to examine the communication costs. As we increase the number of partitions accessed by a transaction, the overhead of committing this transaction increases because the commitment involves agreement of more participants per transaction. This issue is mainly a problem for non-deterministic protocols as the participants need to agree on the order for operations. As the number of participants increases, more coordination is required to commit each transaction.

While Calvin eliminates the overhead of 2PC, it still suffers from increasing the number of partitions accessed per transaction. The reasons for that are: (i) it needs to send the transactions to more participants, and (ii) it needs to wait for acknowledgments from more participants before declaring a transaction as committed. This communication overhead increases as the number of partitions accessed increases. In contrast, Q-Store demonstrates its insensitivity to this factor and achieves a throughput of around a million transactions per second despite the increase in the number of partitions accessed per transaction. Since the workload is uniform, the number of partitions accessed affects only the sizes of remote execution queues, and there is no increase in the number of communicated execution queues.

Scalability For all previous experiments, we have used 16 servers. In this set of experiments, we vary the number of nodes to evaluate the scalability. We set the percentage of multi-partition transactions to 50%, and force each transaction to access all available partitions. Figure 11, shows that Q-Store scales well as the number of server nodes increases in the cluster, achieving over

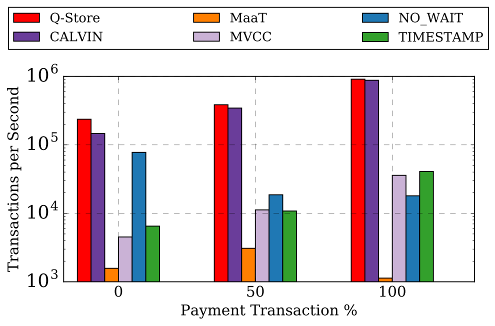


Figure 12: The impact of different TPC-C transaction mixes on the system's throughput. 15% **multi-partition transactions is used.**

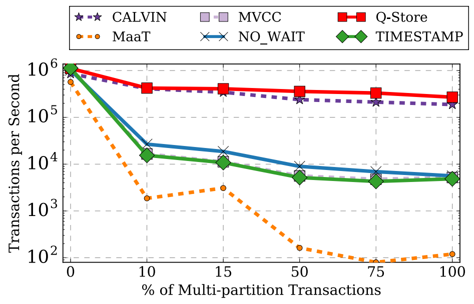


Figure 13: Varying the percentage of multi-partition transaction with equal ratios of Payment and NewOrder transactions.

1 million transactions per second at 16 server nodes. Other approaches do not scale due to the overhead of multi-partition transactions. Calvin's performance cannot scale because of the single-threaded pre-execution phases, while non-deterministic protocols do not scale due to the increased overhead of 2PC.

6.2 TPC-C Experiments

We also evaluate Q-Store with workloads based on the industry-standard TPC-C benchmark. For this set of experiments, we use a total of 16 server nodes, with 4 warehouses per server. Hence, the total number of warehouses is 64. We use three workloads: 100% NewOrder-transaction workload, 50% Payment and 50% NewOrder transactions workload mix, and finally 100% Payment-transaction workload. We use the standard rate of 15% of the payment transactions coming from remote customers as the multi-partition transaction rate, for all the transactions in the workloads. We also restrict the number of partitions accessed to *two* even for NewOrder transactions.

The results are shown in Figure 12. Both deterministic systems Calvin and Q-Store significantly outperform other algorithms by a significant margin due to their use of 2PC. Q-Store outperforms Calvin by up to 1.8×. Remarkably, Q-Store outperforms NO-WAIT, which is the best performing non-deterministic protocol, by up to 55.2×. NO-WAIT suffers from high abort rates due to contended warehouse records (to avoid deadlocks) and the overhead of 2PC for multi-partition transactions. On the other hand, Q-Store eliminates the overhead of 2PC and execution-induced aborts.

The second set of experiments that use TPC-C workloads study the effects of multi-partition transaction rates (Figure 13). The transaction profiles in TPC-C are more complicated than their YCSB counterparts. It involves data dependencies among operations, which can reduce the performance of Q-Store. For example, in the NewOrder transaction, many operations require the new value of the OrderId, which is updated by the same transaction. Our current implementation creates an execution-queue per warehouse, which serializes all operations accessing records belonging to a given warehouse. Despite this unfavorable data partitioning scheme, Q-Store's throughput still outperforms Calvin's throughput. [1]

7 RELATED WORK

Research on distributed transaction processing systems started several decades ago. One of the key challenges in distributed transaction processing is managing the execution of concurrent transactions such that they produce serializable execution histories. Bernstein and Goodman [3] give a comprehensive overview of distributed concurrency control techniques. In this section, we cover some of the recently proposed distributed transaction processing systems and transaction processing techniques that are mostly related to Q-Store. We categorize them as follows.

Non-deterministic Transaction Processing When a transaction updates multiple partitions of the distributed database, there is a need for a commit protocol to ensure that the updates are consistent across all the partitions because nodes may arrive at distinct order of execution for the transaction operations. As a result, aborts may occur non-deterministically. The two-phase commit protocol is typically used to resolve this problem, but naive implementations of 2PC suffer from costly overheads, which negatively impact the system performance. Therefore, many optimizations for 2PC have been proposed (e.g., [26, 29, 31, 39]) while preserving the non-deterministic nature of execution. However, due to this non-determinism, these systems suffer from execution-induced aborts and cannot eliminate the overhead of 2PC [1]. In contrast to these approaches, Q-Store processes transactions deterministically and eliminates the overhead of 2PC and non-deterministic aborts during execution.

Eliminating Multi-partition Transactions Some proposed approaches avoid the cost of 2PC by avoiding the need to process multi-partition transactions. For example, G-store [6] allows applications to declare arbitrary groups of records and moves these groups to a single node to avoid the overhead of processing multi-partition transactions. In a similar spirit, LEAP [27] avoids the cost of 2PC by moving records accessed by a given transaction to a single node at run-time implicitly. Q-Store, on the other hand, embraces multi-partition transactions, and deterministically orders operations into execution-queues; thus avoiding the need for a 2PC protocol.

Deterministic Transaction Processing Deterministic approaches to transaction processing showed great potential in the academic research literature and even had commercial offerings, e.g., [12, 42]. For single-partitioned workloads, H-Store[25] uses single-threaded serial execution per partition. For workloads having multi-partition transactions, H-Store provides limited concurrency by employing a coarse-grained locking mechanism that locks all the partitions prior to the start of a transaction. Jones et al. [24] studies the application of speculative concurrency control to multi-partition transactions in H-Store, which allows transactions to read uncommitted updates of transactions that are performing distributed commitment protocol. Unlike H-Store, Q-Store does not lock partitions to produce a serializable execution for operations of multi-partition transactions. Instead, Q-Store creates execution-queues that capture the serializable order of conflicting operations, and it assigns these execution-queues to worker threads. After that, each worker thread executes its assigned execution-queues according to the pre-determined priority of execution-queues, which allows Q-Store to maintain its high performance despite the multi-partition workloads.

In Gargamel [4], a single dedicated load-balancing node pre-serializes (using static analysis) possibly conflicting transactions before their execution. The load-balancing node can easily become the bottleneck for the system. Unlike Gargamel, Q-Store is centered around the notion of priority and exploits multiple nodes for planning.

Calvin [40, 44] uses determinism to eliminate the cost of two-phase-commit protocol when processing distributed transactions. T-Part [44] relies on the same system architecture of Calvin, but its scheduling layer constructs transaction dependency graphs to reduce the stalling of worker threads. There are fundamental architectural differences between Calvin and Q-Store. The planning phase performs the same functionality as the two-step (sequencing and scheduling) pre-processing phases, but *in parallel*, and the execution phase of Q-Store does not rely on any locking mechanism and employs a queue-oriented (speculative and conservative) processing design. Additionally, in contrast to Calvin, which assigns a transaction to a worker thread for processing, Q-Store assigns an execution-queue to a worker. Because of this thread-to-transaction mapping, Calvin cannot exploit intra-transaction parallelism opportunities within a single node.

Intra-transaction Parallelism Most transaction processing systems perform a thread-to-transaction assignment, which makes these systems unable to exploit intra-transaction parallelism efficiently. Several research studies proposed techniques for exploiting this kind of parallelism in centralized environments (e.g., [10, 34, 35, 43]). Q-Store goes beyond these proposals and exploits intra-transaction parallelism within and across nodes in the context of distributed transaction processing.

8 CONCLUSIONS AND FUTURE WORK

We presented Q-Store, which efficiently processes distributed multi-partition transactions via queue-oriented priority-based execution model. We present a formalization of our system and describe its design and implementation. We perform an extensive evaluation of Q-Store using different workloads from standard benchmarks (that is, YCSB and TPC-C). We demonstrate that Q-Store, consistently and significantly achieves higher performance than existing non-deterministic and deterministic distributed transaction processing systems. We experimentally demonstrate that Q-Store out-performs the state-of-the-art deterministic distributed transaction processing protocol by up to $22.1\times$ with YCSB workloads. Against non-deterministic distributed transaction processing protocols, Q-Store achieves up to two orders of magnitude better throughput with YCSB workloads, and up to $55\times$ with TPC-C workloads.

There are renewed research interests in byzantine fault-tolerance for transaction processing [14–17, 21, 32]. In future, we plan to support byzantine fault-tolerance for database transactions in Q-Store. On the one hand, blockchain transactions are deterministic, which aligns with the kind of transactions that Q-Store's supports. On the other hand, it is very challenging to design and implement

[1] For TPC-C like workloads, unlike Calvin, Q-Store's performance can be further optimized by further splitting execution-queues and exploit parallelism instead of serializing operations per warehouse. However, such optimization is beyond the scope of this paper, and we leave it to future work.

efficient, Byzantine fault-tolerant protocols. We believe that the design principles behind Q-Store can lead to efficient Byzantine fault-tolerant protocols, as very few blockchain proposals look at optimizing execution.

REFERENCES

[1] Daniel J. Abadi and Jose M. Faleiro. 2018. An Overview of Deterministic Database Systems. *Commun. ACM* 61, 9 (Aug. 2018), 78–88. https://doi.org/10.1145/3181853

[2] Jason Evans April. 2006. A Scalable Concurrent Malloc(3) Implementation for FreeBSD.

[3] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (June 1981), 185–221. https://doi.org/10.1145/356842.356846

[4] P. Cincilla, S. Monnet, and M. Shapiro. 2012. Gargamel: Boosting DBMS Performance by Parallelising Write Transactions. In *Proc. ICPADS'12.* 572–579. https://doi.org/10.1109/ICPADS.2012.83

[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. SoCC'10.* ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[6] Sudipto Das, Divyakant Agrawal, and Amr E. Abbadi. 2010. G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In *Proc. SoCC'10 (SoCC '10).* ACM, Indianapolis, Indiana, USA, 163–174. https://doi.org/10.1145/1807128.1807157

[7] 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. In *Proc. SIGMOD'84.* ACM, 1–8. https://doi.org/10.1145/602259.602261

[8] Ramez Elmasri and Shamkant B. Navathe. 2015. *Fundamentals of Database Systems* (7th ed.). Pearson.

[9] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proc. VLDB Endow.* 8, 11 (July 2015), 1190–1201. https://doi.org/10.14778/2809974.2809981

[10] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 613–624. https://doi.org/10.14778/3055540.3055553

[11] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. 2014. Lazy Evaluation of Transactions in Database Systems. In *Proc. SIGMOD'14.* ACM, 15–26. https://doi.org/10.1145/2588555.2610529

[12] FaunaDB. 2019. FaunaDB Website. https://fauna.com/. (2019).

[13] J. N. Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmüller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 393–481.

[14] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2019. Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation. *CoRR* abs/1911.00838 (2019).

[15] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2019. Brief Announcement: Revisiting Consensus Protocols through Wait-Free Parallelization. In *DISC'19.* 44:1–44:3. https://doi.org/10.4230/LIPIcs.DISC.2019.44

[16] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. 2019. Permissioned Blockchain Through the Looking Glass: Architectural and Implementation Lessons Learned. *CoRR* abs/1911.09208 (2019).

[17] Suyash Gupta and Mohammad Sadoghi. 2018. Blockchain Transaction Processing. In *Encyclopedia of Big Data Technologies.* Springer International Publishing, Cham, 1–11. https://doi.org/10.1007/978-3-319-63962-8_333-1

[18] Suyash Gupta and Mohammad Sadoghi. 2018. EasyCommit: A Non-Blocking Two-Phase Commit Protocol. In *Proc. EDBT'18.* https://doi.org/10.5441/002/edbt.2018.15

[19] Suyash Gupta and Mohammad Sadoghi. 2019. Efficient and non-blocking agreement protocols. *Distributed and Parallel Databases* (13 Apr 2019). https://doi.org/10.1007/s10619-019-07267-w

[20] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 553–564. https://doi.org/10.14778/3055540.3055548

[21] Jelle Hellings and Mohammad Sadoghi. 2019. Brief Announcement: The Fault-Tolerant Cluster-Sending Problem. In *DISC'19.* 45:1–45:3. https://doi.org/10.4230/LIPIcs.DISC.2019.45

[22] IBM. 2014. DB2 Isolation Levels. http://disq.us/t/2s92c84. (Oct. 2014).

[23] Jemalloc. 2018. Jemalloc Website. http://jemalloc.net/. (2018).

[24] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *Proc. SIGMOD'10.* ACM, New York, NY, USA, 603–614. https://doi.org/10.1145/1807167.1807233

[25] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[26] Butler W. Lampson and David B. Lomet. 1993. A New Presumed Commit Optimization for Two Phase Commit. In *Proc. VLDB'93.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 630–640.

[27] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *Proc. SIGMOD'16.* ACM, New York, NY, USA, 1659–1674. https://doi.org/10.1145/2882903.2882923

[28] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: Scaling Transactions through Asymmetric Replication. *PVLDB* 12, 11 (2019), 1316–1329.

[29] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2014. MaaT: Effective and Scalable Coordination of Distributed Transactions in the Cloud. *PVLDB* 7, 5 (Jan. 2014), 329–340. https://doi.org/10.14778/2732269.2732270

[30] Microsoft. 2019. SQL Server Isolation Levels. https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql. (2019).

[31] C. Mohan, B. Lindsay, and R. Obermarck. 1986. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.* 11, 4 (Dec. 1986), 378–396. https://doi.org/10.1145/7239.7266

[32] Faisal Nawab and Mohammad Sadoghi. 2019. Blockplane: A Global-Scale Byzantizing Middleware. In *ICDE'19.* 124–135. https://doi.org/10.1109/ICDE.2019.00020

[33] Oracle. 2019. Data Concurrency and Consistency - 11g Release 2 (11.2). https://docs.oracle.com/cd/E25054_01/server.1111/e25789/consist.htm. (2019).

[34] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-Oriented Transaction Execution. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 928–939. https://doi.org/10.14778/1920841.1920959

[35] Thamir M. Qadah and Mohammad Sadoghi. 2018. QueCC: A Queue-Oriented, Control-Free Concurrency Architecture. In *Proc. Middleware '18.* ACM, Rennes, France, 13–25. https://doi.org/10.1145/3274808.3274810

[36] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2014. An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems. *Proc. VLDB Endow.* 7, 10 (June 2014), 821–832. https://doi.org/10.14778/2732951.2732955

[37] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2018. L-Store: A Real-Time OLTP and OLAP System. In *Proc. EDBT'18.* 540–551. https://doi.org/10.5441/002/edbt.2018.65

[38] Mohammad Sadoghi and Spyros Blanas. 2019. Transaction Processing on Modern Hardware. *Synthesis Lectures on Data Management* 14, 2 (March 2019), 1–138. https://doi.org/10.2200/S00896ED1V01Y201901DTM058

[39] George Samaras, Kathryn Britton, Andrew Citron, and C. Mohan. 1993. Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment. In *Proc. ICDE'93.* IEEE Computer Society, Washington, DC, USA, 520–529.

[40] Alexander Thomson, Thaddeus Diamond, Shu C. Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proc. SIGMOD'12.* ACM, 1–12. https://doi.org/10.1145/2213836.2213838

[41] TPC. 2010. *TPC-C, On-Line Transaction Processing Benchmark, Version 5.11.0.* TPC Corporation.

[42] VoltDB. 2019. VoltDB. https://www.voltdb.com/. (2019).

[43] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling Multicore Databases via Constrained Parallel Execution. In *Proc. SIGMOD'16 (SIGMOD '16).* ACM, New York, NY, USA, 1643–1658. https://doi.org/10.1145/2882903.2882934

[44] Shan-Hung Wu, Tsai-Yu Feng, Meng-Kai Liao, Shao-Kan Pi, and Yu-Shan Lin. 2016. T-Part: Partitioning of Transactions for Forward-Pushing in Deterministic Database Systems. In *Proc. SIGMOD'16.* ACM, New York, NY, USA, 1553–1565. https://doi.org/10.1145/2882903.2915227

[45] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220. https://doi.org/10.14778/2735508.2735511

[46] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System. *PVLDB* 11, 10 (2018), 1289–1302. https://doi.org/10.14778/3231751.3231763

Ensemble Grammar Induction For Detecting Anomalies in Time Series

Yifeng Gao
George Mason University
Fairfax, VA
ygao12@gmu.edu

Jessica Lin
George Mason University
Fairfax, VA
jessica@gmu.edu

Constantin Brif
Sandia National Laboratories
Livermore, CA
cnbrif@sandia.gov

ABSTRACT

Time series anomaly detection is an important task, with applications in a broad variety of domains. Many approaches have been proposed in recent years, but often they require that the length of the anomalies be known in advance and provided as an input parameter. This limits the practicality of the algorithms, as such information is often unknown in advance, or anomalies with different lengths might co-exist in the data. To address this limitation, previously, a linear time anomaly detection algorithm based on grammar induction has been proposed. While the algorithm can find variable-length patterns, it still requires preselecting values for at least two parameters at the discretization step. How to choose these parameter values properly is still an open problem. In this paper, we introduce a grammar-induction-based anomaly detection method utilizing ensemble learning. Instead of using a particular choice of parameter values for anomaly detection, the method generates the final result based on a set of results obtained using different parameter values. We demonstrate that the proposed ensemble approach can outperform existing grammar-induction-based approaches with different criteria for selection of parameter values. We also show that the proposed approach can achieve performance similar to that of the state-of-the-art distance-based anomaly detection algorithm.

1 INTRODUCTION

Time series anomaly detection is an important task, with applications in a broad variety of domains. Many approaches have been proposed in recent years, but often they require that the length of the anomalies be known in advance and provided as an input parameter. Recently, this limitation has been addressed by introducing a time series anomaly detection approach that is based on grammar induction and has a linear time complexity with respect to the data size (the time series length).

Typically, the grammar-induction-based anomaly detection follows a four-step process. In the first step, the input time series is converted into a discrete sequence of symbols via a sliding window; this discretization depends on two parameters. In the second step, grammar induction (e.g., via the Sequitur algorithm [15]) is applied to the discrete sequence to quickly identify grammar rules that are repeating strings of symbols. The third step maps the repeating strings back to the time series subsequences that the strings represent. Finally, a meta time series named *rule density curve* is computed, which records the frequency of grammar rules at each point and is used to detect and rank the anomalies. Specifically, it is assumed that anomalies correspond to rarely occurring strings and hence are indicated by minima of the rule density curve.

While the grammar-induction-based algorithm can find variable-length patterns, it still requires preselecting values for at least two parameters at the discretization step. These two discretization parameters are the number of segments (i.e., the length of a word, also called PAA size which will be explained later), and the alphabet size. How to choose the parameter values properly is still an open problem, especially in an unsupervised setting where no training data are available.

Figure 1 presents an example that illustrates the challenge of choosing proper parameter values, even with the knowledge of ground truth information that allows us to evaluate the quality of anomalies detected. Figure 1.top shows a snippet of dishwasher electricity usage time series. An anomalous cycle that has an unusual short power usage period is highlighted in red. We ran the algorithm with different parameter value combinations, and the results are shown in Figure 1.bottom. According to the figure, the best parameter value combination (indicated by the arrow) is significantly different from the second best combination. Moreover, we see that parameter values that are close to the optimal values actually perform badly. As a result, guessing the "best" parameter values can be very tricky.

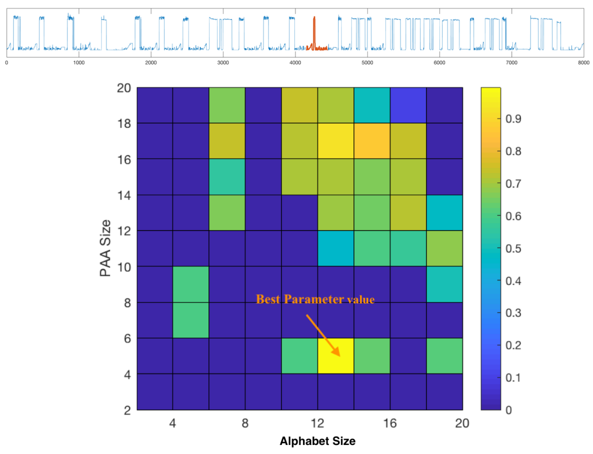


Figure 1: (top) A snippet of dishwasher electricity usage time series. An anomalous cycle is highlighted in red. (bottom) Performance of the grammar-induction-based algorithm with different discretization parameter values in the task of detecting the anomalous subsequence.

To overcome the challenge mentioned above, in this paper, we introduce a robust version of the grammar-induction-based anomaly detection method, which utilizes ensemble learning. Intuitively, instead of using a single combination of parameter values, the method generates the final result based on a set of results obtained using different parameter values. We design an approach that combines the results returned by each ensemble member corresponding to a different combination of parameter values. Furthermore, in the discretization step, we use an approach to compute multi-resolution words to efficiently represent time series subsequences. This approach can dramatically reduce the cost of discretization and increase the scalability of the proposed work.

The contribution of this paper is summarized as follows:

- The proposed ensemble grammar induction can achieve better performance than the grammar induction with a single combination of parameter values.
- The proposed ensemble grammar induction is particularly suitable for unsupervised anomaly detection, where no training set is available to perform a grid search for the best parameter values.
- The proposed approach has a linear time complexity with respect to the data size and can achieve performance comparable to that of the state-of-the-art distance-based approach that has a quadratic time complexity.
- We adapt a fast algorithm to discretize subsequences, which reduces the cost of repeated computation.
- We demonstrate that the ensemble approach can be applied to find meaningful anomalies in real-world application.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 presents definitions and notations used in the paper. The process used for discretization and numerosity reduction is described in Section 4. The grammar-induction-based anomaly detection is presented in Section 5. The ensemble approach is introduced in Section 6. The experimental results are shown in Section 7, and conclusions are summarized in Section 8.

2 RELATED WORK

First, we describe the state-of-the-art time series anomaly detection approach. Keogh et al. [9] introduced a concept named *time series discord*, which is the subsequence that has the largest one–nearest-neighbor (1-NN) distance, hence it represents the most unusual subsequence in the time series. The authors introduced an algorithm named HOTSAX to effectively detect the time series discord. Recently, a series of matrix-profile-based approaches, STOMP [23] and STAMP [21], have been introduced for fast computation of 1-NN distances for every subsequence. It has been shown that compared with the original method, STOMP and STAMP can achieve more stable and generally better performance. However, all these methods have a quadratic time complexity with respect to the data size, and the accuracy is sensitive to the length of the discord that has to be specified in advance as an input parameter [20].

In previous work, we proposed a series of approximate time series pattern discovery algorithms called GrammarViz based on grammar induction [10, 19, 20]. The idea is that by learning a context-free grammar from a discrete sequence of symbols that approximates the original time series, one can identify the repeating strings. Recently, in [18], we have extended the idea of grammar-induction-based anomaly detection by introducing the concept of rule density curve, which is a meta time series that records the frequency of grammar rules (repeating strings) at each point of the original time series. Since anomalies correspond to rarely occurring strings, they are indicated by minima of the rule density curve. It has been shown that this approach can achieve competitive performance compared to the state-of-the-art while having linear time complexity. However, the algorithm requires the user to preselect values of two important parameters, and the performance can be greatly affected by these values.

Several ensemble algorithms have been proposed for unsupervised anomaly detection [17, 22]. Most of them use the average performance as the ground truth and select a very small number of detectors to approximate the average performance. All these approaches are introduced for point-based anomaly detection. How to use them for detecting anomalous subsequences is still unclear. Besides, since subsequences extracted from a time series via a sliding window are highly overlapped, the detector used in these approaches [1] cannot resolve this problem.

Other works [2, 4, 8] also focus on anomaly detection; however, these techniques often focus on detecting anomalies in event logs (discrete or mixed data type time series) and the lengths of the anomalies are often very short.

3 NOTATIONS AND PROBLEM DEFINITION

We first describe the fundamental definitions related to time series and grammar induction. We then formulate the problem of time series anomaly detection.

3.1 Notations and Definitions

We start with the definitions related to time series:

Time series $T = t_1, \ldots, t_m$ is a set of observations ordered by time.

Subsequence $T_{p,q}$ of a time series T is a subsequence of elements in T starting from position p and ending at position q, of length $n = q - p + 1$. Typically, $n \ll m$, and $1 \le p \le m - n + 1$.

Subsequences can be extracted from time series via a sliding window. In many applications, we are interested in finding unusual "shapes." Therefore, anomaly discovery result is more meaningful when the method can maintain offset- and amplitude-invariance during the anomaly detection process. This can be achieved by normalizing all subsequences prior to applying an anomaly detection algorithm. **z-normalization** is a procedure that normalizes the mean and standard deviation of a subsequence to zero and one, respectively.

Since the proposed anomaly detection approach is based on grammar induction, we next introduce the definitions related to grammar induction using a toy example shown in Figure 2. Intuitively, grammar induction is a process that induces a hierarchical grammar structure from a **token sequence** S, which is a sequence of discrete tokens (words). In the example, S consists of nine two-letter tokens. The hierarchical grammar structure is represented by a set of **grammar rules**. Each grammar rule represents a repeating string of token segments in S. In the example, two rules R_1 and R_2 represent the repeating token segments *ab*, *bc* and *cc*, *cc*, respectively. Following the terminology used in previous work [15, 20], each grammar rule is also called a **non-terminal** and each token stored in the token sequence is called a **terminal**.

Using the set of grammar rules, the original token sequence S is represented by the compressed sequence R_0. Since compressibility is a measure of regularity (and hence incompressibility is a measure of anomalousness), the compressed sequence plays an important role in motif (repeated patterns) discovery and anomaly detection.

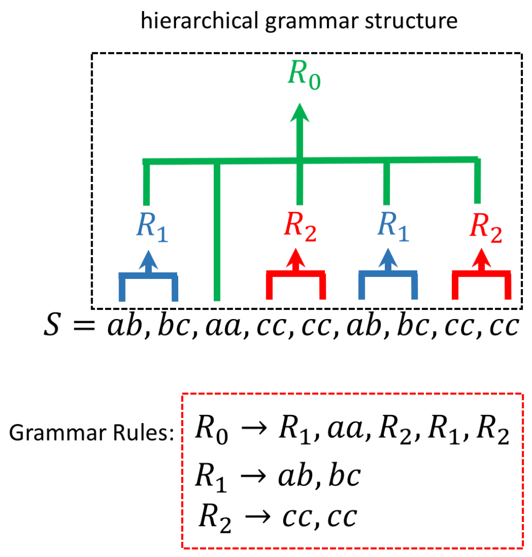

hierarchical grammar structure
R_0
R_1
R_2
R_1
R_2
$S = ab, bc, aa, cc, cc, ab, bc, cc, cc$
Grammar Rules:
$R_0 \rightarrow R_1, aa, R_2, R_1, R_2$
$R_1 \rightarrow ab, bc$
$R_2 \rightarrow cc, cc$

Figure 2: Example of grammar induction applied to a token sequence.

Clearly, the grammar induction approach cannot be directly used for real-valued time series. To use it, we need first to approximate the original time series by a discretized token sequence. We describe the time series discretization process in detail in Section 4.

3.2 Problem of Anomaly Detection

Based on the definitions above, we can formulate the problem of anomaly detection. In this work, we follow the previous anomaly detection framework GrammarViz [20], and determine a hierarchical grammar structure for a time series, through the processes of time series discretization and grammar induction. The anomaly candidates are subsequences that cannot be compressed by induced grammar rules.

To illustrate the basic idea behind this process of anomaly detection, we use another toy example of a simple token sequence:

$$S = aa, bb, cc, xx, aa, bb, cc. \tag{1}$$

The grammar structure induced from S is shown in Table 1. We see that S contains a repeating pattern, aa, bb, cc, represented by the grammar rule R1. The token xx, however, does not appear in any grammar rules (i.e., it is incompressible). It is not hard to see that token xx is *structurally* dissimilar from the rest of the sequence. In the grammar-induction-based time series anomaly detection framework, the subsequence that xx represents is considered an anomaly candidate.

Table 1: Example of grammar rules induced from token sequence S of Eq. (1)

Grammar rules	Expanded sequence
$R_0 \rightarrow R_1, xx, R_1$	$aa, bb, cc, xx, aa, bb, cc$
$R_1 \rightarrow aa, bb, cc$	aa, bb, cc

4 DISCRETIZATION AND NUMEROSITY REDUCTION

Time series discretization [12] is a common step in many time series data mining tasks, including anomaly detection [5, 9]. Since grammar induction requires discrete input, it is necessary to approximate the time series by a token sequence first. In general, discretization offers several advantages including noise removal, dimensionality reduction, and improved efficiency.

In this section, we first describe the algorithm called *Symbolic Aggregate approXimation* (SAX) [12], a widely used time series discretization technique. We then describe *numerosity reduction*, a procedure that removes repeating consecutive tokens to form a more compact token representation of a time series [11, 20].

4.1 Symbolic Aggregate approXimation

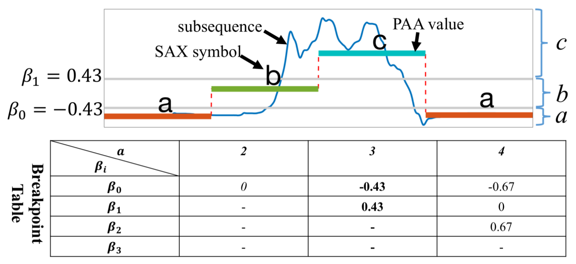
In this section, we describe SAX [12], a popular technique used to discretize univariate time series.

SAX consists of two steps. At the first step, the *Piecewise Aggregate Approximation* (PAA) is used to convert a normalized subsequence of length n from a time series T into a representation of a lower dimension $w < n$ (w is called PAA size). Specifically, the subsequence is divided into w equal-sized windows, and the average value of the elements within each window is computed. In other words, the PAA coefficients vector [12] is a w-dimensional vector that consists of the average values from w equal-sized segments of the input subsequence. PAA coefficients are an approximate representation of the original subsequence.

At the second step, the PAA coefficients vector is mapped to w symbols from an alphabet of size a, according to a breakpoint table [12], defined such that the regions are approximately equal-probable under the Gaussian distribution. This maximizes the chances that the symbols occur with an approximately equal probability. These w symbols form a SAX word.

Figure 3 illustrates the SAX process for an example subsequence (shown as the blue curve). The bold flat lines represent the values of PAA coefficients computed from their respective segments in the subsequence. The breakpoint table with a from 2 to 4 is also shown in the figure. Since we set $a = 3$ in this example, two breakpoints in the second column of the table are used to generate three regions: $(-\infty, -0.43)$, $[-0.43, 0.43)$, $[0.43, \infty)$. The PAA coefficients falling into these three regions are mapped to symbols a, b, and c, respectively. In this example, the SAX word $abca$ is formed to approximate the original subsequence.

This description shows how SAX is applied to a given subsequence. In order to discretize the entire time series, SAX is usually applied via a sliding window of length n.



a / β_i	2	3	4
β_0	0	**-0.43**	-0.67
β_1	-	**0.43**	0
β_2	-	-	0.67
β_3	-	-	-

Figure 3: Example of the SAX process, with $a = 3$ and $w = 4$, which approximates the original subsequence by the word *abca*.

4.2 Numerosity Reduction

In practice, since neighboring subsequences are only off by one point, the neighboring tokens generated by SAX are often identical to each other. This phenomenon, however, will result in an overwhelming number of grammar rules that represent trivial matches, and this redundancy will significantly affect both the scalability and the quality of results. To avoid this problem, a *numerosity reduction* step is added to further compress the token sequence. Specifically, whenever there is a sequence of one token repeating consecutively multiple times, numerosity reduction will output only the first occurrence of the token along with its offset.

For example, a token sequence

$$S = ba, ba, ba, dc, dc, aa, ac, ac \tag{2}$$

can be compressed to

$$S_{\text{NR}} = ba_1, dc_4, aa_6, ac_7, \tag{3}$$

where the subscripts indicate the positions in the original uncompressed token sequence. S_{NR} contains all information needed to retrieve the original token sequence.

5 GRAMMAR INDUCTION

In this section, we first describe Sequitur [15], a grammar induction algorithm with a linear time complexity. We then describe the process of computing the rule density curve and ranking anomalies.

5.1 Sequitur

Sequitur is a linear-time greedy algorithm to induce a context-free grammar structure from a discrete token sequence.

Sequitur maintains a list of grammar rules and a table of digrams based on the input sequence. A digram is a pair of consecutive tokens (terminals or non-terminals) in the sequence. Two principles, digram uniqueness and rule utility, are applied to constrain the rules during grammar induction. Digram uniqueness requires that digrams stored in the digram table should be unique. Rule utility requires that rules that only appear once should be removed to minimize the size of grammar.

To illustrate how Sequitur works, consider an example token sequence generated by SAX with parameters $w = 2, a = 3, n = 16$, after the numerosity reduction step:

$$S_{\text{NR}} = ab_1, bc_8, aa_{15}, cc_{21}, ca_{25}, ab_{29}, bc_{34}, aa_{40}. \tag{4}$$

A step-by-step grammar induction process is shown in Table 2. From Step 1 to Step 6, since neither digram uniqueness nor rule utility is violated, the algorithm simply reads the first six tokens and adds digrams into the digram table, respectively. In Step 7, the algorithm finds that the digram $\{ab, bc\}$ occurs in the digram table twice. Therefore, in Step 8, the algorithm forms a new rule, $R_1 \rightarrow ab, bc$. The new non-terminal symbol R_1 is generated to replace all occurrences of $\{ab, bc\}$, and the digram table is updated to maintain the uniqueness of digrams by removing $\{ab, bc\}$ and adding $\{R_1, aa\}$ and $\{ca, R_1\}$. Similarly, in Step 9, the digram $\{R_1, aa\}$ appears twice and therefore is replaced, in Step 10, by a new rule R_2. After the digram table is updated in Step 10, the algorithm finds that the rule R_1 only appears once. Therefore, in Step 11, R_1 is expanded to satisfy rule utility.

After processing the entire token sequence, the original token sequence is compressed into $R_0 \rightarrow R_2, cc, ca, R_2$ and the string cc, ca is identified as an anomaly candidate since it cannot be compressed.

5.2 Rule Density Curve

We next describe the construction of the rule density curve [20]. Simply stated, a rule density curve is a meta time series, in which each value is equal to the number of grammar rules that "cover" the respective time point. For anomaly detection, we are interested in the intervals for which the rule densities are the lowest. These intervals correspond to subsequences (more precisely, the SAX strings that represent these subsequences) that rarely appear in grammar rules, and hence are potentially anomalous. For the example sequence shown in Table 2, the time series points corresponding to the token subsequence cc, ca will have a count of zero since cc, ca does not appear in any grammar rules.

To construct the rule density curve, we map each instance of a rule back to the subsequence index based on the index recorded in the numerosity reduction step, and then we keep track of the number of rules that cover each time point.

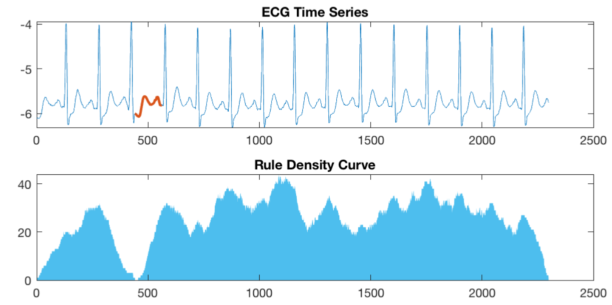


Figure 4: An example of the rule density curve generated for an ECG time series.

Once the rule density curve is constructed, we can locate the potentially anomalous subsequences by finding the local minima of the curve and ranking them based on their respective rule density values.

An example is shown in Figure 4. Figure 4.top shows an electrocardiogram (ECG) time series and Figure 4.bottom shows the rule density curve computed from this time series. An anomaly candidate, highlighted in red in Figure 4.top, corresponds to the minimum of the rule density curve. According to [20], this location corresponds to an anomalous premature heart beat.

5.3 Challenges of Parameter Selection

Although the grammar-induction-based approach described above has been successfully used in time series data mining problems [7, 20], it has two drawbacks that can strongly affect the quality of patterns found. First, approximation errors are inevitably introduced due to the information loss at the discretization step. For example, two subsequences represented by the same SAX word may actually be dissimilar and have a large distance. Second, Sequitur is a greedy algorithm which cannot guarantee the globally optimal result. Therefore, the grammar rules learned may not perfectly reflect the repeating and anomalous patterns in the time series. In summary, a single run of grammar induction with fixed parameter values may simply not be enough to detect high quality anomalies.

In this paper, in order to mitigate the limitations of the existing grammar-induction-based anomaly detection framework while still maintaining high efficiency, we introduce an ensemble approach to generate a rule density curve based on multiple runs with different parameter values.

Table 2: Example of Sequitur inducing grammar from token sequence S_{NR} of Eq. (4)

Step	Grammar rules	Digrams
1.	$S \rightarrow ab_1$	
2.	$S \rightarrow ab_1, bc_8$	$\{ab, bc\}$
3.	$S \rightarrow ab_1, bc_8, aa_{15}$	$\{ab, bc\}, \{bc, aa\}$
4.	$S \rightarrow ab_1, bc_8, aa_{15}, cc_{21}$	$\{ab, bc\}, \{bc, aa\}, \{aa, cc\}$
5.	$S \rightarrow ab_1, bc_8, aa_{15}, cc_{21}, ca_{25}$	$\{ab, bc\}, \{bc, aa\}, \{aa, cc\}, \{cc, ca\}$
6.	$S \rightarrow ab_1, bc_8, aa_{15}, cc_{21}, ca_{25}, ab_{29}$	$\{ab, bc\}, \{bc, aa\}, \{aa, cc\}, \{cc, ca\}, \{ca, ab\}$
7.	$S \rightarrow \boldsymbol{ab_1, bc_8}, aa_{15}, cc_{21}, ca_{25}, \boldsymbol{ab_{29}, bc_{34}}$	$\boldsymbol{\{ab, bc\}}, \{bc, aa\}, \{aa, cc\}, \{cc, ca\}, \{ca, ab\}, \boldsymbol{\{ab, bc\}}$
8.	$S \rightarrow R_1, aa_{15}, cc_{21}, ca_{25}, R_1$ $R_1 \rightarrow ab, bc$	$\{R_1, aa\}, \{bc, aa\}, \{aa, cc\}, \{cc, ca\}, \{ca, ab\}, \{ca, R_1\}$ $\{ab, bc\}$
9.	$S \rightarrow \boldsymbol{R_1, aa_{15}}, cc_{21}, ca_{25}, \boldsymbol{R_1, aa_{40}}$ $R_1 \rightarrow ab, bc$	$\boldsymbol{\{R_1, aa\}}, \{bc, aa\}, \{aa, cc\}, \{cc, ca\}, \{ca, ab\}, \{ca, R_1\}, \boldsymbol{\{R_1, aa\}}$ $\{ab, bc\}$
10.	$S \rightarrow R_2, cc_{21}, ca_{25}, R_2$ $\boldsymbol{R_1 \rightarrow ab, bc}$ $R_2 \rightarrow R_1, aa$	$\{R_2, cc\}, \{cc, ca\}, \{ca, R_2\}$ $\{ab, bc\}$ $\boldsymbol{\{R1, aa\}}$
11.	$S \rightarrow R_2, cc_{21}, ca_{25}, R_2$ $R_2 \rightarrow ab, bc, aa$	$\{R_2, cc_{21}\}, \{cc, ca\}, \{cc_{21}, R_2\}$ $\{ab, bc\}, \{bc, aa\}$

6 ENSEMBLE GRAMMAR INDUCTION

In this section, we introduce the proposed ensemble-based grammar induction approach.

6.1 Ensemble Rule Density Curve

The algorithm that generates the ensemble rule density curve from a set of multiple grammar induction runs is shown in Algorithm 1. First, we generate N rule density curves using values for PAA size w and alphabet size a randomly chosen from intervals $[2, w_{\max}]$ and $[2, a_{\max}]$, respectively (Lines 4–6). Second, we remove low-quality curves based on their standard deviations (Lines 7–10). Third, we normalize each remaining curve to the same scale (Line 11). Finally, we combine the results from all normalized curves by computing the median (Line 14).

Algorithm 1: Ensemble Rule Density Curve

```
1:  Input: time series $T$, sliding window length $n$, ensemble
    size $N$, maximum PAA size $w_{\max}$, maximum alphabet
    size $a_{\max}$, ensemble selectivity $\tau$
2:  Output: ensemble rule density curve $d_e$
3:  $\mathcal{D}$ =[] {Compute $N$ rule density curves with random
    parameter values and compute the standard deviation
    for each of them}
4:  for $i = 1$ to $N$ do
        {Randomly generate parameter values; any $w, a$
    combination is used only once}
5:    $w, a$ = GenerateRandomParam($w_{\max}, a_{\max}$)
6:    $d_i$ = GrammarInduction($T, n, w, a$)
7:    $s_i$ = ComputeStd($d_i$);
8:  end for
9:  index = ArgSort($s$) {Sort the standard deviations in
    descending order}
10: for $i = 1$ to $\tau N$ do {Keep $\tau$% of the density curves and
    normalize each of them}
11:   $d_{\text{norm}} = d_{\text{index}[i]} / \max(d_{\text{index}[i]})$
12:   $\mathcal{D}$.add($d_{\text{norm}}$)
13: end for
14: $d_e$ = ComputeMedian($\mathcal{D}$) {Compute the median for all
    normalized rule density curves}
15: return $d_e$
```

6.1.1 Removing Low-Quality Rule Density Curves.

Not all rule density curves provide reliable information about anomalies. Intuitively, low-quality curves correspond to situations where a grammar rule set has a similar frequency everywhere, and they should be removed from the ensemble to improve the effectiveness of anomaly detection. Towards this end, the algorithm computes the standard deviation for each of the generated rule density curves (Line 8). We then rank the curves based on the standard deviation in descending order and only keep the top τ% of the curves to form the ensemble set $\mathcal{D}$.

An example that illustrates the quality variation among different rule density curves is shown in Figure 5. All four curves in Figure 5 are generated from the ECG time series shown in Figure 4.top. The first two curves colored in blue are the top-2 curves based on the ranking by the standard deviation. The last two curves colored in red are the bottom-2 curves based on this ranking. As seen from the figure, it is very hard to determine the anomaly location from the bottom-2 curves. In contrast, the top-2 curves reveal the anomaly location clearly. This example demonstrates that using the rule density curves with higher standard deviations can help identify anomalies and potentially reduce the number of false positives.

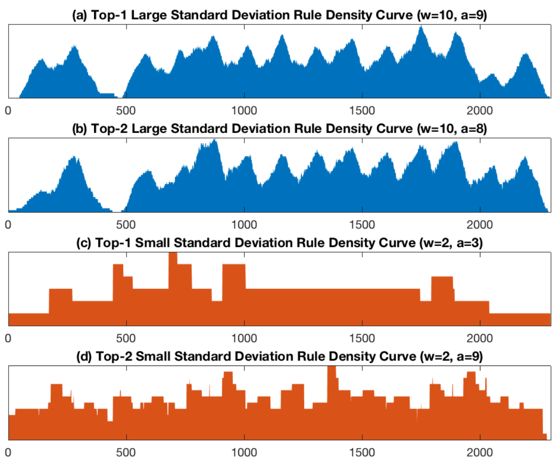


Figure 5: Examples of rule density curves generated from the ECG time series with different parameter values.

6.1.2 *Normalizing Rule Density Curves.*

Intuitively, each rule density curve may have a different scale. For example, using a coarse discretization resolution (i.e., small PAA size and alphabet size) tends to result in a large average frequency of grammar rules since it is more likely to find a match due to a smaller dictionary size. Conversely, using a fine discretization resolution (i.e., large PAA size and alphabet size) tends to result in a small average frequency of grammar rules. Consequently, without a normalization process, some anomaly detectors may undeservedly dominate the decision.

To avoid this problem, we normalize each of the rule density curves in the ensemble set, so that each point of the curve falls in the range [0, 1]. It is worth noting that we do not use min-max normalization because we want to preserve the significance of the locations where the rule density is zero.

6.1.3 *Combining Rule Density Curves.*

At the final step, we compute an ensemble rule density curve d_e based on all normalized curves kept in the set. In this paper, the specific way in which we combine all rule density curves in the ensemble is by computing the median value at each time point. Once the ensemble rule density curve is generated, the algorithm ranks the anomaly candidates through the same process as described in Sec. 5.2.

6.2 Multi-resolution SAX Word Computation

Since the ensemble approach requires computing different SAX words (corresponding to different parameter values) for the same subsequence, it is important to increase the efficiency of the discretization process. Therefore, in this subsection, we describe a fast way to compute multi-resolution SAX words [6].

6.2.1 *Fast Computation of SAX Words with Different Values of w.*

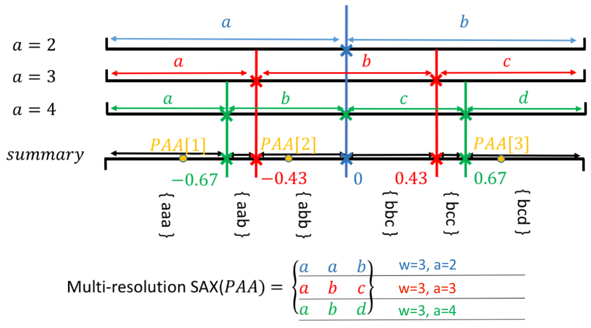
We first describe a fast way to compute the PAA coefficients [16]. First, two vectors of statistical features for a time series T are pre-computed: $\text{ESum}_x(x) = \sum_{i=1}^{x} t_i$ and $\text{ESum}_{xx}(x) = \sum_{i=1}^{x} t_i^2$. Given a subsequence $T_{p,q}$ of length n, the SAX representation can be computed by Algorithm 2. In this algorithm, the mean and variance of $T_{p,q}$ are computed in constant time (Lines 3–5). The cost of computing the PAA coefficients (Lines 6–8) is $O(w)$ for a single resolution, which is faster than the trivial approach whose computation cost is $O(n)$ ($w < n$).

Algorithm 2: Fast Compute PAA (FastPAA)

```
1: **Input**: subsequence $T_{p,q}$, $\text{Esum}_x$,$\text{Esum}_{xx}$, PAA size $w$
2: **Output**: PAA representation $A$
3: $E_x = \text{Esum}_x(q) - \text{Esum}_x(p)$
4: $E_{xx} = \text{Esum}_{xx}(q) - \text{Esum}_{xx}(p)$
5: $n = q - p + 1$, $\mu = E_x/n$, $\sigma = \sqrt{(E_{xx} - E_x^2/n)/(n-1)}$
6: **for** every PAA segment **do**
7:     $A_i = \left(\frac{\text{Esum}_x(A_{i,e}) - \text{Esum}_x(A_{i,s})}{n/w} - \mu\right)/\sigma$ {$A_{i,s}$ and $A_{i,e}$ are the start and end points of the $i$th PAA segment}
8: **end for**
9: **return** $A$
```

6.2.2 *Fast Computation of SAX Words with Different Values of a.*

To efficiently compute SAX words with multiple resolutions, we adapt an algorithm we introduced in a previous work [6]. Specifically, given a maximum alphabet size $a_{\max}$, to fast compute SAX words with different alphabet sizes, we first gather breakpoint tables of all alphabet size values used in the ensemble. For each interval between any two breakpoints, a symbol sequence containing corresponding symbols up to $a_{\max}$ resolution is recorded. We represent any PAA coefficient belonging to the interval by the pre-computed symbol sequence.



$$\text{Multi-resolution SAX}(PAA) = \begin{Bmatrix} a & a & b \\ a & b & c \\ a & b & d \end{Bmatrix}$$

Figure 6: Fast computation of multi-resolution SAX.

An example with alphabet sizes from 2 to 4 is shown in Figure 6. In the figure, the set of breakpoints for all alphabet sizes from 2 to 4 (labeled by '×' in each line) are projected to the line denoted as "summary." All distinct breakpoints with a from 2 to 4 in the breakpoint table (Fig. 3.bottom) create 6 intervals, and each interval stores a sequence of symbols. The i^{th} position in such a sequence stores the corresponding symbol for alphabet size $a = i + 1$. For example, the 3 PAA coefficients that fall in intervals $(-\infty, -0.63]$, $(-0.43, 0]$ and $(0.63, \infty)$ (denoted by yellow dots) are mapped to symbol sequences *aaa*, *abb*, and *bcd*, respectively. A symbol matrix is then created by concatenating such symbol sequences (bottom of Figure 6). The i^{th} row of the symbol matrix represents a SAX word generated with alphabet size $i + 1$. For example, the first row *aab* is the SAX word generated for $a = 2$. To construct the matrix of SAX words with all alphabet sizes, for each PAA coefficient, we only need to perform at most 3 comparisons to determine the interval via binary search, which is the same as generating fixed-resolution SAX. By using binary search to determine which interval the PAA coefficient belongs to, we can find its SAX representations in all resolutions from $a = 2$ to $a = a_{\max}$ with a time complexity of $O(2\log(a_{\max}))$. When $a_{\max} = 20$, the cost of computing all resolutions is similar to computing a fixed resolution.

6.2.3 *Overall Improvement.*

The time complexity of computing SAX words with multiple resolutions in a straightforward manner (without acceleration) is $O(nw_{\max}a_{\max} + w_{\max}^2 a_{\max}^2)$. In contrast, with the proposed acceleration, the computation cost is $O(w_{\max}^2 \log(a_{\max}))$, which is a significant improvement.

7 EXPERIMENTAL EVALUATION

We perform a series of numerical experiments to evaluate the accuracy and speed of ensemble grammar induction applied to time series anomaly detection. In all experiments, unless noted otherwise, the parameter values are: $a_{\max} = 10$, $w_{\max} = 10$, $N = 50$, and $\tau = 40\%$. All the experiments are conducted on a 16 GB RAM laptop with quad core processor of 2.5 GHz. We first show the performance on real-world time series. We then evaluate the impact of parameter sets used in the ensemble. Finally, we analyze the scalability of the algorithm and conduct a case study on an electric load time series.

Table 3: Properties of datasets used for experimental evaluation

Dataset	Time Series Length	Segment Length	Data Type
TwoLeadECG	1772	82	ECG
ECGFiveDay	2772	132	ECG
GunPoint	3150	150	Motion
Wafer	3150	150	Sensor
Trace	5775	275	Sensor
StarLightCurve	21504	1024	Sensor

Table 4: Performance evaluation results (average Score)

Dataset	Proposed Approach	GI-Random	GI-Fix	GI-Select	Discord
TwoLeadECG	0.3951	0.2873	0.0629	0.1663	**0.4931**
ECGFiveDay	0.3903	0.2988	0.2671	0.105	**0.4794**
GunPoint	**0.4728**	0.3715	0.2411	0.056	0.4
Wafer	**0.3179**	0.2126	0.1382	0.248	0.309
Trace	**0.5718**	0.2022	0.3601	0.3408	0.2816
StarLightCurve	**0.9369**	0.6930	0.5301	0.8759	0.9161

Table 5: Performance evaluation results (HitRate)

Dataset	Proposed Approach	GI-Random	GI-Fix	GI-Select	Discord
TwoLeadECG	0.72	0.52	0.4	0.24	0.8
ECGFiveDay	0.8	0.44	0.36	0.24	0.8
GunPoint	0.68	0.56	0.44	0.12	0.68
Wafer	0.72	0.4	0.36	0.4	0.52
Trace	0.96	0.4	0.8	0.6	0.52
StarLightCurve	1.0	0.96	0.76	1.0	1.0

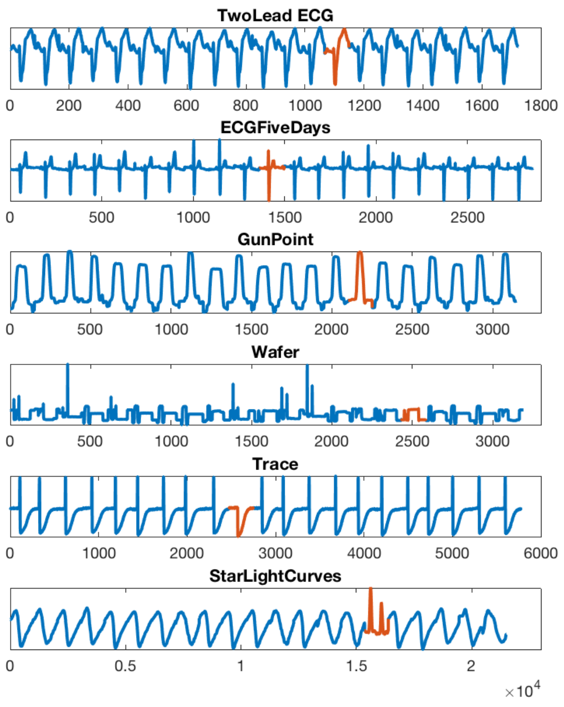


Figure 7: Examples of the test time series generated from six real-world and synthetic time series datasets. In each time series, the segment that belongs to a different class is highlighted in red.

7.1 Performance Evaluation in Comparison to Baseline Methods

Since it is difficult to find annotated time series for anomaly detection, to evaluate our proposed method, we compile our own data using several real-world and synthetic time series datasets from the UCR Time Series Classification Archive [3].

7.1.1 Datasets. We chose six time series datasets with diverse characteristics from the archive. The properties of the data are shown in Table 3. The datasets originate from different application domains including medicine (ECG), 3D motion tracking (GunPoint), manufacturing (Wafer), synthetic sensor data (Trace) and astronomy (StarLightCurve), with different instance lengths (ranging from 82 to more than 1000).

The instances are labeled with class information. For each dataset, we treat all instances that belong to the first class as "normal" data, and all the remaining instances that belong to other classes as "anomalous." We first generate a normal time series by concatenating 20 randomly selected normal instances. Then an anomalous instance is randomly selected and planted into the generated normal time series at a random position between 40% and 80% of the series. In this manner, we generate 25 time series for each of the six datasets. One example of the generated time series for each of the six datasets are shown in Figure 7, with planted anomalous instances highlighted in red. All methods being compared are run to locate the planted anomalous instance in each time series, with sliding window length n equal to the instance length.

7.1.2 Performance Evaluation. Each of the tested methods returns top-3 ranked anomaly candidates (which are required to not overlap with each other) for each time series. We evaluate the performance of each method by quantifying the overlap between the discovered anomalies and the ground truth (the planted anomaly). Specifically, for each anomaly candidate, we compute a quantity called Score, defined as:

$$\text{Score} = 1 - \min\left(1, \frac{|\text{PredictLocation} - \text{GTLocation}|}{\text{GTLength}}\right), \quad (5)$$

where PredictLocation is the location of the anomaly candidate, while GTLocation and GTLength are the location and length of the planted anomalous instance. The maximum value, Score = 1, is obtained when PredictLocation exactly matches the ground truth. The better the overlap between the anomaly candidate and the ground truth, the higher the Score. If the anomaly candidate does not overlap with the ground truth at all, then Score = 0.

For each method and each time series, we only use the maximum Score achieved among the three anomaly candidates. For each method, we record the average Score value computed over the set of 25 time series generated per dataset; the average Score values are reported in Table 4. We also record the quantity called HitRate, which is the fraction of the anomaly candidates that overlap with the ground truth (i.e., satisfy the condition Score > 0); the HitRate values are reported in Table 5. Finally, we record the number of times the ensemble grammar induction method wins/ties/loses against a baseline method; these results are reported in Table 6.

7.1.3 Baselines. The proposed ensemble-based method is compared against four different baseline methods, which are described below.

- **Grammar Induction with Random Parameter Values** (GI-Random): The grammar-induction-based anomaly detection approach with randomly selected parameter values w and a. The ranges from which w and a are chosen are the same as in the ensemble-based approach..

- **Grammar Induction with Fixed Generic Parameter Values** (GI-Fix): The grammar-induction-based anomaly detection approach with fixed parameter values $w = 4$ and $a = 4$. These are popular parameter values that can be used in most of datasets as reported in [20].
- **Grammar Induction with Selected Parameter Values** (GI-Select): The grammar-induction-based anomaly detection approach with w and a values selected via an optimization procedure described in [19], using 10% of the normal time series. The ranges from which w and a are chosen are the same as in the ensemble-based approach.
- **Time Series Discord** (Discord): The state-of-the-art approach that computes the one–nearest-neighbor (1-NN) distance for every subsequence in the time series; the subsequences with the largest 1-NN distances are labeled as anomalies. In the experiments, we use the latest Matrix-Profile-based implementation of this approach [23] to compute the 1-NN distances.

7.1.4 Results. The overall performance results measured by the average Score and HitRate are shown in Tables 4 and 5, respectively.

According to Table 4, the proposed approach achieves the highest average Score values in four out of six datasets, and the second-highest average Score values in two of the datasets. Compared with GI-Random, GI-Fix, and GI-Select, the proposed approach achieves higher average Score values in all six datasets. Compared with Discord, the proposed approach achieves a higher average Score value in four out of six datasets (GunPoint, Wafer, Trace and StarLightCurve). In two datasets (ECGFiveDays and TwoLeadECG), Discord outperforms the proposed approach.

According to Table 5, the proposed approach achieves the highest HitRate values (alone or shared) in five out of six datasets, and the second-highest HitRate value in one of the datasets. These results are consistent with those in Table 4 and indicate that the proposed approach can successfully locate an anomalous subsequence that strongly overlaps with the ground truth.

In addition, Figure 10 shows a detailed comparison summary of the proposed approach against all baselines. Each of the six rows corresponds to one of the datasets, and each of the four columns corresponds to one of the baselines. In each plot, a blue dot located at the point (x, y) denotes a pair of Score values (ensemble Score, baseline Score), computed from Eq. (5) for one of the 25 generated time series. A dot located in the lower triangle (highlighted in pink) of the plot corresponds to a win of the proposed approach (ensemble Score > baseline Score); a point located in the upper triangle corresponds to a loss (ensemble Score < baseline Score), and a point located on the diagonal corresponds to a tie (ensemble Score = baseline Score). The numbers of wins, ties, and losses of the proposed method against the baselines are shown in Table 6.

According to Figure 10, the proposed approach outperforms GI-Random, GI-Fix, and GI-Select. Moreover, in most of the tested datasets, the proposed method often detects ground truth anomalies that are completely missed by these variations of the grammar-induction-based approach (i.e., the respective baseline Score values are zero), while the opposite outcomes (ensemble Score values are zero) are quite rare. Compared with Discord, the proposed approach achieves similar performance. Moreover, in most of the tested datasets, cases where the ensemble-based approach discovers ground truth anomalies missed by Discord are more common than the opposite ones.

According to Table 6, compared with GI-Random, GI-Fix, and GI-Select, the proposed method wins in more than half of the time series in most datasets. Compared with Discord, the results are similar to those measured by Score and HitRate. Specifically, the proposed approach has more wins than losses in three datasets (GunPoint, Wafer, and Trace), loses more often than wins in two ECG datasets, and is in a virtual dead heat with Discord in the StarLightCurves dataset.

In summary, the experiments indicate that using an ensemble of parameter values can significantly improve the performance of the grammar-induction-based approach, compared to the variations of the method where a single combination of parameter values is selected. Also, the proposed method can achieve a competitive performance compared with the state-of-the-art discord approach. Importantly, while the latter has a quadratic time complexity, the former preserves a linear time complexity and hence is much more feasible for anomaly detection in large-scale data.

7.2 Effects of Parameter Value Ranges

In this subsection, we first evaluate how the ensemble grammar induction approach performs for different parameter value ranges determined by $w_{\max}$ and $a_{\max}$. Specifically, we evaluate the performance on the same time series that were used in Sec. 7.1. As the baseline for comparison, we use the best of the GI-Random, GI-Fix, and GI-Select methods for each dataset. We report the number of wins/ties/loses vs. the baseline for all tested parameter value ranges. We then evaluate how values of three hyper parameters — ensemble size N, ensemble selectivity τ, and sliding window length n — affect the result. In these experiments, we evaluate the performance based on Score and HitRate.

7.2.1 Effect of $w_{\max}$ and $a_{\max}$. We first test the proposed approach with values of $w_{\max}$ and $a_{\max}$ equal to each other and varying from 5 to 20. The results are shown in Table 7. According to the results, the smallest ranges ($w_{\max} = a_{\max} = 5$) lead to the worst performance. A possible explanation is that the ranges are too small to generate a sufficient number of high-quality rule density curves. The performance significantly improves for $w_{\max} = a_{\max} = 15$. However, increasing the range values beyond 15 is not useful; furthermore, in two datasets (TwoLeadECG and StarLightCurve) the performance slightly deteriorates when the ranges go up to 20, and in the GunPoint dataset the performance actually peaks for $w_{\max} = a_{\max} = 10$. The lack of performance improvement (or even some deterioration) at range values larger than 15 indicates that long and high-resolution SAX words may capture too much noise in data and, as a result, produce low-quality rule density curves. Nevertheless, the ensemble-based method still outperforms other approaches that rely on a single combination of parameter values.

7.2.2 Effect of $w_{\max}$. We also test the proposed approach with values of $w_{\max}$ varying from 5 to 20 and fixed $a_{\max} = 10$. The results are shown in Table 8. Once again, we observe that when the range is too small ($w_{\max} = 5$), the performance is the worst. The performance significantly improves for larger values of $w_{\max}$, with the peak-performance value depending on the dataset.

7.2.3 Effect of $a_{\max}$. In this experiment, we test the proposed approach with values of $a_{\max}$ varying from 5 to 20 and fixed $w_{\max} = 10$. The results are shown in Table 9. For $a_{\max} = 5$, the performance is subpar for the ECGFiveDays, GunPoint, and

Table 6: Wins/ties/losses of ensemble grammar induction against all baselines

Approach Dataset	TwoLeadECG	ECGFiveDays	GunPoint	Wafer	Trace	StarLightCurve
GI-Random	12/5/8	17/3/5	14/5/6	13/5/7	20/1/4	18/1/6
GI-Fix	17/7/1	13/5/7	15/4/6	17/6/2	14/1/10	24/0/1
GI-Select	14/5/6	18/5/2	16/8/1	9/8/8	14/3/8	17/0/8
Discord	8/4/13	9/1/15	14/7/4	12/5/8	18/1/6	12/0/13

Table 7: Wins/ties/losses of ensemble grammar induction against best GI baseline for different values of $a_{\max}$ and $w_{\max}$

Approach	TwoLeadECG	ECGFiveDays	GunPoint	Wafer	Trace	StarLightCurve
$a_{\max} = 5$, $w_{\max} = 5$	1/12/12	8/9/8	3/9/13	3/14/9	4/11/10	2/0/23
$a_{\max} = 10$, $w_{\max} = 10$	12/5/8	13/5/7	14/5/6	9/8/8	14/1/10	17/0/8
$a_{\max} = 15$, $w_{\max} = 15$	14/4/7	17/2/6	13/4/8	13/7/5	15/0/10	18/0/7
$a_{\max} = 20$, $w_{\max} = 20$	12/4/9	17/2/6	13/4/8	13/7/5	15/0/10	17/1/7

Table 8: Wins/ties/losses of ensemble grammar induction against best GI baseline for different values of $w_{\max}$

Approach	TwoLeadECG	ECGFiveDays	GunPoint	Wafer	Trace	StarLightCurve
$a_{\max} = 10$, $w_{\max} = 5$	5/9/11	6/8/11	5/6/14	7/9/9	4/10/11	1/0/24
$a_{\max} = 10$, $w_{\max} = 10$	12/5/8	13/5/7	14/5/6	9/8/8	14/1/10	17/0/8
$a_{\max} = 10$, $w_{\max} = 15$	10/5/10	18/3/4	11/6/8	18/3/4	15/0/10	19/0/6
$a_{\max} = 10$, $w_{\max} = 20$	12/4/9	18/2/5	10/4/11	14/3/8	16/0/9	20/0/5

Table 9: Wins/ties/losses of ensemble grammar induction against best GI baseline for different values of $a_{\max}$

Approach	TwoLeadECG	ECGFiveDays	GunPoint	Wafer	Trace	StarLightCurve
$a_{\max} = 5$, $w_{\max} = 10$	11/5/9	8/8/9	7/8/10	12/7/6	11/5/9	1/1/23
$a_{\max} = 10$, $w_{\max} = 10$	12/5/8	13/5/7	14/5/6	9/8/8	14/1/10	17/0/8
$a_{\max} = 15$, $w_{\max} = 10$	11/6/8	13/6/6	13/4/8	8/8/9	16/0/9	15/0/10
$a_{\max} = 20$, $w_{\max} = 10$	11/4/10	14/5/6	13/4/8	9/9/7	15/0/10	12/1/12

Trace datasets, and it is terrible for the StarLightCurve dataset (as a rule, this dataset exhibits the worst performance when one or both range values are small). However, $a_{\max} = 5$ actually results in the best performance for the Wafer dataset. For most datasets, the $a_{\max}$ values of 10, 15, and 20 produce very similar results.

The last two experiments indicate that having a larger range for w is more important than for a, which may indicate that a variation in the PAA size has a larger effect on the performance of the algorithm than a variation in the alphabet size, and hence w is a more important parameter to choose, in general.

Table 10: Performance (average Score) vs. N

Dataset	$N = 5$	$N = 10$	$N = 25$	$N = 50$
TwoLeadECG	0.3424	0.3488	0.3912	0.3951
ECGFiveDays	0.37	0.3882	0.4168	0.3903
GunPoint	0.3128	0.4629	0.4965	0.4728
Wafer	0.2308	0.2637	0.2839	0.3179
Trace	0.4767	0.5789	0.5994	0.5718
StarLightCurve	0.8244	0.7593	0.8676	0.9369

Table 11: Performance (HitRate) vs. N

Dataset	$N = 5$	$N = 10$	$N = 25$	$N = 50$
TwoLeadECG	0.52	0.6	0.72	0.72
ECGFiveDays	0.68	0.72	0.76	0.8
GunPoint	0.56	0.76	0.68	0.68
Wafer	0.44	0.64	0.6	0.72
Trace	0.76	0.96	0.96	0.96
StarLightCurve	1.0	1.0	1.0	1.0

7.2.4 Effect of Ensemble Size. We next evaluate the proposed method with the ensemble size N varying from 5 to 50 (recall that all other experiments used the fixed value $N = 50$). The average Score and HitRate of the proposed method for different ensemble sizes are shown in Tables 10 and 11, respectively.

We observe that the performance for a small ensemble size ($N = 5$) is worse than for a larger ensemble size ($N = 25$ or $N = 50$) for all datasets. In general, the Score and HitRate values increase as N grows, but these increases saturate when the ensemble size is large enough (e.g., when $N \geq 25$). The results indicate that selecting $N \geq 25$ may be suitable for most cases.

We also observe that the Score values are, in general, rather low for all methods, for most datasets except StarLightCurve. This could be due to the fact that the Score is normalized by the ground truth length (cf. Eq. (5)). However, the planted anomaly may only have a small section that differs from the "normal" data, and that may be what the algorithms detect. In such case, treating the entire planted instance as anomalous would indeed lower the Score. This explanation is later validated by the results in Table 13, which show that a shorter sliding window length results in higher Score values in some cases. Nevertheless, this is why we also use the HitRate, which is independent of the ground truth length, as an alternate measure.

7.2.5 Effects of Ensemble Selectivity. Next, we evaluate the proposed method with ensemble selectivity τ varying from 5% to 100%. In this experiment, the evaluation of the average Score value (which is computed over 25 time series) is repeated 20 times for each dataset and τ value. We then compute the mean and standard deviation over the set of 20 average Score values. The results are shown in Table 12. We observe that better performance is typically achieved for smaller τ values (e.g., from 5% for ECGFiveDays and Trace datasets to 20% for GunPoint and StarLightCurve datasets), while the standard deviation is typically smaller for τ values of 20% or 40%. Therefore, we recommend choosing τ around 20% since it combines a relatively high level of performance with a relatively small variance of results.

7.2.6 Effects of Sliding Window Length. In this experiment, we investigate how the proposed method performs when the sliding window length n is less than the ground truth anomaly

Table 12: Mean and standard deviation over 20 average Score values, vs. τ

Dataset	τ = 5%	τ = 10%	τ = 20%	τ = 40%	τ = 80%	τ = 100%
TwoLeadECG	0.4149 (0.04)	0.4196 (0.032)	0.4 (0.026)	0.3882 (0.027)	0.3354 (0.036)	0.3071 (0.032)
ECGFiveDays	0.425 (0.042)	0.41 (0.045)	0.38 (0.038)	0.37 (0.037)	0.35 (0.024)	0.32 (0.036)
GunPoint	0.488 (0.042)	0.50 (0.037)	0.505 (0.035)	0.488 (0.025)	0.43 (0.023)	0.412 (0.023)
Wafer	0.339 (0.05)	0.371 (0.042)	0.337 (0.027)	0.311 (0.027)	0.27 (0.032)	0.26 (0.037)
Trace	0.6136 (0.037)	0.6017 (0.035)	0.5972 (0.025)	0.5864 (0.024)	0.4997 (0.046)	0.4166 (0.042)
StarLightCurve	0.9057 (0.017)	0.9183 (0.016)	0.9327 (0.009)	0.9052 (0.012)	0.7359 (0.021)	0.628 (0.021)

Table 13: Performance (average Score) vs. n

Dataset	$n = 0.6n_a$	$n = 0.7n_a$	$n = 0.8n_a$	$n = 0.9n_a$	$n = n_a$
TwoLeadECG	0.4620	0.4605	0.4107	0.4259	0.3951
ECGFiveDays	0.4391	0.3691	0.3535	0.3797	0.3903
GunPoint	0.4373	0.4992	0.4680	0.4371	0.4728
Wafer	0.3095	0.4195	0.3389	0.2824	0.3179
Trace	0.5229	0.5911	0.5689	0.5852	0.5718
StarLightCurve	0.8624	0.8998	0.9216	0.9048	0.9369

Table 14: Performance (HitRate) vs. n

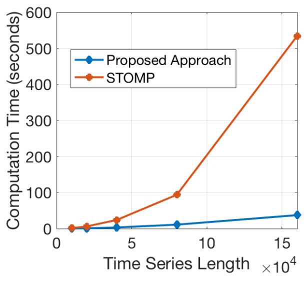
Dataset	$n = 0.6n_a$	$n = 0.7n_a$	$n = 0.8n_a$	$n = 0.9n_a$	$n = n_a$
TwoLeadECG	0.72	0.84	0.8	0.76	0.72
ECGFiveDays	0.96	0.8	0.84	0.72	0.8
GunPoint	0.84	0.68	0.72	0.64	0.68
Wafer	0.56	0.64	0.52	0.52	0.72
Trace	1.0	1.0	1.0	1.0	0.96
StarLightCurve	1.0	1.0	1.0	1.0	1.0

length (denoted as n_a). Specifically, we test five different sliding window lengths equal to 60%, 70%, 80%, 90% and 100% of n_a. The average Score and HitRate values are shown in Tables 13 and 14, respectively. According to the results, while there exists some variation in the performance, the dependence on n is not significant, and the proposed method robustly outperforms the existing grammar-induction-based approaches in most cases.

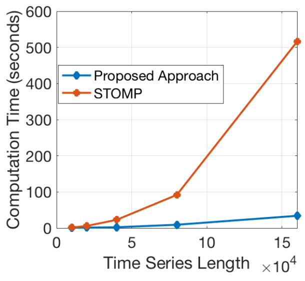
7.3 Scalability

We evaluate the scalability of the proposed ensemble-based approach by applying it to a 160,000 length random walk (RW) time series, ECG data [18] and electroencephalogram (EEG) data [13]. We compare the computation times for the proposed method and for the state-of-the-art discord discovery approach. In the experiment, we use STOMP [23], the latest Matrix-Profile-based algorithm, to detect discords. It has been shown that STOMP is both faster and more robust to different types of data compared to the original discord discovery method HOTSAX [9].

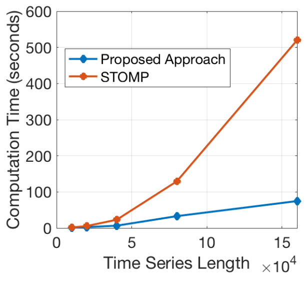
The scalability, evaluated as the computation time versus the time series length, is shown in Figure 8. We see that, as the time series length increases, the computation time grows significantly slower for the proposed method than for STOMP. At the largest time series length, the proposed approach is about one order of magnitude faster than STOMP, for all three types of time series data. We also find that the computation times for both approaches are roughly independent of the sliding window length.



(a) RW time series



(b) ECG time series



(c) EEG time series

Figure 8: Scalability: Computation time vs. time series length.

7.4 Case Study: Anomaly Detection in Electric Power Usage Time Series

Interpreting the time dependence of electric power usage has many potential applications [21]. In this section, we show that the ensemble grammar induction method can find anomalies in large-scale electric power usage time series. We use 100 days of the fridge-freezer power usage data provided in [14], to evaluate the performance of the proposed method. The entire time series, which consists of approximately 600,000 points, is shown in Figure 9(a), and the first 20,000 points are shown in Figure 9(b). We run the proposed method with the sliding window length of 900, which is about the duration of one cycle (shown in red box in Figure 9(b)). The computation time is about one minute. Figures 9(c) and 9(d) show the two top-ranked anomaly candidates detected by the proposed method.

We see that the top-1 anomaly represents a cycle whose shape is unusual compared to the typical cycles shown in Figure 9(b). The top-2 anomaly represents an unusual event that contains normal cycles and short spikes. The two anomalies represent different, unusual power usage patterns. Since this time series is very long, and the anomalies have different lengths, using the state-of-the-art discord discovery approach would be time consuming. In contrast, the proposed method provides an efficient way to detect anomalies.

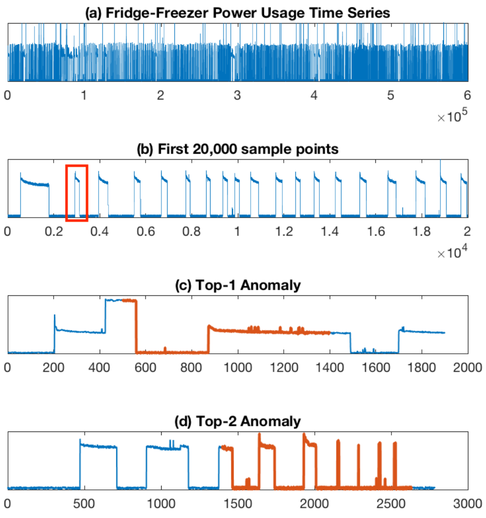


Figure 9: A 600,000 length fridge-freezer power usage time series, and two top-ranked anomalies detected in this series by ensemble grammar induction.

7.5 Detecting Multiple Anomalies

We also investigate the effectiveness of the proposed approach in detecting multiple anomalies in time series. In this experiment, we use the StarlightCurve dataset to generate 10 time series. Each of these time series is of length 43008 and contains two randomly selected and placed anomalies of length 1024. We evaluate the performance by the number of ground truth anomalies detected (a ground truth anomaly is considered detected if it overlaps with at least one of the top-3 ranked anomaly candidates). The proposed method performed well as it successfully identified both anomalies in nine time series and one of the two anomalies in one time series.

8 CONCLUSION

In this paper, we introduce a robust grammar-induction-based anomaly detection approach utilizing ensemble learning. Instead of using a particular combination of parameter values for anomaly detection, the proposed method generates the final result based on a set of results obtained using different parameter values. The experiments performed on datasets with known ground truth show that the proposed ensemble approach can outperform existing grammar-induction-based approaches with different criteria for selection of parameter values. We also show that the proposed approach, which has a linear time complexity with respect to the data size, can achieve performance similar to that of the state-of-the-art distance-based anomaly detection approach that has a quadratic time complexity.

ACKNOWLEDGMENTS

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. 2000. LOF: identifying density-based local outliers. In *ACM sigmod record*, Vol. 29. ACM, 93–104.

[2] 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.

[3] Hoang Anh Dau, Eamonn Keogh, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, Yanping, Bing Hu, Nurjahan Begum, Anthony Bagnall, Abdullah Mueen, Gustavo Batista, and Hexagon-ML. 2018. The UCR Time Series Classification Archive. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/.

[4] Len Feremans, Vincent Vercruyssen, Boris Cule, Wannes Meert, and Bart Goethals. 2019. Pattern-based anomaly detection in mixed-type time series. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases.*

[5] Yifeng Gao, Qingzhe Li, Xiaosheng Li, Jessica Lin, and Huzefa Rangwala. 2017. TrajViz: A Tool for Visualizing Patterns and Anomalies in Trajectory. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases.* Springer, 428–431.

[6] Yifeng Gao and Jessica Lin. 2018. Exploring variable-length time series motifs in one hundred million length scale. *Data Mining and Knowledge Discovery* 32, 5 (2018), 1200–1228.

[7] Yifeng Gao, Jessica Lin, and Huzefa Rangwala. 2016. Iterative Grammar-Based Framework for Discovering Variable-Length Time Series Motifs. In *Machine Learning and Applications (ICMLA), 2016 15th IEEE International Conference on.* IEEE, 7–12.

[8] C Sweetlin Hemalatha, Vijay Vaidehi, and R Lakshmi. 2015. Minimal infrequent pattern based approach for mining outliers in data streams. *Expert Systems with Applications* 42, 4 (2015), 1998–2012.

[9] Eamonn Keogh, Jessica Lin, and Ada Fu. 2005. Hot sax: Efficiently finding the most unusual time series subsequence. In *Fifth IEEE International Conference on Data Mining (ICDM'05).* Ieee, 8–pp.

[10] Yuan Li, Jessica Lin, and Tim Oates. 2012. Visualizing Variable-Length Time Series Motifs.. In *Proceedings of the 2012 SIAM International Conference on Data Mining.* SIAM, 895–906.

[11] Jessica Lin, Eamonn Keogh, Stefano Lonardi, Jeffrey P Lankford, and Donna M Nystrom. 2004. Visually mining and monitoring massive time series. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 460–469.

[12] Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi. 2007. Experiencing SAX: a novel symbolic representation of time series. *Data Mining and knowledge discovery* 15, 2 (2007), 107–144.

[13] Abdullah Mueen. 2013. Enumeration of time series motifs of all lengths. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on.* IEEE, 547–556.

[14] David Murray, Jing Liao, Lina Stankovic, Vladimir Stankovic, Richard Hauxwell-Baldwin, Charlie Wilson, Michael Coleman, Tom Kane, and Steven Firth. 2015. A data management platform for personalised real-time energy feedback. In *Procededings of the 8th International Conference on Energy Efficiency in Domestic Appliances and Lighting.* 1–15.

[15] Craig G. Nevill-Manning and Ian H. Witten. 1997. Identifying hierarchical strcture in sequences: A linear-time algorithm. *J. Artif. Intell. Res.(JAIR)* 7 (1997), 67–82.

[16] 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 *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 262–270.

[17] Shebuti Rayana and Leman Akoglu. 2015. Less is more: Building selective anomaly ensembles with application to event detection in temporal graphs. In *Proceedings of the 2015 SIAM International Conference on Data Mining.* SIAM, 622–630.

[18] Pavel Senin, Jessica Lin, Xing Wang, Tim Oates, Sunil Gandhi, Arnold P Boedihardjo, Crystal Chen, and Susan Frankenstein. 2015. Time series anomaly discovery with grammar-based compression.. In *EDBT.* 481–492.

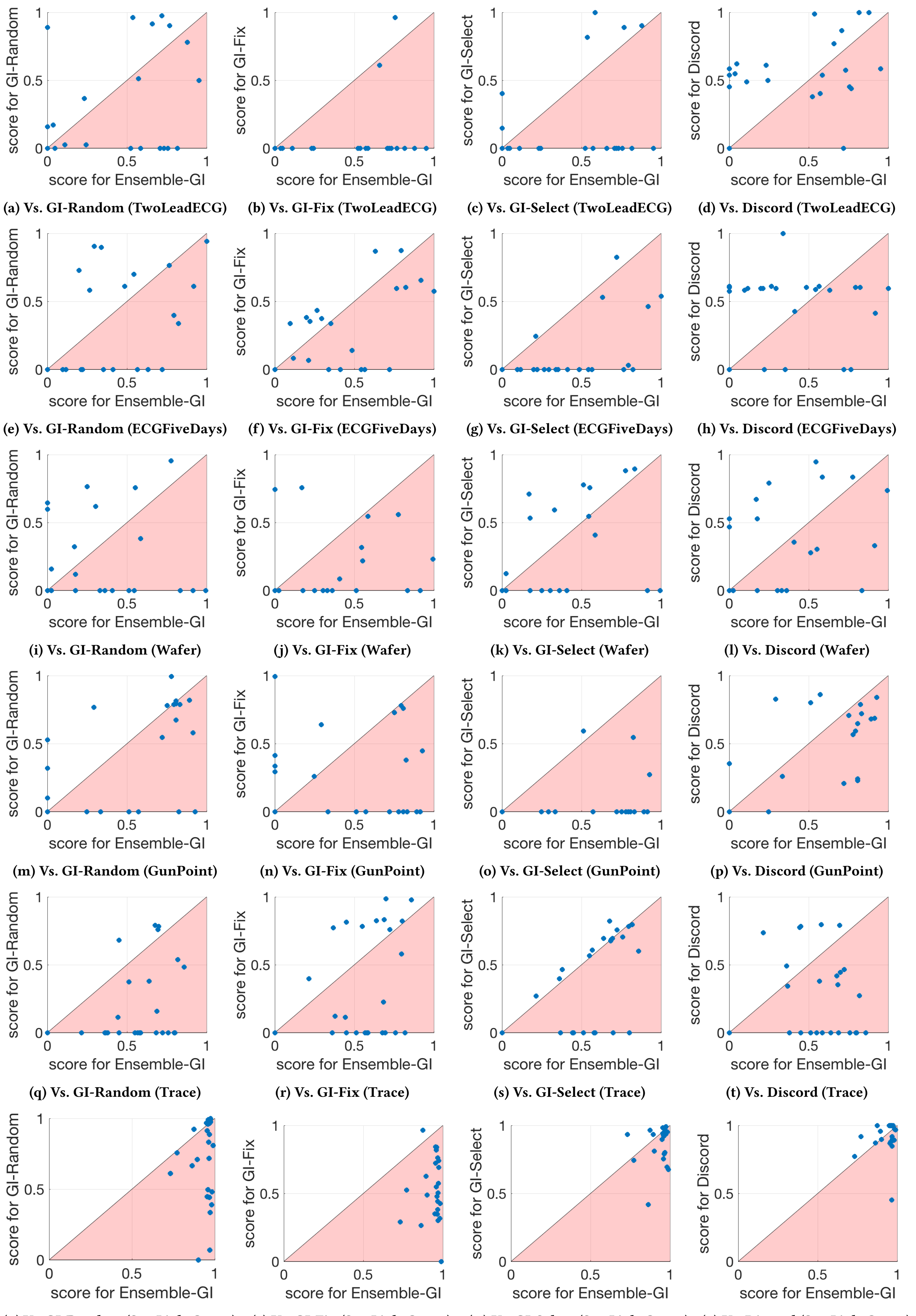
[19] Pavel Senin, Jessica Lin, Xing Wang, Tim Oates, Sunil Gandhi, Arnold P Boedihardjo, Crystal Chen, and Susan Frankenstein. 2018. Grammarviz 3.0: Interactive discovery of variable-length time series patterns. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 12, 1 (2018), 10.

[20] Pavel Senin, Jessica Lin, Xing Wang, Tim Oates, Sunil Gandhi, Arnold P Boedihardjo, Crystal Chen, Susan Frankenstein, and Manfred Lerner. 2014. GrammarViz 2.0: a tool for grammar-based pattern discovery in time series. In *Machine Learning and Knowledge Discovery in Databases.* Springer, 468–472.

[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 *Data Mining (ICDM), 2016 IEEE 16th International Conference on.* 1317–1322.

[22] Yue Zhao, Zain Nasrullah, Maciej K Hryniewicki, and Zheng Li. 2019. LSCP: Locally selective combination in parallel outlier ensembles. In *Proceedings of the 2019 SIAM International Conference on Data Mining.* SIAM, 585–593.

[23] Yan Zhu, Zachary Schall-Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth Funning, Abdullah Mueen, Philip Brisk, and Eamonn J. Keogh. 2016. Matrix Profile II: Exploiting a Novel Algorithm and GPUs to Break the One Hundred Million Barrier for Time Series Motifs and Joins. *2016 IEEE 16th International Conference on Data Mining (ICDM)* (2016), 739–748.



(a) Vs. GI-Random (TwoLeadECG) (b) Vs. GI-Fix (TwoLeadECG) (c) Vs. GI-Select (TwoLeadECG) (d) Vs. Discord (TwoLeadECG)

(e) Vs. GI-Random (ECGFiveDays) (f) Vs. GI-Fix (ECGFiveDays) (g) Vs. GI-Select (ECGFiveDays) (h) Vs. Discord (ECGFiveDays)

(i) Vs. GI-Random (Wafer) (j) Vs. GI-Fix (Wafer) (k) Vs. GI-Select (Wafer) (l) Vs. Discord (Wafer)

(m) Vs. GI-Random (GunPoint) (n) Vs. GI-Fix (GunPoint) (o) Vs. GI-Select (GunPoint) (p) Vs. Discord (GunPoint)

(q) Vs. GI-Random (Trace) (r) Vs. GI-Fix (Trace) (s) Vs. GI-Select (Trace) (t) Vs. Discord (Trace)

(u) Vs. GI-Random (StarLightCurve) (v) Vs. GI-Fix (StarLightCurve) (w) Vs. GI-Select (StarLightCurve) (x) Vs. Discord (StarLightCurve)

Figure 10: Summary of performance comparison of ensemble grammar induction against baseline methods.

Incremental Based Top-k Similarity Search Framework for Interactive-Data-Analysis Sessions

Oded Elbaz, Tova Milo, and Amit Somech

Tel Aviv University, Israel

ABSTRACT

Interactive Data Analysis (IDA) is a core knowledge-discovery process, in which data scientists explore datasets by issuing a sequence of data analysis actions (e.g. filter, aggregation, visualization), referred to as a session. Since IDA is a challenging task, special recommendation systems were devised in previous work, aimed to assist users in choosing the next analysis action to perform at each point in the session. Such systems often record previous IDA sessions and utilize them to generate next-action recommendations. To do so, a compound, dedicated session-similarity measure is employed to find the top-k sessions most similar to the session of the current user. Clearly, the efficiency of the top-k similarity search is critical to retain interactive response times. However, optimizing this search is challenging due to the non-metric nature of the session similarity measure.

To address this problem we exploit a key property of IDA, which is that the user session progresses *incrementally*, with the top-k similarity search performed, by the recommender system, *at each step*. We devise efficient top-k algorithms that harness the incremental nature of the problem to speed up the similarity search, employing a novel, effective *filter-and-refine* method. Our experiments demonstrate the efficiency of our solution, obtaining a running-time speedup of over 180X compared to a sequential similarity search.

1 INTRODUCTION

Interactive Data Analysis (IDA) is an important procedure in any process of data-driven discovery. It is ubiquitously performed by data scientists and analysts who interact with their data "hands-on" by iteratively applying analysis actions (e.g. filtering, aggregations, visualizations) and manually examining the results. This is primarily done to understand the nature of the data and extract knowledge from it, yet is also fundamental for particular data scientific tasks such as data wrangling and cleaning, feature selection and engineering, and explaining decision models.

However, since IDA is long known as a complex and difficult task, extensive research has been devoted to the development of *recommendation systems* that assist users in choosing an appropriate next-action to perform at each point in an IDA session. Many of these IDA recommendation systems [2, 15, 16, 26, 43] rely on a *similarity comparison* between users' sequences of analysis actions (denoted *sessions*). They follow the assumption that if two session *prefixes* are similar, their *continuation* is likely to also be similar. Hence, they utilize a repository of prior analysis sessions (of the same or other users): Given an *ongoing* user's session, such systems first retrieve the top-k most similar session prefixes from the repository, then examine their continuation and use the gathered information to form a possible next-action recommendation [2, 15, 16, 43]. Since these recommender systems are *interactive*, the efficiency of the top-k session similarity search is critical. However, most previous work focuses on the *quality* and *applicability* of the produced IDA recommendations, rather than on their *scalability* and running time *performance*.

Our goal in this paper is thus to devise efficient, scalable algorithms for this particular top-k similarity search problem, a significant computational bottleneck in many IDA recommender systems. Specifically, we focus our attention on a dedicated similarity measure for analysis sessions [3], which we denote by *SW-SIM*. In a comprehensive user study [3], *SW-SIM* was compared, quality wise, against several alternative similarity measures, and was found to be the most suitable for the context of data analysis sessions. Consequently, it has been adopted by multiple IDA recommender systems and applications, e.g., [2, 4, 30, 41, 44]. *SW-SIM* is an extension of the well-known Smith-Waterman algorithm [35] for local sequence alignment. Intuitively, given a similarity metric for individual analysis-actions (e.g. filter, aggregation, visualization), *SW-SIM* compares two sessions s, s' by *aligning* them, i.e. matching similar action pairs using an *alignment matrix*. The measure allows for (yet penalizes) gaps in the alignment, and gives a higher weight to the more recent actions in the session, since they are expected to have higher relevance on the current user's intent.

Given a current user session and a sessions repository, a naive top-k search may be done by sequentially iterating over all sessions in the repository, computing the *SW-SIM* similarity score of the current session w.r.t. each of their prefixes, then selecting the top-k prefixes with the highest score. This simple algorithm, however, is prohibitively time consuming: **Our experiments show that even when employed on a medium size repository of 10K sessions, the naive sequential search takes more than 17 seconds to complete**. This is excessively high for a real-time response.

To optimize it, two key challenges must be addressed:

1. A single similarity comparison of two sessions is expensive to begin with. Computing *SW-SIM* requires to construct an alignment matrix, which results in a high computation time (quadratic in the mean session length).

2. Employing existing optimizations is highly non-trivial. Since *SW-SIM* is a non-metric similarity measure, existing optimizations for similarity search e.g. [8, 33, 37] are inadequate. Furthermore, because analysis sessions are compound sequences of complex analysis actions, they do not have a numeric vector representation, which is a requirement by other top-k optimization works [33].(See Section 2 for an elaborated discussion.)

A key observation underlying our work is that the user session progresses *incrementally* with the top-k similarity search performed by the IDA recommender system *at each step*. We exploit this property to address the two challenges mentioned above: Our first, rather direct optimization, speeds up the processing of a *single* similarity comparison between two sessions by utilizing previous-step computations. Our second, more sophisticated optimization, tackles the *top-k search* by employing a novel filter-and-refine technique. It employs lower and upper bounds stemming from the incremental growth of the sessions accompanied by a dedicated index structure. Our experiments demonstrate that **using our optimizations, a speedup of more than 180X is obtained compared to a non-optimized sequential search**. Importantly, this is done while retaining a perfect accuracy of results, since our algorithms are exact.

We next explicitly state our assumptions and focus, then summarize our contributions and paper organization.

Paper Focus and Assumptions.
1. Although the particular IDA settings may vary, many IDA recommender systems rely on a top-k prefixes search. IDA settings may vary between systems, in terms of, e.g., the type of allowed analysis actions, structure of the data, user expertise, etc. However, the incremental, session-like nature of the process characterizes most IDA settings. Our paper is thus aimed at solving a computational bottleneck in a growing number of IDA-dedicated recommendation systems [2, 15, 16, 26, 43] that rely on finding the top-k most similar session prefixes in order to generate recommendations. Our solution is generic, suitable for a wide range of IDA platforms, regardless of their particular settings.
2. *SW-SIM* is currently the most-suitable measure for IDA sessions similarity. A multitude of definitions exist for measuring the similarity between two arbitrary sequences, such as *Dynamic Time Warping* for time series and *Smith-Waterman* for DNA sequences. In the context of data analysis sessions, *SW-SIM* is considered the most comprehensive and suitable (See [3] for a comparative study).
3. The focus of this paper is on performance and scalability, rather than on the quality of the produced recommendations. Different IDA recommender systems may use the selected top-k sessions in different ways in order to generate recommendations, possibly using additional information such as the data properties, user profiles, etc. But in all of them - response time optimization of the top-k search is clearly a critical issue. We therefore focus on performance and scalability rather than discussing the quality of the recommendations produced. For the latter, we refer the reader to works e.g. [2, 15, 16, 26, 43] in which the focus is on the quality of recommendations produced.

Technical Contributions & Paper Organization.
Section 3: We provide a simple, generic model for representing IDA sessions, and describe the alignment-based session similarity notion. Based on this model, we develop a formal definition for the incremental top-k similarity search problem.
Section 4: We describe an optimization for the incremental construction of a session alignment matrix and means to derive prefixes similarity scores from it.
Section 5: We present a novel threshold-based algorithm for the incremental top-k search, which utilizes effective similarity lower and upper bounds, also stemming from the incremental nature of the search problem.
Section 6: We demonstrate the efficiency of our solution via extensive experiments on artificial and real-life session repositories.

We begin by reviewing related work (Section 2), and finally present our conclusions and limitations in Section 7.

2 RELATED WORK

There is vast literature on sequences similarity, alignment, and top-k search. We survey works in numerous application domains, explaining why existing optimizations are inadequate for the context of IDA.
Bioinformatics-based optimizations for sequence alignment. Local alignment techniques are known to be extremely useful in the bioinformatics domain for tasks such as DNA and protein sequencing [22, 25, 39]. In this application domain, alignment is often performed between a query sequence and an extremely long *reference* string (e.g. a DNA genome), often from a fixed, small size alphabet (comprises, e.g., six nucleotides). Consequently, biology-driven optimizations such as [14, 18, 28, 40] exploit this particular property (small alphabet size) in order to index the large reference string. These solutions typically use a prefix/suffix tree or hash tables mapping small, common substrings to their location in the reference string. In contrast, in our context the “alphabet" (namely the analysis-actions space) is unbounded and, as explained in the next section, there is a predefined notion of distance between letters (analysis actions in our context). Such solutions, therefore, cannot be directly employed in our case.
Edit Distance and Dynamic Time Warping (DTW). Edit distance and DTW are popular alignment-based measures that are used in numerous application domains such as textual auto-correction, speech recognition and the comparison of time series. However, as noted in [3], these measures lack important properties for the context of IDA sessions. Classical edit-distance, which originated from the comparison of textual strings, assumes that letters are either *identical* or *mismatched*, which is inadequate for the context of IDA. This is because the alphabet, containing individual actions, is much larger and more complex. Also, the compound similarity of individual “letters” (actions) should be taken into consideration. DTW, on the other hand, does support the employment of a designated distance metric for the sequences' objects, yet is not compatible with the notion of *gaps* [32]. Also, both measures do not support the important time-discount feature [3], which allows giving a higher weight to the more recent actions in the session. Optimization works for these measures are typically based on specific properties of the measures or application domain [19, 38].
General solutions for similarity search in a non-metric space. *SW-SIM* is a non-metric similarity measure, thus many optimizations that rely on the *triangle inequality* property (e.g.,[8, 27]) cannot be used here. Other solutions for non-metric spaces, e.g. [6, 7, 19, 34], assume that the sequences' objects are numeric vectors (e.g. as in Minkowski distance, Cosine distance), as opposed to sequences of abstract objects (analysis actions in our case). But even ignoring this, employing any such solutions in our context requires indexing each session prefix, which may be prohibitively large (typically by an order of magnitude compared to the number of sessions). Existing solutions for optimized top-k search supporting arbitrary, non-metric similarity measure are Constant Shift Embedding (CSE) [31] and NM-Tree [34]. We examine these solutions in our experimental evaluation (Section 6.2) and show that our algorithm is consistently superior, obtaining a performance improvement of at least two orders of magnitude.
Incremental keyword query autocompletion. Incremental computation is used for optimization in a variety of database applications, e.g., interactive association-rules and sequence mining [29], but most notably for incremental text autocompletion [10, 17, 24]. the latter solutions efficiently perform a *top-k keyword*

similarity search at each key stroke of the user (using string edit distance). However, such solutions also exploit the rather small alphabet size and use a trie-index, a dedicated prefix tree for strings. As explained above, in our case the "alphabet" (i.e., the analysis-actions space) is complex and unbounded. Last, [21] and [20] suggest algorithms for incrementally computing the *string edit distance* of Ax and B (where x is an additional character) in $O(|A| + |B|)$ by using the distance matrix computed for A and B. We employ similar principles for our single-pair incremental alignment computation (Section 4), yet, importantly, augment them with novel techniques for efficient top-k similarity search.
Similarity search of scientific/business workflows. Similarity search has also been considered in the context of workflows [12, 23, 36] in two main contexts - searching for workflow *specifications*, and searching for workflow *traces*. They employ, however, a different notions of similarity focusing mainly on the structure/nesting of the workflow-specification components (different from the the unstructured, free-form nature of IDA sessions). Also, to our knowledge, these works do not address the *incremental* nature of the top-k similarity search problem, that is more typical for IDA sessions than for workflows.

3 PRELIMINARIES

We start by providing basic definitions for IDA, then define the incremental top-k similarity search problem.

Interactive Analysis Sessions. In the process of IDA, users investigate datasets via an interactive user interface which allows them to formulate and issue analysis actions (e.g., filter, grouping, aggregation, visualization, mining) and to examine their results. Formally, we assume an infinite domain[1] of analysis actions Q and model an analysis session as a sequence of actions $s = \langle q_1, q_2, \ldots, q_n \rangle | q_i \in Q$. We use $s[i]$ to denote the i'th action in s (q_i) and s_i to denote the session's *prefix* up to q_i, i.e. $s_i = \langle q_1, q_2, \ldots, q_i \rangle$. Therefore, $s = s_n$, and s_0 is the empty session. An ongoing user session, denoted u, is built incrementally. At time t the user issues an action q_t, then analyzes its results and decides whether to issue a next action q_{t+1} or to terminate the session. The session (prefix) u_t at time t thus consists of the actions $\langle q_1, \ldots, q_t \rangle$, where the following actions in the session, i.e., $q_{t+1}, ..q_n$ are not yet known.

In this work we consider the case of comparing the similarity of the user session u to other sessions (and parts thereof) in a given repository, denoted S, containing prior analysis sessions performed by the same or other users. To formally define sessions similarity, let us first consider the similarity of individual analysis actions, then generalize to sessions.

Individual Actions Similarity. Given a *distance metric* for individual analysis actions $\Delta : Q \times Q \rightarrow [0, 1]$ over the actions domain, the *action similarity function* $\sigma(q_1, q_2)$ is the complement function defined by $\sigma(q_1, q_2) = 1 - \Delta(q_1, q_2)$ for all q_1, q_2 in Q. Several distance/similarity measures for many kinds of analysis actions, e.g. for SQL and OLAP queries visualizations, and web-based analysis actions have been proposed in the literature (e.g. [3, 16, 26]) and our framework can employ any of them as long as the corresponding measure defines a metric space.

Session Similarity. As mentioned in Section 2, there are several ways to lift the similarity of individual elements into similarity of sequences [11]. To assess which measure is the most suitable

[1]The domain is practically infinite since some action types, e.g. "filter", may have an unbounded number of possible parameter assignments.

		a	b	c	A	B
	0	0	0	0	0	0
A	0	0.24	0.19	0.13	0.66	0.58
B	0	0.19	0.53	0.47	0.58	1.47
A	0	0.30	0.47	0.53	1.28	1.38
B	0	0.23	0.66	0.58	1.19	2.28

(a) Session ϕ

		A	B	a	b	c
	0	0	0	0	0	0
A	0	0.48	0.43	0.37	0.30	0.23
B	0	0.43	1.07	1.00	0.93	0.85
A	0	0.59	1.00	1.43	1.35	1.26
B	0	0.52	1.32	1.35	1.88	1.78

(b) Session ψ

Figure 1: Alignment Matrices for u_4 = "$ABAB$"

for comparing analysis sessions, the authors of [3] formulated desiderata for an ideal session similarity measure, based on an in-depth user study:

- It should take the actions' *order of execution* into consideration, i.e. two sessions are similar if they contain a *similar* set of actions, performed in a *similar order*.
- "Gaps" (i.e subsequences of non-matching actions) should be allowed yet penalized.
- Long matching subsequences should be better rewarded than shorter matching subsequences.
- Recent actions in the sessions are more relevant than old ones.

They conclude that the only measure respecting all of the above mentioned desiderata is the *Smith-Waterman similarity measure* [35], a popular measure for local sequence alignment, and propose an extension suitable for the context of analysis actions, denoted *SW-SIM* in our work. We next describe the measure.

The similarity score of two sessions is defined recursively, on increasingly growing prefixes, with the base of the recursion being the empty prefix. At each point, the similarity of the pair of actions at the end of the prefixes is considered, and the best option, score-wise, is chosen. The two actions may either be matched, in which case an award proportional to their similarity is added to the accumulated score for their preceding prefixes, or, alternatively, one of the actions is skipped over. In this case a linear *gap penalty* $0 \le \delta \le 1$ is deducted from the accumulated score. To reflect the fact that the matching/skipping of older actions is less important than that of recent ones, rewards/penalties are multiplied by a decay factor $0 \le \beta \le 1$ with an exponent reflecting how distant the actions are from the sessions' end. More formally, the sessions similarity is defined using an *alignment matrix.*

Definition 3.1 (Alignment Matrix, Similarity Score). Given sessions s, s' of lengths n, m resp., their alignment matrix $A_{s,s'} \in \mathbb{R}^{(n+1)\times(m+1)}$ is recursively defined as follows. For $0 \le i \le n$, $0 \le j \le m$:

$$A_{s,s'}[i,j] = \begin{cases} 0, \text{ if } i = 0 \lor j = 0, \text{ else:} \\ max \begin{cases} A_{s,s'}[i-1, j-1] + \sigma(s[i], s'[j])\beta^{(n-i)+(m-j)} \\ A_{s,s'}[i, j-1] - \delta\beta^{(n-i)+(m-j)} \\ A_{s,s'}[i-1, j] - \delta\beta^{(n-i)+(m-j)} \\ 0 \end{cases} \end{cases}$$

The similarity score between s and s' is defined as:

$$Sim(s, s') := A_{s,s'}[n, m]$$

Note that even though the distance between *individual actions* forms a metric space, as noted in the introduction, *SW-SIM* does not induce a metric on IDA sessions.

The optimal values for the decay factor β and the gap penalty δ are typically chosen by the system administrator using an extrinsic evaluation, as explained in Section 6.1.

The following example illustrates the definition.

Example 3.2. For simplicity, assume that the action space is represented by the English letters, both uppercase and lowercase, and assume the following action similarity for any two distinct uppercase letters X,Y and their corresponding lowercase versions x,y.

$$\sigma(\cdot,\cdot) := \begin{cases} \sigma(X,X) = 1 \\ \sigma(x,x) = 1 \\ \sigma(x,X) = \sigma(X,x) = 0.5 \\ \sigma(x,y) = \sigma(X,Y) = 0.1 \\ \sigma(X,y) = \sigma(y,X) = 0 \end{cases}$$

Identical letters are maximally similar, and two instances of the same letter, but in different case (upper/lower), are 0.5 similar. Different letters but of the same case are 0.1 similar. Let u_4 ="ABAB" be the current user session, and consider two repository sessions ϕ ="abcAB" and ψ ="ABabc". Intuitively, according to the desiderata above, we expect ϕ to be more similar to u_4 than ψ, as their most recent suffix ("AB") is identical. This is reflected also in the alignment matrices of ϕ and ψ, depicted in Figure 1a and Figure 1b (resp.), when setting the gap penalty $\delta = 0.1$ and decay factor $\beta = 0.9$. The bottom-right cell in each matrix reflects the alignment score of the two sessions. The highlighted cells describes the alignment "trace", namely the cells chosen (among the three options in alignment formula) when advancing to the next step in the final score computation. Specifically, when the highlighted trace moves vertically/horizontally we have a gap, and when it moves diagonally the corresponding actions are matched. As we can see, the similarity score $Sim(u_4, \phi)$ is higher than $Sim(u_4, \psi)$.

Our definition of *SW-SIM* follows that of [3, 35] with a minor adaptation to our context. First, we define the similarity score as the *bottom-right* cell of the alignment matrix, as opposed to the *maximal* cell value in [3, 35]. This is because we focus on measuring the similarity of the current session to all *session prefixes* in the repository[2]. Second, we apply a decay factor to both actions matches and gaps, as opposed to only matches in [3], since their significance decreases as the session advances. Also, note that [3] suggests dynamically setting the decay factor and gap penalty, which are both constants in our case. As we shall see, the use of constant parameter values facilitates effective optimization via computation factorization.

Problem Definition. To assist the user in choosing an appropriate next-action, IDA recommender systems search the repository S to identify session prefixes that are most similar to u_t - the current, ongoing user session at time t. The continuation of the retrieved sessions is then processed by the recommender systems and used to derive a next-action recommendation for u_t.[3]

Let $\mathcal{P}refix(S)$ be the set consisting of all session prefixes in repository S, i.e. $\mathcal{P}refix(S) = \{s_i \mid s \in S, 1 \le i \le |s|\}$. Since the user session $u = \langle q_1, q_2, ...\rangle$ is built incrementally, at each step $t = 1, \ldots, |u|$ we are given u_t and wish to identify its top-k most similar session prefixes in the repository.

[2]This is done by IDA recommendation systems in order to process the continuation of matching session-prefixes to next-action recommendations.
[3]Other information is often used as well, e.g. the *interestingness* or *frequency* of the actions to be recommended, the user profile, etc.

u \ φ		a	ab	abc	abcA	abcAB
ABAB	0	0.35	0.90	0.71	1.32	2.28
ABABc	0	0.22	0.71	1.23	1.09	1.95

(a) u, ϕ

u \ ψ		A	AB	ABa	ABab	ABabc
ABAB	0	0.80	1.81	1.67	2.09	1.78
ABABc	0	0.62	1.53	1.47	1.78	2.19

(b) u, ψ

Figure 2: Similarity vectors for u_4 and u_5

Definition 3.3 (Incremental Top-k Similarity Search). Given a user session u_t at time t and a sessions repository S, the set $top_k(u_t, S) \subseteq \mathcal{P}refix(S)$ consists of k session prefixes s.t. $\forall s \in top_k(u_t, S), \forall s' \in \mathcal{P}refix(S) \backslash top_k(u_t, S) : Sim(u_t, s) \ge Sim(u_t, s')$. The incremental search problem is to compute, at each $t = 1, \ldots, |u|$, the set $top_k(u_t, S)$.

For simplicity, we assume that in the course of a user session u, the repository S is unchanged. In Section 5.5 we discuss the minor changes required in our framework to support the case of a dynamic repository where sessions are incremented or added.

Last, note that while session prefixes may be long and contain "historical" actions of less importance, *SW-SIM* (as explained above) favors *recent*, later actions over old ones. Also, it is easy to show that in *SW-SIM*, the similarity of a prefix $s_i = \langle q_1, q_2, ...q_i\rangle$ is always higher (or equal) than the similarity of any shorter *sub-session* of s ending in q_i.

4 INCREMENTAL SIMILARITY COMPUTATION

We first optimize the similarity calculations between the current user session u and all prefixes of a *single* repository session s. Rather than computing the alignment matrix from scratch for each prefix of s, we show that: (1) it is sufficient to compute one alignment matrix between two sessions, then use it to simultaneously derive *all prefixes similarity scores*, and (2) the alignment matrix $A_{u_t,s}$ at time t can be efficiently constructed by reusing the previous matrix $A_{u_{t-1},s}$ computed at time $t-1$.

Given a current user session u_t and a repository session s of size $|s|$, we define a *similarity vector*, $\vec{Sim}(u_t, s)$ containing the similarity score of u_t and each prefix of s, i.e.,

$$\vec{Sim}(u_t, s) = [Sim(u_t, s_0), Sim(u_t, s_1), \ldots, Sim(u_t, s_{|s|})]$$

We use $\vec{Sim}(u_t, s)[j]$, where $j = 1 \ldots |S|$, to denote the j element in the vector. The similarity vector $\vec{Sim}(u_t, s)$ can be derived from the alignment matrix $A_{u_t,s}$ using the following observation:

Observation 4.1. *Given an alignment matrix $A_{u_t,s}$*

$$\forall 0 \le j \le |s| : \vec{Sim}(u_t, s)[j] = \frac{A_{u_t,s}[t,j]}{\beta^{|s|-j}}$$

Namely, the similarity score of u_t and prefix s_j is derived from their corresponding element in the alignment matrix, $A_{u_t,s}[t,j]$, by readjusting the decay factor β. The proof is derived from the more general observation that for arbitrary prefixes s_i and s'_j of session s and s':

$$A_{s_i,s'_j}[i,j] = \frac{A_{s,s'}[i,j]}{\beta^{(|s|-i)+(|s'|-j)}} \qquad (1)$$

Example 4.2. To continue with our running example, the similarity vectors $\vec{Sim}(u_4, \phi)$ and $\vec{Sim}(u_4, \psi)$ are given in the first row of the tables in Figure 2a, and Figure 2b, resp. According to Observation 4.1, an element in the similarity vector, e.g. $\vec{Sim}(u_4, \phi)[3]$, may be computed directly from the corresponding cell in the alignment matrix of u_4 and ϕ: $\vec{Sim}(u_4, \phi)[3] = \frac{A_{u_4,\phi}[4,3]}{\beta^2} = \frac{0.58}{0.81} = 0.71$.

Further exploiting the incremental nature of the problem, we next show how to efficiently construct a similarity vector $\vec{Sim}(u_t, s)$ at time t, from that computed at time $t-1$, thereby avoiding explicitly building the entire alignment matrix. To do so, we generalize the dynamic programming construction of the alignment matrix (Definition 3.1). Vector entries at time t are computed by reusing entries in the previous similarity vector $\vec{Sim}(u_{t-1}, s)$, computed at time $t-1$ (see the colored parts of the formula in Proposition 4.3).

Proposition 4.3. *For every repository session s, user session u and time t > 1,*

$$\vec{Sim}(u_t, s)[j] = \begin{cases} 0, \text{ if } j = 0, \text{ else:} \\ max \begin{cases} \vec{Sim}(u_{t-1}, s)[j-1]\,\beta^2 + \sigma(u_t[t], s[j]) \\ \vec{Sim}(u_t, s)[j-1]\beta - \delta \\ \vec{Sim}(u_{t-1}, s)[j]\,\beta - \delta \\ 0 \end{cases} \end{cases}$$

The proof is obtained by employing Equation 1 on each case of the conditional definition.

Example 4.4. Continuing with our example, assume that the user now issues a new action "c", i.e. at $t = 5$, u_5 = "*ABABc*". The new similarity vectors $\vec{Sim}(u_5, \phi)$ and $\vec{Sim}(u_5, \psi)$, are depicted in the bottom row of Figures 2a and 2b, resp. As before, the similarity scores $Sim(u_5, \phi)$ and $Sim(u_5, \psi)$, appear in the right-most cell of the vectors. Using Proposition 4.3 we can derive each value in the new vectors from the previous corresponding similarity vectors at time $t = 4$. For example, the fourth element in $\vec{Sim}(u_5, \psi)$ is given by:

$$\vec{Sim}(u_5, \psi)[4] = max \begin{cases} \vec{Sim}(u_4, \psi)[3]\beta^2 + \sigma(\text{“c”}, \text{“b”}) \\ \vec{Sim}(u_5, \psi)[3]\beta - \delta \\ \vec{Sim}(u_4, \psi)[4]\beta - \delta \end{cases}$$

$$= max(1.45, 1.22, 1.78) = 1.78$$

Let us analyze the reduction in time complexity resulting from these two simple optimizations. Let $|\widehat{s}|$ be the average session size and λ the complexity of computing similarity for individual actions. The expected time complexity of computing the similarity vector $\vec{Sim}(u_t, s)$ by construcitng all ($|\widehat{s}|$) allignemnt matrices is $O(|\widehat{s}|^3\lambda)$. Employing Observation 4.1 allows us to compute the same vector by constructing only one alignment matrix, in $O(|\widehat{s}|^2\lambda)$. Further employing Proposition 4.3 reduces the expected time to $O(|\widehat{s}|\lambda)$ since only $|s|$ action similarity calculations are required, between $u_t[t]$ (the new action in u_t) and all actions of s.

5 INCREMENTAL TOP-K ALGORITHMS

We are now ready to address the incremental top-k problem - given an ongoing user session u, retrieve the set $top_k(u_t, S)$ of similar prefixes at each time $t = 1, \ldots, |u|$. We first present a simple algorithm, denoted I-TopK, that iterates over the repository S and computes the similarity vector for each session. We then show how a much faster variant, denoted T-TopK, is obtained by employing a novel, incremental-based *filter-and-refine* approach.

Algorithm 1 T-TopK(u_t, S, k)

```
    Input: $u_t$ - current user session,
           $S$ - session repository,
           $k$ - size of the top-k set.
    Output: $top_k(u_t, S)$ - a set of top-k most similar session
            prefixes to $u_t$.
 1: $top \leftarrow MaxHeap(k)$
 2: if t == 1 then
 3:     Use I-TopK to obtain $top_k(u_t, S)$
 4: else
 5:     Compute $inf^t$, the lower-bound similarity threshold
 6:     $C \leftarrow \{s | s \in S \wedge sup^t(s) \geq inf^t\}$
 7:     for session $s \in C$ do
 8:         $m \leftarrow$ latest time $t' < t$ that $\vec{Sim}(u_{t'}, s)$ was computed.
 9:         for ($i = m + 1; i \leq t; i++$) do
10:             Compute $\vec{Sim}(u_i, s)$ using Proposition 4.3
11:         for ($j = 1; j \leq |s|; j++$) do
12:             $top.push(\vec{Sim}(u_t, s)[j], s_j)$
13:         if $|top| == k$ then
14:             $C \leftarrow C \setminus \{s | sup^t(s) < minScore(top)\}$
15: return $top$
```

5.1 Iterative Top-k Algorithm (I-TopK)

Algorithm I-TopK takes the following as input: the current session u_t, the sessions repository S, and the desired size k of the top-k set. It iterates over the repository, computing the similarity vector $\vec{Sim}(u_t, s)$ for each $s \in S$ by employing Proposition 4.3, i.e. by using the similarity vector $\vec{Sim}(u_{t-1}, s)$ calculated at the previous iteration ($t-1$). The top-k similarity scores (along with their corresponding prefixes) are maintained in a max-heap of size k, which will contain the exact set $top_k(u_t, S)$ of the top-k most similar prefixes to u_t at the end the loop.

The time complexity of I-TopK is $O(|S||\widehat{s}|\lambda)$, since it iterates over a session repository of size $|S|$ and computes a similarity vector in $O(|\widehat{s}|\lambda)$. The max-heap is maintained in a negligible cost of $O(log\ k)$, as we assume that $k << |S|$. As for space complexity, I-TopK requires storing the previous similarity vectors computed at $t-1$, therefore it requires $O(|S||\widehat{s}|)$ space, where $|\widehat{s}|$ is the average session size.

5.2 Threshold-Based Algorithm (T-TopK)

The T-TopK algorithm follows a *filter-and-refine* paradigm, s.t. in the *filter* step, repository sessions are pruned according to an *overestimation* of their similarity scores. Then, during the *refinement* step, the similarity vectors are computed only for candidate sessions passing the filter step. Importantly, T-TopK is guaranteed to be exact since the *filter* step always *overestimates* the true similarity scores and *underestimates* the filter thresholds. The novelty and efficiency of the algorithm stem from exploiting the incremental nature of the search problem.

We first describe the outline of the T-TopK algorithm, then in Sections 5.3 we present the techniques used for its *filter* step, i.e., the efficient calculation of the similarity lower and upper bounds.

T-TopK Algorithm Outline. Algorithm 1 depicts the outline of the T-TopK algorithm. For a given user session u_t, a sessions repository S and a number k, the prefixes set $top_k(u_t, S)$ is retrieved in the following manner. First, if u_t contains only one action, the I-TopK algorithm will be used, since we have no previous prefix to rely on. Otherwise we employ a *filter-and-refine* process as follows.

Filter Step. We first find candidate sessions that are likely to contain prefixes in the top-k set $top_k(u_t, S)$:

(1) Form a global similarity threshold. We first compute an initial lower bound threshold, denoted inf^t, for the similarity score of a prefix to be a member of $top_k(u_t, S)$. The threshold inf^t *underestimates* the true minimal similarity score (Line 5).

(2) Compute a similarity upper-bound for each session. We then form a similarity upper-bound for each session s, denoted $sup^t(s)$, that overestimates the maximal similarity of a prefix in s to the current user session u_t.

The algorithm then filters the repository S by retrieving all sessions having similarity upper-bounds greater than inf^t (Line 6). Importantly, since the lower bound is *underestimated* and the upper bounds are *overestimated* - no false negatives can occur. In Section 5.3 we describe how the similarity lower and upper bounds are defined and efficiently calculated.

Refinement Step. We employ a *refinement* step over the candidate sessions and calculate the exact similarity scores using their *similarity-vectors.* Recall that to efficiently construct a similarity vector (Proposition 4.3) at time t, we need to have the vector of time $t-1$. However, we may not have calculated a similarity vector at $t-1$ if a session was pruned in the filter step. Thus, for each such candidate session we reconstruct, if necessary, its previous vectors from time $< t$ (We discuss below how this computation can be performed in user idle-times, allowing T-TopK a further speed-up), then compute the current similarity vector $\vec{Sim}(u_t, s)$ (Lines 9- 10).

Last, we iteratively push each element in the similarity vector into a max-heap *top* of size k (Lines 11- 12), and further prune candidate sessions if their upper bound is lower than the minimum score in *top* (Lines 13-14). Finally, the max-heap *top* holds the set $top_k(u_t, S)$, containing the top-k most similar prefixes to u_t.

Offline computation in user idle-times. User idle-times occur between two consecutive actions - while the user examines the results of her current action, before executing the next one (which typically takes several seconds). Our algorithm utilizes such idle-times, to compute similarity vectors skipped at previous iterations *offline*, so that they are already available when needed in the top-k search.

Correctness of the T-TopK algorithm. We next sketch the correctness proof, showing that the T-TopK algorithm is correct, i.e., always retrieves the *exact* set of top-k most similar session prefixes:

Let *top* denote the output of the algorithm. We need to show that $top_k(u_t, S) = top$. We will consider $t > 1$ (the case of t=1 is trivial). First, we show that for an arbitrary prefix s_j of a repository session s, $s_j \in top_k(u_t, S) \rightarrow s_j \in top$. If $s \in top_k(u_t, S)$ then $Sim(u_t, s) \geq inf^t$ (as inf^t is the similarity lower bound). Hence, for the upper bound $sup^t(s)$ of session s we can easily see that $sup^t(s) \geq inf^t$, thus $s \in C$, i.e. s is retrieved and processed as a candidate session. The proof for the case that $s_j \in top_k(u_t, S) \leftarrow s_j \in top$ follows similar lines.

To complete the picture we still need to explain how (1) the similarity lower bound inf^t, and (2) the upper bounds $sup^t(s)$ are computed and compared.

5.3 Incremental-Based Similarity Bounds

We first explain how the similarity lower and upper bounds are defined, then describe how the candidate sessions are efficiently retrieved.

5.3.1 Similarity Threshold (Lower Bound).

The threshold inf^t forms a lower-bound (underestimated) for the similarity score of a prefix to be a member of $top_k(u_t, S)$. Intuitively, we use the sessions in the (already computed) top-k set of time $t-1$ as a potential representative for the top-k set of time t, and define our threshold w.r.t. them. Let $S^{t-1}_{top} \subseteq S$ denote the set of sessions with prefixes in $top_k(u_{t-1}, S)$. The similarity threshold is defined by:

$$inf^t = minScore(top_k(u_t, S^{t-1}_{top}))$$

As the size of S^{t-1}_{top} is at most k, inf^t is computed efficiently.

The proof that inf^t always underestimates the exact lower-bound can be immediately obtained from the simple observation that:

$$\forall S' \subseteq S, minScore(top_k(u_t, S')) \leq minScore(top_k(u_t, S'))$$

In particular, the observation holds for $S' = S^{t-1}_{top}$.

Nevertheless, we show that inf^t cannot be too far from the *exact* threshold at t-1, using the following observation:

Observation 5.1. $minScore(top_k(u_{t-1}, S))\beta - \delta \leq inf^t$

This stems from the fact that the lowest possible inf^t is obtained when the last query in u_t is not aligned in *any* of the sessions in S^{t-1}_{top} (hence a gap penalty is absorbed).

Example 5.2. Assume that $k = 1$ (i.e. we are interested in the top-1 similar prefixes), and consider the sessions u, ϕ, ψ from our running example. Considering the similarity vectors at $t = 4$ (detailed in Example 4.2), the most similar prefix is ϕ_5 and thus $top_1(u_4, \{\phi, \psi\}) = \{\phi_5\}$. Consequently, its corresponding session is in the set $S^4_{top} = \{\phi\}$. For computing the threshold $inf^5 = minScore(top_1(u_5, S^4_{top}))$, we construct the similarity vector $\vec{Sim}(u_5, \phi)$ (bottom row in Figure 2a), and take the maximal score among its elements, thus $inf^5 = 1.95$.

5.3.2 Upper-Bounding Similarity Scores.

Let $\widehat{sup}^t(s) = max_{1 \leq j \leq |s|} Sim(u_t, s_j)$ denote the *tight*, exact upper bound for the similarity score of all prefixes in s. Recall that for sessions in S^{t-1}_{top}, we already computed their similarity vectors when computed the lower bound inf^t, therefore their tight bound is already at hand. Nevertheless, for the rest of the sessions, i.e. $s \in S \setminus S^{t-1}_{top}$, we show that the incremental nature of the computation can be used to form $sup^t(s)$, an effective over-approximation to $\widehat{sup}^t(s)$. An important observation is that we are only interested in upper-bounding the scores of sessions containing at least one prefix with a similarity scores higher than inf^t (otherwise it can be safely pruned). We therefore define a *proper* similarity upper bound as follows:

Definition 5.3 (Proper upper bound). $sup^t(s)$ *is a proper upper bound w.r.t.* u_t *and* S*, if for every session* s *having some prefix* s_j *where* $Sim(u_t, s_j) > inf^t$*, we have that* $sup^t(s) \geq \widehat{sup}^t(s)$.

A second important observation, is that unless the last action in u_t, denoted q_t, is aligned to an action in $s \in S \setminus S^{t-1}_{top}$, s can be safely pruned.

Observation 5.4. *For a session $s \in S \setminus S^{t-1}_{top}$, if when computing $Sim(u_t, s)$ the action q_t is not matched with any of s's actions, then for every prefix s_j of s, $Sim(u_t, s_j) \leq inf^t$.*

This is because if q_t is not matched with an action in s_j it absorbs a gap penalty, i.e., $Sim(u_t, s_j) = Sim(u_{t-1}, s_j)\beta - \delta$ (Proposition 4.3). On the other hand, since $s_j \notin top_k(u_{t-1}, S)$ we know that $Sim(u_{t-1}, s_j) \leq minScore(top_k(u_{t-1}, S))$. Then, using Observation 5.1 we can see that $Sim(u_t, s_j)$ can never exceed inf^t.

Consequently, in the analysis below we ignore sessions where q_t is not matched with any of s's actions, and for brevity, unless stated otherwise, whenever we refer to a session s we mean one in $S \setminus S^{t-1}_{top}$ where q_t has a match.

We next provide two ways to overestimate $\widehat{sup}^t(s)$. In our efficient implementation of the filter procedure for candidate session retrieval (to be detailed in the next subsection), we use the first bound to quickly prune irrelevant sessions, then use the second one for further pruning the candidates set.

First bound (B_1). The following observation shows that we can bound $\widehat{sup}^t(s)$ from above using the top-k set of the previous iteration and the maximal similarity of the session's individual actions to the new action q_t.

Observation 5.5.

$$\widehat{sup}^t(s) \leq minScore(top_k(u_{t-1}, S))\beta^2 + max_{q\in s}\{\sigma(q_t, q)\}$$

Intuitively, this is because for every prefix not in $top_k(u_{t-1}, S)$, the similarity to u_{t-1} is smaller than $minScore(top_k(u_{t-1}, S))$. Consequently, even if the newly added action q_t is matched to the most similar action in s, by the definition of the similarity measure, the cumulative score cannot be greater than the prefix similarity (multiplied by the decay factor, following the arrival of a new action) plus the similarity of the best match (Proof omitted). We therefore use the following as our first upper-bound:

$$B_1(s) = minScore(top_k(u_{t-1}, S))\beta^2 + max_{q\in s}\{\sigma(q_t, q)\}$$

Note that since $top_k(u_{t-1}, S)$ is already computed, to calculate the bound we only need the similarity of q_t and the actions in s. As individual actions do reside in a metric space (unlike sessions/prefixes), we show in Section 5.4 how this is done efficiently.

Second bound (B_2). An alternative, less intuitive upper-bound for $\widehat{sup}^t(s)$ is achieved by dividing u_t into three disjoint subsequences w.r.t. a session s, and separately bounding the similarity to each part: (a) the segment of u_t that was already compared to s in previous iterations, denoted u_τ (b) the segment containing only the last added action q_t, and (c) the segment in between these two. By adding up the bounds for each segment we obtain a bound for the whole sequence:

Proposition 5.6.

$$\widehat{sup}^t(s) \leq max_{1\leq j\leq |s|}\{\vec{sim}_{u_\tau,s}[j]\}\beta^{2(t-\tau)} + \frac{\beta^{2(t-\tau)} - \beta^2}{\beta^2 - 1} + \\ + max_{q\in s}\{\sigma(u_t[t], q)\}$$

The latter proposition holds for $\beta < 1$[4].

Proof Sketch. We prove by induction. We denote $s_\tau := max_{1\leq j\leq |s|}\{\vec{sim}_{u_\tau,s}[j]\}$. The base case is when $t = \tau + 1$. We can easily show that:

$$\widehat{sup}^{\tau+1}(s) \leq s_\tau\beta^2 + max_{q\in s}\{\sigma(u_t[t], q)\}$$

We know that:

$$\widehat{sup}^{t+1}(s) \leq \widehat{sup}^t(s)\beta^2 + max_{q\in s}\{\sigma(u_{t+1}[t+1], q)\} \quad (2)$$

Assuming the inequality holds for t, we can bound $\widehat{sup}^t(s)\beta^2$:

$$\widehat{sup}^t(s)\beta^2 \leq \beta^2(s_\tau\beta^{2(t-\tau)} + \frac{\beta^{2(t-\tau)} - \beta^2}{\beta^2 - 1} + max_{q\in s}\{\sigma(u_t[t], q)\})$$

Since $max_{q\in s}\{\sigma(u_t[t], q)\} \leq 1$ we have:

$$\widehat{sup}^t(s)\beta^2 \leq s_\tau\beta^{2(t+1-\tau)} + \frac{\beta^{2(t+1-\tau)} - \beta^4 + \beta^2(\beta^2 - 1)}{\beta^2 - 1}$$

We can use the latter expression in (2) to obtain:

$$\widehat{sup}^{t+1}(s) \leq s_\tau\beta^{2(t+1-\tau)} + \frac{\beta^{2(t+1-\tau)} - \beta^2}{\beta^2 - 1} + max_{q\in s}\{\sigma(u_{t+1}[t+1], q)\}$$

Hence the inequality holds for $t + 1$ □

Importantly, note that this bound requires no action-similarity computations besides those already performed for B_1, and that all other values are either predefined constants or already computed in previous iterations. The proper upper bound $sup^t(s)$ is defined as the minimum of the two bounds: $sup^t(s) = min(B_1(s), B_2(s))$

5.4 Efficient Retrieval of Candidate Sessions

As mentioned above, we use the first bound B_1 to quickly identify relevant sessions, then compute the full bound only for the retrieved sessions, for further pruning. We use the following observation, which follows immediately from Observation 5.5.

Observation 5.7. *For $sup^t(s)$ to be not smaller than inf^t, s must contain some action q s.t.*

$$\sigma(q_t, q) \geq inf^t - minScore(top_k(u_{t-1}, S))\beta^2$$

To identify sessions that contain such actions, we employ an index structure that uses the fact that the action similarity measure defines a *metric space* (See Section 3). Specifically, in our implementation we use a *metric-tree* [8] - a popular index structure that harnesses the triangle inequality property of a metric space to facilitate a fast similarity search.

Sessions selection via the actions metric-tree. Individual actions are stored in a metric tree, with pointers from each action to the session it appears in[5]. From Observation 5.7 and our definition of the action distance notion, it follows that all sessions that satisfy the bound B_1 must contain some action q s.t.:

$$\Delta(q_t, q) \leq 1 - inf^t + minScore(top_k(u_{t-1}, S))\beta^2$$

We thus use the metric tree for a fast retrieval of all such actions q, and follow their associated pointers to identify the relevant sessions. Now we only need to compute Bound B_2 for the retrieved sessions to select those with upper bound greater than inf^t.

Example 5.8. For our running example, at $t = 5$ we retrieve candidate sessions via the metric tree, w.r.t. the lower bound 1.95 (computed as in Example 5.2). Recall that our goal is to retrieve sessions having an action q s.t. $\Delta(q_5, q) \leq 1 - inf^5 + minScore(top_1(u_4, \{\phi, \psi\}))\beta^2$, namely actions similar to q_5 = "c" with distance no more than $1 - 1.95 + 2.28 * 0.81 = 0.897$. Only

[4]For the case of $\beta = 1$, the middle part in the compound expression is defined to be 0

[5]The distance function used is the complement of our predefined action similarity function, i.e. $\Delta(q, q') = 1 - \sigma(q, q')$

the letters “C” and “c” meet this constraint, therefore sessions ψ (that contains “c”) is retrieved and added to the initial candidate sessions set. Then, the upper bound for ψ is given by $sup^5(\psi) = min(\{B_1(\psi), B_2(\psi)\}) = min(2.84, 3.09) = 2.84$ (full computation omitted), which is greater than the lower bound 1.95, therefore the final set of candidate sessions is $\{\psi\}$.

5.5 Analysis of Pruning Effectiveness

Intuitively, the effectiveness of T-TopK is dependent on (1) the *amount of reduction* in the examined candidate sessions, on which the exact similarity scores are calculated, as well as on (2) the *cost of retrieving* the candidate sessions.

First, let us assess the expected time complexity of T-TopK. Let $|\widehat{C}|$ be the expected number of candidate sessions meeting the bounds (we assume $|\widehat{C}| >> k$), and let $\widehat{\tau}$ be the expected number of missing similarity vectors that need to be computed for each candidate session. Also, denote by α the cost of the metric-tree search. Thus the average time complexity of T-TopK is given by $O(\alpha + \lambda|\widehat{C}|\widehat{\tau}|\widehat{s}|)$.

The parameters $|\widehat{s}|$ (mean session size) and λ (cost of individual action similarity operation) are not controllable nor affected by the implementation of the T-TopK algorithm. We therefore analyze the performance of T-TopK according to the following parameters: α - the cost of the actions metric tree search, $|\widehat{C}|$ - the expected number of candidate sessions, and $\widehat{\tau}$ the expected number of missing vectors per candidate session.

Cost of searching the actions metric tree (α). Recall that the individual actions metric tree is used to efficiently retrieve all repository sessions that satisfy Bound B_1. Based on the cost model for metric trees proposed in [9], α is given by: $\sum_{l=1}^{L} \lambda M_{l+1} F(\widehat{r_l} + r_q)$, where: $F(x)$ is the action-distance probability distribution (i.e., the probability that the distance of two arbitrary actions $\leq x$), L is the number of levels in the tree, and M_l is the number of nodes in level l of the tree, $\widehat{r_l}$ is the mean *covering radius* of nodes at level l, and r_q is the search range (in our case, $r_q = 1 - inf^t + minScore(top_k(u_{t-1}, S))\beta^2$). Also note that M_{L+1} is the total number of individual actions stored in the metric tree. Based on this cost model, we can see that (1) naturally, increasing the size of the actions domain and the cost of action-distance computation increases α. (2) More importantly, decreasing the range r_q decreases α. The latter is greatly affected by the lower bound inf^t and the upper bound $B1$, as described next.

Expected number of candidate sessions ($|\widehat{C}|$). The number of candidate sessions directly stems from the number of sessions that satisfy both bounds $B1$ and $B2$. Intuitively, $|\widehat{C}|$ depends on three factors: (1) *How high is the lower-bound.* Naturally, the *higher* the lower-bound inf^t is, the *less* the repository sessions may surpass it. The lower bound inf^t, computed by $minScore(top_k(u_t, S_{top}^{t-1}))$ is higher, if the least similar prefix in the top-k set, computed over the sessions in S_{top}^{t-1}, is of high similarity to u_t. In turn, this holds if the probability that q_t, the newly added action to u_t will be similar enough to actions contained in the sessions of S_{top}^{t-1}, and also on the *exact* similarity threshold computed at t-1 (higher is better). (2) *How low are the upper bounds.* Similarly, *lower* upper-bounds reduce the chance that repository sessions can surpass the lower bound. Both upper bounds are lower when the decay factor β is lower (this directly stems from Observation 5.5 and Proposition 5.6). Also, for both upper bounds, if the probability that q_t is aligned to an action in $s \in S \setminus S_{top}^{t-1}$ is lower - so are the upper bounds. (3) *The underlying session similarity probability distribution* - intuitively, how many sessions, on expectation, may contain prefixes with high similarity to u_t - sufficient to satisfy Bounds $B1$ and $B2$. Naturally, if many sessions are likely to have prefixes similar to u_t, more candidates will be retrieved.

Expected number of missing similarity vectors $\widehat{\tau}$. A higher number of missing similarity vectors decreases the performance of $T - TopK$. It may happen if a session did not make it to the candidates set C for several consecutive iterations, yet is suddenly considered at time t. However, recall that many of these missing vectors may be effectively computed in user idle times. We also show in Section 6.3 that even idle-times shorter than the minimal human reaction time (250ms) are sufficient to compute a significant portion of the missing vectors.

In summary, the effectiveness of the pruning is dependent on multiple, intertwined factors that stem both from the underlying structure of the analysis sessions (and actions) as well as the problem parameters (such as the gap penalty δ and the size of k). In our experimental evaluation (See Section 6) we have empirically examined the effect of various such parameters on the performance of T-TopK. Besides using a real-life session repository, we used a multitude artificially crafted repositories, each with different underlying structure, as well as different settings of the search problem. We show that in almost all such configurations, T-TopK is highly efficient in comparison to I-TopK and surpasses other baseline algorithms by 2-3 orders of magnitude.

Additional Remarks. We conclude with two remarks. First, considering the space complexity of T-TopK - on top of storing the similarity vectors there is an additional cost induced by maintaining the metric tree. However, assuming that the metric tree is balanced, the overall space complexity of the algorithm is thus $O(|S||\widehat{s}|)$, which is the same as for I-TopK. In Section 6 we show that the additional cost induced by the metric tree is marginal, even when the session repository contains over $1.5M$ individual actions.

Last, note that in practical settings the repository is dynamic, i.e., new sessions are added and existing ones are incremented. T-TopK can be easily adapted to this setting, as newly added or incremented repository sessions merely induce more *missing similarity vectors.* Recall from Section 5.2 that missing vectors can be computed during *user idle-times* (between her consecutive actions) thereby minimizing the effect on computation time.

6 EXPERIMENTAL STUDY

As mentioned above, since the speedup achieved by our optimizations may be affected by the underlying structure of the session repositories, one would ideally like to evaluate their performance on a *multitude* of real-life session repositories with different characteristics.

However, the only publicly available repository (to our knowledge) of real-life analysis sessions is rather small [1]. Therefore, we first performed experimental evaluations on a variety of artificially crafted session repositories, each having different internal characteristics that may effect the performance of our solutions. Then, we used the session repository of [1] to verify that the same performance trends hold on real life sessions as well.

Last, let us recall that our experiments focus on running-time performance. For an evaluation of the quality of recommendations generated based on *SW-SIM* we refer the reader to, e.g., [2, 3].

In what comes next, we describe the experimental setup and the construction method for the artificial repositories in Section 6.1, then show the performance of our solution compared to numerous baseline approaches in Section 6.2. Last, we examine the scalability and performance trends w.r.t. numerous parameters of the search problem and session repositories (Section 6.3).

6.1 Experimental Datasets & Setup

Artificial Session Repositories Construction. In the absence of real-life session repositories that are large enough and publicly available, we constructed a multitude of artificial repositories, each with different underlying characteristics that may affect the performance of our solution. Each repository was constructed by first building an action (metric-) space with certain, configurable properties, then constructing repository sessions by drawing actions from this space (also w.r.t. a configurable setting). The controllable parameters and the ranges we use are depicted in Table 1.

Generating the individual action space. Recall that our algorithms model analysis actions as abstract objects in a given metric space. We represent here actions as points in an N-dimensional Euclidean space, with the Euclidean distance as the distance metric. To obtain values in $[0, 1]$ range, each element is restricted to the range $[0, \frac{1}{\sqrt{N}}]$. We the simulate a controlled degree of similarity between actions as follows: first draw a (controllable) number of action points uniformly at random (u.a.r.), to serve as cluster centers, then generate additional actions around each center using a (multi-variant) normal distribution, with the cluster-center as mean. In our experiments we varied the number N of action-space dimensions, the number of action clusters, and their radius (standard deviation), obtaining different degrees of underlying action-similarity in the repository session.

Generating repository sessions. The sessions repository is constructed in an analogous manner. We first generate a set of "seed" sessions, then use them to generate sessions with varying degrees of similarity to the seed. For each session (seed or other) we draw its length from a given normal distribution. To construct a seed-session we select (u.a.r) a sequence of actions of the given length. The rest of the repository sessions are constructed based on one of the seed-sessions. To further control the similarity of an arbitrary session s to its corresponding seed, we set a fraction p of "random" actions in s. Namely, $p \cdot l$ actions in s are randomly drawn (u.a.r) from the entire action space. The rest $(1-p)\cdot l$ actions in s are chosen from the same action-clusters as the actions in the corresponding seed session. Intuitively, setting a high value of p of "random" actions significantly decreases the potential of two sessions to be similar.

In the experiments below we used multiple repository configurations, as depicted in Table 1.

REACT-IDA Session Repository. We used the only publicly available repository (to our knowledge) of real-life sessions [1], collected as part of the experimental evaluation of an existing IDA recommender system [26]. The user sessions were performed by 56 analysts who used a web-based IDA interface in order to explore four different datasets from the cyber-security domain. The repository contains 1100 distinct actions. The average session length is 8.5 actions, and the median user idle-time is 40 seconds.

Action Similarity. Recall that both algorithms require a measure for action similarity. When using the real-life repository, we employed the similarity measure of the IDA recommender system [26] whose analysis UI was used to record the sessions.[6] For the artificial repositories, where the actions reside in a multi-dimensional metric-space, we used Euclidean distance.

Parameter	Min	Max	Default
Problem Parameters			
Decay Factor β	0.1	1	0.9
Gap Penalty δ	0.05	1	0.1
Output Size k	4	24	12
Controlled Repository Parameters			
Action Dim.	5	500	25
#Action Clusters	3K	24K	6K
Cluster Rad. (std)	0.0075	0.012	0.003
#Seed Sessions	6%	16%	10%
%Random Actions (p)	0%	100%	80%
Idle Time (s)	0	5	1
Repository Scale Parameters			
#Sessions	1K	100K	10K
Session len.	$\mathcal{N}(4, 3^2)$	$\mathcal{N}(32, 12^2)$	$\mathcal{N}(16, 3^2)$

Table 1: Problem & Repository Parameters

Default Parameters Selection. The search problem parameters (namely, the decay factor β, the gap penalty δ, and the size of k) may affect the performance of our solution. However, to capture the real-life setting where the algorithms are embedded in actual IDA recommender systems, the default values are set to optimize the *quality* of the *SW-SIM* measure, rather than the performance of our optimizations. To do so, we embedded *SW-SIM* as the top-k search component in the IDA next-step recommender system [26] (code available in [1]), and performed the hyper-parameters selection routine as described in [26]. Briefly, this is done by executing a grid search for the systems' parameters (including the top-k problem parameters), and selecting the configuration that allows the recommender system to achieve the highest qualitative performance (as determined in a predictive accuracy evaluation).

As for the artificial repository construction, intermediate values were chosen as default values, as stated in Table 1. Nevertheless, in Section 6.3 we examine the effect of varying each problem/repository parameter on the running time performance of our solution.

Hardware and Software Specifications. We implemented the algorithms presented in the previous sections in *Java 8*, using Guava (https://github.com/google/guava) for the max-heap. For the metric-tree we used an M-tree [8] Java implementation (https://github.com/erdavila/M-Tree), employing *Minimum Sum of Radii* split policy (See [8]), and a maximum node capacity of 20 objects. All experiments were conducted over *Intel Core i7-4790, 3.6GHz* machine (4 dual cores), equipped with 8GB RAM.

6.2 Baseline Comparison

We compared the performance of the T-TopK algorithm to the following baseline algorithms, each computing the exact set $top_k(u_t, S)$ according to the alignment based similarity measure (Definition 3.1). Among these, we show the results of the following baselines: (1) **Naive Sequential Search (NSS)** that retrieves the set of similar prefixes by iteratively comparing u_t to each prefix of each repository session in S using a direct implementation of Definition 3.1 with no further optimizations, then selects

[6]The similarity notion considers both the action syntax and (a signature of) the results. (See [26]).

Baseline	REACT-IDA (1.1K actions)		Repo-10 (160K actions)		Repo-100 (1.6M actions)	
	Time (ms)	#ops	Time (ms)	#ops	Time (s)	#ops
NM-Tree	42.3	27.5K	18,503	12.3M	186	123M
CSE	55	36.6K	18,224	12.1M	183	121M
NSS	51.9	35K	17,204	11.5M	170	123M
OSS	7.6	5.1K	3,242	1.36M	32	13.6M
I-TopK	1	1.1K	237	160K	2.4	1.6M
T-TopK	**0.5**	**641**	**59**	**39K**	**0.9**	**722K**

Table 2: Baseline Comparison. We compared various baseline algorithms in terms of running times and the number of similarity operations

the top-k similar prefixes. (2) **Optimized Sequential Search (OSS)**, which employs the first optimization described in Section 4, instead of computing the alignment matrix for each prefix. Namely, it iterates over all sessions in $s \in S$, constructs a single alignment matrix $A_{u_t,s}$, and uses Observation 4.1 to derive the similarity scores of u_t and each prefix of s. (3) **I-TopK**, the iterative algorithm depicted in Section 5.1 which employs both of the optimizations in Section 4. (4) **T-TopK**, the optimized algorithm described in Sections 5.2-5.4. (5) **Constant Shift Embedding (CSE)[31]** and (6) **NM-Tree [34]**, are general purpose solutions for top-k search in a non-metric space. Both use *metrization* techniques: CSE use a simple solution that increments each distance score by a predefined constant, so the triangle inequality is enforced. Then, all session prefixes in S are stored in a *metric tree* (w.r.t. the new metric space). When given the ongoing session u_t, the metric tree is traversed, using the new metric to obtain the exact set of top-k similar prefixes. NM-tree uses a more sophisticated similarity-preserving transformation on the original (non-metric) distance measure, then employs an extension of the metric tree to index and query the transformed metric space.

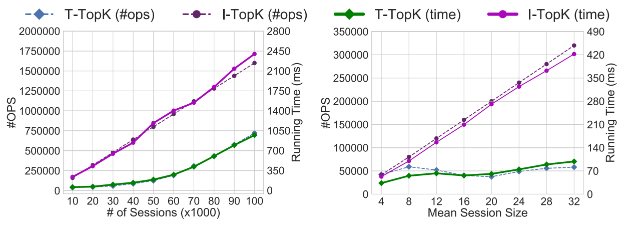
Finally, recall from Section 2 that optimization techniques dedicated to other similarity measures (e.g., DTW, Global Sequence Alignment) are inadequate for *SW-SIM*, therefore could not serve as baselines. Also, approximation-based solutions were omitted as well since this work concerns with the case of exact computation of the similarity scores.

Evaluation Process. We evaluated all baselines using a multitude of configurations for the top-k search problem (i.e. the constants β, δ of the similarity measure, and k), on various configurations of artificial repositories.

The evaluation process is as follows. Given a session repository S, in each trial we draw a random session prefix as u_t, then employ each baseline to retrieve $top_k(u_t, S)$. We performed 100 trials for each repository, capturing the average execution time and memory consumption, and the average number of action similarity operations performed in each run.

Results. Table 2 presents a representative sample of the results, showing the average running time and the number of action similarity operations (denoted #ops) obtained by each baseline. The comparison is presented for the REACT-IDA repository (Containing 1.1K actions) as well as Repo-10 and Repo-100, which are two artificial session repositories (with the default configuration stated in Table 1) containing 10K and 100K sessions (resp., 160K and 1.6M actions).

First, for both Repo-10 and Repo-100, the performance of *NM-Tree* and *CSE* is almost on-par with NSS (the naive sequential search). This could be due to the fact that transforming the non-metric space may induce high overlap between the metric-tree nodes, therefore the search deteriorates to sequential search plus additional overhead induced by the metric tree (See [34]). Second, note that OSS improves running times by $5X$ (and #ops by $9X$)



(a) Repository Size **(b) Mean Session len.**

Figure 3: Effect of Scale Parameters

over NSS due to its efficient use of Observation 4.1. However, a more significant improvement of $15X$ in running times (and $9X$ in terms of #ops) is achieved by I-TopK which utilizes the incremental computation. **Finally, an additional 2-4X speedup is obtained by T-TopK, resulting in an overall improvement of 189X to 291X over the NSS baseline**.

The performance trends on the REACT-IDA repository (comprising the smaller collection of real-life sessions) are similar, with some minor variation. We can first see that NM-Tree performs slightly better than NSS and CSE. This is due to the underlying structure of the dataset that allows the NM-Tree to perform (and store) the distance-preserving transformations more efficiently.

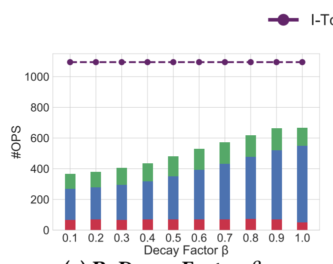
However, both our optimized algorithms perform significantly better, with T-TopK dominating. Interestingly, while the REACT-IDA is fairly small, the pruning-based optimizations of T-TopK are still highly effective. T-TopK retains a $2X$ speedup compared to I-TopK, and significantly outperforms the other baselines.

Next, we present a performance comparison of the algorithms when employed on a multitude of different repositories and with different problem parameters, as well as an examination of the computation segments in T-TopK. Since in all configurations, the performance of T-TopK was significantly better (by at least an order of magnitude) than the baseline algorithms, we omit them from presentation and use only I-TopK as a baseline.

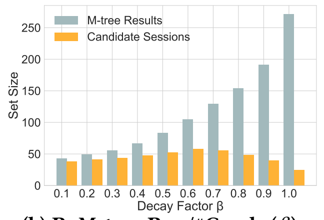
6.3 Scalability & Parameters Effect

We next analyze the performance of the T-TopK Algorithm compared to I-TopK, by varying the problem parameters as well as the repository parameters, one at a time, while keeping the rest at their default values (As in Table 1). We measured both the running times, number of action similarity operations (#*ops*), and memory consumption.

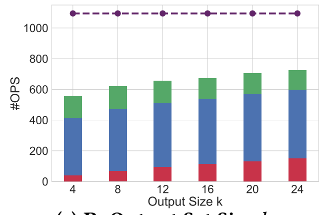
Scalability Parameters. Figure 3a depicts the performance of the algorithms when increasing the number of sessions in the repository. For both algorithms we can see a rather linear increase in running times (correspondingly, #*ops* however the two plots overlap), but T-TopK shows a significant speed up of 3X (on average) over I-TopK. Moreover, as we can see in Figure 3b, when the mean *session length* increases, T-TopK performance stays stable compared to a linear increase for I-TopK. This demonstrates that



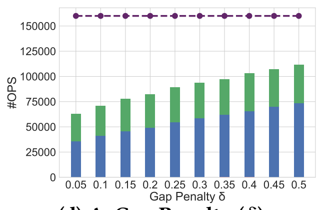
(a) R: Decay Factor β.



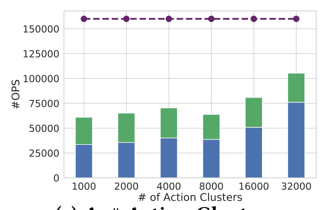
(b) R: M-tree Res./#Cands (β).



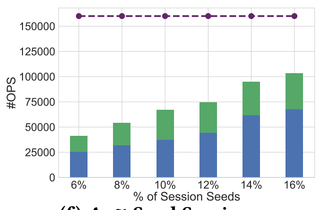
(c) R: Output Set Size k



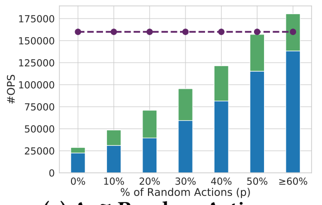
(d) A: Gap Penalty (δ)



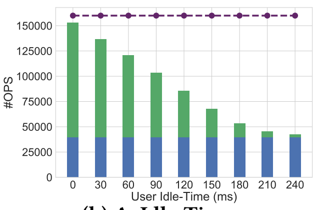
(e) A: # Action Clusters



(f) A: % Seed Sessions



(g) A: % Random Actions



(h) A: Idle-Time

Figure 4: Parameters Effect on REACT-IDA (R) and Artificial (A) Datasets

the effectiveness of the threshold-based approach further grows for longer sessions. As for memory consumption, the maximal usage for T-TopK did not exceed $74MB$ even for a repository as big as 100K sessions (1.6M individual actions), which is negligible in practice. We therefore do not further discuss memory consumption in the next experiments.

Problem Parameters. Recall that the problem parameters are the gap penalty δ, the decay factor β and the output set size k. We examine their effect on the performance, when varying each parameter, and keeping the rest at their default values. To gain a better insight on the computation of T-TopK, we break the total *#ops* performed into three computational segments: (1) forming the initial similarity lower bound, (2) the metric-tree search, and (3) computing similarity vectors for candidate sessions. The corresponding running time trends are similar, therefore omitted from the figures below. Also, as Segment (3) is negligibly small when using the default idle-time, we further restrict the default user idle-time to 150ms, which naturally stresses T-TopK. The performance of I-TopK in terms of #ops is represented by a dotted flat line since it is not dependent on the problem parameters.

We observe that increasing each of the problem parameters results in a minor increase in #ops for T-TopK. Figure 4a and Figure 4c show the effect of the decay factor β and the output size k on performance, when the algorithms are employed on the REACT-IDA repository, and Figure 4d shows the effect of the gap penalty δ on the default artificial repository (the effect of δ on the REACT-IDA repository was marginal, therefore the figure is omitted). The increase in performance is expected, since the values of the problem parameters play a part in the lower/upper bound computations: (1) The decay factor β affects the upper bound, therefore lower values induce more restrictive candidate selection, and thereby better performance. (2) Increasing the output size k causes a decrease in the lower bound threshold inf^t, thus more candidate sessions are examined.

To gain a deeper insight on the performance w.r.t. the computation segments, we examine in Figure 4b, the number of sessions retrieved in the metric-tree search results (i.e. sessions satisfying bound B_1) and the number of sessions among them that also meet bound B_2 (hence satisfy the upper bound), for varying β values. We can see that the first set grows with β (hence the cost of Segment 2 increases), but many sessions are pruned via bound B_2 (thus the cost of Segment 3 does not increase).

Repository Parameters. As is the case for most top-k search optimizations and dedicated data structures, the underlying structure of the data points may impact the performance of the T-TopK algorithm (note that the incremental similarity optimizations presented in Section 4 are not affected by such parameters).

Therefore, to properly evaluate the effectiveness of T-TopK we constructed a multitude of artificial repositories (as described above), each with a different underlying structure that stems from a particular repository-parameters configuration. We varied each construction parameter and examined its effect on the performance of T-TopK (keeping the rest of the parameters at their default configuration). Figures 4e, 4f and 4g depict the performance of T-TopK compared to I-TopK when varying the amount of individual action clusters, percentage of "seed sessions", and the percentage of random actions in a session, respectively. Intuitively, increasing the values of each of these parameters inflict more "randomness" on the underlying structure of the data points (sessions), therefore the mean similarity score of two arbitrary sessions decreases. In turn, lower similarity scores imply smaller lower bounds, therefore more candidate sessions are examined. With the exception of one case, in all observed settings, even when increasing these parameters, T-TopK was still effective, outperforming I-TopK. The only setting where I-TopK outperformed T-TopK was when setting a high ($> 60\%$) percentage of random actions in the repository sessions, which significantly decreases the average session similarity score (Importantly, further increasing this parameter had a negligible effect on performance). However, as we show next, given slightly higher user idle-times, (which is a very reasonable assumption) T-TopK still outperforms I-TopK, even when the percentage of random actions is high, as the computation segment of calculating similarity vectors (in green) is significantly diminished.

User Idle-Times. We measured how the mean user idle-time affects the performance of T-TopK. Such idle time between consecutive actions of the same user often takes several seconds. Figure 4h depicts the performance when varying the expected time ranges from 0 to 0.24 seconds. Interestingly, even the latter, which is lower than the minimal human reaction time to a visual stimulus, was already sufficient to compute all missing vectors offline. As expected, T-TopK improves when the idle-time increases, and when all missing vectors are computed offline - T-TopK obtains almost a 3X speedup over I-TopK.

7 CONCLUSION

In this work, we show (for the first time, to our knowledge) that by utilizing the progressive nature of IDA sessions, a major running-time speedup (of over $180X$) can be obtained, compared to current solutions for top-k similarity search of analysis sessions. Our solution allows IDA recommendation systems to effectively rely on much larger session repositories while retaining interactive response times.

However, the scope of our work is limited in two main aspects: First, our solution is dedicated to the *SW-SIM* similarity measure for analysis sessions. While considered a very comprehensive, useful measure, some IDA recommendation systems use different session similarity measures. Nevertheless, we believe that the core principles of our solution may be applicable (with some tweaking and adaptations) to other similarity measures - even outside the scope of IDA, (e.g. scientific/business workflows, prescriptive analytics, mashups [5, 13, 42]).

Second, as common in many filter-and-refine based frameworks, the efficiency of the pruning techniques used by the T-TopK algorithm may be affected by underlying characteristics of the session repository, as well as the parameters of the search problem. In the absence of large enough, publicly available real-life IDA workloads, our experimental evaluation is performed, primarily, over a multitude of carefully crafted artificial repositories, each with different characteristics. Although we have shown that similar performance trends occur on the real-life session repository as well (the only one that is publicly available), it is still left to demonstrate the performance speedup on other real-life IDA workloads (when such become publicly available). Nevertheless, recall that the efficiency of the single-alignment optimizations, and the computation of the similarity vectors (as presented in Section 4), is independent of the session repository or problem parameters - and even these alone still provide a significant speedup of $40 - 80X$ compared to the currently available solutions.

Acknowledgments. This work has been partially funded by the Israel Innovation Authority (MDM), the Israel Science Foundation, the Binational US-Israel Science foundation, Len Blavatnik and the Blavatnik Family foundation, and Intel® AI DevCloud.

REFERENCES

[1] [n. d.]. REACT: IDA Benchmark Dataset. ([n. d.]). https://git.io/fhyOR.

[2] Julien Aligon, Enrico Gallinucci, Matteo Golfarelli, Patrick Marcel, and Stefano Rizzi. 2015. A collaborative filtering approach for recommending OLAP sessions. *DSS* 69 (2015).

[3] Julien Aligon, Matteo Golfarelli, Patrick Marcel, Stefano Rizzi, and Elisa Turricchia. 2014. Similarity measures for OLAP sessions. *KAIS* 39 (2014).

[4] Marie Aufaure, Nicolas Kuchmann-Beauger, Patrick Marcel, Stefano Rizzi, and Yves Vanrompay. 2013. Predicting your next OLAP query based on recent analytical sessions. *DaWaK* (2013).

[5] Ralph Bergmann and Yolanda Gil. 2014. Similarity assessment and efficient retrieval of semantic workflows. *Information Systems* 40 (2014), 115–127.

[6] Michael W Berry and Murray Browne. 2005. *Understanding search engines: mathematical modeling and text retrieval.* Vol. 17. Siam.

[7] Paolo Ciaccia and Marco Patella. 2002. Searching in metric spaces with user-defined and approximate distances. *ACM Transactions on Database Systems (TODS)* 27, 4 (2002), 398–437.

[8] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB.*

[9] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1998. A cost model for similarity queries in metric spaces. In *SIGCART.* ACM.

[10] Dong Deng, Guoliang Li, He Wen, HV Jagadish, and Jianhua Feng. 2016. META: an efficient matching-based method for error-tolerant autocompletion. *PVLDB* 9, 10 (2016).

[11] Michel Marie Deza and Elena Deza. 2009. Encyclopedia of distances. Springer, 1–583.

[12] Remco Dijkman, Marlon Dumas, Boudewijn Van Dongen, Reina Käärik, and Jan Mendling. 2011. Similarity of business process models: Metrics and evaluation. *Information Systems* 36, 2 (2011), 498–516.

[13] Fan Du, Catherine Plaisant, Neil Spring, and Ben Shneiderman. 2016. EventAction: Visual analytics for temporal event sequence recommendation. In *2016 IEEE Conference on Visual Analytics Science and Technology (VAST).* IEEE, 61–70.

[14] Robert C Edgar. 2010. Search and clustering orders of magnitude faster than BLAST. *Bioinformatics* 26, 19 (2010), 2460–2461.

[15] Magdalini Eirinaki, Suju Abraham, Neoklis Polyzotis, and Naushin Shaikh. 2014. Querie: Collaborative database exploration. *TKDE* (2014).

[16] Arnaud Giacometti, Patrick Marcel, and Elsa Negre. 2009. *Recommending multidimensional queries.* Springer Berlin Heidelberg.

[17] Shengyue Ji, Guoliang Li, Chen Li, and Jianhua Feng. 2009. Efficient interactive fuzzy keyword search. In *Proceedings of the 18th international conference on World wide web.* ACM, 371–380.

[18] W James Kent. 2002. BLAT—the BLAST-like alignment tool. *Genome research* 12, 4 (2002), 656–664.

[19] Eamonn Keogh and Chotirat Ann Ratanamahatana. 2005. Exact indexing of dynamic time warping. *Knowledge and information systems* 7, 3 (2005), 358–386.

[20] Sung-Ryul Kim and Kunsoo Park. 2000. A dynamic distance table. *CPM* (2000).

[21] Gad M Landau, Eugene W Myers, and Jeanette P Schmidt. 1998. Incremental string comparison. *SIAM J. Comput.* 27, 2 (1998), 557–582.

[22] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg. 2009. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology* (2009).

[23] David Leake and Joseph Kendall-Morwick. 2008. Towards case-based support for e-science workflow generation by mining provenance. In *European Conference on Case-Based Reasoning.* Springer, 269–283.

[24] Guoliang Li, Jiannan Wang, Chen Li, and Jianhua Feng. 2012. Supporting efficient top-k queries in type-ahead search. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval.* ACM, 355–364.

[25] Elaine R Mardis. 2008. The impact of next-generation sequencing technology on genetics. *Trends in genetics* 24, 3 (2008), 133–141.

[26] Tova Milo and Amit Somech. 2018. Next-Step Suggestions for Modern Interactive Data Analysis Platforms. In *KDD.*

[27] David Novak and Pavel Zezula. 2006. M-Chord: a scalable distributed similarity search structure. In *ICSIS.*

[28] Panagiotis Papapetrou, Vassilis Athitsos, George Kollios, and Dimitrios Gunopulos. 2009. Reference-based alignment in large sequence databases. *VLDB* (2009).

[29] Srinivasan Parthasarathy, Mohammed Javeed Zaki, Mitsunori Ogihara, and Sandhya Dwarkadas. 1999. Incremental and interactive sequence mining. In *CIKM.*

[30] Stefano Rizzi and Enrico Gallinucci. 2014. CubeLoad: a parametric generator of realistic OLAP workloads. In *CAISE.*

[31] Volker Roth, Julian Laub, Motoaki Kawanabe, and Joachim M Buhmann. 2003. Optimal cluster preserving embedding of nonmetric proximity data. *TPAMI* (2003).

[32] Hiroaki Sakoe and Seibi Chiba. 1978. Dynamic programming algorithm optimization for spoken word recognition. *TASSP* (1978).

[33] Victor Sepulveda and Benjamin Bustos. 2010. CP-Index: using clustering and pivots for indexing non-metric spaces. In *SISAP.*

[34] Tomáš Skopal and Jakub Lokoč. 2008. NM-tree: Flexible approximate similarity search in metric and non-metric spaces. In *DEXA.*

[35] Temple F Smith and Michael S Waterman. 1981. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197.

[36] Johannes Starlinger, Sarah Cohen-Boulakia, Sanjeev Khanna, Susan B Davidson, and Ulf Leser. 2016. Effective and efficient similarity search in scientific workflow repositories. *FGCS* 56 (2016), 584–594.

[37] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, et al. 2013. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *VLDB* (2013).

[38] Yu Tang, Yilun Cai, Nikos Mamoulis, Reynold Cheng, et al. 2013. Earth mover's distance based similarity search at scale. *VLDB* (2013).

[39] Tatiana A Tatusova and Thomas L Madden. 1999. BLAST 2 Sequences, a new tool for comparing protein and nucleotide sequences. *FEMS Microbiol. Lett.* 174, 2 (1999).

[40] Sebastian Wandelt, Johannes Starlinger, Marc Bux, and Ulf Leser. 2013. RCSI: Scalable similarity search in thousand (s) of genomes. *VLDB* (2013).

[41] Ting Xie, Varun Chandola, and Oliver Kennedy. 2018. Query log compression for workload analytics. *Proceedings of the VLDB Endowment* 12, 3 (2018), 183–196.

[42] Sen Yang, Xin Dong, Leilei Sun, Yichen Zhou, Richard A Farneth, Hui Xiong, Randall S Burd, and Ivan Marsic. 2017. A Data-driven Process Recommender Framework. In *KDD.*

[43] Xiaoyan Yang, Cecilia M Procopiuc, and Divesh Srivastava. 2009. Recommending join queries via query log analysis. In *ICDE.*

[44] Youwei Yuan, Weixin Chen, Guangjie Han, and Gangyong Jia. 2017. OLAP4R: A Top-K Recommendation System for OLAP Sessions. *TIIS* (2017).

pgFMU: Integrating Data Management with Physical System Modelling

Olga Rybnytska, Laurynas Šikšnys, Torben Bach Pedersen, and Bijay Neupane
Department of Computer Science, Aalborg University
olga@cs.aau.dk,siksnys@cs.aau.dk,tbp@cs.aau.dk,bj21.neupane@gmail.com

ABSTRACT

By expressing physical laws and control strategies, interoperable physical system models such as Functional Mock-up Units (FMUs) are playing a major role in designing, simulating, and evaluating complex (cyber-)physical systems. However, existing FMU simulation software environments require significant user/developer effort when such models need to be tightly integrated with actual data from a database and/or model simulation results need to be stored in a database, e.g., as a part of larger user analytical workflows. Hence, users encounter substantial complexity and overhead when using such physical models to solve analytical problems based on real data. To address this issue, this paper proposes pgFMU - an extension to the relational database management system PostgreSQL for integrating and conveniently using FMU-based physical models inside a database environment. pgFMU reduces the complexity in specifying (and executing) analytical workflows based on such simulation models (requiring on average 22x fewer code lines) while maintaining improved overall execution performance (up to 8.43x faster for multi-instance scenarios) due to the optimization techniques and integration between database and an FMU library. With pgFMU, cyber-physical data scientists are able to develop a typical FMU workflow up to 11.74x faster than using the standard FMU software stack. When combined with an existing in-DBMS analytics tool, pgFMU can increase the accuracy of Machine Learning models by up to 21.1%.

1 INTRODUCTION

Cyber-physical system experts, cyber-physical data scientists, and cyber-physical developers often need to analyze, predict, and simulate physical systems. For this purpose, *physical system models* are often used to capture time-dependent behaviour and dynamics of such systems [1]. They offer powerful, rigorous, and cost-effective means to characterize and reason about such systems, without the need to build, interact, and/or interfere with such systems. Physical system models (*physical models* for short) are well supported by a number of physical system modelling software tools and environments. However, each such modelling environment often uses a specialized form and format of a physical model with limited possibilities to utilize such models across different tools and environments. To mitigate this problem, *Functional Mock-up Interface (FMI)* [2] has emerged as a de-facto standard [3] to facilitate physical model exchange and co-simulation across a large number of modelling tools. In FMI, physical models are compiled into a standard representation, denoted as *functional mock-up units* (FMUs). FMUs reflect real physical systems composed of physical and digital components interacting in complex ways according to a set of pre-defined control strategies and physical laws. FMUs allow accurately defining system behaviour even in the physical states that are not observable in the real world, unlike what is required by traditional AI/ML models (e.g., artificial neural networks). Due to these advantages, FMUs continue gaining popularity in relevant physical modeling communities [5]. FMI has already been broadly adopted and supported by 130+ software tools, including well-known simulation environments Simulink (Matlab) [4] and EnergyPlus [6] (80.000+ downloads), as well as JModelica [8]/Open Modelica [10] (more than 20 companies and 30 universities in the consortium) with 1600+ model components and 1350+ functions from many domains available in a standard library alone.

Despite comprehensive physical modelling support, existing FMI-based simulation environments and tools offer poor database (DB) integration and lack built-in support for conveniently including physical models into user-defined analytical workflows. Thus, user data (e.g., model parameters, measurements, control inputs) cannot be conveniently supplied to the model from a DB and model results cannot be effectively used in larger analytical workflows (e.g., those encompassing multiple simulations) while offering convenient declarative approaches to manage such physical models within a common data management and physical modelling environment. Without these capabilities, model-driven analytical tasks become complicated and slow in terms of both development time and execution, and less usable for users working with, e.g., prescriptive analytics applications [12]).

As a running example, consider a prediction problem from the energy field. The aim is to predict and analyze indoor temperatures inside a house that is heated by an electric heat pump (HP) under different heating scenarios (e.g., *no heating*, *heating at max power*). For this task, a physical model represented as an FMU needs to be calibrated and simulated using measurements and weather data stored in the database. Predictions need to be stored in the database, for further analysis and visualization. In this case, as shown in Figure 1, the user has to pick relevant FMU-compliant software tools (e.g., JModelica [8] or Python [7] + PyFMI [14] + ModestPy [3]) and then use these tools to (1) load a pre-generated FMU file or manually build an FMU file from a model specification file, (2) read historical measurements and (future) control inputs from a database, (3) recalibrate the model (e.g, using ModestPy) in case the model cannot ensure the good fit with the historical measurements, (4) validate the model against real measurements, and update the model and/or parameters values, (5) simulate the updated model to generate temperature predictions for different scenarios (e.g, using PyFMI), (6) export predicted values back to the database, and (7) perform further analysis utilizing a DBMS. This imposes limitations for the user in terms of number of software tools and libraries to use, and in terms of the overall complexity and ability to effectively utilize physical models in larger analytical workflows where both simulation and optimization are required. In this and similar user workflows, interleaved data exchange between a database and a

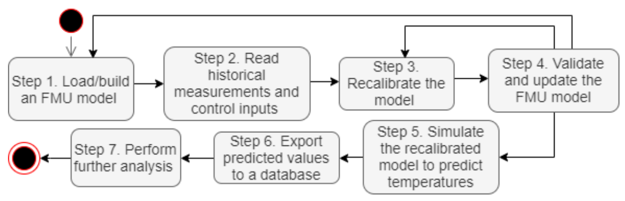

Step 1. Load/build an FMU model
Step 2. Read historical measurements and control inputs
Step 3. Recalibrate the model
Step 4. Validate and update the FMU model
Step 5. Simulate the recalibrated model to predict temperatures
Step 6. Export predicted values to a database
Step 7. Perform further analysis

Figure 1: Running example workflow

modelling tool makes the workflows difficult to specify and evaluate, leading to significant implementation complexity, developer and performance overheads. This calls for new solutions offering more tight integration between a database and a modeling tool.

This paper proposes pgFMU – an extension to the PostgreSQL [9] DBMS to address these issues. pgFMU is a SQL-based model- and data management environment that brings benefits to cyber-physical data scientists and cyber-physical software developers. pgFMU facilitates FMU model storage, simulation, and parameter estimation tasks by effectively integrating FMI inside PostgreSQL. For this purpose, pgFMU offers a number of User-Defined Functions (UDFs) accessible by simple SQL queries for each necessary operation. As such, our extension exhibits the following advantages: (1) increased user productivity (on average 11.74x faster for user-defined workflows in terms of development time) due to the usage of a single integrated system for FMU-specific use cases, (2) reduced system complexity (22x fewer code lines), (3) increased performance due to the reduction in I/O operations and optimization for multi-instance workflows (up to 8.43x faster), (4) reduced number of error-prone actions by minimizing the number of software systems and tools a data scientist is required to work with, and 5) when used in combination with traditional Machine Learning models, pgFMU can increase model accuracy by up to 21.1%. These are supported by the results of our experiments which are based on the real-world use cases. Lastly, pgFMU is an open-source project and can be found on GitHub[1].

The rest of the paper is organized as follows: Section 2 elaborates on solving the aforementioned use case using traditional stack, Section 3 discusses the related work Section 4 highlights technical challenges of integrating FMUs inside a FMS, Section 5 discusses how pgFMU tackles model storage and specification, Section 6 presents our pgFMU-based approach for parameter estimation, Section 7 discusses the model simulation, Section 8 presents the experimental evaluation, and Section 9 concludes the paper and suggests future research directions.

2 RUNNING EXAMPLE

In this section, we follow Figure 1 and elaborate on the steps of solving the exemplified case of predicting indoor temperatures of a house. This example illustrates a typical process, steps, and software packages a cyber-physical data scientist would adopt to address this specific and similar problems based on FMU models.

Load / build an FMU model. A Functional Mockup Unit (FMU) is a .zip file consisting of a number of XML files, model and underlying solver implementation C-language files and/or their binary compilations for execution on different hardware platforms. The XML files describe model variables and parameters, attributes, default stop time, tolerance, etc., to be used, mostly, by FMU libraries and software tools and not the end-user. The cyber-physical data scientist can either download a pre-compiled FMU file from the (third-party) system modellers, or build an FMU file manually using a modelling tool, e.g., Open Modelica [10]. The latter option requires domain knowledge and modeling skills to mathematically capture all instantaneous thermal energy losses (and gains), intensities, and their balance within the building (e.g., windows, walls). For this problem, at time $t_k, k = 1 \ldots n$, temperature dynamics of the house can be captured by a generic *linear time invariant state space* model (1) (adapted from [11]):

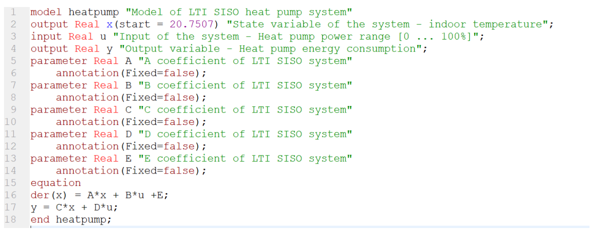
$$\begin{aligned} x(t_{k+1}) &= F(x(t_k), u(t_k), \theta(t), t) \\ y(t_k) &= H(x(t_k), u(t_k), \theta(t), t), \end{aligned} \tag{1}$$

where $x(t_0) = x_0$ is the initial state, F is a linear or non-linear function, $F : \Re^n \times \Re^m \times \Re^p \times \Re \to \Re^n$, $x(\cdot) \in \mathbb{R}^n$ is the state variable vector, $u(\cdot) \in \mathbb{R}^m$ is the model input vector, $\Theta(\cdot) \in \mathbb{R}^p$ is the parameter vector, $y(\cdot) \in \mathbb{R}^O$ is the model output vector, and $H : \Re^n \times \Re^m \times \Re^p \times \Re \to \Re^O$ is the output function.

This model often needs to be translated into a representation required by a specific modelling tool, e.g., the Modelica [13] program shown in Figure 2. There, u is the *input variable* – heat pump power rating settings in the range [0 ... 1], corresponding to [0 .. 100%] of HP power operation; x is the *state variable* – indoor temperatures; y is the *output variable* – energy consumed by a heat pump. The parameters are represented by A, B, C, D, and E; $A = \left(-\frac{1}{R \cdot Cp}\right)$, $B = \left(\frac{P \cdot \eta}{Cp}\right)$, $C = P$, $D = 0$, $E = \left(\frac{\theta_a}{R \cdot Cp}\right)$, where $Cp = 1.5kWh/^{\circ}C$ is the thermal capacitance (the amount of energy needed to heat up by 1 $^{\circ}C$ within 1 hour); $R = 1.5^{\circ}C/kW$ is the thermal resistance; $P = 7.8kW$ is the rated electrical power of the heat pump; $\eta = 2.65$ is the performance coefficient (the ratio between energy usage of the heat pump and the output heat energy); $\theta_a = -10^{\circ}C$ is the outdoor temperature.

Read historical measurements and control inputs. Modelling tools often have poor support for database integration and require model parameters and inputs to be provided in a predefined format, usually a text file. If model inputs are stored in a database, the users are required to either manually export them to use it within the modelling software, or to download and familiarize themselves with often complex library functionality (e.g. Matlab Database Explorer app [4] or *psycopg2* [16] Python package). Yet, storing the measurements in a database has a number of advantages while handling the concurrent I/O operations such as managing the information from the multiple sensors and serving end-user applications. Lastly, the user is required to retrieve the control inputs from the FMU, and manually match them with the input data series obtained from the measurements.

Recalibrate the model. Values of one or more model parameters (e.g., A, B, E in Figure 2) are often either not known, or tuned for another physical system. This may result in poor model predictions, or erroneous decisions. *Parameter estimation* (or model calibration) is the operation of fitting model parameters to actual measurements, such that simulated model states and



```
model heatpump "Model of LTI SISO heat pump system"
output Real x(start = 20.7507) "State variable of the system - indoor temperature";
input Real u "Input of the system - Heat pump power range [0 ... 100%]";
output Real y "Output variable - Heat pump energy consumption";
parameter Real A "A coefficient of LTI SISO system"
    annotation(Fixed=false);
parameter Real B "B coefficient of LTI SISO system"
    annotation(Fixed=false);
parameter Real C "C coefficient of LTI SISO system"
    annotation(Fixed=false);
parameter Real D "D coefficient of LTI SISO system"
    annotation(Fixed=false);
parameter Real E "E coefficient of LTI SISO system"
    annotation(Fixed=false);
equation
der(x) = A*x + B*u +E;
y = C*x + D*u;
end heatpump;
```

Figure 2: Heat pump LTI SISO model in .mo format

[1]https://github.com/OlgaRyb/pgFMU.git

outputs acceptably match measured system states and outputs. In our problem, the unknown model parameter values are A, B and E (see Figure 2), and the sum of squared errors between the measured and simulated indoor temperatures is to be minimized.

Parameter estimation can be performed, for instance, using the ModestPy Python package [3] and its built-in function `Estimation`. In this situation, `Estimation` requires a path to the working directory, a path to the FMU model file, input measurements, known values of the parameters, values to be estimated, and a dataset with real measurements to calibrate upon. All these formal parameters need to be specified explicitly by the user. Additionally, in case the user needs to further use the updated model, parameters update is required to be done manually by calling a specific function from the *PyFMI* [14] package.

Often, a set of relevant parameters to be estimated has to be identified by the user. This operation demands an extensive domain knowledge, or assistance of the external domain expert. As an alternative, the user can use the functionality of *PyFMI* Python package to retrieve the relevant parameters of an FMU model. However, the list of retrieved parameters always needs to be accompanied by often complex filtering. For example, by default the *PyFMI* fetches the full list of parameters, including the parameters related to the built-in solver internally connected with the FMU model. Such parameters are not relevant for the parameter estimation operation, therefore, should be filtered out.

Validate and update the FMU model. Once parameter estimation is performed, the model needs to be validated before it can be used for subsequent predictions, e.g., using cross-validation. In case the model cannot generate predictions with sufficient accuracy, the model has to be either re-calibrated (e.g., with a different set of parameters) or further refined or replaced (i.e., using another FMU). In our case, validation should be performed using user-defined Python scripts.

Simulate the recalibrated model to predict temperatures. Next, we need to generate predictions by simulating the calibrated FMU model based on the desired input (HP power rating setting in the range [0 ... 1]/heating scenario). During simulation, model outputs and states are computed based on the provided inputs (and previous states), as seen in Figure 3. To simulate the model in this workflow, the user need to write a (Python) program (e.g. using the *PyFMI*) to load the *.fmu* file, read input data, map data to the form required by the simulation library (e.g. using *numpy* and *pandas* [15]), simulate the model (*PyFMI*), and insert simulation results back in the database (*psycopg2*).

Export predicted values to a database. For further analysis and visualization, the user needs to export data either directly to a database or first to a text file and then import the text file into a database using the respective SQL command.

Perform further analysis. Stored predicted indoor temperatures may further be used for subsequent visualization, analysis, optimization, and/or decision support. However, if predictions

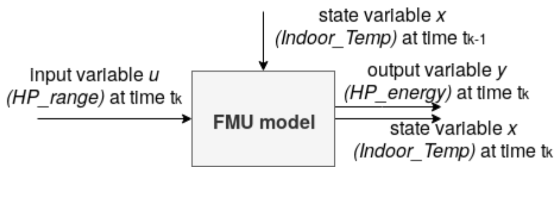


Figure 3: Simplified simulation FMU schematic of the HP system

Table 1: Workflow operations

Operation	**Package**	**Code lines**	
		Python	*pgFMU*
Load/build an FMU model	PyFMI	4	1
Read historical measurements and control inputs	psycopg2, PyFMI, pandas	12	-
Recalibrate the model	ModestPy, pandas	15	1
Validate & update the FMU model	PyFMI, pandas	7	-
Simulate the recalibr. model to predict temp.	PyFMI, Assimulo, numpy	24	1
Export predicted values to a DB	psycopg2, pandas	4	-
Perform further analysis	psycopg2, PyFMI	22	1
Total		**88**	**4**

under different model inputs are required, or by using different models, this overall process needs to be repeated. When predictions with multiple FMU model need to be performed, the cyber-physical data scientist faces additional complexity of managing all these models, their parameters values, and predictions. As seen in Table 1, for this particular example, the data scientist would have to perform 7 steps utilizing 6 different Python packages and writing 88 lines of code. The necessity to utilize multiple software packages makes development time-consuming and error-prone. At the same time, the usage of different programming interfaces makes such tasks difficult to perform, manage, and customize. FMU integration inside a DBMS, as in pgFMU, results in an order of magnitude reduction in the code (as seen in last table column).

3 RELATED WORK

FMU model simulation is widely supported by over 130 tools and packages, most of them being commercial ones. Among open-source projects are OpenModelica[10], PyFMI [14], and FMI Library as a part of JModelica[8]. However, even while providing GUI (e.g., OpenModelica), these tools do not have the solid support for the import and export of data from/to a DBMS. For example, OpenModelica requires an intermediary text file to be created as an input to the simulation engine; this text file should follow a strict format to be able to continue the simulation.

Another well-known environment for FMU simulation and parameter estimation is JModelica [8]. JModelica offers a Python interface, and serves the user with a number of functions required for a broad range of simulation- and optimization-related tasks. Another advantage of JModelica is package orchestration: even though some packages (e.g. PyFMI or Assimulo [20]) are claimed to be independent of other software, individual package management still remains a challenge. Nonetheless, JModelica does not provide yet a direct database input support, so the users are required to use extra packages (e.g. psycopg2).

At the same time, attempts were made to develop in-DBMS analytics: SAS [21] in collaboration with Teradata suggested the integration of DBMS with predictive modelling (including regression analysis and time series analytics); Tiresias [22] - a PostgreSQL-based system that supports "how-to" queries for defining and integrated evaluation of constrained optimization problems, and SolveDB - an extensible DBMS for SQL-based optimization problems [23]. The existing DBMSes offer only limited analytical support, such as in-DBMS linear optimization and, to

Table 2: Comparison between in-DBMS analytics tools and pgFMU

Feature	MADlib	Microsoft SQL Server ML Services	pgFMU
Data query language	SQL	SQL	SQL
Model integration approach	UDFs	Stored procedures	UDFs
In-DBMS machine learning	✓	✓	✗
In-DBMS physical models	✗	✗	✓
- FMU management	✗	✗	✓
- FMU simulation	✗	✗	✓
- FMU parameter estimation	✗	✗	✓

some extent, forecasting and simulation. To our best knowledge, no systems have until now supported in-DBMS storage, analysis, simulation, and calibration of the FMI-compliant models.

There exists a number of tools that incorporate traditional Machine Learning (ML) models into a DBMS, e.g., the hugely popular MADlib [27] and the recently released Microsoft (MS) SQL Server ML Services [28]. Table 2 shows a comparison of pgFMU with these systems. As we can see, neither MADlib, nor Microsoft SQL Server ML Services support physical system models (like FMUs), unlike pgFMU. MADlib follows the same integration approach as pgFMU by offering a set of UDFs specialized for ML. MS SQL Server ML Services provides a single T-SQL stored procedure for executing external Python and R scripts for ML, which is considerably less user-friendly as two languages (SQL + Python/R) are involved. While the UDFs of pgFMU can be installed as a DBMS extension, pgFMU can be used as a complement to existing ML tools, e.g., MADlib, which we demonstrate in Section 8.

4 CHALLENGES OF INTEGRATING FMUS INSIDE A DBMS

In the next section, we highlight 3 essential technical challenges of integrating FMUs inside a DBMS:

Challenge 1. Language Integration Challenge: *How to integrate and expose FMUs in database queries?* The challenge is to offer effective new SQL constructs, so that FMUs and FMU instances can be created, analyzed, manipulated, and used effectively together with existing SQL constructs.

Challenge 2. FMU Meta-data Challenge. *How to optimally take advantage of FMU meta-data to semi-automate task specification and data mapping?* Traditionally, the specification of FMU tasks and mapping between (discrete) data stored in a database and the FMU variables is manual and very verbose, but there is a huge opportunity to take advantage of the meta-data stored as a part of a FMU to automate such specifications.

Challenge 3. FMU Performance Challenge. *How to increase system performance when a (generic) FMU needs to be instantiated and used many times?* Individual tasks within FMU workflows (e.g., Figure 1) often require the same FMU to be instantiated and simulated many times. When model parameters and inputs change only marginally, FMU computations/results of FMU simulation can be reused to increase overall performance. The next sections elaborate on how pgFMU addresses these challenges.

5 MODEL SPECIFICATION, STORAGE, AND MANIPULATION

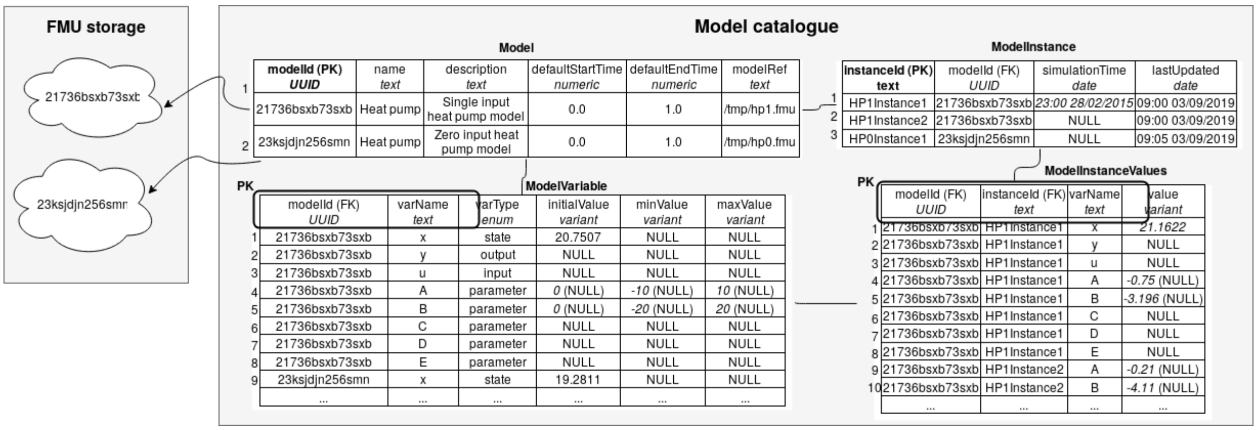
pgFMU aims to facilitate arbitrary user-specified FMU-based workflows by enabling creation, storage, and manipulation of FMU models (e.g., in terms of model parameters to be estimated) in a DBMS through user-specified SQL queries. This leads to Challenge 1: Language Integration Challenge (Section 4). Here, pgFMU takes a "session-like" approach where FMU instances are managed and used by explicitly calling commands in a particular order for FMU creation, deletion, etc. For this, pgFMU employs the same approach as MADlib [27] and offers a set of easy-to-use User-Defined Functions (UDFs) that support the full range of model management operations. pgFMU packs and offers these UDFs as a PostgreSQL extension, for ease-of-use and installation. Thus, FMU-specific functionality of pgFMU can be used with, e.g., the functionality of MADlib, for combined machine learning and physical simulations, as shown later. In this section, we describe in detail the new DBMS constructs / UDFs for model specification and storage, explain how they work, and show how users can take advantage of them when working with FMU models.

In pgFMU, a user can create an instance of an FMU model by using the function (UDF) *fmu_create (modelRef, [instanceId])* $\mapsto$ *instanceId*. As input, the function takes a mandatory textual argument *modelRef*, that represents either a path to a pre-compiled FMU (.fmu) file, a Modelica (.mo) file, or inline Modelica model code. If *modelRef* is a Modelica argument, it will be automatically compiled into an FMU file. The function returns a textual model instance identifier *instanceId*, which uniquely identifies the model instance in the subsequent model management function calls. The model instance identifier can be set manually by the user by supplying *instanceId* as an optional argument.

As explained in Section 4, FMU meta-data can be used to semi-automate task specification and data mapping (Challenge 2). For this, pgFMU reads meta-data from the user-defined FMUs once during FMU load and stores it in a *model catalogue* in a database. Thus, pgFMU automatically detects simulation parameters (start/stop timestamp, time step), variable causalities (e.g., *input*, *output*, *parameter*) and data types (e.g., *float*, *integer*, *string*), automatically configures simulation, and performs implicit data conversions when needed. The user can manually override the detected parameters. As seen in Figure 4, the model catalogue consists of four basic tables: *Model*, *ModelVariable*, *ModelInstance*, and *ModelInstanceValues*. *Model* captures the essential model attributes. FMU models are identified with a Universally Unique Identifier (UUID) [25] - a 128-bit string for unique object identification. All loaded models are stored in the *FMU storage* (non-volatile memory). *ModelVariable* describes the model variables: the variable name, variable type, and the initial, minimum, and maximum values. The combination of *ModelId* and *varName* attributes serves as the primary key. *initialValue*, *minValue*, and *maxValue* have the *variant* [24] type - a specialized PostgreSQL data type that allows storing any data type in a column, while keeping track of the original data type. *ModelInstance* stores the information about model instances. It uses *instanceId* as the primary key, and *modelId* as a foreign key. In this way, multiple instances of the same parent model can be stored. *ModelInstanceValues* stores the model instance variable values. Here, the combination of *modelId, instanceId*, and *varName* is a primary key. Lastly, we show some fields in all four tables decoded in regular and italic font. Here, the italic indicates the changes occurred after executing a query example (e.g., *fmu_parest*, Section 6). The initial field value is decoded with regular font.

A user can create an instance of the heat pump model (see Section 2) using the following intuitive SQL query in pgFMU:

```sql
SELECT fmu_create('/tmp/hp1.fmu', 'HP1Instance1');
```



Model

	modelId (PK) *UUID*	name *text*	description *text*	defaultStartTime *numeric*	defaultEndTime *numeric*	modelRef *text*
1	21736bsxb73sxb	Heat pump	Single input heat pump model	0.0	1.0	/tmp/hp1.fmu
2	23ksjdjn256smn	Heat pump	Zero input heat pump model	0.0	1.0	/tmp/hp0.fmu

ModelInstance

	instanceId (PK) **text**	modelId (FK) *UUID*	simulationTime *date*	lastUpdated *date*
1	HP1Instance1	21736bsxb73sxb	*23:00 28/02/2015*	09:00 03/09/2019
2	HP1Instance2	21736bsxb73sxb	NULL	09:00 03/09/2019
3	HP0Instance1	23ksjdjn256smn	NULL	09:05 03/09/2019

ModelVariable

	modelId (FK) *UUID*	varName *text*	varType *enum*	initialValue *variant*	minValue *variant*	maxValue *variant*
1	21736bsxb73sxb	x	state	20.7507	NULL	NULL
2	21736bsxb73sxb	y	output	NULL	NULL	NULL
3	21736bsxb73sxb	u	input	NULL	NULL	NULL
4	21736bsxb73sxb	A	parameter	*0* (NULL)	*-10* (NULL)	*10* (NULL)
5	21736bsxb73sxb	B	parameter	*0* (NULL)	*-20* (NULL)	*20* (NULL)
6	21736bsxb73sxb	C	parameter	NULL	NULL	NULL
7	21736bsxb73sxb	D	parameter	NULL	NULL	NULL
8	21736bsxb73sxb	E	parameter	NULL	NULL	NULL
9	23ksjdjn256smn	x	state	19.2811	NULL	NULL
	...	...	...	...	...	...

ModelInstanceValues

	modelId (FK) *UUID*	instanceId (FK) *text*	varName *text*	value *variant*
1	21736bsxb73sxb	HP1Instance1	x	*21.1622*
2	21736bsxb73sxb	HP1Instance1	y	NULL
3	21736bsxb73sxb	HP1Instance1	u	NULL
4	21736bsxb73sxb	HP1Instance1	A	*-0.75* (NULL)
5	21736bsxb73sxb	HP1Instance1	B	*-3.196* (NULL)
6	21736bsxb73sxb	HP1Instance1	C	NULL
7	21736bsxb73sxb	HP1Instance1	D	NULL
8	21736bsxb73sxb	HP1Instance1	E	NULL
9	21736bsxb73sxb	HP1Instance2	A	*-0.21* (NULL)
10	21736bsxb73sxb	HP1Instance2	B	*-4.11* (NULL)
	...	...	...	...

Figure 4: pgFMU model catalogue (filled with example data values)

The result of query execution is shown in Row 1 in *Model*, Row 1 in *ModelInstance*, Rows 1-8 in *ModelVariable* and *ModelInstance-Values*, and the object *21736bsxb73sxb* in *FMU storage*. Similarly, the user can use *fmu_create* to interact with Modelica model files:

```sql
SELECT fmu_create ('HP0Instance1', '/tmp/model.mo');
SELECT fmu_create('HP0Instance1', 'model heatpump
    output Real x, ..., y = C*x + D*u;end heatpump;');
```

The result of executing either of these calls corresponds to Row 2 in *Model*, Row 3 in *ModelInstance*, Row 9 in *ModelVariable*, and the object *23ksjdjn256smn* in *FMU storage*. The user can also create a copy of the model instance by using *fmu_copy (instanceId, [InstanceId2])* ↦ *instanceId2* (e.g., when managing many heat pumps of the same type); it takes the initial instance *instanceId* as input, and outputs the copy of this instance with the new instance identifier *InstanceId2* (user-defined or pgFMU-generated). The code snippet below illustrates the usage of *fmu_copy* (leading to changes in Row 2 in *ModelInstance* and Rows 9, 10 in *ModelInstanceValues*):

```sql
SELECT fmu_copy ( 'HP1Instance1', 'HP1Instance2');
```

In pgFMU, the initial copy of the FMU file is reused when either creating a new instance of the same FMU model (Row 2 in *ModelInstance*), copying a model instance, or changing a model state in the database environment. Once loaded (see Figure 4, *FMU storage*), an FMU model is used for all further operations; in this way, we avoid the creation and load of superfluous FMU model files, and we control and manipulate model instances while minimizing memory and computational resources. Algorithm 1 presents the logic behind *fmu_create*.

Furthermore, pgFMU provides a number of utility functions to analyse and manipulate the model instance variable values. For example, the function *fmu_variables (instanceId)* ↦ *(instanceId, varName, varType, initialValue, minValue, maxValue)* returns the details of all variables and parameters of the supplied model instance *instanceId*. The result includes the variables' initial, minimum, and maximum values, which can also be retrieved using the function *fmu_get(instanceId, varName)* ↦ *(initialValue, minValue, maxValue)*. In addition, a user can set the initial, maximum and minimum values of *HP1Instance1 A* and *B* model instance parameters using *fmu_set_initial(instanceId, varName, initialValue)* ↦ *instanceId*, *fmu_set_minimum (instanceId, varName, minValue)* ↦ *instanceId*, and *fmu_set_maximum (instanceId, varName, maxValue)* ↦ *instanceId* (Row 4 in *ModelVariable*, italic font):

```sql
SELECT fmu_set_initial('HP1Instance1', 'A', 0);
SELECT fmu_set_minimum('HP1Instance1', 'A', -10);
SELECT fmu_set_maximum('HP1Instance1', 'A', 10);
```

Furthermore, a user can retrieve all "HP1Instance1" model variables that serve, for example, as model parameters using the following query (the query output is shown in Table 3):

```sql
SELECT * FROM fmu_variables('HP1Instance1') AS f WHERE
f.varType = 'parameter'
```

The model instance can be brought back to its initial state using *fmu_reset (instanceId)* ↦ *instanceId* (see Rows 4, 5 in *ModelVariable*, and Rows 4, 5, 9, 10; the regular font indicates the initial values for the specific model instance). The user can delete either a specific model instance using *fmu_delete_instance (instanceId)*, or an entire FMU model using *fmu_delete_model (modelId)*. In the

Algorithm 1: fmu_create

```
Input:
    Stored model representation: modelRef;
    Optional: Unique model instance identifier [instanceId];
Output:
    Model instance identifier instanceId;
 1: if modelRef = .fmu file then
 2:     fmuModel ← Construct a model object using PyFMI
        function load_fmu(modelRef);
 3: else
 4:     if modelRef = .mo file or inline Modelica model
        specification then
 5:         fmuModel ← Construct a model object using PyFMI
            function compile_fmu(modelRef);
 6:     end if
 7: end if
 8: if instanceId is not given then
 9:     instanceId ← pgFMU-generated instanceId;
10: end if
11: Store the FMU model file in FMU storage;
12: Retrieve model variable names varName, types varType and
    values value by means of PyFMI function
    fmuModel.get_model_variables;
13: Insert the related sets of records into Model, ModelVariable,
    ModelInstances, ModelInstancesValues;
14: Return instanceId;
```

Table 3: *fmu_variables* example query output

instanceId	varName	varType	initial-Value	min-Value	max-Value
HP1Instance1	A	parameter	0	-10	10
HP1Instance1	B	parameter	0	-20	20
...	...	...	...	...	...

latter case, all model instances associated with this FMU model will be automatically removed from the database.

6 MODEL PARAMETER ESTIMATION

Model parameter estimation is a key functionality in pgFMU. Model instance parameters in pgFMU can be estimated using *fmu_parest (instanceIds, input_sqls, [pars])* $\mapsto$ *estimationErrors*. It takes a list of model instances *instanceIds* as input, updates the model instances with updated parameter values, and returns the list of estimation errors for each model instance *estimationErrors* (Root Mean Square Errors (RMSEs) by default). The user must also specify a list of SQL queries (*input_sqls*, one SQL query for each model instance) that produce the data to use in the parameter estimation, i.e., model training input and output variable pairs at different time instances. By default, the function estimates all model parameters. Optionally, the user can override this list by supplying a list of specific parameter names *pars*.

For example, the user can estimate the model parameters "A" and "B", and then store the updated model instance in the model catalogue using the following query (output is shown in Rows 4 and 5 in *ModelInstanceValues*, italic font):

```sql
SELECT fmu_parest('{HP1Instance1}', '{SELECT * FROM
    measurements}', '{A, B}')
```

fmu_parest() adopts the functionality of the ModestPy [3] Python package, which uses multiple optimization runs of the Global (denoted as G) and the Local (denoted as LaG) Search algorithms on different subsets of inputs to ensure result optimality even in the case of non-convex problems (later Figure 5 will illustrate the intuition behind this). In pgFMU, we use the ModestPy genetic algorithm implementation as G, and gradient-based method implemented by scikit-learn as LaG.

fmu_parest is designed to reduce the required amount of computation when parameters of multiple FMU instances need to be estimated (Challenge 3). Therefore, it automatically detects the number of model instances supplied and uses different algorithms when estimating parameters of a single model instance (SI scenario) or multiple instances (MI scenario).

Single instance parameter estimation Algorithm 2 provides implementation details for *fmu_parest* within the SI scenario. *fmu_parest* not only estimates the parameters of the FMU model instances, but also validates and updates the model instance with the new parameter values.

Multi-instance optimization In many scenarios, a user has a number of instances of the same model, e.g., 20 instances (*HP1Instance1*, *HP1Instance2*,.., *HP1Instance20*) of the heat pump model *HP1* corresponding to 20 different houses located in the same neighbourhood. In this case, pgFMU can apply its MI optimization when estimating parameters for multiple instances.

For example, the user can estimate the parameters "A" and "B" of *HP1Instance1* and *HP1Instance2* using the following query (leading to Rows 4, 5, 9, and 10 in *ModelInstanceValues*, italic font):

Algorithm 2: fmu_parest_SI

```
Input:
    Unique model instance identifier: instanceId;
    Query to retrieve measured data: input_sql;
    Optional: List of parameters [pars];
Output:
    estimationError;
1: Result set measurements ← Execute input_sql;
2: uuid ← Retrieve FMU model UUID from ModelInstance
   table identified by instanceId;
3: if pars is not given then
4:     pars ← Retrieve parameter variables from ModelVariable
       table identified by uuid;
5: end if
6: Retrieve the input variable values from the result set
   measurements;
7: parsEstimated, estimationError ← Run G & LaG for
   pars;
8: Update ModelInstanceValues with parsEstimated;
9: Return estimationError;
```

```sql
SELECT fmu_parest('{HP1Instance1, HP1Instance2}', '{
    SELECT * FROM measurements, SELECT * FROM
    measurements2}', '{A, B}')
```

Figure 5 illustrates the logic behind the MI optimization of *fmu_parest*. This optimization occurs in two stages. During the first stage, we estimate the parameters of *HP1Instance1* (solid blue line). In most cases, only the lower and upper bounds of each model parameter are known, and such bounds are usually defined by real-world physical constraints (e.g., HP performance coefficient cannot be negative). We set the initial parameter values to random numbers between the lower and the upper bounds mentioned above. Then, we follow the steps described in Algorithm 2, i.e., firstly, we run Global Search (G) to reduce the search space. Here, *1_G*, *2_G*, and *3_G* (filled circles) indicate the G iterations for *HP1Instance1*. After finding a good place in the search space, Local Search after Global (LaG) finetunes the parameter values to find optimal ones. We denote LaG iterations as *1_LaG*, *2_LaG*, and *3_LaG* (empty circles). *3_LaG* (exemplified by Rows 4, 5 in *ModelInstanceValues*) is the optimal parameter values of *HP1Instance1*.

The next stage is to perform parameter estimation for *HP1Instance2* (red dashed line). First, we check that the model instances belong to the same parent FMU. Next, we check whether the condition for the MI optimization invocation holds, i.e., we only

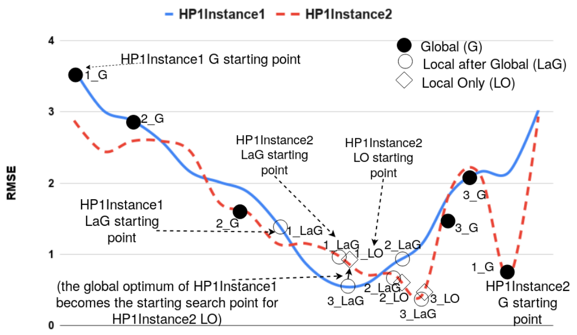


Figure 5: *fmu_parest* MI optimization

invoke the MI optimization after ensuring similarity (by calculating the L2 norm) between the input (and output) measurements of *HP1Instance1* and *HP1Instance2*. L2 norm (or the Euclidean norm) is one of the simplest and widely known, but still accurate and robust metrics for measuring the similarity between time series [26]. It is important to check the similarity between model instances time series to make sure the optimal solutions for these model instances lie within the same neighbourhood (see Figure 5). In case the difference between measurement time series is greater than a threshold, we do not invoke the MI optimization, and instead, run Algorithm 2 (a combination of G+LaG) for every instance. The *1_G*, *2_G*, and *3_G* points (filled circles) indicate the G iterations, while *1_LaG*, *2_LaG*, and *3_LaG* (empty circles) illustrate the LaG iterations. *3_LaG* would, in this case, be the optimal parameter values of *HP1Instance2*.

If the measured time series are sufficiently similar, i.e., the difference between them is less than *threshold*, we invoke the MI optimization. This means the optimal parameters values of *HP1Instance1* (solid blue line, *3_LaG*) become the initial parameter values of *HP1Instance2* (red dashed line, *1_LO*). In this way, we run Local Only Search (LO) (which is essentially the same algorithm as LaG, but with different initial parameter values) to obtain optimum parameter values of *HP1Instance2*, as its solution lies within the neighbourhood of the best solution of *HP1Instance1*. The points *1_LO*, *2_LO*, and *3_LO* (diamonds) are converging to those found by LaG. *3_LO* (exemplified by Rows 9, 10 in *ModelInstanceValues*) is the optimum parameter values of *HP1Instance2*. Algorithm 3 describes MI optimization for n model instances.

In pgFMU, the MI optimization significantly speeds up parameter estimation for multiple model instances. This speed-up is possible because G is much more expensive than LaG (see Section 8), and instead of G+LaG only LO is run. pgFMU provides the gradient-based Local Search algorithm with the good initial parameter values. In [3] authors emphasize, that "the initial global search would not be needed if the approximate initial values of parameters were known. In such a case the gradient-based methods would easily outperform GA [genetic algorithm]". As it will be shown later in Section 8, this statement holds. It should also be noted that the quality of the solution is dependent on the internal ModestPy algorithms. Within pgFMU, we adapt and enhance the functionality of ModestPy to best capture user preferences, streamline parameter estimation, and facilitate user interaction. The empirical evaluation of the MI parameter estimation shows identical accuracy with and without MI optimization.

7 MODEL SIMULATION

Users can simulate models by utilizing the function *fmu_simulate (instanceId, [input_sql], [time_from], [time_to])* $\mapsto$ *(simulationTime, instanceId, varName, values)*, which performs simulation on the supplied model instance and returns simulation results as a table of a timestamp, a model instance identifier, a variable name, and the variable's simulated value. By default, the simulation results for all state and output variables are returned. It is also possible to supply a time series of model input variable values by specifying an SQL query *input_sql*. If desired, the user can also specify a time window for simulation using the *time_from* and *time_to* parameters; otherwise, the start and end time will be determined by *defaultStartTime* and *defaultEndTime*. The system raises an error, e.g., if insufficient model input time series or an incomplete simulation time interval is provided.

Algorithm 3: fmu_parest_MI

```
Input:
    List of unique model instance identifiers: instanceIds;
    List of queries to retrieve measured data: input_sqls;
    Optional: List of parameters [pars], float [threshold].
Output:
    List of estimationErrors;
 1: if pars is not given then
 2:     pars ← Retrieve parameter variables from ModelVariable
        table identified by InstanceIds;
 3: end if
 4: for i = 0 to length (instanceIds)-1 do
 5:     Result set measurements[i] ← Execute input_sql[i];
 6:     parsEstimated[0], estimationError[0] ← Run
        fmu_parest_SI for instanceIds[0];
 7:     for i = 1 to length (instanceIds)-1 do
 8:         if modelId[0] ! = modelId[i] then
 9:             Run fmu_parest_SI for instanceIds[i];
10:         else
11:             $\delta$ = L2 norm of measurements[i] from
                measurements[0];
12:             if $\delta \geq$ threshold then
13:                 Run fmu_parest_SI for instanceIds[i];
14:             else
15:                 Update all the initial parameter values of
                    instanceIds[i] to parsEstimated[0];
16:                 parsEstimated[i], estimationError[i] ←
                    Run LO for instanceIds[i];
17:             end if
18:         end if
19:     end for
20:     Update ModelInstanceValues with parsEstimated;
21: end for
22: Return estimationErrors;
```

Table 4: *fmu_simulate* example query output

simulationTime	instanceId	varName	value
08:00 28/02/2015	HP1Instance1	x	20.7507
08:30 28/02/2015	HP1Instance1	y	0.0041
...	...	...	...

For example, the user can simulate the model by supplying model input values from the table *measurements* (Table 6) using the following query (also updating *ModelInstanceValues*, Row 1 and *ModelInstance*, Row 1, values in italic):

```sql
SELECT simulationTime, instanceId, varName, value
FROM fmu_simulate('HP1Instance1','SELECT * FROM
    measurements') WHERE varName IN ('y', 'x');
```

Table 4 shows the output of this query. If needed, the user can filter values of the desired columns using the standard WHERE clause predicates. We note that returning variable values as separate columns is not possible due to PostgreSQL limitations. In PostgreSQL, UDFs require fixed output schemas; flexible schemas are not supported in a convenient and easy-to-use way.

If multi-instance simulations are needed, the user can call a standard LATERAL join:

```sql
SELECT * FROM generate_series(1, 100) AS id,
LATERAL fmu_simulate('HP1Instance' || id::text ,
'SELECT * FROM measurements') AS f
```

Using *fmu_simulate*, model simulation is performed in two stages. In the first stage, an input object required for model simulation is created according to the FMU meta-data. This object consists of a set of input time series, automatically transformed for each input variable by taking into account their data types and variabilities (Challenge 2). In the second stage, an underlying model instance is simulated within the time interval (user-specified or from the model catalogue) while feeding the simulation algorithm with an input object. The simulation results are then emitted as a table (Table 4). Algorithm 4 depicts the logic behind *fmu_simulate*.

The UDFs described in Sections 5, 6, and 7 can be combined to solve specific problems. For instance, the example regarding HP temperature prediction can be specified using a single query (producing the same results and updates as *fmu_create, fmu_parest*, and *fmu_simulate* example queries from Sections 5, 6, and 7):

```
SELECT time, value
FROM fmu_simulate (fmu_parest {fmu_create
 ('HP1Instance1', 'C:\temp\hp1.fmu')},
'{SELECT * FROM measurements}', '{A, B}'),
'SELECT time, 'u' AS varName, value
 FROM generate_series('''2015-01-01'''::timestamp,
'''2015-01-02'''::timestamp, '''1 hour'''::
interval) AS time WHERE varName = 'x'
```

In addition, there are few important points. Firstly, all pgFMU UDFs are independent of each other and can be used in any order. Similar to other modelling software, e.g., Matlab and JModelica,

```
Algorithm 4: fmu_simulate
Input:
    Unique model instance identifier: instanceId;
    Optional: Query to retrieve measured data: [input_sql],
    [time_from], [time_to];
Output:
    Output table (simulationTime, instanceId, varName, values);
 1: uuid ← Retrieve FMU model UUID from ModelInstance
    table identified by instanceId;
 2: fmuModel ← Load FMU model identified by uuid from FMU
    storage;
 3: Result set measurements ← Execute input_sql;
 4: inputs ← Retrieve input variables from ModelVariable
    identified by uuid;
 5: input_object ← Empty hash map of (name, time series)
    pairs;
 6: For each name n in inputs: Insert (n, measurements[n])
    into input_object;
 7: if time_from and time_to are not given then
 8:     (time_from, time_to) ← (Model.DefaultStartTime,
        Model.DefaultEndTime) where Model.modelid = uuid;
 9: end if
10: result ← fmuModel.simulate(input_object,
    time_from, time_to);
11: time ← (time_from, time_to)
12: output ← [];
13: for time i in result.time do
14:     for varName in result.variables do
15:         Append (i, instanceId, varName, result[varName][i]) to
            output;
16:     end for
17: end for
18: Return output;
```

the user is only requested to create the in-DBMS instance of an FMU model. pgFMU provides full flexibility when configuring user workflows: if parameter estimation is not required, the user can simulate the model right away, or change the order of actions and perform model simulation followed by parameter estimation. Next, in pgFMU, we eliminate explicit I/O operations (e.g. historical measurements import, or simulation results export from a third-party software into DBMS) as all the computations are done "in-place" inside DBMS. This affects the essential operations (parameter estimation, model simulation) due to the data-driven nature of these operations. Furthermore, when operating many instances of the same model, we eliminate the necessity to load the same FMU file multiple times. We store one single FMU model, and operate with in-DBMS model instances. These instances share the essential information about the initial FMU through model catalogue, and can be considered as sufficient substitutes of FMUs. pgFMU is also able to operate several in-memory FMU model files at a time. Lastly, as we use prepared SQL queries, we avoid the repeated reevaluation of database queries for measurements retrieval.

8 EXPERIMENTAL EVALUATION

In this section, we evaluate pgFMU in real-world circumstances. Firstly, we describe our experimental setup. Then, we present the results of the evaluation with regard to model quality, performance, and usability.

8.1 Experimental Setup

As a baseline, we follow a traditional workflow for model storage, calibration, simulation, and validation, and perform the steps described in Figure 1. We consider two scenarios: *single instance (SI) scenario* and *multi-instance (MI) scenario*. In the SI scenario, we execute parameter estimation and simulation using a single model instance only. In the MI scenario, we execute parameter estimation and simulation for many model instances.

Three main system configurations are compared: (1) workflow execution within a Python IDE (referred as *Python*), (2) workflow execution using the non-optimized version of pgFMU (referred as *pgFMU-*), and (3) workflow execution using the optimized version of pgFMU (referred as *pgFMU+*). *Python* is based on the usage of standard Python packages. In *pgFMU-* configuration we use the pgFMU system with no multi-instance optimization activated, and *pgFMU+* takes advantage of the MI optimization. All three configurations are evaluated using Ubuntu 17.1 OS on a Lenovo ThinkPad with a four-core Intel i7 processor, and 8 GB of DDR3 main memory. For parameter estimation within pgFMU, we utilize genetic algorithm (GA) for Global Search and sequential quadratic programming (SQP) for Local Search. The study argues [3] that this combination of algorithms produces optimal results in terms of accuracy and performance. For GA, we use default settings with a fixed randomly derived seed. For SQP, the default settings were used as well. More information about default parameters settings is available on the library homepage.

Generalizability was one of the main criteria for choosing the test models. The models should represent real-world physical systems with varying numbers of inputs and outputs, and different physical meaning of model parameters. We follow a workflow process identical to the one outlined in Figure 1, i.e., we estimate the unknown parameters of the model (model parameter estimation), validate the model with regard to the real input measurements (model validation), and simulate the model to predict the values of the model state variable (model simulation). We have chosen three different FMU models, each of them representing a real-world physical system. These models are denoted *HP0, HP1*, and *Classroom*, respectively. *HP1* corresponds to the running example model described in Section 2. *HP0* is a modification of *HP1* with zero inputs, such that we keep the heat pump power at a constant rate. *Classroom* is a thermal network model [17] represented by a classroom in a 8500 m^2 university building at the SDU Campus Odense (Odense, DK). Table 5 summarizes the inputs, outputs and parameters of the three models. For experimental evaluation within the MI scenario, we construct 100 synthetic datasets for each FMU model. We multiply the original dataset time series values with a constant $delta$ from the numerical range $\delta \in \{0.8, \ldots, 1.2\}$, meaning we amplify or decrease the numerical values by up to 20% while ensuring the same data distribution as the original datasets. In Section 6 we explain the reasoning behind such numerical range of δ. We also ensure that the datasets respect the physical constraints of the real-world systems.

8.2 Model quality

In this subsection, we compare and evaluate model quality for *Python*, *pgFMU-*, and *pgFMU+*. We perform parameter estimation and model simulation within the *SI scenario* and *MI scenario*.

SI scenario. Within *Python*, parameter estimation is performed using the ModestPy package, whereas *pgFMU-* and *pgFMU+* utilize the *fmu_parest* UDF. *HP1* was calibrated using the NIST dataset [18]. *HP0* was calibrated using the same dataset with u being kept at a constant rate of 1.38%. For both models, we estimate the parameters based on the hourly aggregated data from February 1-21, while using February 22-28 for validation. The *Classroom* model was calibrated using measurements data from University building O44 in Odense, DK. Table 6 shows an excerpt of the datasets for all three models. We compare parameter values and error value for *HP0, HP1* and *Classroom* for *Python*, *pgFMU-*, and *pgFMU+*. Table 7 shows this comparison. To evaluate the quality of the model, we use the RMSE metric. RMSE and Mean Absolute Error (MAE) are two commonly used metrics for model evaluation. However, a study [19] argues that RMSE is more appropriate to be used when unfavourable conditions should be given a higher weight, i.e., RMSE penalizes large errors stricter than MAE. In our case, we want to distinguish every occurrence when a model fails to secure a good fit with the measured data, therefore, we choose RMSE for model quality evaluation.

For *HP0* (Table 7), the model parameter values have converged to the same values within *Python*, *pgFMU-* and *pgFMU+*. The RMSEs when performing parameter estimation for *Python*, *pgFMU-*, and *pgFMU+* are near identical (the relative difference is only 0.013%). For *HP1*, the RMSEs in all three configurations are exactly the same, and for *Classroom* the relative difference is at most 0.018%. We consider these differences negligible. Thus, *pgFMU-* and *pgFMU+* handles single model workflow computations with the same accuracy as *Python*. The identical accuracy for all three configurations is achieved through the usage of the same Python ModestPy library; however, both *pgFMU-* and *pgFMU+* use a modified version of ModestPy to provide generalizability and handle all types of FMU models, and perform data binding and pre-processing steps discussed in Section 2.

MI scenario. For the MI scenario, we compare RMSE values for 100 instances of each model (*HP0, HP1* and *Classroom*) side-by-side. Each model instance is supplied with a synthetic dataset based on the measured data. For *Classroom*, the RMSE values for all three configurations are matching, resulting in the same average RMSE values (1.61°*C*). For *HP1*, RMSE values differ a bit more, but are still very close, with either *Python*, *pgFMU-*, or *pgFMU+* as the better one. The average RMSE values yield in 2.03°*C* for all three configurations. Thus, the model quality is also the same here. We observe a similar behaviour for *HP0*, where for *pgFMU-* and *Python* the average RMSE is 0.68°*C*, while *pgFMU+* estimates the model parameters with 0.66°*C* average accuracy.

When enabling MI optimization, one must remember the conditions for this feature to produce acceptable results. By default, parameter estimation is performed using Algorithm 2 (Section 6), unless the user alters the threshold value. In our case, we have set the threshold to 20% (Section 8.1) based on the series of experiments reflected in Figure 6. In this Figure, the x-axis represents dataset dissimilarity in terms of L2 norm distances, the y-axis encodes the corresponding RMSE values, and the secondary y-axis shows the execution time of G+LaG and LO. RMSE is represented using the same unit as the dependent variable (for *HP1*, the indoor temperature in °*C*). The Figure shows that the execution time of G+LaG is significantly larger than LO. We see that G takes approximately 90% of the execution time (considering that LaG and LO are the same algorithms, with different initial parameter values). The Figure shows that there is no difference in G+LaG and LO RMSEs until maximum dissimilarity reached approximately 30%; after this, the difference grows linearly. This is because the optimal solution for *instanceId[0]* used as initial parameter values for the remaining models *instanceId[1], ..., instanceId[n-1]* (see Algorithm 3) was not able to provide satisfactory results. Further, the choice of a threshold value is always contextually dependent, and a user has to decide about the acceptable RMSE values.

Combining pgFMU and MADlib. To improve model quality, pgFMU can be used in combination with other in-DBMS analytics tools, e.g., MADlib. If in our *Classroom* model the number of occupants in the room is unknown, we can use MADlib to predict occupancy, e.g., using the ARIMA model and the following query to train the model:

```sql
SELECT arima_train(
'occupants', -- Source table
'occupants_output', -- Output table
'time', -- Timestamp column
'value'); -- Timeseries column
```

In this experiment, we used the original dataset to train the model and perform the prediction. We divided the dataset into training (80%) and validation (20%) sets. We created two model instances: without occupancy information, and with occupancy values predicted by the MADlib ARIMA model. Then, we simulated the two models, and compared model RMSEs. The *Classroom* model

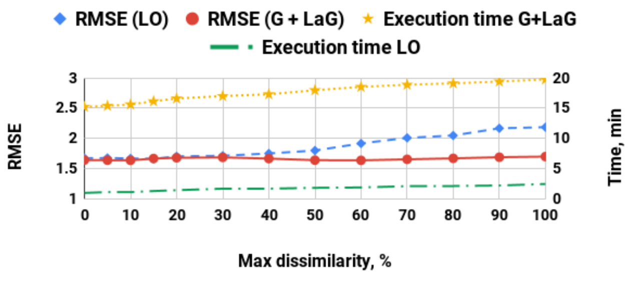


Figure 6: Avg. RMSE & execution time overhead of LO & G+LaG for datasets of different dissimilarity (HP1 model)

Table 5: FMU models

ModelID	Measurements dataset	Inputs	Outputs	Parameters
HP0	NIST Engineering Lab's Net-Zero Energy Residential Test Facility	No inputs	HP power consumption *y*, Indoor temperature *x* (state variable).	Thermal capacitance *Cp*, thermal resistance *R*.
HP1	NIST Engineering Lab's Net-Zero Energy Residential Test Facility	HP power rating setting in the range [0 .. 100%] *u*	HP power consumption *y*, Indoor temperature *x* (state variable).	Thermal capacitance *Cp*, thermal resistance *R*.
Classroom	Data from the classroom in the test facility in Odense, DK	Solar radiation *solrad*, outdoor temperature *tout*, number of occupants *occ*, damper position *dpos*, radiation valve position *vpos*.	Indoor temperature *t* (state variable)	Solar heat gain coeff. *shgc*, zone thermal mass factor *tmass*, ext. wall thermal resistance *RExt*, occupant heat generation effectiveness *occheff*.

Table 6: Dataset for *HP0, HP1* (top), *Classroom* (bottom)

No	Timestamp	x	y	u
1	2015/02/01 00:00	20.7507	0	0
2	2015/02/01 01:00	23.6231	0.1381	0.0177
...	...	...	...	...

No	Timestamp	T	solrad	Tout	occ	dpos	vpos
1	2018/04/04 08:00	21.5727	364.37	11	19.7	0	13.165
2	2018/04/04 08:30	20.8667	396.05	10.5	20.033	0	19.4
...	...	...	...	...	...	...	...

with the occupancy values predicted by MADlib ARIMA showed up to 21.1% increased accuracy in terms of RMSE.

Reversely, pgFMU can be used to improve the quality of traditional ML models. In the next experiment, we used the indoor temperatures of the *Classroom* computed using pgFMU to increase the accuracy of the logistic regression model that identifies the position (open/closed) of the ventilation damper *dpos*. When we include indoor temperature *t* into the feature vector of the model, this yields 5.9% increased model accuracy.

8.3 Performance evaluation

In this subsection, we look into the performance comparison of *Python*, *pgFMU-*, and *pgFMU+* for *HP0, HP1*, and *Classroom* within SI and MI scenarios.

Single instance scenario. To evaluate pgFMU performance within the SI scenario, we compare the execution time for all three models. Table 8 shows this comparison. For all the models, we do not observe a significant difference in execution time between *Python*, *pgFMU-* and *pgFMU+* (for *HP0*, *Python* is faster by 0.14%, for *HP1* and *Classroom pgFMU-* and *pgFMU+* perform better by 0.10%, and 0.12%, respectively). We conclude that within *Python*, *pgFMU-*, and *pgFMU+* the performance is practically identical, as expected. However, in Table 8 we see that model calibration takes more than 99% of execution time. This imposes limitations when estimating the parameters of multiple model instances.

Table 7: SI scenario, model calibration comparison

	Python		*pgFMU-, pgFMU+*	
	Param. values	RMSE	Param. values	RMSE
HP0	Cp: 1.53	0.7701	Cp: 1.53	0.7702
	R: 1.51		R: 1.51	
HP1	Cp: 1.49	0.5445	Cp: 1.49	0.5445
	R: 1.481		R: 1.481	
Classroom	RExt: 4	1.6445	RExt: 4	1.6442
	occheff: 1.478		occheff: 1.478	
	shgc: 3.246		shgc: 3.246	
	tmass: 50		tmass: 50	

MI scenario. For the MI scenario, we use *Python*, *pgFMU-* and *pgFMU+*, and scale the number of model instances to 100 for each FMU model. We match the synthetic dataset with each FMU model instance. Figure 7 shows the execution time for *Python*, *pgFMU-*, and *pgFMU+* for *HP0, HP1* and *Classroom.* The execution workflow of storing, calibrating, simulating, and validating 100 *HP0* model instances takes 1083.7 min (18.06 hours) for *Python*, 1073.7 min (17.9 hours) for *pgFMU-*, and 204.2 min (3.4 hours) for *pgFMU+*. As we can see, *pgFMU+* outperforms *Python* and *pgFMU-* by 5.31x. The execution workflow of storing, calibrating, simulating, and validating 100 *HP1* model instances takes 1329.63 min (22.16 hours) for *Python*, 1319.48 min (21.97 hours) for *pgFMU-*, and 241.47 min (4.02 hours) for *pgFMU+*. In this case, *pgFMU+* outperforms *Python* and *pgFMU-* by 5.51x. We observe an even larger difference in workflow execution time for 100 *Classroom* model instances. It takes 1380.68 min (23.01 hours) for *Python*, 1370.95 min (22.85 hours) for *pgFMU-*, and 163.6 min (2.73 hours) for *pgFMU+*; *pgFMU+* is faster by 8.43x.

As we can see, the execution time grows linearly with more model instances, but the growth rate is different for different models. The common pattern for all three models is that *Python* and *pgFMU-* exhibit a similar growth rate. For *pgFMU+*, the execution time also grows linearly, but slower compared to *Python* and *pgFMU-*. This means *pgFMU+* performs user-defined workflows based on the FMU model calibration and simulation on average 6.42x faster. Such gains in the runtime are achieved through a number of optimization steps described in Sections 6 and 7.

8.4 Usability

We conducted usability tests to evaluate how the functionality of pgFMU reflects user needs. We asked a group of 6 PhD-candidates and 24 master students from four different universities in Denmark, Poland, Spain, and Belgium to individually perform the task of *HP1* and *Classroom* model calibration and simulation. All the participants were asked to complete the SI workflow based on Figure 1 using *Python* and *pgFMU.* The execution of the MI workflow was optional. During the usability testing, we recorded the issues with both configurations, and observed the learning curve of the participants as they progressed with the task. All participants were timed.

Table 8: Configurations comparison, SI scenario

ID	Operation	execution time, s					
		HP0		HP1		Classroom	
		Python	pgFMU±	Python	pgFMU±	Python	pgFMU±
1	Load FMU	0.02	0.025	0.02	0.021	0.03	0.03
2	Read historical measurements & control inputs	0.02	0.021	0.03	0.031	0.04	0.041
3	(Re)calibrate the model	842.99	844.18	834.68	833.88	830.2	829.16
4	Validate and update FMU model	0.01	-	0.01	-	0.01	-
5	Simulate FMU model	0.16	0.214	0.2	0.22	0.35	0.44
6	Export predicted values to a DBMS	0.06	-	0.06	-	0.05	-
	Total	**843.26**	**844.44**	**835**	**834.15**	**830.68**	**829.67**

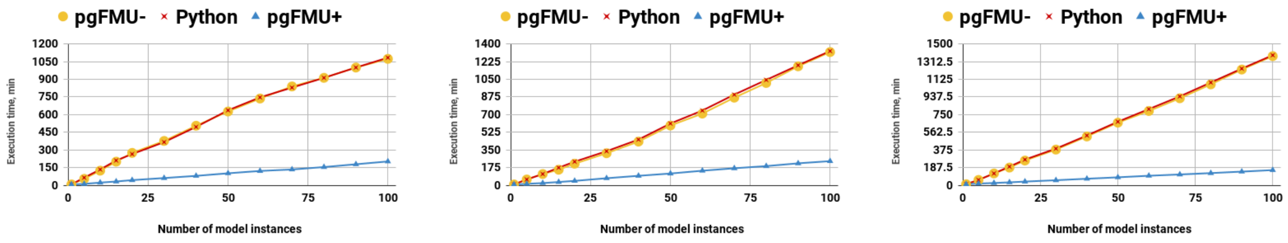


Figure 7: *HP0, HP1, Classroom* (from the left to the right) parameter estimation execution time

In the beginning, the users were asked to answer a pre-assessment questionnaire aimed at identifying their knowledge about physical systems modelling. By using the scale from 1(very little) to 5(very much), the participants answered the following questions:

Q1. How familiar are you with energy systems and physical system modelling?
Q2. How familiar are you with model simulation and calibration?
Q3. How familiar are you with model simulation software?
Q4. How comfortable are you with using Python IDE(s)?
Q5. How comfortable are you with using SQL?

Based on the participants' answers, they did not consider themselves experts in the energy domain (Q1). Only 2 people estimated the familiarity with energy systems as "very much". For Q1, Q2 and Q3, the majority (27 out of 30 for Q1 and Q2, 26 out of 30 for Q3) ranked their knowledge in energy systems, model simulation and calibration processes, and model simulation software as "very little" or "little". When speaking about programming environments, 25 out of 30 students knew "much" or "very much" about the SQL, with only 14 out of 30 giving the same score to the Python. We concluded that graduate and post-graduate students possessed more knowledge about SQL, and felt more comfortable with using SQL-based functions rather than Python packages.

For this session, we set the time limit to 3 hours. The participants tested the *pgFMU* functionality first, then the *Python.* All participants but one were able to finalize the task within the defined time range. Figure 8 illustrates the time distribution for every user performing the steps described in Figure 1 for *HP1* and *Classroom* models. With pgFMU it took under 20 minutes for all participants to become familiar with the syntax, and complete the task. The minimum learning time for pgFMU was reported to be 9.6 minutes, and the maximum was 17.6 minutes, respectively. On average, with pgFMU the participants finished the task (excluding the runtime) 11.74x faster than with *Python.*

The main criticism regarding the suggested setup was the necessity to use multiple Python packages, and the inability to

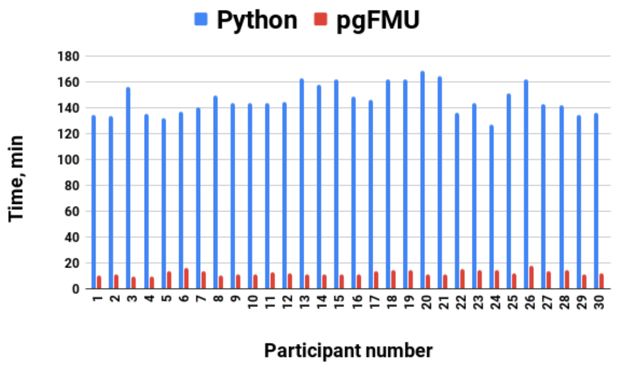


Figure 8: Users learning and development time (combined)

perform all the tasks with a single programming package/tool. The whole workflow was described as "intuitive" and "easy to understand", but the need to use different libraries was characterized as "confusing" and "unsettling". One of the participants reported that both PyFMI and ModestPy packages in their documentation use domain-specific language which was difficult to grasp. Table 9 shows highlighted feedback that the participants expressed about the strengths and weaknesses of both systems.

At the end, all users were asked to answer a post-assessment questionnaire aimed at identifying participants' opinion regarding the functionality of *Python* and *pgFMU.* By using the scale from 1(very unsatisfactory) to 5(very satisfactory), the participants were asked to answer the following questions :

Q1. How was it to retrieve information about model variables?
Q2. How was it to set model parameters?
Q3. How was it to calibrate the model?
Q4. How was it to simulate the model?
Q5. How was your satisfaction with systems' functionality?

Table 9: Strengths and weaknesses of *Python* and *pgFMU*

Strong points of *Python*	Strong points of *pgFMU*
“Better to debug, analyze"; “More functionality”; “Control over program flow”; “Data visualization option”;	"Data and model in one place"; "Easy to run and understand"; "Simplicity, no need to use or import external tools"; "Familiar SQL syntax";
Weak points of *Python*	Weak points of *pgFMU*
"A lot of unknown new modules and packages"; "No one ready package to do everything"; "A lot of code to set up configuration, some functions not intuitive"; "You need practise[*sic*] to understand";	"Not so much configuring available"; "I don’t see any significant [*weak points*] besides maybe installation of the package on postgres [*PostgreSQL*];" "Specific database implementation";

The questionnaire results show a clear advantage of using *pgFMU* over *Python*. When performing the workflow depicted in Figure 1, *pgFMU* scores better when retrieving (3.7 out of 5 for pgFMU vs 3.1 out of 5 for Python) or setting the model variables (3.83 out of 5 vs 3.26 out of 5 respectively), calibrating (3.66 out of 5 vs 3.1 out of 5 respectively) and simulating the model. In particular, the model simulation functionality of *pgFMU* was commended, scoring 4.17 out of 5 (for the same operation, *Python* scored only 3.53 out of 5). *pgFMU* outperforms *Python* in terms of overall participants’ satisfaction, scoring 4.2 and 3.6 out of 5, respectively.

Lastly, *pgFMU* requires substantially fewer lines of code than does *Python*; Table 1 shows this comparison. In addition, *pgFMU* does not require any customization, meaning it is capable of working with any number of model inputs, outputs and parameters without UDF adjustment for every specific model or use case. On the contrary, *Python* requires the user to manually retrieve and match model inputs with measurements data.

9 CONCLUSION AND FUTURE WORK

This paper presents pgFMU - the first DBMS extension to support storage, simulation, calibration, and validation of physical system models defined as Functional Mock-up Units (FMUs) within a single DBMS environment. This extension is developed for cyber-physical data scientists and cyber-physical software developers. For such users, pgFMU provides a set of powerful User-Defined Functions (UDFs) invoked by traditional SQL queries. The UDFs are designed as stand-alone functions within pgFMU, and can be used in a user-defined sequence. Furthermore, pgFMU provides efficient functionality to store, simulate an arbitrary number of FMU model instances, and estimate the parameters of such instances. The aforementioned properties all contribute to supporting FMU model simulation and parameter estimation tasks within a DBMS environment. Due to its in-DBMS implementation, pgFMU demonstrates increased performance (up to 8.43x, and on average 6.42x faster for multi-instance workflows) for data-dependent workflows and improves user productivity (11.74x times faster in terms of development time). Moreover, it can increase Machine Learning model accuracy by up to 21.1% when used in combination with existing in-DBMS analytics tools, e.g., MADlib.

pgFMU has been tested by the entry level cyber-physical data scientists. The usability testing showed positive results, reporting the development time more than an order of magnitude lower than the traditional approach. As reported by the participants, pgFMU was capable of addressing the major point of dissatisfaction, namely the variety of software packages and libraries the user is obliged to use, and the domain-specific nature of such packages. pgFMU simplifies the overall analytical procedure, and minimizes the efforts required to specify and calibrate a particular model; the users are able to perform the necessary operations with approximately 22x fewer code lines. pgFMU was also reported to provide a more intuitive way of scripting and a better data organisation.

Future work will continue the development of functionality to support in-DBMS FMU-based dynamic optimization. This includes the adoption of various model predictive control means, covering the optimization of control inputs. Additionally, we will look into FMU integration challenges in the Big Data setting, including just-in-time (JIT) FMU compilation to optimize user queries, and scheduling FMU execution on multi-core multi-node environments.

REFERENCES

[1] Ilic, M. D., Xie, L., Khan, U. A., & Moura, J. M. 2010. Modeling of future cyber–physical energy systems for distributed sensing and control. In *IEEE Trans. on SMC-Part A: Systems and Humans*.
[2] Functional Mock-up Interface. 2018. https://fmi-standard.org/
[3] Arendt, K., Jradi, M., Wetter, M., & Veje, C. T. 2018. ModestPy: An Open-Source Python Tool for Parameter Estimation in Functional Mock-up Units. In *American Modelica Conference*.
[4] MATLAB & Simulink. 2019. https://www.mathworks.com/
[5] Modelica Association. 2019. https://www.modelica.org/association/
[6] EnergyPlus. 2019. https://energyplus.net/
[7] Python.org. 2019. https://www.python.org/about/
[8] Åkesson, J., Gäfvert, M., & Tummescheit, H. 2009. Jmodelica—an open source platform for optimization of modelica models. In *MATHMOD*.
[9] Momjian, B. 2001. PostgreSQL: introduction and concepts.
[10] Fritzson, P., Aronsson, P., Pop, A., Lundvall, H., Nystrom, K., Saldamli, L., ... & Sandholm, A. 2006. OpenModelica-A free open-source environment for system modeling, simulation, and teaching. In *IEEE ISCSIC*.
[11] Bonvini, M., Wetter, M., & Sohn, M. D. 2014. An fmi-based framework for state and parameter estimation. In *10th International Modelica Conference*.
[12] Frazzetto, D., Nielsen, T. D., Pedersen, T. B., & Šikšnys, L. 2019. Prescriptive analytics: a survey of emerging trends and technologies. In *VLDBJ*.
[13] Fritzson, P., & Bunus, P. 2002. Modelica-a general object-oriented language for continuous and discrete-event system modeling and simulation. In *35th Annual Simulation Symposium*.
[14] Andersson, C., Åkesson, J., & Führer, C. 2016. PyFMI: A Python package for simulation of coupled dynamic models with the functional mock-up interface.
[15] McKinney, W. 2012. Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.
[16] PostgreSQL + Python. Psycopg. 2019.http://initd.org/psycopg/
[17] Arendt, K., Veje, C. T. 2019. MShoot: an Open Source Framework for Multiple Shooting MPC in Buildings. In *IBPSA*.
[18] Healy, W., Fanney, A. H., Dougherty, B., Payne, W. V., Ullah, T., Ng, L., and Omar, F. 2017. Net Zero Energy Residential Test Facility Instrumented Data; Year 1. https://doi.org/10.18434/T46W2X
[19] Chai, T., & Draxler, R. R. 2014. Root mean square error (RMSE) or mean absolute error (MAE)?–Arguments against avoiding RMSE in the literature. In *GMD*.
[20] Andersson, C., Führer, C., & Åkesson, J. 2015. Assimulo: A unified framework for ODE solvers. In *IMACS*.
[21] Webb, B. 2008. Sas in-database processing with teradata: an overview of foundation technology. https://support.sas.com/resources/papers/teradata08.pdf
[22] Meliou, A., & Suciu, D. 2012. Tiresias: the database oracle for how-to queries. In *SIGMOD*.
[23] Šikšnys, L., & Pedersen, T. B. 2016. Solvedb: Integrating optimization problem solvers into sql databases. In *SSDBM*.
[24] variant: Variant data type for PostgreSQL / PostgreSQL Extension Network. 2019. https://pgxn.org/dist/variant/0.7.0/doc/variant.html
[25] The Telecommunications Standardisation. 2019. https://www.ietf.org/rfc/rfc4122.txt
[26] Wang, X., Mueen, A., Ding, H., Trajcevski, G., Scheuermann, P., & Keogh, E. 2013. Experimental comparison of representation methods and distance measures for time series data. In *DMKDFD*.
[27] Hellerstein, J. M., Ré, C., Schoppmann, F., Wang, D. Z., Fratkin, E., ... & Kumar, A. 2012. The MADlib analytics library: or MAD skills, the SQL. In *VLDB*.
[28] SQL Server Machine Learning Services (Python and R). 2019. https://docs.microsoft.com/en-us/sql/advanced-analytics/

Differentially-Private Next-Location Prediction with Neural Networks

Ritesh Ahuja
University of Southern California
riteshah@usc.edu

Gabriel Ghinita
University of Massachusetts Boston
gabriel.ghinita@umb.edu

Cyrus Shahabi
University of Southern California
shahabi@usc.edu

ABSTRACT

The emergence of mobile apps (e.g., location-based services, geo-social networks, ride-sharing) led to the collection of vast amounts of trajectory data that greatly benefit the understanding of individual mobility. One problem of particular interest is next-location prediction, which facilitates location-based advertising, point-of-interest recommendation, traffic optimization, etc. However, using individual trajectories to build prediction models introduces serious privacy concerns, since exact whereabouts of users can disclose sensitive information such as their health status or lifestyle choices. Several research efforts focused on privacy-preserving next-location prediction, but they have serious limitations: some use outdated privacy models (e.g., k-anonymity), while others employ learning models with limited expressivity (e.g., matrix factorization). More recent approaches (e.g., DP-SGD) integrate the powerful differential privacy model with neural networks, but they provide only generic and difficult-to-tune methods that do not perform well on location data, which is inherently skewed and sparse.

We propose a technique that builds upon DP-SGD, but adapts it for the requirements of next-location prediction. We focus on user-level privacy, a strong privacy guarantee that protects users regardless of how much data they contribute. Central to our approach is the use of the skip-gram model, and its negative sampling technique. Our work is the first to propose differentially-private learning with skip-grams. In addition, we devise data grouping techniques within the skip-gram framework that pool together trajectories from multiple users in order to accelerate learning and improve model accuracy. Experiments conducted on real datasets demonstrate that our approach significantly boosts prediction accuracy compared to existing DP-SGD techniques.

1 INTRODUCTION

The last decade witnessed a rapid development in mobile devices capabilities, accompanied by the emergence of numerous locations-centric applications, such as point-of-interest (POI) search, geo-social networks, ride-sharing services, etc. As a result, vast amounts of rich trajectory data have become available. Coupled with recent advances in machine learning, these data can benefit numerous application domains, such as traffic analysis, location-based recommendations, homeland security, etc.

Mobile users share their coordinates with service providers (e.g., Google Maps) in exchange for receiving services customized to their location. The service providers analyze the data and create powerful machine learning models. Subsequently, these models can be *(i)* placed on user devices to improve the quality of location-centric services; *(ii)* shared with business affiliates interested in expanding their customer base; or *(iii)* offered in a Machine-Learning-as-a-Service (MLaaS) infrastructure to produce business-critical outcomes and actionable insights (e.g., traffic optimization). Figure 1 illustrates these cases. Given historical trajectories, several approaches exploit recent results in neural networks to produce state-of-the-art POI recommender systems [10, 35, 58]. Even though individual trajectory data are not disclosed directly, the model itself retains significant amounts of specific movement details, which in turn may leak sensitive information about an individual's health status, political orientation, entertainment preferences, etc. The problem is exacerbated by the use of neural networks, which have the tendency to overfit the data, leading to unintended memorization of rare sequences which act as quasi-identifiers of their owners [9, 13]. Hence, significant privacy risks arise if individual location data are used in the learning process without any protection.

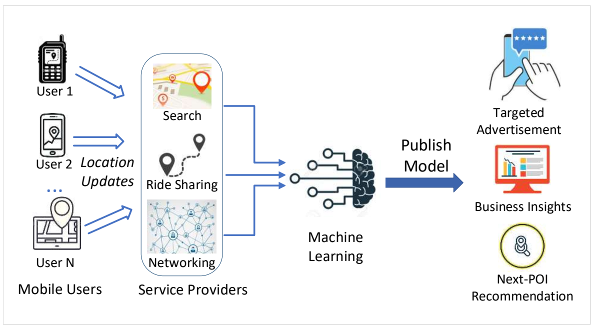


Figure 1: System Model

The research literature identified several fundamental privacy threats that arise when performing machine learning on large collections of individuals' data. One such attack is *membership inference* [25, 52] where an adversary who has access to the model and some information about a targeted individual, can learn whether the target's data was used to train the model. Another attack called *model inversion* [56] makes it possible to infer sensitive points in a trajectory (e.g., a user's favorite bar) from non-sensitive ones (e.g., a user's office). Within the MLaaS setting—where a third party is allowed to only query the model—this implies extracting the training data using only the model's predictions [20].

Iterative procedures such as stochastic gradient descent (SGD) [7] are often used in training deep learning models. Due to the repeated accesses to the data, they raise additional challenges when employing existing privacy techniques. In order to prevent the inference of private information from the training data, recent approaches rely on the powerful *differential privacy (DP)* model [14]. Sequential querying using differentially private mechanisms degrades the overall privacy level. The recent work in [2] provides a tight-bound analysis of the composition of the Gaussian Mechanism for differential privacy under iterative training procedures, enabling the utility of a deep learning model to remain high [39], while preventing the exposure of the training data [6, 27]. While integrating differential privacy techniques into training procedures like stochastic gradient descent is relatively straightforward, computing a tight bound of the privacy loss over multiple iterations is extremely challenging (see Section 6 for a summary of results).

The seminal work in [2] provided *record-level* privacy for a simple feed-forward neural network trained in a centralized manner. The approach provides protection only when each individual contributes a single data item (e.g., a single trajectory). When an individual may contribute multiple data items, a more strict protection level is required, called *user-level* privacy. McMahan et. al. [39] showed that one can achieve *user-level* privacy protection in a federated setting for simple learning tasks. However, ensuring good utility of the trained model for datasets with various characteristics remains a challenge. McMahan et. al. [39] remove skewness in their inputs by pruning each user's data to a threshold, thus discounting the problems of training neural models on inherently sparse location datasets, usually having density around 0.1% [60]. Existing work on privacy-preserving deep learning either assume large and dense datasets, or are evaluated only on dummy datasets [21] that are replicated to a desired size using techniques such as [38]. Such techniques overlook the difficulty of training models on smaller or sparse datasets, which often prevent models from converging [40]. Moreover, they require extensive hyperparameter tuning to achieve good accuracy, and the rough guidelines offered to tune these parameters [37] do not extend to more complex neural architectures, or to datasets different from those used in their work.

We propose a technique that can accurately perform learning on trajectory data. Specifically, we focus on next-location prediction, which is a fundamental and valuable task in location-centric applications. The central idea behind our approach is the use of the skip-gram model [41, 43]. One important property of skip-grams is that they handle well sparse data. At the same time, the use of skip-grams for trajectory data increases the dimensionality of intermediate layers in the neural network. This creates a difficult challenge in the context of privacy-preserving learning, because it increases data sensitivity, and requires a large amount of noise to be introduced, therefore decreasing accuracy.

To address this challenge, we capitalize on the *negative sampling (NS)* technique that can be used in conjunction with skip-grams. NS turns out to be extremely valuable in private gradient descent computation, because it helps reduce the gradient update norms, and thus boosts the ratio of the useful signal compared to the noise introduced by differential privacy. In addition, we introduce a data grouping mechanism that makes learning more effective by combining multiple users into a single bucket, and then training the model per bucket. Grouping has a dual effect: on the positive side, it increases the information diversity in each bucket, improving learning outcomes; on the negative side, it heightens the adverse effect of the introduced Gaussian noise. We study closely this trade-off, and investigate the effect of grouping factors in practice.

Our specific contributions are:

(1) We propose a private learning technique for sparse location data using skip-grams in conjunction with DP-SGD. To our knowledge, this is the first approach to combine skip-grams with DP to build a private ML model. Although our analysis and evaluation focus on location data, we believe that DP-compliant skip-grams can also benefit other scenarios that involve sparse data.
(2) We address the high-dimensionality challenge introduced by skip-grams through the careful use of *negative sampling*, which helps reduce the norm of gradient descent updates, and as a result preserves a good signal-to-noise ratio when perturbing gradients according to the Gaussian mechanism of DP. In addition, we group together data from multiple users into buckets, and run the ML process with each bucket as input. By increasing the diversity of the ML input, we are able to significantly boost learning accuracy.
(3) We perform an extensive experimental evaluation on real-world location check-in data. Our results demonstrate that training a differentially private skip-gram for next-location recommendation clearly outperforms existing approaches for DP-compliant learning. We also perform a thorough empirical exploration of the system parameters to understand in-depth the behavior of the proposed learning model. Our findings show that DP-compliant skip-grams are a powerful and robust approach for location data, and some of the trends that we uncovered can also extend to other types of sparse data, beyond locations.

The rest of the paper is organized as follows: we provide background information in Section 2. Section 3 introduces the system architecture, followed by the details of our private location recommendation technique in Section 4. We perform an extensive experimental evaluation in Section 5. We survey related work in Section 6, followed by conclusions in Section 7.

2 BACKGROUND

2.1 Differential Privacy

Differential Privacy (DP) [17] represents the de-facto standard in protecting individual data. It provides a rigorous mathematical framework with formal protection guarantees, and is the model of choice when releasing aggregate results derived from sensitive data. The type of analyses supported by DP range from simple count or sum queries, to the training of machine learning models. A popular DP flavor that is frequently used in gradient descent due to its refined composition theorems is (ε, δ)-differential privacy. Given non-negative numbers (ε, δ), a randomized algorithm $\mathcal{M}$ satisfies (ε, δ)-differential privacy iff for all datasets D and D' differing in at most one element, and for all $E \subseteq \text{Range}(\mathcal{M})$, the following holds:

$$Pr[\mathcal{M}(D) \in E] \leq e^{\varepsilon} Pr[\mathcal{M}(D') \in E] + \delta \tag{1}$$

The amount of protection provided by DP increases as ε and δ approach 0. Dwork et al. [17] recommend setting δ to be smaller than $1/n$ for a dataset of cardinality n. The parameter ε is called *privacy budget*.

Datasets D and D' that differ in a single element are said to be *neighboring*, or *sibling*. When the adjacency between the datasets is defined with respect to a single data record, then the DP formulation provides *record-level* privacy guarantees. The amount of protection can be extended to account for cases when a single individual contributes multiple data records. In this case, the sibling relationship is defined by allowing D and D' to differ only in the records provided by a single individual. This is a stronger privacy guarantee, called *user-level* privacy.

To achieve (ε, δ)-DP, the result obtained by evaluating a function (e.g., a query) f on the input data must be perturbed by adding noise sampled from a random variable Z. The amount of noise required to ensure the mechanism $\mathcal{M}(D) = f(D) + Z$ satisfies a given privacy guarantee depends on how sensitive the function f is to changes in the input, and the specific distribution chosen for Z. The Gaussian mechanism (GM) [16] is tuned to the sensitivity S_f computed according to the global ℓ_2-norm as $S_f = \sup_{D \simeq D'} ||f(D) - f(D')||_2$ for every pair of sibling datasets D, D' . GM adds zero-mean Gaussian noise calibrated to the function's sensitivity as follows:

Theorem 2.1. *For a query $f : D \rightarrow \mathbb{R}$, a mechanism $\mathcal{M}$ that returns $f(D) + Z$, where $Z \sim \mathcal{N}(0, \sigma^2 S_f^2)$ guarantees (ε, δ)-DP if $\sigma^2\varepsilon^2 \geq 2ln(1.25/\delta)$ and $\varepsilon \in [0, 1]$ (see [17] for the proof).*

The *composability* property of DP helps evaluate the effect on privacy when multiple functions are applied to the data (e.g., multiple computation steps). Each step is said to consume a certain amount of privacy budget, and the way the budget is *allocated* across multiple steps can significantly influence data utility.

2.2 Neural Networks

Modern machine learning (ML) models leverage the vast expressive power of artificial neural networks to dramatically improve learning capabilities. Convolutional networks have shown exceptional performance in processing images and video [30]. Recurrent networks can effectively model sequential data such as text, speech and DNA sequences [12, 28]. A neural network is composed of one or more interconnected multilayer stacks, most of which compute non-linear input-output mappings. These layers transform the representation at one level (starting with the raw input) into a representation at a higher, more abstract level. The key to improving inference accuracy with a neural net is to continually modify its internal adjustable parameters.

Stochastic gradient descent (SGD) is the canonical optimization algorithm for training a wide range of ML models, including neural networks. It is an iterative procedure which performs parameter updates for each training example x_i and label y_i. Learning the parameters of a neural network is a nonlinear optimization problem. At each iteration, a batch of data is randomly sampled from the training set. The error between the model's prediction and the training labels, also called *loss*, is computed after each iteration. The loss is then differentiated with respect to the model's parameters, where the derivatives (or gradients) capture their contribution to the error. A *back-propagation* step distributes this error back through the network to change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer. Each internal parameter of the model θ is brought closer to predicting the correct label as follows:

$$\theta = \theta - \eta \cdot \nabla_\theta \mathcal{J}(\theta; x^{(i)}; y^{(i)})$$

where η is the *learning rate* hyper-parameter and $\mathcal{J}$ is the loss function. Iteratively recomputing gradients and applying them to update the model's parameters is referred to as *descent*, and this operation is performed until the model's performance is satisfactory.

2.3 Differentially Private-SGD (DP-SGD)

Introduced in [1], DP-SGD integrates (ε, δ)-DP with neural networks. It modifies traditional SGD in that after calculating the changes in its internal parameters, it obfuscates the gradient values with noise sampled from the Gaussian distribution.

Table 1: Summary of Notations

Notation	Definition
U, P	Sets of users and check-in locations, respectively
N, L	Cardinalities of sets U and P, respectively
U_u	Historical record of user u's check-ins
dim	Dimension of location embedding space
b, η	Batch size and learning rate, respectively
q	User sampling probability per step
m	Expected user sample size per step
ε, δ	Privacy parameters of Gaussian mechanism
σ	Noise scale
λ	Data grouping factor
$\mathcal{H}$	Set of training buckets
C	Per-layer clipping norm

DP-SGD averages together multiple gradient updates induced by training-data examples, *clips* (i.e., truncates) each gradient update to a specified maximum ℓ_2-norm, and adds Gaussian random noise to their averaged value. Clipping each gradient bounds the influence of each training-data example on the model. Accordingly, the sensitivity of the average query can be adjusted as desired, and due to the added noise tuned to the sensitivity of the query, differential privacy is ensured in each iteration. Typically, repeatedly executing a query results in sharp degradation of the privacy protection, as more information is leaked by multiple usages of private iterations. The *moments accountant* technique [1] computes the privacy loss resulting from the composition of Gaussian mechanisms under random sampling. It tracks the moments of the privacy loss variable in each step of the descent, and provides a much tighter upper bound on privacy budget consumption than the standard composition theorem [17].

3 SYSTEM ARCHITECTURE

In Section 3.1 we define the problem statement. We outline the learning model architecture in Section 3.2 and we show how it is utilized in Section 3.3. Table 1 summarizes notations used throughout the paper.

3.1 Problem Statement

Data Representation. The input to our learning model consists of check-in data from a set of N users $U = \{u_1, u_2, ..., u_N\}$. The set of L check-in locations (e.g., points of interest) is denoted as $P = \{l_1, l_2, ..., l_L\}$. Each user $u \in U$ has a historical record of check-ins denoted as $U_u = \{c_1, c_2, ...\}$, where each element c_i is a triplet $\langle u, l, t\rangle$ comprised of user identifier, location and time.

Learning Objective. The objective of our model is to predict the location that a given user u will check into next, given a time-ordered sequence of previous check-ins of the user. The past check-ins can represent the user's current trajectory or his entire check-in history. For each scenario, we describe the usage of the model in Section 3.3. In an initial step, we employ an unsupervised learning method, specifically the skip-gram model [43], to learn the latent distributional context [50] of user movements over the set P of possible check-in locations. A latent representation of every location in a reduced-dimension vector space is the intermediate output. Next, we determine for each user

u its inclination to visit a particular location l by measuring how similar l is in the latent vector space to the locations previously visited by u.

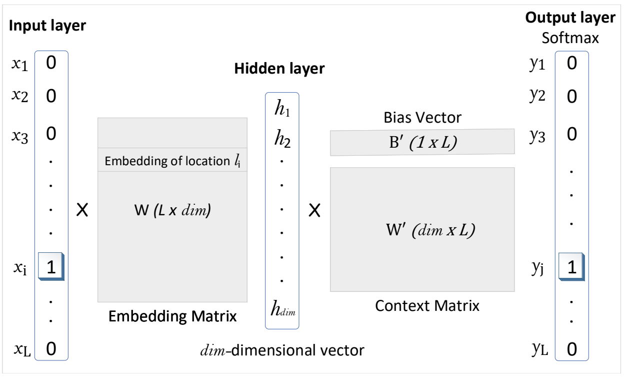


Figure 2: Architecture of the location-recommendation model

3.2 Learning Model

The skip-gram negative sampling (SGNS) model [41, 43] was initially proposed to learn word embeddings from sentences. However, several recent efforts [10, 35, 58] show that the model is also appropriate for location recommendation tasks. Specifically, the model is used to learn location embeddings from user movement sequences, where each location corresponds to a word, and a user's check-in history to a sentence.

Given the set of check-ins of a user, we treat the consecutively visited locations as a trajectory that reflects her visit patterns. A data pre-processing step is required to make the data format compatible with the input of a neural network: every location in P is tokenized to a word in a vocabulary of size $L = |P|$. Given a *target* location check-in c, a symmetric window of win context locations to the left and win to the right is created to output multiple pairs of target and context locations as training samples. The assumption is that if a model can distinguish between actual pairs of target and context words from random noise, then good location vectors will be learned.

Figure 2 illustrates the neural network used in our solution. The model parameters consist of three tensors $\theta = \{W, W', B'\}$ and two hyper-parameters representing the embedding dimensions dim and the negative samples drawn neg. Consider a target-context location pair (l_x, l_y). First, both locations are one-hot encoded into binary vectors $\vec{x}$ and $\vec{y}$ of size L. The multiplication of $\vec{x}$ with embedding matrix W produces the embedding vector for the input location l_x (i.e., the i^{th} row of matrix W). $W \times x$ represents the mapping of input location x to a vector $\vec{h}$ in an dim-dimensional space. Next, for each positive sample (i.e., true target/context pair), a neg number of negative samples are drawn. The context location vector $\vec{y}$ along with the negative samples are passed through a different weight matrix W' and bias vector B'. Finally, a sampled softmax loss function is applied to calculate the prediction error. At a high level (we refer the reader to [49] for a detailed look), the parameters are modified such that the input word (and the corresponding embedding) is tugged closer to its neighbors (i.e., paired context locations), and tugged away from the negative samples. As a result, during back-propagation, only $neg + 1$ vectors in W or W' are updated instead of entire matrices. In the original work [41, 43], negative sampling was devised to improve computational efficiency, as updating the entire model in each iteration can be quite costly. In private learning, it also plays an important role in controlling the adverse affects of noisy training.

We remark here that techniques such as Noise Contrastive Estimation [23] and Negative Sampling use a non-uniform distribution for drawing the samples—for example, by decreasing the sampling weight for the frequent classes—whereas, we use a sampled softmax function with a uniform sampling distribution. This is a necessity for preserving privacy, since estimating the frequency distribution of locations from user-submitted data will cause privacy leakage. Lastly, the embedded vectors are normalized to unit length for efficient use in downstream applications. On top of improving performance [32, 55], normalizing the vectors assists similarity calculation by making cosine similarity and dot-product equivalent.

We detail the privacy-preserving learning model in Section 4. In the remainder of this section, we show how the model, once computed in a privacy-preserving fashion, can be utilized.

3.3 Model Utilization

We provide an overview of how our proposed privacy-preserving next-location prediction model is utilized. Once our privacy-preserving learning technique is executed, the resulting model can be shared with consumers, since the users who contributed the data used in the training are protected according to the semantic model of DP. While the utilization of our model is orthogonal to our proposal, we include it in this section in order to provide a complete, end-to-end description of our solution's functionality.

A typical use of our model is for a mobile user to download[1] it to her device, provide her location history as input, and receive a next-location recommendation. Alternatively, a service provider who already has the locations of its subscribers, will perform the same process to provide a next-location suggestion to a customer. We emphasize that, the model utilization itself does not pose any privacy issues. In both cases above, neither the input, nor the output to the model are shared, so there is no privacy concern. The only time we need to be concerned about privacy is when *training* the model, since a large amount of trajectories from numerous users is required for that task.

Consider a user who has recent check-ins ζ in a relatively short time period (e.g., last few hours). This set of locations forms the basis for recommending to the user the next location to visit. The normalized embedded matrix W in the fully-trained model encodes the latent feature vector of all locations. For each location check-in $l_i \in \zeta$, the embedding vectors $w(l_i)$ are extracted and stacked on top of each other. More precisely, to obtain the embedding vector $w(l_i)$, the binary vector of l_i is multiplied with W (similar to the first step of the training process). This process is equivalent to extracting the dim-dimensional row corresponding to location l_i. Then, the average of elements across dimensions of the stacked vectors is computed to produce a representation $\mathcal{F}(\zeta)$ of the recent check-ins of the user. Finally, cosine similarity scores are computed as the dot-product of the vector $\mathcal{F}(\zeta)$ to the embedding vector of each location in the universe L. We rank all locations by their scores and select the top-K locations as the potential recommendations for the user.

In the case when the user has no recent check-ins, the representation $\mathcal{F}()$ can be computed over her movement profile

[1]To reduce communication costs, only the embedding matrix is deployed.

comprising of historical check-ins. Other methods include training an additional model to learn latent feature vectors of each user from her preferences and locations visited. As in [19, 58], a user's feature representation can be used to determine her inclination to visit a particular location. However, modeling each user with such personalized representations, while at the same time preserving *user*-level privacy, is a fundamentally harder problem (in terms of both system design and privacy framework), and is left as future work.

When the model is deployed at an untrusted location-based service provider (LBS), additional privacy concerns must be addressed. In this case, the mobile user must protect the set ζ (or $\mathcal{F}(\zeta)$) *locally*. Techniques such as geo-indistinguishability [3] can be applied to protect the check-in history (discussed in Section 6). For example, the check-in coordinates can be obfuscated to prevent adversaries from pinpointing the user to a certain location with high probability. Addressing these vulnerabilities in the MLaaS setting is orthogonal to the scope of this paper.

4 PRIVATE LOCATION PREDICTION (PLP)

Section 4.1 presents in detail our proposed approach for private next-location prediction. Section 4.2 provides a privacy analysis of our solution.

4.1 Private Location Prediction (PLP)

PLP is a customized solution to location recommendation. It learns latent factor representations of locations while controlling the influence of each user's trajectory data to the training process. Bounding the contribution of a *single* data record in the SGD computation has been proposed in previous work [2, 53]. We make several extensions and contribute data grouping techniques to boost model performance. Even while combining data of multiple users, we guarantee user-level privacy (such as in [21, 39]). By grouping data records of multiple users, we benefit from cross-user learning to improve model performance.

Algorithm 1 depicts the procedure of this learning process. Model hyperparameters labeled batch size β, learning rate η and loss function $\mathcal{J}$ are related to gradient descent optimization, whereas hyperparameters labeled grouping factor λ, sampling probability q, gradient clipping norm bound C, noise scale σ and privacy parameters ε, δ are introduced to create an efficient and privacy-preserving system. We briefly describe each component in isolation before coupling them together to illustrate the big picture.

User Sampling. Given a sampling probability $q = m/N$, each element of the user set is subjected to an independent Bernoulli trial which determines whether the element becomes part of the sample. As a consequence, the size of sampled set of users U_{sample} is equal to m only in expectation. This is a necessary step in correctly accounting for the privacy loss via the moments accountant [2].

Data Grouping. Data grouping is essentially a pre-processing technique that significantly boosts model performance. It has a dual purpose. The first is to reduce the effects of skewness and sparsity inherent to location data, where the frequency of check-ins of users at locations follows the Zipf's law [11]. The second is to provide cross-user learning to smooth updates in the model parameters produced by the function in lines 15-22. The underlying intuition is simple: to ensure good performance of the context model, each update of a training step must contribute to the final result. By combining the profiles of multiples users

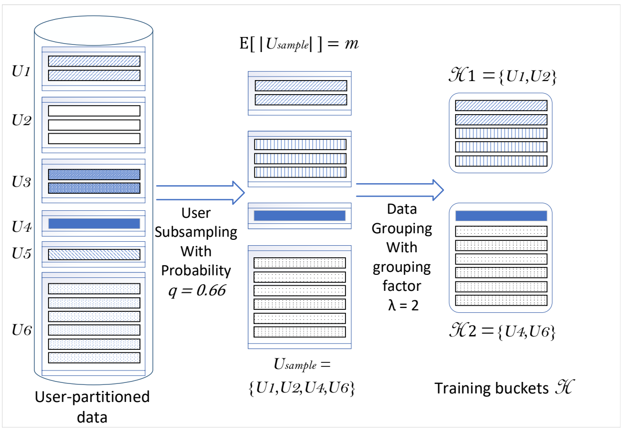


Figure 3: Data sampling and grouping

we also reduce minor observation errors that may be produced from specific data points in a user's profile.

Our data grouping technique agglomerates the data of multiple users into buckets $\mathcal{H}$. Given a grouping factor λ, users (and their entire data) are randomly assigned to buckets such that each bucket contains λ users. This operation is encapsulated in the *groupData*$(\cdot)$ function in line 6. As a separate method, we also tried equal frequency grouping, where a global pass over the record count of each user is used to produce buckets such that each contains approximately the same number of records (while ensuring that the data records of each user are not split into multiple buckets). However, we noticed no statistically significant benefit in model accuracy from equal frequency grouping than with a random grouping. Accordingly, we use the latter in the rest of the work.

Figure 3 illustrates the data sampling and grouping process (corresponding to lines 5-6) for a sampling probability of 0.66 and $\lambda = 2$. Grouped data in each bucket is organized as a single array for processing by gradient descent optimization. Recall from Section 3.2 that a symmetric moving window is applied to create training examples, after the array is read by the *generateBatches*() function (in line 17). A number β of target-context location pairs are placed in each batch.

In brief, at each step of PLP, we sample a random subset of users (line 5), combine the data of multiple users into buckets (line 6), compute a gradient update with bounded ℓ_2 norm from each bucket (lines 7-8), add noise to the sum of the clipped gradients (line 9), take their approximate average, and finally update the model by adding this approximation (line 10). Alongside, a privacy ledger is maintained to keep track of the privacy budget spent in each iteration by recording the values of σ and C (lines 3 and 11). This tracker has the added benefit of allowing privacy accounting at any step of the training process. Given a value of δ and the recorded ledger, the moments accountant can compute the overall privacy cost in terms of ε. This functionality is provided by the cumulative_budget_spent() function in line 12, which implements the moments accountant from [2].

Privacy Mechanism. The gradient values computed in line 20 do not have an a-priori bound. This complicates the application of the Gaussian Mechanism (GM), which is generally tuned to the sensitivity of the performed query. In this particular use case, we employ a Gaussian sum query in line 9, the results of which are then averaged using a fixed-denominator estimator. To bound the sensitivity of this query, a maximum sensitivity of C is enforced

Algorithm 1 Algorithm for Private Location Prediction with user-level privacy.

```
Input: loss function $\mathcal{J}(\theta)$, grouping factor $\lambda$, learning rate $\eta$, sampling probability $q = m/N$, gradient norm bound $C$, batch size $\beta$, privacy parameters $\varepsilon, \delta$
1:  procedure TrainPrivateLocationEmbedding
2:      Initialize: Model $\theta_0 = \{W, W', B'\}$,
3:              Privacy Accounting ledger $\mathcal{A}(\delta, q)$
4:      for each step $t = 1, \ldots$ do
5:          $U_{sample} \leftarrow$ a random sample of $m_t$ users
6:          Initialize buckets $\mathcal{H} \leftarrow groupData(U_{sample}, \lambda)$
7:          for each data bucket $d_h \in \mathcal{H}$ do
8:              $\bar{g}_h \leftarrow$ ModelUpdateFromBucket$(\theta_t, d_h)$
9:          $\widehat{g}_t = \frac{1}{|\mathcal{H}|}(\sum_{h \in \mathcal{H}} \bar{g}_h + \mathcal{N}(0, \sigma^2 C^2 I))$      ▷ Noise.
10:         $\theta_{t+1} = \theta_t + \widehat{g}_t$      ▷ Model Update.
11:         $\mathcal{A}$.track_budget$(C, \sigma)$
12:         if $\mathcal{A}$.cumulative_budget_spent() $\geq \varepsilon$ then:
13:             return $\theta_{t-1}$
14:
15: function ModelUpdateFromBucket$(\theta_t, d_h)$
16:     $\Phi \leftarrow \theta_t$
17:     $\mathcal{B} \leftarrow generateBatches(d_h, \beta)$
18:     for each $b \in \mathcal{B}$ do
19:         $\Phi \leftarrow \Phi - \eta \frac{1}{|b|} \sum_{(x_i, y_i) \in b} \nabla_{\Phi} \mathcal{J}(\Phi, x_i, y_i)$
20:     $g_h = \Phi - \theta_t$
21:     $\bar{g}_h = g_h / \max(1, \frac{\|g_h\|_2}{C})$      ▷ Gradient Norm Clipping.
22:     return $\bar{g}_h$
```

on every gradient computed on bucket h as follows (equivalent to line 21):

$$||\bar{g}_h||_2 = \begin{cases} ||g_h||_2 & \text{for } ||g_h||_2 \leq C \\ C & \text{for } ||g_h||_2 > C. \end{cases}$$

Gradient clipping places a strict limit on the maximum contribution—in terms of its ℓ_2 norm—of the gradient computed on a bucket. Formally, $||\bar{g}_h||_2 \leq C$. The sensitivity of the scaled gradient updates with respect to the summing operation is thus upper bounded by C. Finally, dividing the GM's output by the number of buckets $|\mathcal{H}|$ yields an approximation of the true average of the buckets' updates.

We note that increasing the number of users in each bucket increases the valuable information in each gradient update. At the same time, the noise introduced by the Gaussian mechanism is scaled to the sensitivity of each bucket's update (i.e., C). If too few buckets are utilized, this distortion may exceed a limit, meaning that too much information output by the summing operation is destroyed by the added noise. This will impede any learning progress. We treat the grouping factor λ as a hyper-parameter and tune it.

In a multi-layer neural network such as the one described in our work, each tensor can be set to a different clipping threshold. However, we employ the *per-layer clipping* approach of [37], where given an overall clipping magnitude C, each tensor is clipped to $C/\sqrt{|\theta|}$. In the skip-gram model, $\theta_0 = \{W, W', B'\}$, hence $|\theta| = 3$, so we clip the ℓ_2-norm of each tensor to $C/\sqrt{3}$. However, the effect of clipping on the three tensors is rather different due to the difference in their dimensionality. Context matrix W' is clipped to the same degree as bias vector B', despite the fact that they have dimensions $(L \times dim)$ and $(1 \times L)$, respectively. While the dimensionality of the embedding matrix W is $(L \times dim)$, only a fraction of the weights—proportional to *neg*, instead of L—are considered for clipping due to the sampling of *neg* number of negative examples in the sampled softmax function. Simply put, $||W||_2$ is proportional to *neg* and when carefully tuned, the clipping parameter is large enough that nearly all updates are smaller than the clip value of $C/\sqrt{|\theta|}$, improving the signal-to-noise ratio over iterative computations. We discuss the effect of this parameter in controlling the distortion of Gaussian noise in Section 5.

4.2 Privacy Analysis

Recall that, our proposed system provides *user-level* differential privacy to individuals who contribute their check-in history to the training data. This ensures that all individuals are protected, regardless of how much data they contribute (i.e., even if the length of the check-in history varies significantly across users). Let U_k denote the data of a single user. The sensitivity of the Gaussian Sum Query (GSQ) function w.r.t. to neighboring datasets that differ in the records of a single user is defined as

$$S_{GSQ} = \max_{\{U_{sample}, U_k\}} ||GSQ(U_{sample} \cup U_k) - GSQ(U_{sample})||_2$$

In Algorithm 1, GSQ is executed over the bucket gradients, which complicates the analysis of the privacy properties of the algorithm. We consider two distinct scenarios where a user's data may be assigned to: (i) exactly one bucket; or (ii) more than one bucket. We define ω as the *data split factor*, meaning that a user's data may be placed in at most ω buckets.

Case 1 [$\omega = 1$]. This represents the scenario where multiple (up to λ) users' data may be present in a single bucket, but a single user's data may be allocated to at most one bucket. Figure 4(a) depicts this case, which is assumed by default in Algorithm 1. This is a sufficient condition to ensure that the per-user contribution to a bucket's gradient update is tightly bounded. Formally, there exists a unique $d_h \in \mathcal{H}$ s.t. $U_k \subseteq d_h$. In addition, when the ℓ_2 norm of the gradient $||\bar{g}_h||_2$ computed on a data-bucket d_h is upper-bounded by the clipping factor C, we get

$$S_{GSQ} \leq \max_{\{\mathcal{H}, d_h\}} ||GSQ(\mathcal{H} \cup d_h) - GSQ(\mathcal{H})||_2 \leq C$$

An informal proof that this approach satisfies (ε, δ)-DP is as follows: The sensitivity of the gaussian sum query $GSQ = \sum_{h \in \mathcal{H}} \bar{g}_h$ is bounded as $S_{GSQ} \leq C$, if for all buckets we have $||\bar{g}_h||_2 \leq C$. By extension, if a sampled user (and his location visits) can be assigned to exactly one bucket, *sibling* datasets that differ in the data of a single user can change the output of GSQ by at most C. Therefore, Gaussian Noise drawn from $\mathcal{N}(0, \sigma^2 C^2 I)$ guarantees user-level (ε, δ)-DP.

Case 2 [$\omega > 1$]. If the data of a single user is split over multiple buckets, then it is possible that even after scaling the bucket gradients to C, the sensitivity of the Gaussian sum query is no longer C w.r.t. to user-neighboring datasets. Figure 4(b) illustrates an example with $\omega = 2$. A similar split strategy (proposed in [38]) is used in the empirical evaluation of [21], wherein a small dataset is scaled up to amplify privacy accounting. However, the authors fail to regulate their noise scale to reflect the altered data sensitivity or alternatively recompute the achieved privacy guarantee. We show that when the data of a user U_k is split across multiple buckets, the sensitivity of the query increases to

ω. Assuming that $|\mathcal{H}| \leq |U_{sample}|$, we can write,

$$\omega = \max_{\{U_k \in U_{sample}\}} |\{d_h : d_h \in \mathcal{H} \text{ and } d_h \cap U_k \neq \emptyset\}|$$

meaning that the data of a user can influence the gradients of at most ω buckets. Accordingly, if for all buckets $||\bar{g}_h||_2 \leq C$, a single user can change the output of *GSQ* by at most ωC. Therefore, to guarantee user-level (ε, δ)-DP, Gaussian Noise must be drawn from $\mathcal{N}(0, \sigma^2\omega^2C^2I)$.

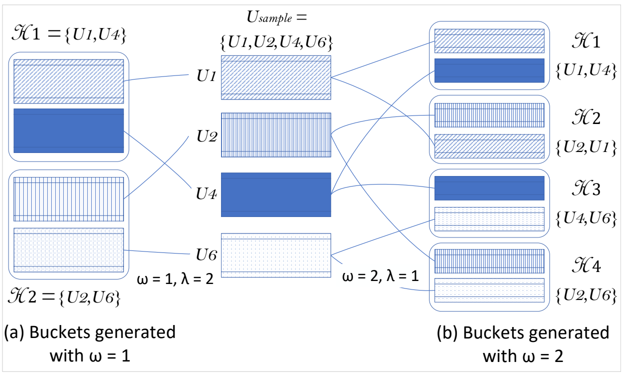

$\mathcal{H}1 = \{U1,U4\}$
$\mathcal{H}2 = \{U2,U6\}$
$U_{sample} = \{U1,U2,U4,U6\}$
U1
U2
U4
U6
ω = 1, λ = 2
ω = 2, λ = 1
$\mathcal{H}1$ $\{U1,U4\}$
$\mathcal{H}2$ $\{U2,U1\}$
$\mathcal{H}3$ $\{U4,U6\}$
$\mathcal{H}4$ $\{U2,U6\}$
(a) Buckets generated with ω = 1
(b) Buckets generated with ω = 2

Figure 4: Sensitivity of Gaussian Sum Query over U_{sample} users: (a) $\omega = 1$, a single user's data is placed in exactly one bucket; (b) $\omega = 2$, a single user's data is split across two buckets. Since gradients computed over the generated buckets $\mathcal{H}_1, ..., \mathcal{H}_4$, are bounded by C, a user can contribute at most $2C$ to the computed sum.

We remark here that values of $\omega > 1$ produced no positive effect in our evaluation. We experimented with $\omega = 2$ by splitting a user's data to exactly two random buckets. We found that the signal-to-noise ratio is adversely affected, since the marginally improved signal from the split data is offset by the now quadrupled (proportional to ω^2) noise variance. In the rest of the work, we set ω to 1.

5 EXPERIMENTS

Section 5.1 provides the details of the experimental testbed. Section 5.2 focuses on the evaluation of the proposed technique in comparison with the state-of-the-art DP-SGD approach. In Section 5.3 we evaluate in detail our approach when varying system parameters, and provide insights into hyper-parameter tuning.

5.1 Experimental Settings

Dataset. We use a real dataset collected from the operation of a prominent geo-social network, namely *Foursquare* [59]. The data consist of a set of user check-ins. Every check-in is described by a record comprising of user identifier, the latitude and longitude of the check-in, and the identifier of the POI location. In order to simulate a realistic environment of a city and its suburbs, we focus on check-ins within a single urban area, namely Tokyo, Japan. In particular, we consider a large geographical region covering a 35×25km^2 area bounded to the South and North by latitudes 35.554, 35.759, and to the West and East by longitudes 139.496, 139.905. We filter out the users with fewer than ten check-ins, as well as the locations visited by fewer than two users (such filtering is commonly performed in the location recommendation literature [33, 61]). The remainder of the data contains a total of 739, 828 check-ins from 4, 602 unique users over 5, 069 locations during a time period of 22 months from April 2012 to January 2014.

Implementation. All algorithms were implemented in Python on a Ubuntu Linux 18.04 LTS operating system. The experiments were executed on an Intel Xeon Platinum 8164 CPU, with 64GB RAM. All data and intermediate structures (e.g., neural network parameters, gradients) are stored in main memory. The proposed neural model is built using Google's Tensorflow library [1]. To account for the privacy budget consumption of the complex iterative mechanism used in learning, we use the privacy accounting method from [54], which allows for a tight composition of privacy-preserving steps. At each step of the computation, we calculate the (ε, δ) tuple from moment bounds, according to the moments accountant procedure introduced in [37].

Evaluation Metric. To evaluate the performance of location recommendation, we adopt the "leave-one-out" approach, which has been widely used in the recommender systems literature [10, 19, 26, 35, 57, 58]. This metric simulates the behavior of a user looking for the next location to visit. Given a time-ordered user check-in sequence, recommendation models utilize the first $(t - 1)$ location visits as an input and predict the t^{th} location as the recommended location. The recommendation quality is measured by *Hit-Rate (HR)*. *HR@k* is a recall-based metric, measuring whether the test location is in the top-k locations of the recommendation list. The outcome of the evaluation is binary: 1 if the test location is included in the output set of the recommender, and 0 otherwise. In the rest of the section, we use the terms *prediction accuracy* and *HR@k* interchangeably.

Model Training. Our testing and validation sets consist of location visits of users who are not part of the training set. Since we do not train models to learn user specific representations (such as in [10, 35, 58]), this is an accurate representation of real-life model utilization at a user's device. Validation and testing sets are created in a similar fashion. First, a randomly selected set of 100 users and their corresponding check-ins are removed from the dataset. From these, time ordered sequences of trajectories are generated. Each individual trajectory does not exceed a total duration of six hours (following the work in [10, 34]). The remaining 4402 users and their check-ins represent the training dataset for learning the parameters of the proposed model.

To train the model, we utilize Adam [29], a widely adopted optimization algorithm for deep neural network models that has specific properties to mitigate disadvantages of traditional SGD, such as its difficulty in escaping from saddle points, or extensive tuning of its learning rate parameter. We implement the optimizer in a differentially private manner by tracking an exponential moving average of the noisy gradient and the squared noisy gradient, as illustrated in [24]. We found that tuning the initial learning rate and decay scheme for Adam only affects the learning in the very first few steps. Typically, Adam does not require extensive tuning [29] and a learning rate between 0.001 to 0.1 is most often appropriate. In our experiments, we found that a learning rate value $\eta \in [0.02, 0.07]$ produces similar results, so we set it to 0.06 for all our runs.

Parameter Settings. We select the training hyper-parameters of the skip-gram model via a cross validation grid search. Figure 5 depicts the validation accuracy over 200 data epochs using the non-private learning approach. We plot the validation Hit-Rate for $k = 5$, 10 and 20 candidates, respectively. We look for those models that reach the highest accuracy. The embedding dimension *dim* is set to 50. While a larger number of hidden units allows more predictive power, the accuracy improvement reaches a plateau when the embedding dimension is in the range [50, 150]. In non-private training, it is preferable to use more units, whereas for private learning a larger model increases the sensitivity of the gradient. We keep our model at the lower end of the *dim* range to keep the number of internal parameters of the models low. The batch size is set to $b = 32$, and the context window parameter $win = 2$ (for a total window size of 5). These parameters are also consistent with those utilized in previous work [10, 58]. Varying the number of negative examples sampled (denoted by *neg*) marginally affects the non-private model, whereas with private learning we find that it directly controls the sensitivity of the private sum query (in Section 5.3 we show experiments on how to tune it). The default value for negative samples is $neg = 16$.

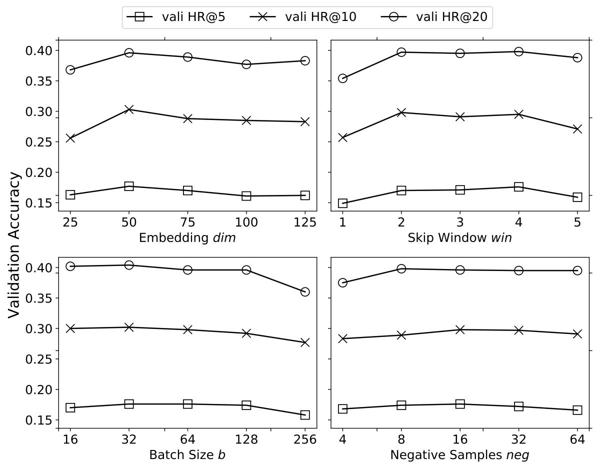


Figure 5: Non-private model hyperparameter tuning

For the privacy parameters, we fix the value of $\delta = 2 \times 10^{-4} < 1/N$ as recommended in previous work on differentially-private machine learning [2, 39]. For a given value of δ, the privacy budget ε affects the amount of steps we can train until we exceed that budget threshold. We set the default value of the hyper-parameters to $q = 0.06$, $\sigma = 2.5$, $C = 0.5$, $\lambda = 4$ (please see Table 1 for a summary of notations). Recall that, the sampling ratio of each lot is $q = m/N$, so each epoch consists of $1/q$ steps.

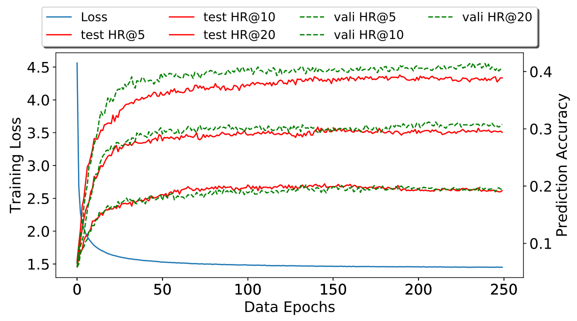


Figure 6: Non-private model performance

5.2 Comparison with Baseline

We evaluate the performance of our proposed approach in comparison with two baselines: *(i)* a non-private learning approach using SGD, and *(ii)* the state-of-the-art user-level DP-SGD approach from [2, 39].

First, we evaluate the non-private location prediction model described in Section 3.2. Figure 6 illustrates the validation and testing Hit-Rate at $k = 5$, 10 and 20. The model generalizes well to the test set, and there appears to be no evidence of overfitting up to 250 data epochs. The presented results are competitive with existing approaches in [35, 58], suggesting that the model hyper-parameters are suitable to capture the underlying semantics of mobility patterns. The best testing accuracy of the non-private model for the *HR*@10 setting is 29.5%.

Throughout our evaluation we found that, when the model is trained in a differentially private manner, there is only a small difference between the model's accuracy on the training and the test sets. This is consistent with both the theoretical argument that differentially private training generalizes well [5, 15], and the empirical evidence in previous studies [2, 39]. For brevity of presentation, in the rest of this section we only show *HR*@10 evaluation results (similar trends were recorded for *HR*@5 and *HR*@20).

Next, we evaluate our proposed Private Location Prediction (PLP) approach in comparison with DP-SGD [2], which is summarized in Section 2. We adapt the model to work on user-partitioned data, so that it guarantees user-level privacy. The improvements of PLP over DP-SGD passed the paired t-test with significance value $p < 0.01$.

Figure 7 plots the prediction accuracy of the privately trained models for varying levels of privacy ε. For each ε value, we consider two settings each for sampling probability $q = 0.06$ (upper left) and $q = 0.10$ (bottom right). We set $\sigma = 1.5$. We compare PLP against the baseline DP-SGD for two values of the grouping factor λ. As expected, a general trend we observed is that providing more privacy budget allows the models to train to a higher accuracy. However, for the baseline approach, the convergence of the model is thwarted because the model update computed on the data of a single user contributes a limited *signal*, which is often offset by the introduced Gaussian noise. On the other hand, the results show that by incorporating data grouping in its design, PLP is able to ameliorate the data sparsity problem inherent to location datasets. The gain is more pronounced when the grouping factor increases (i.e., higher λ).

Next, we measure the effect of sampling probability q on accuracy. From the theoretical model [2], we know that q directly affects the amount of privacy budget utilized in each iteration (q is also called "privacy amplification factor"). A lower sampling rate includes less data in each iteration, hence the amount of budget consumed in each step is decreased. Our results in Figure 8 confirm this trend. We vary the rate of user sampling q from 4% to 12%. For all runs, we fixed the budget allowance at $\varepsilon = 2$. For a higher sampling probability, the privacy budget is consumed faster, hence the count of total training steps is smaller, leading to lower accuracy. Our proposed PLP method clearly outperforms DP-SGD, whose accuracy drops sharply with an increase in q. We note that, due to the proposed grouping strategy, PLP is more robust to changes in sampling rate, as its accuracy degrades gracefully. In general, a larger bucket cardinality leads to better accuracy, except for the lowest considered sampling rate, where the small fraction of records included in the computation at each

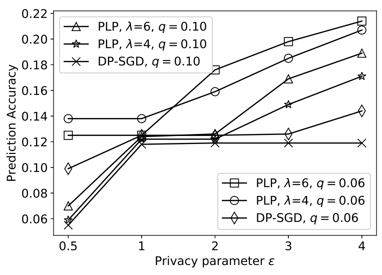


Figure 7: PLP vs DP-SGD: varying privacy budget ε

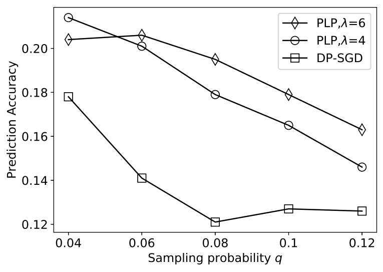


Figure 8: PLP vs DP-SGD: varying sampling ratio q

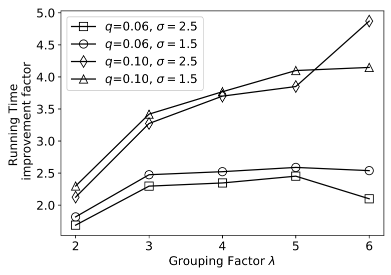


Figure 9: Running time, varying grouping factor λ

step prevents buckets from reaching a significant diversity in their composition.

Finally, we provide a result on the runtime improvements offered by PLP. The y-axis in Figure 9 depicts the multiplicative factor by which PLP is faster that DP-SGD. We show results for two values of q, and for each we present the runtime with two values of noise scale. Linearly scaling the grouping factor has two opposing effects: on the one hand, fewer buckets implies that equally few bucket gradients need to be computed and averaged. On the other hand, as each bucket gets assigned more users, it takes longer to compute each bucket gradient. When fewer users are sampled (i.e., $q = 0.06$) the latter effect begins to dominate for $\lambda > 5$, whereas for $\lambda \in [2, 5]$, the computational efficiency scales from 1.6× to 2.5×. In the setting where sampling rate is higher, at $q = 0.10$, the runtime improvements scale monotonically, to over 4.8× for $\lambda = 6$. These results are consistently observed even with a different number of total iterations (as a larger σ allows more iterations).

In summary, our results so far show that PLP clearly outperforms the existing state-of-the-art DP-SGD. Furthermore, its accuracy was observed to reach values as high as 24%, which is quite reasonable compared to the maximum of 29.5% reached by the non-private learning approach. In the rest of the evaluation, we no longer consider DP-SGD, and we focus on tuning the parameters of the proposed PLP technique.

5.3 Hyper-parameter Tuning

The objective of tuning model hyper-parameters is to obtain a good balance of accuracy and computational overhead of learning. We focus on the following parameters, which we observed throughout the experiments to have a significant influence: grouping factor λ, noise scale σ, the magnitude of ℓ_2 clipping norm, and the number of negative samples *neg*.

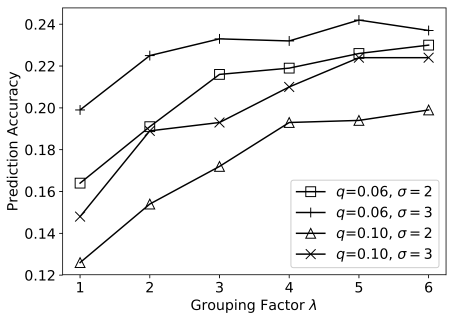


Figure 10: Effect of varying λ

Effect of Grouping factor λ. Figure 10 shows the influence on accuracy of grouping factor λ. We consider two distinct settings each of sampling parameter q and noise scale σ (for a total of four lines in the graph). To limit sensitivity, we clip the gradient norm of each tensor to a maximum l_2 norm of 0.5. Choosing the grouping factor must balance two conflicting objectives: on the one hand, assigning the data of multiple users to the same bucket improves the signal in each bucket gradient, by improving the data diversity within the bucket. On the other hand, the Gaussian noise must be scaled to the sensitivity of a bucket gradient, and a larger bucket size results to fewer buckets, which in turn increases the effect of added noise. Our results confirm this trade-off: initially, when λ increases there is a pronounced increase in accuracy. After a certain point, the accuracy levels off, and reaches a plateau around the value of $\lambda = 5$. When the grouping factor is increased further (not shown in the graph), the accuracy starts decreasing, because there is no significant gain in per-bucket diversity, whereas the relative noise-to-signal ratio keeps increasing.

Effect of Noise Scale σ. The noise scale parameter σ directly controls the noise added in each step. A larger σ leads to more noise, but at the same time it decreases the budget consumption per step, which in turn allows the execution of more learning steps. Figure 11 depicts the model accuracy for varying settings of noise scale. The results presented correspond to two settings each of sampling rate and privacy budget (for a total of four lines). We observe that for the lower-range of σ values, the accuracy is rather poor, especially for smaller settings of privacy budget ε. This is explained by the fact that too little noise is added per step, and the privacy consumption per step is high. As a result, only a small number of steps can be executed before the privacy budget is exhausted, leading to insufficient learning. For larger ε settings, the effect is less pronounced, because there is sufficient budget to allow more steps, even when the noise scale is low.

Conversely, a larger σ allows more steps to be executed, so the best accuracy is obtained for the largest $\sigma = 3.0$ setting. However, we also note that the accuracy levels off towards that setting. For larger σ values (not showed in the graph), we observed that the noise magnitude is too high, and even if budget is slowly exhausted, the training loss in each learning step is excessively high, preventing the model from converging, and leading to very low accuracy. We conclude that the choice of noise scale must be carefully considered relative to the total privacy budget, such that a sufficient number of steps are allowed to execute, while at the same time the loss function value per step is not excessive.

The total number of executed steps also influences the computational overhead of learning. If execution time is a concern, one may want to reduce the number of steps by reducing σ, in an attempt to accelerate the learning (intuitively, since less noise is added at each step, the model will converge faster). This approach is still subject to ensuring that a sufficient number of steps are executed, as neural networks need to perform several complete passes over the dataset.

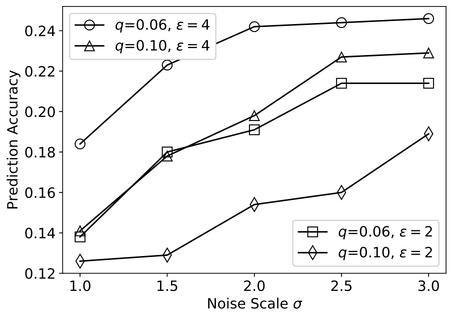


Figure 11: Effect of varying σ

Effect of Clipping norm C. We vary the clipping bound of every tensor in the model $\theta_0 = \{W, W', B'\}$. The value C represents the magnitude of per-tensor clipping, which is set to be equal for every tensor in the model. Clipping more aggressively decreases sensitivity, which in turn leads to a lower privacy budget consumption per step, and allows additional learning iterations to be executed. Conversely, setting the threshold too low also limits the amount of learning that the model can achieve per step. Figure 12 plots the obtained results for several combinations of sampling probability and grouping factor.

We observe that the for the range of values considered, the decrease in sensitivity has a more pronounced impact, and as a result the smaller clipping bounds lead to better accuracy. Of course, one cannot set the clipping bound arbitrarily low, as that will significantly curtail learning. Another factor to consider is the nature of the data, and the effect on gradient values. If the norm of the resulting tensors following gradient computation is high, then a low clipping threshold will destroy the information and prevent learning. In our case, we were able to keep the gradient norm low by using negative sampling, which in turn allowed us to obtain good accuracy for that setting. In cases where this is not possible, it is recommended to increase the clipping threshold value.

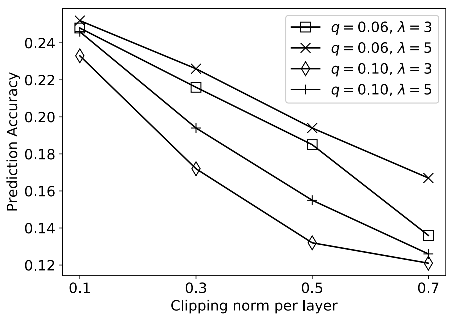


Figure 12: Effect of varying ℓ_2 clipping norm

Effect of Negative samples *neg*. In our final experiment, we investigate the effect on accuracy of negative sampling, which is an important factor in the training success of a skip-gram model. We plot the model accuracy for various values of negative sampling in Figure 13. The number of negative samples *neg* controls the total fraction of weights that are updated for each training sample, and as a side effect it helps keeping the gradient norm low. We can observe a clear 'U'-shaped dependency, reaching a maximum at $neg = 16$. The observed trend is the result of two conflicting factors: if the number of negative samples is too low, training is slowed down, due to the fact that only a small part of the layers are updated per step. Conversely, if too many samples are drawn, then the correspondingly many parameters that need to be updated lead to a large norm. Gradient clipping has an aggressive effect, and as a result, the amount of information that can be learned in each update is obliterated by the noise.

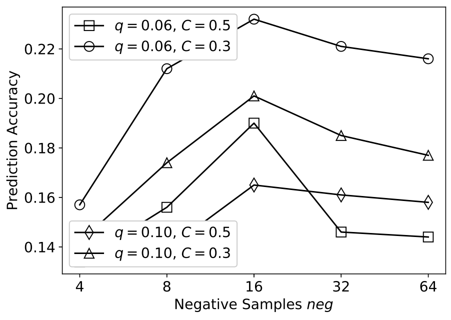


Figure 13: Effect of varying *neg*

6 RELATED WORK

Location recommendation. The problems of location recommendation and prediction have received significant attention in the last decade. Recommending a location to visit to a user necessitates modeling human mobility for the sequential prediction task. Markov Chain (MC) based methods, Matrix Factorization (MF) techniques, and Neural Network models (NN) are the schemes of choice for this objective. MC-based methods utilize a per-user transition matrix comprised of location-location transition probabilities computed from the historical record of check-ins [62]. The m^{th}-order Markov chains emit the probability of the user visiting the next location based on the latest m visited locations. Private location recommendation over Markov Chains is studied in [63]. Aggregate counts of check-ins in discretized regions are published as differentially private statistics. However, due to the sparsity in check-in behavior and the general-purpose privacy mechanisms, their method can only extend to coarse spatial decompositions (e.g., grids having larger than $5km^2$ cells). Factorizing Personalized Markov Chains (FPMC) [47] extend MC by factorizing this transition matrix for the collaborative filtering task. Matrices containing implicit user feedback on locations can also be exploited for location recommendation via weighted matrix factorization [33]. Private Matrix Factorization has been explored in [36, 51], but we are not aware of any proposal for their application to the problem we are considering. Neural Networks have become a powerful tool in recommender applications due to their flexibility, expressive power and non-linearity. Recurrent Neural Networks (RNN) can model sequential data effectively, especially language sentences [42]. Recurrent nets have also been adapted for location sequences [34, 64]. However, RNNs assume that temporal dependency changes monotonically with the position in a sequence. This is often a poor assumption in sparse location data. As a result, the state-of-art [10, 19, 35, 58] employs the skip-gram model [43] to learn the distributional context of users check-in behavior. Extensions incorporate temporal [19, 35, 61], textual [10] and other contextual features [58]. However, none of these studies provide any privacy features, which is the crux of our work.

Differential Privacy (DP) and Neural Networks. A recent focus in the differential privacy literature is to reason about cumulative privacy loss over multiple private computations given the values of ε used in each individual computation [8, 18, 44, 54]. A fundamental tool used in this task is privacy amplification via sampling [4], wherein the privacy guarantees of a private mechanism are amplified when the mechanism is applied to a small random subsample of records from a given dataset. Abadi et. al. [2] provide an amplification result for the Gaussian output perturbation mechanisms under Poisson subsampling. Their technique, called moments accountant, is based on the moment generating function of the privacy loss random variable. Other privacy definitions that lend themselves to tighter composition include Rényi Differential Privacy [44] and zero-Concentrated Differential Privacy [8], and their application to private learning with data subsampling ([54],[31] respectively). However, these privacy models are relatively new and the distinctions in privacy guarantees at the user-end remain to be investigated. In practice, (ε, δ)-differential privacy is the de-facto privacy standard [21, 39].

Location Privacy We overview literature that focuses on preventing the location based service provider (the adversary) from inferring a mobile user's location in the online setting. Spatial k-anonymity (SKA) [22] generalizes the specific position of the querying user to a region that encloses at least k users. The resulting anonymity set bounds the adversary's probability of identifying the query user to at most $1/k$. However, this syntactic notion of privacy can be easily circumvented when the data are *sparse*, i.e., the distribution of the number of location visits of an average user over the universe of POIs is long-tailed. Moreover, check-ins in sparse regions are especially vulnerable to an adversary with background knowledge, significantly increasing the probability that de-anonymization succeeds [45]. Another source of leakage is when the querying user moves, disconnecting himself from the anonymity set. DP can be used in the context of publishing statistics over datasets of locations or trajectories collected from mobile users. The Local Differential Privacy paradigm is well suited for this purpose, and its application to location data is explored in [46]. The Randomized Response mechanism is used to report, in addition to users actual locations, a large number of erroneous locations. Recommendation models that utilize these statistics can at best leverage spatial proximity queries [48] or apply to coarse spatial decompositions [46], and are incapable of cross-user learning such as in the case of the skip-gram model. Lastly, adapting the powerful guarantees of DP to protecting exact location coordinates, Geo-indistinguishability (GeoInd) [3] relaxes the DP definition to the euclidean space. It is the privacy framework of choice for obfuscating user check-ins in the absence of a trusted data curator.

Note that, SKA and GeoInd rely on obfuscating individual location records that make up the larger dataset, making them suitable only for applications that utilize spatial proximity queries (e.g., a user that sends noisy coordinates to obtain points of interest in her vicinity). Utilizing these methods to publish data for training ML models is not viable, since adding noise to the coordinates wipes out any contextual information on the POI visited (beginning with the POI identifier). Moreover, since the same user may have numerous check-in records in a longitudinal location dataset, data publishing with the common techniques suffers from serious privacy leakages. *User-level correlations* (e.g., multiple checkins of a user that are closely related) severely increase the possibility of de-anonymization in the case of SKA. Likewise, in the case of GeoInd, the cumulative privacy loss variable calculated via a standard composition theorem exceeds reasonable privacy levels.

7 CONCLUSIONS

We proposed a new approach for differentially-private next-location prediction using the skip-gram model. To the best of our knowledge, ours is the first technique that deploys DP-SGD for skip-grams. We made use of negative sampling to reduce the norms of gradient updates when dealing with high-dimensional internal neural network layers, and provided a data grouping technique that can improve the signal-to-noise ratio and allows for effective private learning. Our extensive experiments show that the proposed technique outperforms the state-of-the-art, and they also provide insights into how to tune system parameters.

Although our results focus on location data, we believe that our findings can be extended to other types of sparse data. In future work, we plan to test the viability of our approach for other learning tasks. Furthermore, we plan to investigate flexible privacy budget allocation strategies across different stages of the learning process, such that accuracy is further improved. Finally, we will study more sophisticated data grouping approaches that make informed decisions on which users to place together in the same bucket. Since such decisions are data dependent, a careful trade-off must be considered between the budget consumed performing the grouping and the remaining budget used for learning, such that prediction accuracy is maximized.

Acknowledgements. This research has been funded in part by NSF grants IIS-1910950 and IIS-1909806, the USC Integrated Media Systems Center (IMSC), and unrestricted cash gifts from Microsoft and Google. 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 National Science Foundation.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.
[2] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 308–318. ACM, 2016.
[3] M. E. Andrés, N. E. Bordenabe, K. Chatzikokolakis, and C. Palamidessi. Geo-indistinguishability: Differential privacy for location-based systems. In *ACM CCS*, 2013.
[4] B. Balle, G. Barthe, and M. Gaboardi. Privacy amplification by subsampling: Tight analyses via couplings and divergences. In *Advances in Neural Information Processing Systems*, pages 6277–6287, 2018.
[5] R. Bassily, K. Nissim, A. Smith, T. Steinke, U. Stemmer, and J. Ullman. Algorithmic stability for adaptive data analysis. In *Proceedings of ACM Symposium on Theory of Computing*, pages 1046–1059, 2016.
[6] R. Bassily, A. Smith, and A. Thakurta. Private empirical risk minimization: Efficient algorithms and tight error bounds. In *IEEE Symposium on Foundations of Computer Science*, pages 464–473, 2014.
[7] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. 2010.
[8] M. Bun and T. Steinke. Concentrated differential privacy: Simplifications, extensions, and lower bounds. In *Theory of Cryptography Conference*, pages 635–658. Springer, 2016.
[9] N. Carlini, C. Liu, J. Kos, Ú. Erlingsson, and D. Song. The secret sharer: Measuring unintended neural network memorization & extracting secrets.

arXiv preprint arXiv:1802.08232, 2018.

[10] B. Chang, Y. Park, D. Park, S. Kim, and J. Kang. Content-aware hierarchical point-of-interest embedding model for successive poi recommendation. In *IJCAI*, pages 3301–3307, 2018.

[11] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: user movement in location-based social networks. In *Proc. of ACM SIGKDD Conf. on Knowledge discovery and data mining*, pages 1082–1090, 2011.

[12] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(Aug):2493–2537, 2011.

[13] Y.-A. De Montjoye, C. A. Hidalgo, M. Verleysen, and V. D. Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific reports*, 3:1376, 2013.

[14] C. Dwork. Differential privacy: A survey of results. In *Theory and Applications of Models of Computation*, pages 1–19, 2008.

[15] C. Dwork, V. Feldman, M. Hardt, T. Pitassi, O. Reingold, and A. Roth. Generalization in adaptive data analysis and holdout reuse. In *Advances in Neural Information Processing Systems*, pages 2350–2358, 2015.

[16] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284. Springer, 2006.

[17] C. Dwork, A. Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[18] C. Dwork and G. N. Rothblum. Concentrated differential privacy. *arXiv preprint arXiv:1603.01887*, 2016.

[19] S. Feng, G. Cong, B. An, and Y. M. Chee. Poi2vec: Geographical latent representation for predicting future visitors. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[20] M. Fredrikson, S. Jha, and T. Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1322–1333. ACM, 2015.

[21] R. C. Geyer, T. Klein, and M. Nabi. Differentially private federated learning: A client level perspective. *arXiv preprint arXiv:1712.07557*, 2017.

[22] 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. ACM, 2003.

[23] M. U. Gutmann and A. Hyvärinen. Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. *Journal of Machine Learning Research*, 13(Feb):307–361, 2012.

[24] R. Gylberth, R. Adnan, S. Yazid, and T. Basaruddin. Differentially private optimization algorithms for deep neural networks. In *2017 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*, pages 387–394. IEEE, 2017.

[25] J. Hayes, L. Melis, G. Danezis, and E. De Cristofaro. Logan: Membership inference attacks against generative models. *Proceedings on Privacy Enhancing Technologies*, 2019(1):133–152, 2019.

[26] X. He, Z. He, J. Song, Z. Liu, Y.-G. Jiang, and T.-S. Chua. Nais: Neural attentive item similarity model for recommendation. *IEEE Transactions on Knowledge and Data Engineering*, 30(12):2354–2366, 2018.

[27] B. Hitaj, G. Ateniese, and F. Pérez-Cruz. Deep models under the gan: information leakage from collaborative deep learning. In *Proc. of ACM Conf. on Computer and Communications Security*, pages 603–618, 2017.

[28] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[29] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[31] J. Lee and D. Kifer. Concentrated differentially private gradient descent with adaptive per-iteration privacy budget. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1656–1665. ACM, 2018.

[32] O. Levy, Y. Goldberg, and I. Dagan. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*, 3:211–225, 2015.

[33] D. Lian, C. Zhao, X. Xie, G. Sun, E. Chen, and Y. Rui. Geomf: joint geographical modeling and matrix factorization for point-of-interest recommendation. In *Proc. of ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 831–840, 2014.

[34] Q. Liu, S. Wu, L. Wang, and T. Tan. Predicting the next location: A recurrent model with spatial and temporal contexts. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[35] X. Liu, Y. Liu, and X. Li. Exploring the context of locations for personalized location recommendations. In *IJCAI*, pages 1188–1194, 2016.

[36] Z. Liu, Y.-X. Wang, and A. Smola. Fast differentially private matrix factorization. In *Proceedings of the 9th ACM Conference on Recommender Systems*, pages 171–178. ACM, 2015.

[37] H. B. McMahan and G. Andrew. A general approach to adding differential privacy to iterative training procedures. *arXiv preprint arXiv:1812.06210*, 2018.

[38] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, et al. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629*, 2016.

[39] H. B. McMahan, D. Ramage, K. Talwar, and L. Zhang. Learning differentially private language models without losing accuracy. *ICLR*, 2018.

[40] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *IEEE Symposium on Security and Privacy*, 2019.

[41] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[42] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

[43] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[44] I. Mironov. Rényi differential privacy. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 263–275. IEEE, 2017.

[45] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, May 2008.

[46] D. Quercia, I. Leontiadis, L. McNamara, C. Mascolo, and J. Crowcroft. Spotme if you can: Randomized responses for location obfuscation on mobile phones. In *2011 31st International Conference on Distributed Computing Systems*, pages 363–372. IEEE, 2011.

[47] S. Rendle, C. Freudenthaler, and L. Schmidt-Thieme. Factorizing personalized markov chains for next-basket recommendation. In *Proc. of Intl. Conf. on World Wide Web*, pages 811–820, 2010.

[48] D. Riboni and C. Bettini. Differentially-private release of check-in data for venue recommendation. In *2014 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 190–198. IEEE, 2014.

[49] X. Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.

[50] M. Sahlgren. The distributional hypothesis. *Italian Journal of Disability Studies*, 20:33–53, 2008.

[51] H. Shin, S. Kim, J. Shin, and X. Xiao. Privacy enhanced matrix factorization for recommendation with local differential privacy. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1770–1782, 2018.

[52] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2017.

[53] S. Song, K. Chaudhuri, and A. D. Sarwate. Stochastic gradient descent with differentially private updates. In *2013 IEEE Global Conference on Signal and Information Processing*, pages 245–248. IEEE, 2013.

[54] Y.-X. Wang, B. Balle, and S. Kasiviswanathan. Subsampled renyi differential privacy and analytical moments accountant. *arXiv preprint arXiv:1808.00087*, 2018.

[55] B. J. Wilson and A. M. Schakel. Controlled experiments for word embeddings. *arXiv preprint arXiv:1510.02675*, 2015.

[56] X. Wu, M. Fredrikson, S. Jha, and J. F. Naughton. A methodology for formalizing model-inversion attacks. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 355–370. IEEE, 2016.

[57] X. Xin, X. He, Y. Zhang, Y. Zhang, and J. Jose. Relational collaborative filtering: Modeling multiple item relations for recommendation. *arXiv preprint arXiv:1904.12796*, 2019.

[58] C. Yang, L. Bai, C. Zhang, Q. Yuan, and J. Han. Bridging collaborative filtering and semi-supervised learning: A neural approach for poi recommendation. In *Proc. of ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 1245–1254, 2017.

[59] D. Yang, B. Qu, J. Yang, and P. Cudre-Mauroux. Revisiting user mobility and social relationships in lbsns: A hypergraph embedding approach. In *Proc. of Intl. Conf. on World Wide Web*, 2019.

[60] D. Yang, D. Zhang, L. Chen, and B. Qu. Nationtelescope: Monitoring and visualizing large-scale collective behavior in lbsns. *Journal of Network and Computer Applications*, 55:170–180, 2015.

[61] Q. Yuan, G. Cong, Z. Ma, A. Sun, and N. M. Thalmann. Time-aware point-of-interest recommendation. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 363–372. ACM, 2013.

[62] C. Zhang, K. Zhang, Q. Yuan, L. Zhang, T. Hanratty, and J. Han. Gmove: Group-level mobility modeling using geo-tagged social media. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1305–1314. ACM, 2016.

[63] J. D. Zhang, G. Ghinita, and C. Y. Chow. Differentially private location recommendations in geosocial networks. In *2014 IEEE 15th International Conference on Mobile Data Management*, volume 1, pages 59–68. IEEE, 2014.

[64] S. Zhao, T. Zhao, I. King, and M. R. Lyu. Geo-teaser: Geo-temporal sequential embedding rank for point-of-interest recommendation. In *Proc. of Intl, Conf. on World Wide Web*, pages 153–162, 2017.

Explaining Differences Between Unaligned Table Snapshots

Manuel Fink
Data and Web Science Group
University of Mannheim
manuel@informatik
.uni-mannheim.de

Christian Meilicke
Data and Web Science Group
University of Mannheim
christian@informatik
.uni-mannheim.de

Heiner Stuckenschmidt
Data and Web Science Group
University of Mannheim
heiner@informatik
.uni-mannheim.de

ABSTRACT

We study the problem of explaining differences between two snapshots of the same database table including record insertions, deletions and in particular record updates. Unlike existing alternatives, our solution induces transformation functions and does not require knowledge of the correct alignment between the record sets. This allows profiling snapshots of tables with unspecified or modified primary keys. In such a problem setting, there are always multiple explanations for the differences. Our goal is to find the simplest explanation. We propose to measure the complexity of explanations on the basis of minimum description length in order to formulate the task as an optimization problem. We show that the problem is NP-hard and propose a heuristic search algorithm to solve practical problem instances. We implement a prototype called Affidavit to assess the explanatory qualities of our approach in experiments based on different real-world data sets. We show that it can scale to both a large number of records and attributes and is able to reliably provide correct explanations under practical levels of modifications.

1 INTRODUCTION

When the content of a database table is frequently changing, it is difficult to find and understand the differences manually. For this reason, a large number of tools has been developed with the goal of supporting database administrators in situations like these [1, 7–9, 24]. Most of them cannot only identify deleted and inserted records but also highlight changes of individual attribute values of records that exist in both snapshots. However, the existing solutions share a big limitation. They require knowledge of the correct record alignment, usually derived from primary key attributes. In certain use cases though, immutability of primary keys is not a valid assumption. Our research is motivated by a use case of an industry project that aims to understand database updates caused by proprietary software updates. We found existing solutions not well suited because keys of the same records sometimes get reassigned during the update.

Figure 1 serves as a running example for such a problem instance. It shows two table snapshots $\mathcal{S}_1$ and $\mathcal{T}_1$ whose uncolored records have been deleted and inserted respectively. Equally colored records resemble a correctly aligned pair of records in which the record from $\mathcal{T}_1$ was derived from the record in $\mathcal{S}_1$ with the transformations specified below the table.

Snapshots $\mathcal{S}_1$ and $\mathcal{T}_1$ could belong to a company's ERP system whose database was transformed as part of an update to a newer software revision. While the attribute value changes were likely done to meet a new data format specification, the deletions and insertions constitute changes of the table content or noise from continued use of both databases between transformation and snapshotting. The company might be interested in an explanation for the changes because the conversion script is unavailable, proprietary or legacy code that is difficult to understand. A direct benefit of reverse-engineering the transformation is that, additional full system conversions can be avoided if more data needs to be transformed later on, reducing both costs and downtime.

Other application domains include data integration, e.g. duplicate detection when integrating multiple sources with redundant records in the target schema, as well as analysis of changes of third-party data sources without access to the transaction log.

What makes the running example interesting, are the changes to the composite primary key {*ID1, ID2, Date*} that make it necessary to identify other suitable attributes for record linking. *ID2* looks very promising because it is part of the primary key and has perfect discriminability and coverage [3]: The provenance of every single target record is reduced to exactly one source record. However, the correct alignment shows that these characteristics can be highly misleading. *ID2* in $\mathcal{T}_1$ was most likely filled using a skolem function [2] as part of an auto-increment policy. Linking with *Date* is another promising option, yet it would fail to explain the provenance of the three records *T13, T14, T15* in which *'99991231'* in *Date* was replaced by *'20180701'*. On the other hand, once the correct transformation function for Val has been learned, it would be very helpful for aligning the records without missing out on these three pairs. Learning this function without the alignment is difficult though.

Intuitively, we can expect at least *some* attributes to be unchanged in practice and use them to partially resolve the alignment problem. For example, *Type* and *Org* suggest an alignment of records *S11* and *T13*. The division function of *Val* implied by the input-output example *'65'* $\mapsto$ *'0.065'* generalizes to other alignment clusters, too, often resolving them.

Extending snapshot comparison with record linking and function synthesis creates a challenging duality. Scalable unsupervised record linking methods need domain knowledge on how to use the attributes to cluster records into blocks that are small enough for detailed similarity comparisons. In the case of attributes whose values have been systematically changed, algorithms that induce string transformations from examples are needed to learn how to use the attribute for blocking. However, the records need to be aligned already to produce the required input-output examples. Hence, these two sub-problems affect each other and cannot be solved independently.

The core of our contribution is an unsupervised search algorithm that iteratively learns which attributes have likely been changed and induces the corresponding value transformation functions. The resulting solution can deal with transformed or unspecified primary keys and produces more than a report of the differences. It yields an explanation that can be used to transform additional, unseen records of the source table because it generalizes the value changes instead of only listing them.

ID1	**ID2**	**Date**	Type	Val	Unit	Org
S01	**0000**	**20130416**	A	80000	USD	IBM
S02	**0001**	**20120128**	A	180000	USD	IBM
S03	**0002**	**20130315**	A	220000	USD	IBM
S04	**0003**	**20120128**	B	3780000	USD	IBM
S05	**0004**	**20120731**	B	425000	USD	IBM
S06	**0005**	**20120731**	C	21000	USD	IBM
S07	**0006**	**20140503**	C	422400	USD	IBM
S08	**0007**	**20140503**	C	6540	USD	SAP
S09	**0008**	**20131021**	C	9800	USD	SAP
S10	**0009**	**20121125**	C	0	USD	SAP
S11	**0010**	**99991231**	D	65	USD	SAP
S12	**0011**	**99991231**	D	180000	USD	BASF
S13	**0012**	**99991231**	D	220000	USD	BASF
S14	**0013**	**20150203**	D	21000	USD	BASF
S15	**0014**	**20150213**	D	65	USD	BASF
S16	**0015**	**20160807**	E	80000	USD	BASF
S17	**0016**	**20161231**	E	80000	USD	BASF

Source table $\mathcal{S}_1$

ID1	**ID2**	**Date**	Type	Val	Unit	Org
T01	**0000**	**99991231**	A	80	k $	IBM
T02	**0001**	**20120128**	A	180	k $	IBM
T03	**0002**	**20120731**	C	21	k $	IBM
T04	**0003**	**20120731**	B	425	k $	IBM
T05	**0004**	**20121125**	B	0.022	k $	DAB
T06	**0005**	**20130315**	A	220	k $	IBM
T07	**0006**	**20130416**	A	80	k $	IBM
T08	**0007**	**20131021**	C	9.8	k $	SAP
T09	**0008**	**20140503**	C	422.4	k $	IBM
T10	**0009**	**20140503**	C	6.54	k $	SAP
T11	**0010**	**20150213**	D	0.065	k $	BASF
T12	**0011**	**20161231**	E	80	k $	BASF
T13	**0012**	**20180701**	D	0.065	k $	SAP
T14	**0013**	**20180701**	D	180	k $	BASF
T15	**0014**	**20180701**	D	220	k $	BASF
T16	**0015**	**99991231**	F	0.45	k $	SAP

Target table $\mathcal{T}_1$

$$\mathcal{F}^{\mathcal{E}_1} = \left(\begin{array}{lllllll} f^{\mathcal{E}_1}_{ID1}: & \{S01 \mapsto T07, & S02 \mapsto T02, & S03 \mapsto T06, & S05 \mapsto T04, & S06 \mapsto T03, & S07 \mapsto T09, \quad S08 \mapsto T10, \\ & S09 \mapsto T08, & S11 \mapsto T13, & S12 \mapsto T14, & S13 \mapsto T15, & S15 \mapsto T11 & S17 \mapsto T12\}, \\ & \{0000 \mapsto 0006, & 0001 \mapsto 0001, & 0002 \mapsto 0005, & 0004 \mapsto 0003, & 0005 \mapsto 0002, & 0006 \mapsto 0008, \\ f^{\mathcal{E}_1}_{ID2}: & 0007 \mapsto 0009, & 0008 \mapsto 0007, & 0010 \mapsto 0012, & 0011 \mapsto 0013, & 0012 \mapsto 0014, & 0014 \mapsto 0010, \\ & 0016 \mapsto 0011\}, & f^{\mathcal{E}_1}_{Date}: & \text{‘9999123’}x & \mapsto \text{‘2018070’}x, & \textit{otherwise} & x \mapsto x, \\ f^{\mathcal{E}_1}_{Type}: & x \mapsto x, & f^{\mathcal{E}_1}_{Val}: & x \mapsto x\ /\ 1000, & f^{\mathcal{E}_1}_{Unit}: & x \mapsto \text{'k \$'}, & f^{\mathcal{E}_1}_{Org}: \quad x \mapsto x \end{array} \right)$$

Figure 1: Problem Instance $\mathcal{I}_1 = (\mathcal{S}_1, \mathcal{T}_1, \mathcal{A}_1, \mathcal{F}_1)$ that shows the content of two snapshots of the same table. Primary key attributes are bolded. The attribute functions specified in $\mathcal{F}^{\mathcal{E}_1}$ are part of a possible explanation $\mathcal{E}_1$ for the changes that uses the colored records in $\mathcal{S}_1$ to create the records of the same color in $\mathcal{T}_1$. Note that the alignment of the colored records is also given by $f^{\mathcal{E}_1}_{ID1}$. Uncolored records are records which $\mathcal{E}_1$ explains as deleted and inserted respectively.

2 RELATED WORK

The problem presented in this work extends the classic task of reporting differences of table snapshots in two dimensions: record linking and function synthesis. Handling both efficiently at once is not as trivial as combining the best solutions of both fields. It is not even straight-forward how to apply the respective state-of-the-art techniques at all in this context due to their requirements. Our contribution is an exploration of the intersection of the two problems. It is not our goal to improve the state-of-the-art in either of these fields. Instead, we aim to solve them in the context of a comparison tool that requires minimal user effort to make it practical to profile database snapshots with hundreds of tables.

Table Comparison Tools In the industry, there is a high demand for database analysis software which resulted in a large number of both free and commercial solutions for the comparison of relational database tables. Exploring this market, we found, among others, options such as ApexSQL Data Diff [1], Replicator [8], Redgate SQL Data Compare [24], Devart Data Compare [9] or SQL Delta [7]. However, to the best of our knowledge, they can only be used on tables for which a primary key is specified and none of the available products is able to cope with changes of the primary key attributes as it is common standard to use them to link the records for comparison. If they do include functionality to link records in a different way, it requires manual effort by the user, for example by explicitly defining linkage rules. Furthermore, while most of the products are able to export executable SQL scripts that implement the transformation of the data, they do not generalize well to unknown records because the value changes are explicitly stated per record and there is no learning of systematic transformation functions on the level of attributes. As a consequence, the generated reports lack an explanation of the changes in case a systematic pattern exists.

Record Linking Record linking has been frequently studied in academic research and is also known as record, entity or instance matching, identity resolution or deduplication [5]. There are several solutions available that implement both supervised and unsupervised algorithms for linking structured data, for example Magellan [17] inlcuding DeepMatcher [20], JedAI [21], dedupe [4], SILK [16], or WInte.R [18].

In a supervised setting, a set of annotated examples is given. Each example corresponds to a record described in two different representations that usually share some attributes but not necessarily the whole schema. While powerful, we do not consider techniques centered around supervision as users typically use comparison tools as a first attempt to understand changes in large amounts of data without investing manual annotation effort.

JedAI is a suite of unsupervised algorithms that do not require an annotated training dataset to link records. It contains methods for data cleaning, blocking and matching that can be configured by a user to guide the matching process. JedAI offers a rather generic approach where it is not required to define attribute-specific similarity functions. However, the lack of annotated examples means, the user needs to choose a suitable configuration of the different algorithms manually. This depends on the individual data and requires domain knowledge. Our solution is unsupervised without user guidance by using a universally applicable cost function to search for good linking criteria.

A major difference to both supervised and unsupervised approaches, is that we aim to learn transformation functions that transform records from one representation to the other. At the same time, this yields a strong criterion for the task of separating deleted or inserted records from transformed data. Most record linking algorithms however, link records purely based on a fuzzy notion of similarity. They are designed to support use cases in which no transformation function might exist, e.g. linking data from different sources.

Induction of Transformation Functions Learning string manipulations from input-output examples is a research field that has real-world applications in widespread tasks such as data preparation and spreadsheet manipulation. This is exemplified by several works of Gulwani et al. [12–14, 23] that laid the foundations for different add-ins of Microsoft Excel and Azure, marketed under the names QuickCode and FlashFill. FlashMeta [23] on the other hand, is a framework into which the authors were able to cast several different instances of the problem by separating the induction algorithm from the domain-specific language of the transformations.

The language of transformation functions that can be induced by these algorithms spans a subset of regular expressions that includes loops and conditionals. These kinds of transformations are more expressive but have many more parameters than the function families we consider in this work. As it is usually impossible to learn all function parameters from a single input-output example, a set of examples is used to unambiguously induce a specific function. Typically, a user is supposed to give a handful of examples.

Unfortunately, the authors in [26] found that current methods do not scale well to a large number of examples which led them to develop an iterative algorithm that re-uses intermediate search results from previous examples. The third-party scala re-implementation of FlashFill [19] that we were able to try was indeed too slow to be used on large tables as its induction time was in the range of seconds for a single example. On the other hand, there is no mention in [26] of how it deals with mislabeled or noisy examples, leading us to belief that it is not well suited for such scenarios. While in our setting one can reasonably expect to be able to provide a set of input-output examples that includes some correct ones, the examples can be extremely noisy due to record deletions and insertions as well as the unknown alignment of records. This makes it difficult to use state-of-the-art techniques that support complex transformations if the goal is to scale to large tables.

An alternative to induction for learning transformation functions, is the retrieval of a fitting transformation from a corpus. TDE [15] follows such an approach with $50K$ functions crawled from Github and Stackoverflow and recently showed substantially better performance than induction-based systems.

3 PROBLEM STATEMENT

Given two table snapshots with unaligned records, our goal is to explain the differences with a set of operations that transform the source snapshot to the target snapshot. Allowed operations are record insertions, deletions and transformation functions on an attribute level, for example to express a primary key mapping.

Definition 3.1. **(Problem Instance)** A *problem instance* $\mathcal{I} = (\mathcal{S}, \mathcal{T}, \mathcal{A}, \mathcal{F})$ is a set of source ($\mathcal{S}$) and target ($\mathcal{T}$) records given as value tuples under the same schema $\mathcal{A}$ which is a tuple of d attributes. $\mathcal{F} \supset \{id\}$ contains candidate transformation functions.

We will describe $\mathcal{F}$ implicitly with meta functions (see Table 1) used to solve a problem instance, such as prefixing or integer addition. $\mathcal{F}$ contains all their instantiations (i.e. parameter choices) that transform at least one source value to a target value of the same attribute, e.g. $x \mapsto x + 5$. We do not define a problem instance directly by meta functions because the space of possible explanations depends on the concrete instantiations (see Def. 4.1).

The definition of all other symbols is to be understood in relation to one specific, fixed problem instance. It should be clear from the context which problem instance they refer to.

Definition 3.2. **(Explanation)** An *explanation* is a tuple $\mathcal{E} = (\mathcal{S}^{\mathcal{E}-}, \mathcal{T}^{\mathcal{E}+}, \mathcal{F}^{\mathcal{E}})$ of source records labeled as deleted ($\mathcal{S}^{\mathcal{E}-} \in \mathcal{S}$), target records labeled as inserted ($\mathcal{T}^{\mathcal{E}+} \in \mathcal{T}$) and a tuple $\mathcal{F}^{\mathcal{E}} = (f^{\mathcal{E}}_{a_1}, \ldots, f^{\mathcal{E}}_{a_d})$ of attribute functions from $\mathcal{F}$.

Definition 3.3. **(Core)** $\mathcal{S}^{\mathcal{E}} := \mathcal{S} \setminus \mathcal{S}^{\mathcal{E}-}$ is called the *core* of an explanation $\mathcal{E}$.

The core contains all source records which are not labeled as deleted. It is used to produce the target records which are not labeled as inserted.

Definition 3.4. **(Core Image)** The result $\mathcal{T}^{\mathcal{E}}$ of applying the attribute functions of $\mathcal{F}^{\mathcal{E}}$ to the core $\mathcal{S}^{\mathcal{E}}$ is called *core image*:

$$\begin{aligned} \mathcal{T}^{\mathcal{E}} := \mathcal{F}^{\mathcal{E}}(\mathcal{S}^{\mathcal{E}}) \quad &:= \Pi_{f^{\mathcal{E}}_{a_1}, \ldots, f^{\mathcal{E}}_{a_d}}(\mathcal{S}^{\mathcal{E}}), \text{ and so for } s \in \mathcal{S}^{\mathcal{E}}: \\ \mathcal{F}^{\mathcal{E}}(s) \quad &= \left(f^{\mathcal{E}}_{a_1}(\Pi_{a_1}(s)), \ldots, f^{\mathcal{E}}_{a_d}(\Pi_{a_d}(s))\right). \end{aligned}$$

We are only interested in explanations that state the origin of every single target record, either as the result of transforming a source record or as an insertion. Moreover, we demand that each target record provenance is unambiguous by enforcing $\mathcal{F}^{\mathcal{E}}$ to be a bijection between $\mathcal{S}^{\mathcal{E}}$ and $\mathcal{T}^{\mathcal{E}}$.

Definition 3.5. **(Validity)** An explanation $\mathcal{E}$ is called *valid* if $\mathcal{T}^{\mathcal{E}+} = \mathcal{T} \setminus \mathcal{T}^{\mathcal{E}}$ and $|\mathcal{S}^{\mathcal{E}}| = |\mathcal{T}^{\mathcal{E}}|$. The set of valid explanations for a problem instance $\mathcal{I}$ is denoted by $\mathcal{E}^{\mathcal{I}}$.

From now on, when we talk about explanations, we implicitly mean valid explanations.

PROPOSITION 3.6. *Given a problem instance $\mathcal{I}$ and attribute functions $\mathcal{F}^{\mathcal{E}}$, a valid explanation $\mathcal{E}$ can be constructed from $\mathcal{F}^{\mathcal{E}}$ by appropriately choosing $\mathcal{S}^{\mathcal{E}-}$ and $\mathcal{T}^{\mathcal{E}+}$.*

PROOF. Let $\mathcal{S}^{\mathcal{E}} = \{s \mid s \in \mathcal{S} \text{ and } \mathcal{F}^{\mathcal{E}}(s) \in \mathcal{T}\}$. If multiple core records are transformed into the same target record, remove all but one such record from $\mathcal{S}^{\mathcal{E}}$. This yields $\mathcal{T}^{\mathcal{E}} = \mathcal{F}^{\mathcal{E}}(\mathcal{S}^{\mathcal{E}})$ (Definition 3.4) with $|\mathcal{S}^{\mathcal{E}}| = |\mathcal{T}^{\mathcal{E}}|$. Finally, construct $\mathcal{S}^{\mathcal{E}-} = \mathcal{S} \setminus \mathcal{S}^{\mathcal{E}}$ (Definition 3.3) and $\mathcal{T}^{\mathcal{E}+} = \mathcal{T} \setminus \mathcal{T}^{\mathcal{E}}$ (Definition 3.5). □

We use Figure 1 to elaborate on these definitions. It visualizes some problem instance $\mathcal{I}_1$ with $\mathcal{S}_1$, $\mathcal{T}_1$, $\mathcal{A}_1$ as depicted by the two tables. $\mathcal{F}_1$ could be defined implicitly by the following meta functions which have a varying number of parameters: identity, constant value, division, prefix replacement and value mappings (see Table 1).

The record coloring visualizes a possible explanation $\mathcal{E}_1$. The colored target records are producible from the source records of the same color using the specified attribute functions. $\mathcal{E}_1$ has the following formal components:

$$\begin{aligned} \mathcal{S}^{\mathcal{E}_1 -} &= \left\{ s \in \mathcal{S}_1 \mid \Pi_{\mathrm{ID1}}(s) \in \{\mathrm{S10}, \mathrm{S04}, \mathrm{S14}, \mathrm{S16}\} \right\} \\ \mathcal{T}^{\mathcal{E}_1 +} &= \left\{ t \in \mathcal{T}_1 \mid \Pi_{\mathrm{ID1}}(t) \in \{\mathrm{T01}, \mathrm{T05}, \mathrm{T16}\} \right\} \\ \mathcal{F}^{\mathcal{E}_1} &= \textit{as shown below the tables in Figure 1} \end{aligned}$$

For instance, applying its attribute functions to the first source record (S01) produces the seventh target record (T07):

$$\begin{aligned} &\mathcal{F}^{\mathcal{E}_1}((\mathrm{S01}, 0000, 20130416, \mathrm{A}, 80000, \mathrm{USD}, \mathrm{IBM})) \\ =\ &\big(f^{\mathcal{E}_1}_{ID1}(\mathrm{S01}), f^{\mathcal{E}_1}_{ID2}(0000), f^{\mathcal{E}_1}_{Date}(20130416), f^{\mathcal{E}_1}_{Type}(\mathrm{A}), \\ &\ f^{\mathcal{E}_1}_{Val}(80000), f^{\mathcal{E}_1}_{Unit}(\mathrm{USD}), f^{\mathcal{E}_1}_{Org}(\mathrm{IBM})\big) \\ =\ &(\mathrm{T07}, 0006, 20130416, \mathrm{A}, 80, \mathrm{k}\ \$, \mathrm{IBM}) \end{aligned}$$

$\mathcal{E}_1$ is a valid explanation because every target record is either producible from exactly one core record or is included in $\mathcal{T}^{\mathcal{E}_1 +}$.

3.1 Explanation Quality

The example above can be used to demonstrate that, even if systematic operations are used to change a table, it is in general impossible to be sure about the correct explanation given two snapshots. Besides $\mathcal{E}_1$, there are many more valid explanations using the same meta functions. For instance, the twelfth target record (T12) could also be created from the sixteenth (S16) source record instead of the seventeenth (S17). A second valid explanation $\mathcal{E}_2$ could therefore be constructed by adjusting the set of core records and replacing two value mappings in $f^{\mathcal{E}_1}_{ID1}$ and $f^{\mathcal{E}_1}_{ID2}$. This would not affect the brevity of the explanation. However, we would also have to replace $f^{\mathcal{E}_1}_{Date}$ with a function that additionally maps *20160807* to *20161231*. For this, we could not instantiate a simple meta function anymore and we would need a value mapping. As it would have more parameters than a prefix replacement, explanation $\mathcal{E}_2$ is less intuitive than $\mathcal{E}_1$.

Our formal definition of an explanation's quality is motivated by [11] in which the cost of a schema mapping induced from data instances is measured by the number of variables and constants in its minimum repair. Schema mapping repairs are defined as first-order formulas that state exceptions to the schema mapping to make it fit the given data example. The corresponding concept in our problem are records outside of the core whose origin can not be explained with the induced functions. The central task becomes the balancing of the size of the core with the complexity of the functions which does not have a straight-forward solution. We decide to loosely follow the concept of minimum description length [25] and prefer explanations that maximally compress the problem instance. Specifically, we evaluate the description length of our input data $\mathcal{S}$ and $\mathcal{T}$ under an explanation and ignore the contribution of inputs $\mathcal{A}$ and $\mathcal{F}$ as their description length is independent of the choice of $\mathcal{E}$.

Proposition 3.7. *$\mathcal{S}$ and $\mathcal{T}$ are implicitly described by a valid explanation $\mathcal{E} = (\mathcal{S}^{\mathcal{E}-}, \mathcal{T}^{\mathcal{E}+}, \mathcal{F}^{\mathcal{E}})$ and its core $\mathcal{S}^{\mathcal{E}}$.*

Proof. $\mathcal{S} = (\mathcal{S} \setminus \mathcal{S}^{\mathcal{E}-}) \cup \mathcal{S}^{\mathcal{E}-} \overset{(Def.\ 3.3)}{=} \mathcal{S}^{\mathcal{E}} \cup \mathcal{S}^{\mathcal{E}-}$,

$$\mathcal{T} \overset{(Def.\ 3.5)}{=} \mathcal{T}^{\mathcal{E}+} \cup \mathcal{T}^{\mathcal{E}} \overset{(Def.\ 3.4)}{=} \mathcal{T}^{\mathcal{E}+} \cup \mathcal{F}^{\mathcal{E}}(\mathcal{S}^{\mathcal{E}}) \qquad \square$$

Note that every source record is described exactly once (in the two disjoint sets $\mathcal{S}^{\mathcal{E}}$ and $\mathcal{S}^{\mathcal{E}-}$) and the choice of $\mathcal{E}$ only affects the distribution of the source records to these two sets. However, from the records in $\mathcal{T}$, only those contained in $\mathcal{T}^{\mathcal{E}+}$ need to be described. Therefore, an explanation compresses inputs $\mathcal{S}$ and $\mathcal{T}$ if its attribute functions $\mathcal{F}^{\mathcal{E}}$ can be described shorter than the core image $\mathcal{T}^{\mathcal{E}}$ which are the records from $\mathcal{S}$ and $\mathcal{T}$ that can be reconstructed from $\mathcal{E}$ and $\mathcal{S}^{\mathcal{E}}$. For this reason, we optimize the description lengths of $\mathcal{T}^{\mathcal{E}+}$ and $\mathcal{F}^{\mathcal{E}}$ but not $\mathcal{S}^{\mathcal{E}-}$ or $\mathcal{S}^{\mathcal{E}}$.

To measure the description length of the records in $\mathcal{T}^{\mathcal{E}+}$ by strictly following the definition of minimum description length, we would need to determine the minimum number of bits needed to represent $\mathcal{T}^{\mathcal{E}+}$. We decide to loosen this requirement both for semantic and practical reasons. In the context of this work, it is sufficient to count the number of data values that appear in the formal description of an explanation.

Definition 3.8. The description length of the record set $\mathcal{T}^{\mathcal{E}+}$ is defined by $\mathcal{L}(\mathcal{T}^{\mathcal{E}+}) := |\mathcal{A}| \cdot |\mathcal{T}^{\mathcal{E}+}|$.

Concerning the description length of an explanation's attribute functions, it is difficult to find a general definition that captures the brevity of a function's signature. We decide to use the smallest number of parameters that are needed to instantiate the function from a meta function, which again is a count of data values. It shall be denoted by $\psi(f)$.

Definition 3.9. The description length of an explanation's attribute functions $\mathcal{F}^{\mathcal{E}}$ is defined by $\mathcal{L}(\mathcal{F}^{\mathcal{E}}) := \sum_{a \in \mathcal{A}} \psi(f^{\mathcal{E}}_a)$.

Definition 3.10. **(Costs of Explanations)** The costs $c(\mathcal{E})$ of an explanation are defined by $c(\mathcal{E}) := 2\alpha\mathcal{L}(\mathcal{T}^{\mathcal{E}+}) + 2(1-\alpha)\mathcal{L}(\mathcal{F}^{\mathcal{E}})$.

Parameter $\alpha \in [0, 1]$ can be used to prioritize one of the components. For example, in the standard setting $\alpha = 0.5$, explanation $\mathcal{E}_1$ of Figure 1 has costs $c(\mathcal{E}_1) = \mathcal{L}(\mathcal{T}^{\mathcal{E}_1 +}) + \mathcal{L}(\mathcal{F}^{\mathcal{E}_1}) = 7|\mathcal{T}^{\mathcal{E}_1 +}| + \sum_{a \in \mathcal{A}} \psi(f^{\mathcal{E}}_a) = (7 \cdot 3) + (13 \cdot 2 + 13 \cdot 2 + 2 + 0 + 1 + 1 + 0) = 21 + 56 = 77$.

Our cost definition captures two desirable qualities of an explanation. The first term rewards explanations that use a large core to produce a big subset of the records in $\mathcal{T}$. Therefore, explanations that successfully align many records are preferred. The second term promotes simple explanations, as complicated attribute functions with many parameters, such as value mappings, are penalized for their lengthy description.

We can now formally express the requirements of an optimal solution for a problem instance.

Definition 3.11. **(Optimal Solution)** Given a problem instance $\mathcal{I}$, the set of optimal solutions is defined by $\mathcal{E}^* := \underset{\mathcal{E} \in \mathcal{E}^{\mathcal{I}}}{argmin}\ c(\mathcal{E})$.
We call the problem of finding such an optimal solution Explain-Table-Delta.

In practice, $\mathcal{E}^*$ is unlikely to contain more than one optimal explanation. In some problem instances though, multiple source records can be used to produce the same target record and more than one provenance leads to an explanation with optimal costs. Note that $\mathcal{E}^{\mathcal{I}} \neq \emptyset$ as $\mathcal{E}_{\emptyset} = (\mathcal{S}, \mathcal{T}, \{id\}^d)$ is a trivial explanation for every problem instance $\mathcal{I}$ that can always be given. It lists all source records as deleted and all target records as inserted. For example, on $\mathcal{I}_1$ and $\alpha = 0.5$, this explanation has costs $|\mathcal{A}_1| \cdot |\mathcal{T}_1| = 7 \cdot 16 = 112$.

3.2 Problem Complexity

Theorem 3.12. ***(NP-Hardness)*** *The problem Explain-Table-Delta is NP-hard for $\alpha > 0$.*

$\mathbf{c_i}$	#	v_1	v_2	v_3	v_4
$\mathbf{v_1 \vee v_2 \vee \overline{v_3}}$	c1	1	1	0	-
$\mathbf{\overline{v_1} \vee v_4}$	c2	0	-	-	1
$\mathbf{v_3}$	c3	-	-	1	-

Source records $\mathcal{S}$

#	v_1	v_2	v_3	v_4	$\mathbf{c_i}$
c1	0	0	0	-	$\mathbf{v_1 \vee v_2 \vee \overline{v_3}}$
c1	0	1	0	-	
c1	0	1	1	-	
c1	1	0	0	-	
c1	1	1	0	-	
c1	1	0	1	-	
c1	1	1	1	-	
c2	0	-	-	0	$\mathbf{\overline{v_1} \vee v_4}$
c2	0	-	-	1	
c2	1	-	-	1	
c3	-	-	1	-	$\mathbf{v_3}$

Target records $\mathcal{T}$

Figure 2: Reduction of an example 3-sat instance $c = (v_1 \vee v_2 \vee \overline{v_3}) \wedge (\overline{v_1} \vee v_4) \wedge v_3$ to a problem instance of Explain-Table-Delta with 3 source and 11 target records.

Proof. *Via polynomial-time reduction from 3-sat.*
Figure 2 shows an example reduction. Let $c = \bigwedge_{i=1}^{n} c_i$ be an instance of 3-sat with clauses c_i over variables from a set $\mathcal{V} = \{v_1, ..., v_d\}$. Then, we can create a problem instance $\mathcal{I} = (\mathcal{S}, \mathcal{T}, \mathcal{A}, \mathcal{F})$ for which the optimal solution specifies a model of c if c is satisfiable. For this, we set $\mathcal{A} := (\#, v_1, ..., v_d)$. We let $\mathcal{F}$ contain only two possible attribute functions, *id* ($x \mapsto x$) and *boolean negation* ($x \mapsto \overline{x}$). The latter shall swap the truth values {'0', '1'} and otherwise behave like *id*.

We construct $\mathcal{S}$ to have n records. For each clause c_i, $\mathcal{S}$ contains a record s_i such that $\Pi_{\#}(s_i) = \text{'c'} \circ i$ and

$$\Pi_{v_j}(s_i) = \begin{cases} \text{'1'}, & v_j \text{ positive in } c_i \\ \text{'0'}, & v_j \text{ negative in } c_i \\ \text{'-'}, & v_j \text{ not in } c_i. \end{cases}$$

$\mathcal{T}$ is constructed to contain a maximum of $7n$ records. For a clause c_i with k literals, the $2^k - 1$ different models over the variables in c_i are used to define one target record each. Let m_k be such a model for clause c_i. Then, the corresponding target record $t_{i,k}$ has $\Pi_{\#}(t_{i,k}) = \text{'c'} \circ i$. and

$$\Pi_{v_j}(t_{i,k}) = \begin{cases} \text{'1'}, & v_j \text{ positive in } c_i \text{ and } v_j \text{ true in } m_k \text{ or} \\ & v_j \text{ negative in } c_i \text{ and } v_j \text{ false in } m_k \\ \text{'0'}, & v_j \text{ positive in } c_i \text{ and } v_j \text{ false in } m_k \text{ or} \\ & v_j \text{ negative in } c_i \text{ and } v_j \text{ true in } m_k \\ \text{'-'}, & v_j \text{ not in } c_i. \end{cases}$$

The description length $\mathcal{L}(\mathcal{F}^{\mathcal{E}})$ is 0 for every explanation as the two possible functions *id* and *boolean negation* have no parameters that need to be instantiated. Consequently, for $\alpha > 0$, the costs of explanations for $\mathcal{I}$ are solely determined by $|\mathcal{T}^{\mathcal{E}+}|$.

Note that any explanation $\mathcal{E}$ can only produce exactly one target record from the source record of clause c_i because of the functionality of $\mathcal{F}^{\mathcal{E}}$. Because of attribute #, this target record needs to belong to a model of c_i, independent of the choice of $f^{\mathcal{E}}_{\#}$. For the same reason, a target record describing a model of clause c_i can only be produced by the source record corresponding to clause c_i. This means that for a fixed clause c_i, it is impossible to produce more than one target record that describes a model of c_i. Hence, each clause c_i for which $\mathcal{E}$ produces a target record from the corresponding source record reduces $|\mathcal{T}^{\mathcal{E}+}|$ by 1. An optimal solution is one that fulfills this criterion on the most clauses.

Lastly, note that $\mathcal{F}^{\mathcal{E}}$ describes an interpretation over the variables in $\mathcal{V}$. A variable $v_i \in \mathcal{V}$ in this interpretation is *true* if $f^{\mathcal{E}}_{v_i} = id$ and *false* if $f^{\mathcal{E}}_{v_i} =$ *boolean negation*. Applying $\mathcal{F}^{\mathcal{E}}$ to the source record of clause c_i produces a record that contains the truth values of all variables occuring in c_i under this interpretation. If the resulting record is a target record, the interpretation satisfies the clause as target records of c_i describe models. We can conclude that if the 3-sat instance c is satisfiable, an optimal solution $\mathcal{E}^*_0$ for the corresponding Explain-Table-Delta problem is able to produce a target record for every single clause by letting $\mathcal{F}^{\mathcal{E}^*_0}$ describe a model of c. Therefore, given the optimal solution $\mathcal{E}^*_0$, $|\mathcal{S}^{\mathcal{E}^*_0 -}| = 0$ can be used to check if c is satisfiable and if it is, a model can be extracted from $\mathcal{F}^{\mathcal{E}^*_0}$. □

4 AFFIDAVIT

In this section, we describe the components of the search algorithm presented as Algorithm 1 to solve practical instances of Explain-Table-Delta. We implement it in a prototype that is called Affidavit[1] (Algorithm For Function-Inducing Delta Analysis Via Integration of Tables). For a given problem instance, it produces an explanation that serves as an affidavit to declare that, to the best of its knowledge, the specified modifications were used to generate the target from the source table.

Thanks to Proposition 3.6, the task of finding explanations can be reduced to a search for attribute functions. Consequently, Explain-Table-Delta can be understood as a constraint satisfaction problem. If the set of possible attribute functions is finite, a brute-force solution could enumerate and assess all possible attribute function tuples by treating each attribute as a variable with domain $\mathcal{F}$. Clearly, this approach does not scale and works poorly in practice because meta functions like value mappings cause $\mathcal{F}$ to grow exponentially with the size of the data.

We propose a best-first search instead, which starts from a set of empty or partial function assignments and efficiently navigates towards a full assignment with good quality. A transition in the search space corresponds to deciding on the function of an attribute. Each of the assignments of a search state acts as a constraint on the possible alignment of source and target records. The more attribute functions have been assigned, the more it becomes clear which records belong together under these assumptions, making it easier to decide on functions for the remaining attributes. A bad function choice eventually leads to high costs because it results in a small core or complicated functions for the remaining attributes. The search is guided by estimations of the final explanation costs of partial function assignments.

4.1 Preliminaries

Definition 4.1. **(Search Space)** Given $\mathcal{I}$ with $|\mathcal{A}| = d$, the *search space* is defined by $\mathcal{H}^{\mathcal{I}} := \{(h_1, ..., h_d) \mid h_i \in \{*, \diamond\} \cup \mathcal{F}\}$.

This means, a search state $\mathcal{H} \in \mathcal{H}^{\mathcal{I}}$ is a d-tuple whose component h_i assigns either $*$, $\diamond$ or some function f from $\mathcal{F}$ to attribute a_i. In the case of $*$, the function of a_i is still undecided. A $\diamond$ means that Affidavit has identified the attribute as one for which a value mapping is best suited. It will be resolved at the very end of the search when the alignment is maximally determined.

Definition 4.2. **(End State)** $\mathcal{H}$ is called *end state* if the function of each attribute is determined, i.e. if $\{a_i \in \mathcal{A} \mid h_i \in \{*, \diamond\}\} = \emptyset$.

[1] https://github.com/Finkman7/affidavit

Algorithm 1 AFFIDAVIT

```
function AFFIDAVIT($\mathcal{I}$)
    $Q \leftarrow$ INIT-START-STATES($\mathcal{I}$)                ▷ Init Priority Queue $Q$
    while $Q \neq \emptyset$ do
        $\mathcal{H} \leftarrow$ POLL($Q$)                          ▷ Remove Best State
        if IS-END-STATE($\mathcal{H}$) then break
        else
            $Q \leftarrow Q \cup$ EXTENSIONS($\mathcal{H}$)
    return CONVERT-TO-EXPLANATION($\mathcal{H}$)   ▷ Proposition 3.6

function EXTENSIONS($\mathcal{H}$)
    $\mathcal{A}^* \leftarrow$ ORDER-BY-INDETERMINACY($\{a_i \in \mathcal{A} \mid h_i = *\}$)
    $\mathcal{H}^{ext} \leftarrow \emptyset, \mathcal{A}^\diamond \leftarrow \emptyset$          ▷ Extensions and $\diamond$-attributes
    $\mathcal{A}' \leftarrow$ POLL($\mathcal{A}^*, \beta$)                        ▷ Poll $\beta$ attributes
    $\mathcal{R} \leftarrow$ SAMPLE-RANDOM-ALIGNMENT($\Phi^{\mathcal{H}}$)
    while $\mathcal{H}^{ext} = \emptyset$ and $\mathcal{A}' \neq \emptyset$ do
        for $a$ in $\mathcal{A}'$ do
            $\mathcal{H}^a \leftarrow \emptyset$                ▷ Promising attribute extensions
            $g \leftarrow$ INDUCE-GREEDY-MAP($\mathcal{R}$, $a$)
            $\mathcal{H}_g \leftarrow$ EXTEND($\mathcal{H}$, $a$, $g$)
            for $f$ in INDUCE-FUNCTIONS($\Phi^{\mathcal{H}}$, $a$) do
                $\mathcal{H}_f \leftarrow$ EXTEND($\mathcal{H}$, $a$, $f$)
                if $c(\mathcal{H}_f) < c(\mathcal{H}_g)$ then
                    $\mathcal{H}^a \leftarrow \mathcal{H}^a \cup \{\mathcal{H}_f\}$
            if $\mathcal{H}^a \neq \emptyset$ then
                $\mathcal{H}^{ext} \leftarrow \mathcal{H}^{ext} \cup \mathcal{H}^a$
            else                 ▷ a map function is best suited for $a$
                $\mathcal{A}^\diamond \leftarrow \mathcal{A}^\diamond \cup \{a\}$
        $\mathcal{A}' \leftarrow$ POLL($\mathcal{A}^*$)                     ▷ Poll next attribute
    if $\mathcal{H}^{ext} = \emptyset$ then
        for $a$ in $\mathcal{A}^\diamond$ do                          ▷ $\mathcal{A}^\diamond = \mathcal{A}^*$
            $\mathcal{H} \leftarrow$ EXTEND($\mathcal{H}$, $a$, $\diamond$)
        $\mathcal{H}^{ext} \leftarrow$ FINALIZE($\mathcal{H}$)                  ▷ Resolve $\diamond$s
    return $\mathcal{H}^{ext}$
```

Given a search state $\mathcal{H}$, its function assignments can be used as criteria for standard blocking [10] to group source and target records together.

Definition 4.3. **(Blocking Index)** The *blocking index* of a source or target record r under a search state $\mathcal{H}$ is determined by the projection $\xi^{\mathcal{H}}$ to those attributes whose functions are already determined. In the case of source records, the attribute functions are applied during projection:

$$\xi^{\mathcal{H}} := r \mapsto \begin{cases} \Pi_{\{h_i(a_i) \mid h_i \notin \{*, \diamond\}\}} & (r) \quad \text{if } r \in \mathcal{S} \\ \Pi_{\{a_i \mid h_i \notin \{*, \diamond\}\}} & (r) \quad \text{if } r \in \mathcal{T}. \end{cases}$$

$\Xi^{\mathcal{H}}$ denotes the set of all blocking indices:

$$\Xi^{\mathcal{H}} := \{\xi^{\mathcal{H}}(s) \mid s \in \mathcal{S}\} \cup \{\xi^{\mathcal{H}}(t) \mid t \in \mathcal{T}\}.$$

To address source records, target records and the block belonging to an index κ, we define:

$$\phi^{\mathcal{H}}_{\mathcal{S}} := \kappa \mapsto \{s \in \mathcal{S} \mid \xi^{\mathcal{H}}(s) = \kappa\}$$
$$\phi^{\mathcal{H}}_{\mathcal{T}} := \kappa \mapsto \{t \in \mathcal{T} \mid \xi^{\mathcal{H}}(t) = \kappa\}$$
$$\phi^{\mathcal{H}} := \kappa \mapsto \left(\phi^{\mathcal{H}}_{\mathcal{S}}(\kappa), \phi^{\mathcal{H}}_{\mathcal{T}}(\kappa)\right).$$

Definition 4.4. **(Blocking Result)** The *blocking result* under search state $\mathcal{H}$ is the set of all blocks $\Phi^{\mathcal{H}} := \{\phi^{\mathcal{H}}(\kappa) \mid \kappa \in \Xi^{\mathcal{H}}\}$.

Figure 3 visualizes parts of an example blocking result.

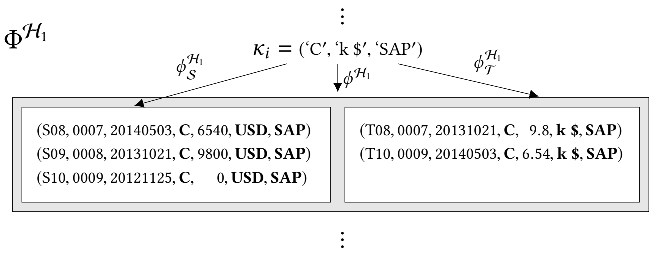

$\Phi^{\mathcal{H}_1}$
κ_i = ('C', 'k \$', 'SAP')
$\phi^{\mathcal{H}_1}_{\mathcal{S}}$
$\phi^{\mathcal{H}_1}$
$\phi^{\mathcal{H}_1}_{\mathcal{T}}$
(S08, 0007, 20140503, **C**, 6540, **USD**, **SAP**)
(S09, 0008, 20131021, **C**, 9800, **USD**, **SAP**)
(S10, 0009, 20121125, **C**, 0, **USD**, **SAP**)
(T08, 0007, 20131021, **C**, 9.8, **k \$**, **SAP**)
(T10, 0009, 20140503, **C**, 6.54, **k \$**, **SAP**)

Figure 3: A block with index κ_i within blocking result $\Phi^{\mathcal{H}_1}$ of search state $\mathcal{H}_1 := (*, *, *, id, *, x \to$ 'k \$', $id)$ on $\mathcal{I}_1$.

4.2 Initialization Strategy

A natural choice for the set of start states $\mathcal{H}^0$ of the search is $\mathcal{H}^\emptyset = \{(*, ..., *)\}$ in which all assignments are empty. While it begins the search without any wrong assumptions and can in theory lead to any explanation, it comes with a drawback. Without any assumption on how to link the records, producing input-output examples to learn the first function is very difficult.

A second natural way of beginning the search comes to mind which dampens this issue. Given the assumption that there is at least one attribute which has not been changed, the search can be started from the set

$$\mathcal{H}^{id} := \{(id, ..., *), (*, id, *, ..., *), ..., (*, ..., id)\}.$$

We prefer it to $\mathcal{H}^\emptyset$ as this assumption should be valid for nearly all practical use cases.

Furthermore, we find overlap scores another reasonable choice to determine a start state. It can drastically improve the runtime by beginning the search from a state in which most of the attributes have already been assigned. The idea is to independently assume for each attribute that it has not been changed and use it to link source and target records that have the same value on this attribute. Giving a score of 1 per attribute on which two records are identical, the score of each pair denotes their similarity in terms of an attribute overlap. Assuming that k unchanged attributes exist, the score of correctly aligned record pairs will be at least k and it is very likely that among the pairs with the highest overlap score, their large overlap will stem mainly from these attributes. We take advantage of this by using the target record with the highest overlap for each source record to build the most likely a-priory alignment over all source records. Sorting the attributes by how often their values overlap on these pairs, we use the k' most frequently overlapping ones to build a set $\mathcal{A}^{id}$. Our choice of k' is determined by the most frequent overlap score among these pairs. This leads to the set of start states

$$\mathcal{H}^s := \{(h_1, \ldots, h_d)\} \text{ with } h_i = \begin{cases} id & \text{if } a_i \in \mathcal{A}^{id} \\ * & \text{otherwise.} \end{cases}$$

To compute record overlaps without a quadratic comparison of all records, we calculate it only for record pairs that share at least one value. Very frequent attribute values that are shared by nearly every record generate an enormous amount of alignment pairs. Therefore, we ignore value overlaps in which the number of resulting pairs would be above a configurable threshold. This limits the a-priori matching to pairs that share at least one value that is not too frequent.

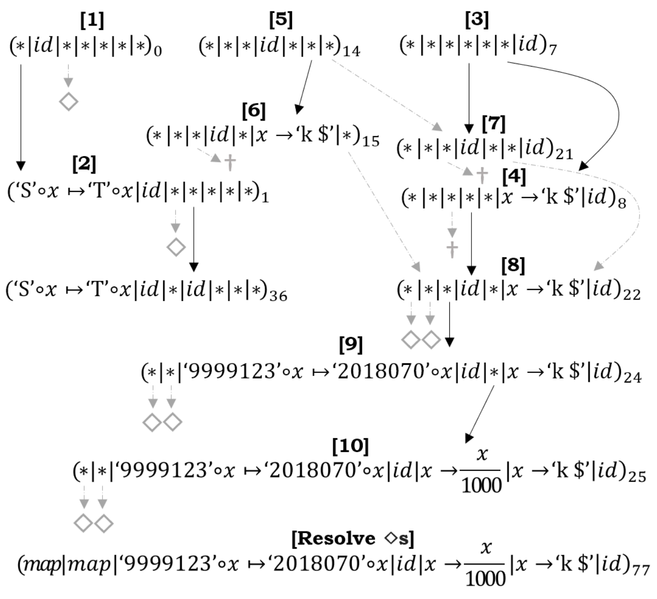

[1]
$(*|id|*|*|*|*|*)_0$
[5]
$(*|*|*|id|*|*|*)_{14}$
[3]
$(*|*|*|*|*|*|id)_7$
[6]
$(*|*|*|id|*|x \rightarrow$'k \$'$|*)_{15}$
[7]
$(*|*|*|id|*|*|id)_{21}$
[2]
$($'S'$\circ x \mapsto$'T'$\circ x|id|*|*|*|*|*)_1$
[4]
$(*|*|*|*|*|x \rightarrow$'k \$'$|id)_8$
[8]
$($'S'$\circ x \mapsto$'T'$\circ x|id|*|id|*|*|*)_{36}$
$(*|*|*|id|*|x \rightarrow$'k \$'$|id)_{22}$
[9]
$(*|*|$'9999123'$\circ x \mapsto$'2018070'$\circ x|id|*|x \rightarrow$'k \$'$|id)_{24}$
[10]
$(*|*|$'9999123'$\circ x \mapsto$'2018070'$\circ x|id|x \rightarrow \frac{x}{1000}|x \rightarrow$'k \$'$|id)_{25}$
[Resolve ◇s]
$(map|map|$'9999123'$\circ x \mapsto$'2018070'$\circ x|id|x \rightarrow \frac{x}{1000}|x \rightarrow$'k \$'$|id)_{77}$

Figure 4: Search Tree on $\mathcal{I}_1$ with $\alpha = 0.5, \beta = 2, \varrho = 3$.

4.3 Extending Search States

AFFIDAVIT discovers potential attribute functions from the blocking result $\Phi^{\mathcal{H}}$ of a search state $\mathcal{H}$ as described in Section 4.4. The resulting functions are used to extend $\mathcal{H}$ by assigning an induced function to the corresponding attribute.

To extend a state $\mathcal{H}$, we first decide on a set of attributes $\mathcal{A}'$ for which functions are induced. The decision is based on an estimation of the indeterminacy of the undecided attributes of $\mathcal{H}$. We estimate it for an attribute by computing its maximum number of distinct source values over all blocks that have both target and source records. This corresponds to an upper bound for the number of source values that need to be considered as the origin of a target value. $\mathcal{A}'$ consists of the β most determined attributes from this estimation. In the next step, we create extensions of $\mathcal{H}$ with the β most promising function candidates of each attribute in $\mathcal{A}'$. The branching factor β is configurable to limit the number of extensions that are produced.

For each $a \in \mathcal{A}'$, we compare its resulting extensions to $\mathcal{H}_g$ which is an extension of $\mathcal{H}$ on a with a map function constructed from a random alignment of all records that respects $\Phi^{\mathcal{H}}$. We build it by mapping each source value to the target value with the highest co-occurrence in the random record alignment. If $\mathcal{H}_g$ has the lowest costs, we store a in the set $\mathcal{A}^{\diamond}$ and reject the attribute's extensions, otherwise we keep all extensions with costs lower than those of $\mathcal{H}_g$. If the function of a in the optimal solution is a value mapping, there typically is no other function type that can be instantiated to a good function candidate for that attribute. This is why, once their indeterminacy is low, this check allows the detection of attributes for which map functions are likely needed.

If we did not keep any extension from the β most determined attributes, we try the next most determined one until we have found at least one extension or we have come to the conclusion that all undecided attributes should receive a map function. In the latter case, we mark all attributes from $\mathcal{A}^{\diamond}$ with $\diamond$ and finalize the state by replacing one $\diamond$ after another. The replacement is a greedy value mapping from the random alignment, just like $\mathcal{H}_g$. We re-sample a new random alignment after each $\diamond$ is replaced in order to have the next map respect the previous assignment.

Figure 4 demonstrates how AFFIDAVIT finds the optimal solution on $\mathcal{I}_1$ starting from $\mathcal{H}^0 = \mathcal{H}^{id}$. The number in square brackets indicates the order in which the states are expanded. $\mathcal{A}'$ is implicitly given by the origin of the arrows. Greyed out arrows lead to extensions that are not added to the queue because no induced function was better than a greedy map ($\diamond$), the queue was full with better states (†, see Section 4.6) or because of duplicate detection.

4.4 Inducing Functions

4.4.1 Supported Meta Functions. Our framework supports any meta function whose parameters are learnable from one input-output example. An example for such a meta function is the conversion of a date attribute. An input-output example such as *'Sep 31 2019'* $\mapsto$ *'20190931'* contains enough information to learn to split the source value by white spaces to extract month, day and year (in that order) and express the date in '*yyyymmdd*' format. Note that there can still be input-output examples of that function such as *'Oct 10 2019'* $\mapsto$ *'20191010'* for which the function instantiation is not unambiguous. It would not be clear from this example if the target format is '*yyyymmdd*' or '*yyyyddmm*'. However, one could simply generate both candidate functions or learn the function from a different input-output example. On the other hand, any linear function of the form $x \mapsto mx + c$ needs at least two input-output examples to learn its parameters m and c. After one example, the number of possible meta function instantiations is still infinitely large. There is no single example that would be enough to induce the function and it is impossible too, to generate all possible candidate functions from one example.

To transform values of an attribute that was the target of a function type that is not supported, Affidavit tries to learn the full value mapping. This way it can still produce explanations with a correct record alignment, even if a more concise function can not be learned. As a mapping with more than one entry needs an input-output example for every value it transforms, value mappings are not learned during the search like other functions. Instead, they are learned at the very end when the record alignment is maximally defined by regular functions.

4.4.2 Inducing Function Candidates. To induce functions for an attribute, AFFIDAVIT uses noisy input-output examples that it samples from blocks that have both source and target records inside them. It does this by randomly selecting up to k distinct target records from these blocks and trying for each one to produce its attribute value from the value of any source record in the same block. We do so by instantiating functions from the meta functions. For example, if target record *T08* from Figure 3 was sampled to learn functions for Val, the following functions could be induced: $x \mapsto x - 6530.2$ (from *'6540'*), $x \mapsto \frac{x}{1000}$ (from *'9800'*), $x \mapsto x + 9.8$ (from *'0'*), $x \mapsto$ *'9.8'* (from any source value).

The set of induced functions over the sampled target records is filtered to include only functions that have been generated a statistically significant amount of time. The idea behind this, is that the function of the optimal solution would be generated each time we sample a target record from the core image of the optimal solution. However, it is only generated from examples in which the effect of the optimal function is actually visible. How often it gets generated, depends on the fraction θ of records with this property in relation to the number of target records. For example, the optimal function might be the one that removes trailing zeroes if there are any but it would not be generated from correct examples without trailing zeroes in the source value.

We regard the sampling of a target record as Bernoulli experiment with success chance θ. Assuming $|\mathcal{T}| >> k$, we treat each experiment as independent, such that the number of records X from which the best function would be generated is a random variable that follows a Binomial distribution with success chance θ and sample size k. We set k to the smallest integer for which $p(X \geq 5) \geq \rho$. Both the estimated fraction θ and the confidence level ρ are configurable parameters. Choosing a larger θ speeds up the algorithm but risks that functions of the optimal solution will not be sampled if θ underestimates the amount of noise in the target records or the rarity of the function's effect. A function that was generated n times, is filtered if $p(X = n) < \rho$. If θ is set lower than the actual fraction, the function of the optimal solution will be found with a probability of a least ρ.

4.4.3 *Ranking Function Candidates.*

In the next step, we determine the best β candidate functions for each attribute. This time, the fact that some functions are not induced from all value pairs which they cover, prevents us from simply ranking the candidates by how often they were generated. While the function from the optimal solution is very likely to be contained in the candidate set after filtering, it is not necessarily the one that was generated most often.

A complete evaluation would consist in traversing all blocks and applying every function candidate to the block's source record values in order to compare the resulting histogram with the block's target values. As this can be very expensive, we use sampling to estimate the fraction of records that each function would align. This time, we sample k' distinct source records and to penalize functions that map too many source values to the same target value, we evaluate on the level of blocks that contain them instead of the individual records. In each block of a sampled source record, we apply all function candidates to every source record value of the block to create a value histogram each in which every resulting value has a frequency equal to the sum of the frequencies of all source values from which it was created. For example, $x \mapsto \frac{x}{1000}$ on block κ_i from Figure 3 results in the histogram $\{1 \times \textit{`6.54'}, 1 \times \textit{`9.8'}, 1 \times \textit{`0'}\}$, while $x \mapsto \textit{`9.8'}$ produces $\{3 \times \textit{`9.8'}\}$. For each function candidate of an attribute, we compute the overlap of the resulting histogram and the target value histogram of the block ($\{1 \times \textit{`9.8'}, 1 \times \textit{`6.54'}\}$ for $\kappa_i$) by summing up the minimum of the frequency of each value present in both histograms. On block $\kappa_i$, $x \mapsto \frac{x}{1000}$ would have an overlap of 2, whereas $x \mapsto \textit{`9.8'}$ has an overlap of 1. We rank the candidate functions in descending order by the size of their total overlap minus their description length to determine the best β candidates.

We choose the smallest integer k' for which it holds that if we use $p = \theta$ in Cochran's formula [6] for determining sample sizes:

$$k' \geq \frac{z^2 \cdot p \cdot (1-p)}{e^2}.$$

For this, we choose $z = 1.96$ and $e = 0.05$ which yields a confidence of 95% that the overlap on the sampled blocks is within ±5% of the overlap over all blocks. If $\Phi^{\mathcal{H}}$ is already very fine-grained with many blocks, this results in a huge speed-up over an evaluation over all blocks. If $\Phi^{\mathcal{H}}$ is still very vague with few blocks, we usually evaluate on many more source records than we sample because we fully evaluate the blocks in which they are contained. This makes the sampling actually less risky than the guarantees of the formula imply.

4.5 Evaluating Search States

The cost function from Definition 3.10 is defined for explanations which can only be constructed from end states. However, during the search, it is necessary to assess the quality of partial search states. In this section, we describe how we extend this definition to partial search states (and end states) in a coherent way to arrive at the cost function used by Affidavit.

The cost component $\mathcal{L}(\mathcal{F}^{\mathcal{E}})$ that measures the description length of an explanation's attribute functions has an obvious counterpart for search states:

$$c_f(\mathcal{H}) := \sum_{h_i, h_i \notin \{*, \diamond\}} \psi(h_i).$$

The value of $\mathcal{L}(\mathcal{T}^{\mathcal{E}+})$ however, depends on $|\mathcal{T}^{\mathcal{E}+}|$ which is not yet determined by a partial search state $\mathcal{H}$. A lower bound is given by the record set for which it is already clear from the partial function assignments of $\mathcal{H}$ that no source record will be aligned with it in any end state to which this search state can lead. Any record in a block without source records is such a record.

On the other hand, the blocks that do have source records can still be used to improve this lower bound. Because a valid explanation's attribute function tuple is a bijection, the number of those records can be estimated from the blocking result of $\mathcal{H}$ from the blocks which have more target than source records:

$$c_t(\mathcal{H}) := \sum_{\kappa \in \Xi^{\mathcal{H}} | \, |\phi_{\mathcal{T}}^{\mathcal{H}}(\kappa)| > |\phi_{\mathcal{S}}^{\mathcal{H}}(\kappa)|} |\phi_{\mathcal{T}}^{\mathcal{H}}(\kappa)| - |\phi_{\mathcal{S}}^{\mathcal{H}}(\kappa)|.$$

Moreover, there is an alternative way of computing $|\mathcal{T}^{\mathcal{E}+}|$ that can be useful to estimate costs during search.

Corollary 4.5. *Let* $\Delta = |\mathcal{S}| - |\mathcal{T}|$. *The validity properties can be leveraged to compute* $|\mathcal{T}^{\mathcal{E}+}|$ *in terms of* $|\mathcal{S}^{\mathcal{E}-}|$ *and* Δ.

Proof. $|\mathcal{T}^{\mathcal{E}+}| \overset{(Def.\ 3.5)}{=} |\mathcal{T}| - |\mathcal{T}^{\mathcal{E}}| = (|\mathcal{T}| + \Delta) - \Delta - |\mathcal{T}^{\mathcal{E}}|$

$= |\mathcal{S}| - \Delta - |\mathcal{T}^{\mathcal{E}}| \overset{(Def.\ 3.5)}{=} |\mathcal{S}| - \Delta - |\mathcal{S}^{\mathcal{E}}|$

$= (|\mathcal{S}| - |\mathcal{S}^{\mathcal{E}}|) - \Delta \overset{(Def.\ 3.3)}{=} |\mathcal{S}^{\mathcal{E}-}| - \Delta$ □

Just like $|\mathcal{T}^{\mathcal{E}+}|$, $|\mathcal{S}^{\mathcal{E}-}|$ cannot be completely calculated for a partial search state $\mathcal{H}$. However, as before we can compute a lower bound by using the blocking result of $\mathcal{H}$:

$$c_s(\mathcal{H}) := \sum_{\kappa \in \Xi^{\mathcal{H}} | \, |\phi_{\mathcal{S}}^{\mathcal{H}}(\kappa)| > |\phi_{\mathcal{T}}^{\mathcal{H}}(\kappa)|} |\phi_{\mathcal{S}}^{\mathcal{H}}(\kappa)| - |\phi_{\mathcal{T}}^{\mathcal{H}}(\kappa)|$$

Definition 4.6. **Costs of Search States** The cost of a search state $\mathcal{H}$ is defined by

$$c(\mathcal{H}) := 2\alpha \cdot c_f(\mathcal{H}) + 2(\alpha - 1) \cdot \max(c_t(\mathcal{H}), c_s(\mathcal{H}) - \Delta).$$

It depends on both the problem instance and the search state which lower bound underestimates less.

4.6 Queue

The best-first search tends to spend most of its time visiting search states with few assignments which is most pronounced when starting the search from $\mathcal{H}_\emptyset$ or $\mathcal{H}_{id}$. This stems from the fact that costs monotonically increase when a function is added to an undecided attribute. What makes this behavior problematic, is the fact that there are exponentially many search states from which one can reach an end state. For instance, there are $2^{|\mathcal{A}|}$ states in the search lattice on a path from $\mathcal{H}_\emptyset = (*, \ldots, *)$ to the end state $\mathcal{H}_{id} = (id, \ldots, id)$. Even with duplicate elimination, unless $\mathcal{H}_{id}$ aligns all records, a complete search would possibly try exponentially many subsets of *id* assignments to check if the remaining attributes can have other functions assigned that result in a better explanation. This behavior cripples performance as the number of attributes grows.

Fortunately, apart from (direct as well as indirect) parents of the optimal solution, it is in practice very unlikely to find many different search states that are at least as good as the optimal solution and will therefore be extracted before it. The likelihood of finding a search state with this property strongly decreases with the number of assignments of a state. For instance, setting *ID1* to *id* in $\mathcal{I}_1$ is an assignment that makes the state $(*, *, id, *, *, *, *)$ look promising at first and even still after extending it to $(*, *, id, *, *, x \mapsto \text{'}k\ \$\text{'}, *)$. These states align a lot of records with relatively cheap functions. This is why they are extracted first in Figure 4. However, the costs of states that result in an incorrect record alignment eventually increases fast when assigning functions to the remaining attributes. In this case, costly value mappings are neededÃű on attributes that could be transformed with a simple operation under the correct alignment. In addition, if the same source value is aligned with multiple different target values (which is more likely under a wrong alignment), the number of aligned records will drop even when using value mappings. For an increasing number of assignments, this makes it more and more unlikely to find states with *ID1* set to *id* that have lower costs than the optimal solution.

Therefore, we decide to use a modified priority queue that is bounded to hold $max(1, \varrho - i + 1)$ search states at the same time on the i-th level of the lattice, i.e. the level on which states have i attributes assigned. If a level is full, it only accepts a new state if it is not worse than all states on the same level. If an inserted state is accepted, it drops the worst state on the same level to make space. Polling the queue still returns the state with the lowest costs independent of the level. In case of equal costs, it returns states with a higher number of assignments first. Heuristically, it is quite unlikely in practice to skip the optimal solution due to this limitation. The most important decisions of the search happen at the early levels. While the cost of search states with only one or a few assignments might still be underestimated, a handful of assignments is in practical problem instances usually enough to identify the best foundation for inducing the remaining attribute functions.

5 EVALUATION

To evaluate Affidavit, we have implemented the meta functions described in Table 1 which include basic string and number transformations. The experiments are meant to demonstrate the core functionality and scalability of the framework. In practice, one might encounter problem instances with functions not supported by our prototype. However, we decided to evaluate on self-created problem instances based on these meta functions because this gives us certainty about the correct transformations and alignment as well as control over the degree of change and noise. This way, we can judge the given explanations in-depth. Furthermore, we can evaluate on the same table multiple times with different transformations, giving a more trust-worthy result. We describe our synthetic transformation of real-world datasets in Section 5.1 and the evaluation protocol in Section 5.2. In Section 5.3, we report about the quality of the produced explanations. Finally, we evaluate in Section 5.4 how our algorithm scales with the number of records and attributes of a problem instance.

Name	Operation	Parameters
Identity	$x \mapsto x$	–
Uppercasing	$x \mapsto \textsc{Uppercase}(x)$	–
Constant Value	$x \mapsto c$	c
Addition (Numeric)	$x \mapsto x + y$	y
Division (Numeric)	$x \mapsto x/y$	y
Front Masking	$.\{\lvert m \rvert\} \circ x \mapsto m \circ x$	m
Front Char Trimming	$[c]^* \circ x \mapsto x$	c
Prefixing	$x \mapsto y \circ x$	y
Prefix Replacement	$y \circ x \mapsto z \circ x$	y, z
Value Mappings	$x \mapsto \begin{cases} y_1 & \text{if } x = x_1 \\ \dots & \\ y_n & \text{if } x = x_n \end{cases}$	$x_1, \dots, x_n,$ $y_1, \dots, y_n$

Table 1: Meta functions implemented in Affidavit. The inverse variants of these functions are also supported, e.g. suffixing in addition to prefixing. String concatenation is denoted by $\circ$.

5.1 Datasets

We perform our experiments on the datasets[2] described in more detail in [22] which have already been used to evaluate algorithms for detecting functional dependencies. They cover a wide range of topics (e.g., flight routes, chess game logs, web log data, etc.) and feature different structural properties, both in terms of the number of attributes (5 to 223) and records (100 to 250000).

For each dataset used in [22], we create ten problem instances in three settings of varying difficulty. Each problem instance is the result of choosing some records of the table as core, transforming it with randomly sampled functions and using the remaining records as noise for the source and target snapshots. A setting consists of two parameters τ and η. The transformation percentage τ denotes the likelihood to sample a function different from *id* for an attribute. This means, it can happen that every attribute gets transformed. In this case, we reject the sampling and generate another one. To sample a function for an attribute that is to be transformed, we randomly instantiate a function from the meta functions described in Table 1. We make sure to generate functions that fit the domain of the attribute, e.g. we do not use uppercasing on numerical attributes. In the case of value mappings, we instantiate it as a random permutation of the source values. These are potentially the hardest transformations to learn due to the high number of parameters and can easily lead to a wrong alignment when confused with *id*. The noise percentage η refers to the fraction of source and target records that are outside the core of the generated problem instance.

To create two table snaphots from a dataset, we first determine the source and target noise by randomly selecting two record subsets. We choose the size of the noise sets such that these records make up a fraction η each of the resulting snapshots. The number of records in the resulting snapshots decreases to a fraction $\frac{1}{\eta+1}$ of the dataset as the noise records are distributed over both snapshots. The rest of the dataset records resembles the core of our reference explanation. We create the core image by applying the sampled transformations to the corresponding attributes of these core records. We also apply the sampled transformations

[2]https://hpi.de/naumann/projects/repeatability/data-profiling/fds.html

to the target noise as its data format should be similar to the core image records. Finally, we add the the source and target noise to the core and core image, respectively.

In addition to the random attribute transformations, we augment each dataset with a new attribute that contains a set of running integers to simulate a simple primary key. We use the same integers in both snapshots in two different permutations. The resulting attribute results in a wrong record alignment if it is used for blocking and is supposed to challenge AFFIDAVIT's ability to deal with transformed primary key attributes. If the dataset already has attributes in which the fraction of distinct attribute values is larger than 0.7, we remove these attributes for our experiments as it might make the alignment too easy when that attribute is not transformed. In Table 2, $|\mathcal{A}|$ denotes the resulting number of attributes after these modifications. Dataset attributes that are completely empty prior to the transformations are ignored as well and do not count towards this number.

5.2 Evaluation Protocol

On each problem instance, we evaluate AFFIDAVIT with two different configurations on a unix system with 24 cores at 2.6 GHz and 200GB memory. The first configuration uses $\mathcal{H}^0 = \mathcal{H}^s$ (start states) determined with a maximum block size of 100000 for the overlap matching, $\beta = 1$ (branching factor) and $\varrho = 1$ (queue width). The second configuration uses $\mathcal{H}^0 = \mathcal{H}^{id}$, $\beta = 2$ and $\varrho = 5$. Both configurations were run with $\alpha = 0.5$, $\theta = 0.1$ (core size estimation) and $\rho = 0.95$ (confidence).

We have chosen two configurations of AFFIDAVIT that resemble different approaches of tackling the problem. Using overlap sampling to begin the search from a promising start state follows the spirit of unsupervised record linking and assumes that similarities in the data can be leveraged a-priori to align the records with sufficient accuracy for the induction of transformation functions. The intention of this approach is a reduction of runtime compared to a more exhaustive search. As such, we chose to push further into that direction by limiting both branching factor and queue width to see how well one can induce the remaining functions when starting from an a-priori alignment. The resulting configuration corresponds to a greedy search that induces one attribute after another without backtracking. We compare this configuration with a search that begins with the set of start states $\mathcal{H}^{id}$ in which each state corresponds to the assumption that one particular attribute was not changed. We use parameters that allow the search to traverse a larger part of the search lattice as this setting is supposed to be more robust in exchange for runtime.

Table 2 describes the macro average over the ten problem instances per dataset per setting with four numbers comprised of runtime t, relative core size Δ_{core}, relative costs Δ_{costs} and accuracy *acc*. The latter three numbers are computed by comparing the resulting explanation $\mathcal{E}_{res}$ to the reference explanation $\mathcal{E}_{ref}$ that correctly describes the attribute functions and separation of core and noise records that were used to create the problem instance. For example, $\Delta_{core} = 0.8$ means that $\mathcal{E}_{res}$ aligned 20% less records than $\mathcal{E}_{ref}$ and $\Delta_{costs} = 0.99$ says that the cost $c(\mathcal{E}_{res})$ was 1% lower (better) than $c(\mathcal{E}_{ref})$. Accuracy is calculated by applying the learned functions $\mathcal{F}^{\mathcal{E}_{res}}$ to each core record $r \in \mathcal{S}^{\mathcal{E}_{ref}}$ and comparing it with the correct transformation $\mathcal{F}^{\mathcal{E}_{ref}}(r)$. For computing accuracy, we ignore the artificial primary key attribute that we added and measure it as the fraction of cells in $\mathcal{S}^{\mathcal{E}_{ref}}$ that are correctly translated this way.

5.3 Result Quality

The results presented in Table 2 show that $\mathcal{H}^{id}$ performs very well in the setting ($\eta = 0.3, \tau = 0.3$) as it learned to correctly translate the core in every run, with minor deviations only in the *balance* and *nursery* datasets. This is a hint that AFFIDAVIT can be used with this setting out-of-the-box to produce high quality explanations for problem instances with a reasonable amount of noise and transformations. However, we see a definite decline in accuracy on some datasets in the setting ($\eta = 0.7, \tau = 0.7$). We attribute it mainly to the high noise, as we can see on *balance*, *chess* and *nursery* that AFFIDAVIT was able to produce explanations cheaper than the reference, aligning a larger number of records by including noise in the core. This shows that our search is effective at minimizing costs but that our cost definition does not reliably lead to the correct explanation when the majority of the records are noise. Consequently, this effect is more pronounced in tables with few attributes. Nevertheless, we see on some datasets that AFFIDAVIT can still correctly learn transformations when the majority of attributes has been changed. This holds especially for tables with a large number of attributes which supports our claim that at least a handful of unchanged attributes is needed to correctly bootstrap the alignment in the beginning of the search – but not necessarily more.

As expected, we find most of the runtimes of $\mathcal{H}^s$ to be significantly lower. The performance in terms of accuracy is mostly comparable to that of $\mathcal{H}^{id}$ which shows that our use of overlap sampling is a promising way to start the search. However, we can see obvious gaps in performance on datasets such as *chess*, *letter* or *nursery*. We manually investigated this and found out that $\mathcal{H}^s$ begins the search assuming that the artificial primary key attribute was unchanged. As highlighted by $\Delta_{core} = 0$, this leads to a trivial explanation because AFFIDAVIT was not able to find functions for the remaining attributes to support anything but an empty core. The reason for this behavior is the fact that these tables contain only attributes with very few distinct values in relation to the number of records. As such, the maximum block size is exceeded when producing records pairs based on overlapping attribute values. The exception is the maximally distinct attribute of running integers which leads to a wrong a-priori alignment. Increasing the maximum block size to the point where a correct start state is found, we found the initial matching already took longer than the total runtime of the more exhaustive configuration. This shows that there are problem instances for which the search from $\mathcal{H}^{id}$ is preferable because it is independent of record similarity and can correct wrong decisions by backtracking.

5.4 Scalability

AFFIDAVIT was designed with the goal of scaling to large problem instances. In the context of database tables, this means that the runtime should increase at most linearly with both a growing number of records and attributes.

5.4.1 *Row Scalability.* We begin by experimentally measuring the scalabilty of AFFIDAVIT to tables with a large number of records. For this, we run the $\mathcal{H}^{id}$ configuration on scaled versions of a ($\eta = 0.3, \tau = 0.3$) problem instance of *flight-500k* which comes from the same source as *flight-1k* but by default has 500000 records and 20 attributes. To scale the problem instance to $x\%$ of the original size, we use $x\%$ of the core records as well as $x\%$ of the source and target noise. We then create the corresponding core image from the scaled core. The sampled transformations

				$\eta = 0.3, \tau = 0.3$				$\eta = 0.5, \tau = 0.5$				$\eta = 0.7, \tau = 0.7$			
Dataset	$\lvert\mathcal{A}\rvert$	Records	$\mathcal{H}^0$	t	Δ_{core}	Δ_{costs}	acc	t	Δ_{core}	Δ_{costs}	acc	t	Δ_{core}	Δ_{costs}	acc
iris	6	150	$\mathcal{H}^s$	0.12s	1.01	1	1	0.09s	0.99	1.01	0.99	0.10s	1.04	0.99	0.99
			$\mathcal{H}^{id}$	0.69s	1.01	1	1	0.51s	1.02	0.99	1	0.38s	1.05	0.99	0.99
balance	6	625	$\mathcal{H}^s$	0.23s	1.01	0.99	0.99	0.21s	0.96	1.02	0.92	0.19s	1.42	0.9	0.84
			$\mathcal{H}^{id}$	0.82s	1.01	0.99	0.99	0.63s	0.93	1.03	0.9	0.79s	1.44	0.89	0.86
chess	8	28056	$\mathcal{H}^s$	2.83s	0	2.11	0.43	2.16s	0.24	1.46	0.56	2.00s	0.45	1.16	0.6
			$\mathcal{H}^{id}$	7.70s	1.03	0.96	1	6.37s	1.05	0.97	0.98	12.97s	1.24	0.93	0.86
abalone	9	4177	$\mathcal{H}^s$	1.49s	0.98	1.02	1	1.01s	0.98	1.01	1	0.88s	0.82	1.04	0.89
			$\mathcal{H}^{id}$	8.70s	1	1	1	3.44s	1	1	1	3.61s	0.97	1.01	1
nursery	10	12960	$\mathcal{H}^s$	1.58s	0	2.27	0.51	1.36s	0.16	1.56	0.56	1.41s	0	1.32	0.48
			$\mathcal{H}^{id}$	4.24s	1	1.01	0.98	5.26s	0.96	1.03	0.85	4.63s	1.55	0.83	0.87
bridges	10	108	$\mathcal{H}^s$	0.05s	0.99	1.02	1	0.08s	0.96	1.04	0.99	0.08s	1.05	1.11	0.9
			$\mathcal{H}^{id}$	0.43s	1	1	1	0.50s	1	1.01	0.99	0.69s	1.15	1.04	0.96
echo	10	132	$\mathcal{H}^s$	0.07s	0.99	1.02	1	0.13s	0.93	1.06	0.98	0.11s	0.89	1.13	0.93
			$\mathcal{H}^{id}$	0.79s	0.99	1.02	1	0.89s	0.93	1.04	0.99	0.95s	0.87	1.11	0.94
breast	11	699	$\mathcal{H}^s$	0.39s	1.07	0.91	1	0.42s	1.21	0.85	0.99	0.42s	1.49	0.83	0.98
			$\mathcal{H}^{id}$	1.02s	1.1	0.86	1	1.08s	1.26	0.81	1	1.37s	1.6	0.8	0.99
adult	15	48842	$\mathcal{H}^s$	6.42s	0.96	1.06	1	5.57s	0.97	1.05	0.99	4.17s	0.99	1.03	0.97
			$\mathcal{H}^{id}$	14.33s	1	1.01	1	19.91s	0.93	1.1	0.99	17.38s	1.1	0.99	0.98
ncvoter-1k	16	1000	$\mathcal{H}^s$	0.58s	0.95	1.08	1	0.57s	0.99	1.01	1	0.85s	0.88	1.06	0.97
			$\mathcal{H}^{id}$	1.81s	0.99	1.02	1	2.33s	0.98	1.01	1	3.50s	0.87	1.07	0.96
letter	18	20000	$\mathcal{H}^s$	4.41s	0	2.65	0.86	5.04s	0.31	1.55	0.82	5.59s	0.68	1.12	0.79
			$\mathcal{H}^{id}$	12.73s	1.02	0.97	1	10.78s	1.04	0.97	1	9.40s	1.14	0.95	1
hepatitis	19	155	$\mathcal{H}^s$	0.11s	0.95	1.09	1	0.14s	0.97	1.02	1	0.19s	0.83	1.09	0.98
			$\mathcal{H}^{id}$	0.79s	0.94	1.1	1	0.71s	0.96	1.03	1	0.76s	0.82	1.09	0.97
horse	28	368	$\mathcal{H}^s$	0.23s	0.99	1.01	1	0.38s	0.89	1.09	0.99	0.56s	0.99	1.01	1
			$\mathcal{H}^{id}$	1.19s	0.97	1.06	1	1.36s	0.94	1.05	0.99	1.82s	0.82	1.07	0.98
fd-red-30	31	250000	$\mathcal{H}^s$	261.18s	1.03	1.06	1	190.49s	0.96	1.04	1	132.03s	0.98	1.01	1
			$\mathcal{H}^{id}$	281.46s	1	1	1	342.02s	1	1	1	242.51s	1	1	1
plista	43	1000	$\mathcal{H}^s$	1.70s	0.9	1.2	1	2.35s	0.89	1.1	0.99	2.52s	1.06	0.98	1
			$\mathcal{H}^{id}$	4.34s	0.98	1.05	1	6.74s	1.01	0.99	1	8.28s	0.93	1.03	0.99
flight-1k	75	1000	$\mathcal{H}^s$	2.67s	0.81	1.41	0.99	3.85s	0.68	1.3	0.98	4.82s	0.69	1.13	0.98
			$\mathcal{H}^{id}$	14.98s	1	1.01	1	26.58s	0.95	1.05	1	35.89s	0.9	1.05	0.99
uniprot	182	1000	$\mathcal{H}^s$	2.95s	0.45	2.23	0.99	2.80s	0.33	1.65	0.99	3.96s	0.77	1.1	1
			$\mathcal{H}^{id}$	49.52s	1	1.01	1	40.55s	1	1.01	1	33.70s	0.85	1.08	1

Table 2: Experimental results of two Affidavit configurations $\mathcal{H}^s$ and $\mathcal{H}^{id}$ on problem instances of varying difficulty.

stay the same. However, we remove value mapping entries defined over attribute values that do not exist anymore in the scaled version. Otherwise the costs of the reference explanation would be unnecessarily high. The resulting run times in Figure 5 confirm that Affidavit scales linearly in the number of records. Moreover, it was able to produce the reference explanation in every run on these problem instances.

5.4.2 Attribute Scalability. Attribute scalability is difficult to assess experimentally because removing attributes from a problem instance can completely alter the difficulty. However, because of the modified priority queue, we can give a rough theoretical upper-bound in ϱ for the worst-case complexity that suggests linear scalability in the number of attributes. For a fixed ϱ, Affidavit begins the search with ϱ search states with one assignment each. Ignoring duplicate elimination, in the absolute worst-case, each of these search states and its (direct and indirect) children are visited in depth-first order. Assuming $\varrho < |\mathcal{A}|$, this results in visiting level ϱ with $O(\varrho!)$ states that are each followed by $|\mathcal{A}| - \varrho$ extensions to produce a full assignment which gives $|\mathcal{A}|O(\varrho!) - \varrho O(\varrho!)$ total extensions. In the case of $\mathcal{H}^0 = \mathcal{H}^{\emptyset}$, this number is at most one larger, for $\mathcal{H}^0 = \mathcal{H}^s$ it is smaller.

Technically, there are operations inside each extension that are not constant in the number of attributes, leading to a polynomial complexity. However, during the extension of a state, the runtime is dominated by the induction of functions for a fixed attribute. As the number of attributes for which this is performed is bounded by β, a linear runtime increase with a growing number of attributes should be the result in practice. The normalized runtimes in Figure 6 support this expectation. The resulting data is very noisy though for a low number of attributes which can be explained by the fact that individual differences in difficulty of the datasets have a proportionally bigger impact on the runtime than the difference in the already low number of attributes.

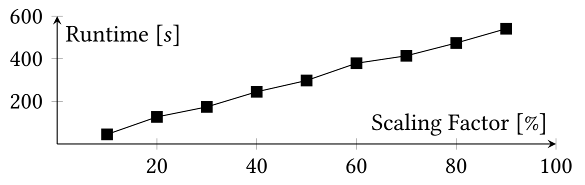


Figure 5: Runtimes on a ($\eta = 0.3, \tau = 0.3$) problem instance of *flight-500k* scaled to different numbers of records.

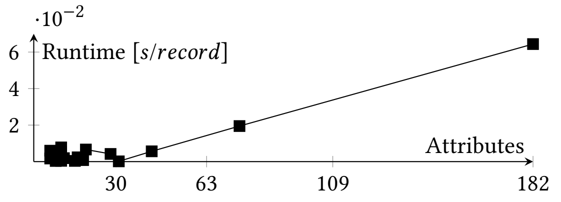


Figure 6: Runtimes of $\mathcal{H}^{id}$ taken from Table 2 for the setting ($\eta = 0.3, \tau = 0.3$) normalized by the number of records of each dataset in relation to the number of its attributes.

6 CONCLUSIONS AND FUTURE WORK

Motivated from an industrial use case in the domain of data exchange, we found a lack of solutions for reverse-engineering updates of relational tables without knowledge of the record alignment. In particular, this pertains snapshots of tables with unspecified or modified primary keys. The resulting task requires record linking and function induction at the same time. To the best of our knowledge, we presented the first theoretical framework that explores both problems at once. As there are no straight-forward criteria that define the best solution, we suggested to measure the quality of a solution on the basis of minimum description length. While we could prove that the resulting optimization problem is NP-hard, we proposed an algorithm based on a best-first search to solve practical instances of the problem. We implemented a prototype of our algorithm called Affidavit and evaluated it on several problem instances of varying difficulty based on real-world datasets. The results confirmed that Affidavit scales linearly with the number of records and attributes. Moreover, we have identified a parameter configuration that can be used out-of-the-box to reliably produce correct explanations under practical levels of noise and transformations of the data. As our algorithm is completely unsupervised, this setting can be used to compare snapshots of databases with many tables without prior linking or labeling of the data by hand.

In practical problem instances, the meta functions implemented so far, would likely not be versatile enough to explain all data transformations. In its current form, Affidavit is most usable by administrators with domain knowledge about which meta functions commonly occur in their domain. They are able to customize Affidavit by adding further meta functions via implementation of a small Java interface. In Future Work, the prototype could be updated to support a richer set of functions by default. For instance, we recently added support for date conversions. Furthermore, it would be interesting to integrate a function corpus like it was done in TDE [15] instead of manually extending the supported functions.

Future work could also investigate a problem variant without knowledge of the schema alignment. Consequently, table modifications like attribute renaming, merging or splitting could be supported. We think the insights and methods of this work would be valuable for such a task as well.

REFERENCES

[1] ApexSQL. 2019. Data Diff. Retrieved March 25, 2019 from https://www.apexsql.com/sql-tools-datadiff.aspx

[2] Patricia C. Arocena, Boris Glavic, and Renee J. Miller. 2013. Value Invention in Data Exchange. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13).* ACM, New York, NY, USA, 157–168. https://doi.org/10.1145/2463676.2465311

[3] Manuel Atencia, Jérôme David, and Jérôme Euzenat. 2014. Data interlinking through robust linkkey extraction.. In *ECAI.* 15–20.

[4] Mikhail Yuryevich Bilenko. 2006. *Learnable similarity functions and their application to record linkage and clustering.* Ph.D. Dissertation.

[5] Peter Christen. 2012. *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection.* Springer Science & Business Media.

[6] William Gemmell Cochran. 1977. Sampling Techniques (Third Edition). (1977).

[7] Australian Software Company. 2019. SQL Delta. Retrieved March 25, 2019 from http://www.sqldelta.com/

[8] Spectral Core. 2019. Replicator. Retrieved March 25, 2019 from https://www.spectralcore.com/replicator

[9] devart. 2019. Data Compare. Retrieved March 25, 2019 from http://www.devart.com/dbforge/sql/datacompare/

[10] Ivan P Fellegi and Alan B Sunter. 1969. A theory for record linkage. *J. Amer. Statist. Assoc.* 64, 328 (1969), 1183–1210.

[11] Georg Gottlob and Pierre Senellart. 2010. Schema mapping discovery from data instances. *Journal of the ACM (JACM)* 57, 2 (2010), 6.

[12] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11).* ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[13] Sumit Gulwani, William R Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105.

[14] Sumit Gulwani, Mikaël Mayer, Filip Niksic, and Ruzica Piskac. 2015. StriSynth: Synthesis for Live Programming. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15).* IEEE Press, Piscataway, NJ, USA, 701–704. http://dl.acm.org/citation.cfm?id=2819009.2819142

[15] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narasayya, and Surajit Chaudhuri. 2018. Transform-data-by-example (TDE): An Extensible Search Engine for Data Transformations. *Proc. VLDB Endow.* 11, 10 (June 2018), 1165–1177. https://doi.org/10.14778/3231751.3231766

[16] Robert Isele and Christian Bizer. 2012. Learning expressive linkage rules using genetic programming. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1638–1649.

[17] Pradap Konda, Sanjib Das, Paul Suganthan GC, AnHai Doan, Adel Ardalan, Jeffrey R Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, et al. 2016. Magellan: Toward building entity matching management systems. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1197–1208.

[18] Oliver Lehmberg, Alexander Brinkmann, and Christian Bizer. 2017. WInte.R - a web data integration framework. In *Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks co-located with 16th International Semantic Web Conference (ISWC 2017),* Vol. 1963. RWTH.

[19] Mikael Mayer. 2017. String-Solver. Retrieved April 18, 2019 from https://github.com/MikaelMayer/StringSolver

[20] 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 *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18).* ACM, New York, NY, USA, 19–34. https://doi.org/10.1145/3183713.3196926

[21] George Papadakis, Leonidas Tsekouras, Emmanouil Thanos, George Giannakopoulos, Themis Palpanas, and Manolis Koubarakis. 2018. The return of jedAI: end-to-end entity resolution for structured and semi-structured data. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1950–1953.

[22] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proceedings of the VLDB Endowment* 8, 10 (2015), 1082–1093.

[23] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015).* ACM, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

[24] Redgate. 2019. SQL Data Compare. Retrieved March 25, 2019 from http://www.red-gate.com/products/SQL_Data_Compare/index.htm

[25] Jorma Rissanen. 1983. A Universal Prior for Integers and Estimation by Minimum Description Length. *The Annals of Statistics* 11, 2 (1983), 416–431. http://www.jstor.org/stable/2240558

[26] Bo Wu and Craig A. Knoblock. 2015. An Iterative Approach to Synthesize Data Transformation Programs. In *Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI'15).* AAAI Press, 1726–1732. http://dl.acm.org/citation.cfm?id=2832415.2832489

Provenance for Probabilistic Logic Programs

Shaobo Wang
Georgetown University
sw1001@georgetown.edu

Hui Lyu
University of Pennsylvania
huilyu@seas.upenn.edu

Jiachi Zhang
Georgetown University
jz598@georgetown.edu

Chenyuan Wu
Beijing Jiao Tong University
16221213@bjtu.edu.cn

Xinyi Chen
Shanghai Jiao Tong University
cxinyic@sjtu.edu.cn

Wenchao Zhou
Georgetown University
wzhou@cs.georgetown.edu

Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

Susan B. Davidson
University of Pennsylvania
susan@seas.upenn.edu

Chen Chen
Megagon Labs
chen@megagon.ai

ABSTRACT

Despite the emergence of *probabilistic logic programming* (PLP) languages for data driven applications, there are currently no debugging tools based on provenance for PLP programs. In this paper, we propose a novel provenance model and system, called P3 (Provenance for Probabilistic logic Programs) for analyzing PLP programs. P3 enables four types of provenance queries: traditional explanation queries, queries for finding the set of most important derivations within an approximate error, top-K most influential queries, and modification queries that enable us to modify tuple probabilities with fewest modifications to program or input data. We apply these queries into real-world scenarios and present theoretical analysis and practical algorithms for such queries. We have developed a prototype of P3, and our evaluation on real-world data demonstrates that the system can support a wide-range of provenance queries with explainable results. Moreover, the system maintains provenance and execute queries efficiently with low overhead.

1 INTRODUCTION

In many data intensive applications, there has been a paradigm shift towards probabilistic and statistical reasoning. In some cases, it is in support of programs that rely on probability distributions. Probabilistic reasoning is used as a basis to trade off performance and accuracy, when collecting and aggregating readings from sensors. In other cases, probabilistic reasoning is dictated by use of external libraries, typically involving programs that rely on outputs of machine learning libraries which are intrinsically probabilistic.

Consequently, over the past few years, there is an emergence of *probabilistic logic programming* (PLP) languages. Many of these languages use database-style declarative conjunctive rules that are loosely based on Datalog semantics with extensions to handle probability. These include SLP [21], Datalog$_P$ [7], PRISM [28], ICL [23], ProbLog [24]. A common thread across these systems is that they allow both data and rules to be probabilistic, and are based on Sato's distribution semantics [27] among possible worlds under a given probability distribution. These languages cover the bulk of PLP languages in use today.

In addition, some PLP languages are used as machine learning models, such as Markov Logic Networks (MLN) [26] and Probabilistic Soft Logic (PSL) [3]. These PLP programs are combined with probabilistic graphical models (PGM), or converted to weighted Boolean formulas [6], for inference and learning. Beyond these languages, BAYONET [9], which introduces their own flavor of probabilistic network programming language.

Despite the proliferation of declarative PLP languages, there are currently no tools that enable us to debug and analyze programs. Given the declarative feature, a natural question to ask is whether *data provenance* [11] can be used for debugging these declarative data-driven systems. However, prior provenance work falls short in enabling debugging capabilities for PLP programs because they are geared primarily towards traditional relational databases. The most obvious candidate is provenance in probabilistic databases [25]. However, these systems do not work for PLP programs, given that only tuples are labeled with independent probabilities while the operators in SQL remain deterministic. This is unlike PLP programs where the rules (and hence the operators used for executing these rules) are probabilistic. Consequently, systems that support provenance in probabilistic databases do not work on queries or programs with uncertainty built into the algorithm rather than the underlying data.

In this paper, we present a model and system called P3 (Provenance for Probabilistic logic Programs). P3 enables a novel form of provenance, which we term *probabilistic provenance*. P3 is aimed at the first class of declarative PLP programs described above. All of these languages consist of programs that are a union of weighted conjunctive rules with recursion (without negation), and adhere to the possible world semantics. A representative example of this language that we use throughout our paper is ProbLog, although the approach can be generalized to similar languages. In ProbLog, tuples and rules are labeled with probabilities. ProbLog-like languages encompasses a wide range of PLP programs that involve reasoning with uncertainty over data. To the best of our knowledge, our work is the first one providing a comprehensive case study on using probabilistic provenance for PLP analysis.

By allowing PLP programs to be analyzable using probabilistic provenance, P3 enables a whole series of novel provenance queries not previously possible, including (1) showing the derivation graphs that explain tuples and their probability values; (2) finding the set of most important derivations for a derived tuple based on its provenance; (3) finding the top-K most influential variables (including base tuples and rules) for a derived tuple based on its provenance; and (4) supporting modification queries, where we can answer how to modify variables' probabilities to efficiently change the derived tuple probability to a target score with small cost.

The key contributions of this paper are as follows:

- **Probabilistic provenance model.** We propose a provenance based approach to reason PLP programs with semantics similar to ProbLog. Provenance is maintained through both graph-based representation (provenance graph) and algebraic representation (provenance polynomials).
- **Probabilistic provenance queries.** We demonstrate how the provenance model can enable provenance queries to answer explanation, derivation, influence and modification questions about derived tuples. We further demonstrate how these queries can generate meaningful results in the presence of cycles in recursive rules.
- **Implementation and evaluation.** We have implemented P3, and evaluated the system over use cases based on real-world data. Through our use cases, we also demonstrate how P3 can help debug and fix errors in ProgLog programs. Our results demonstrate that P3 can enable a range of novel provenance queries with low overhead at maintenance and query time.

2 BACKGROUND

Given our choice of ProbLog as a representative example, we first provide an introduction to the salient features of the language. ProbLog's syntax is based on Datalog, with the main difference being that all base tuple clauses and rule clauses are labeled with probabilities. A ProbLog program specifies a probability distribution over all possible non-probabilistic subprograms of the ProbLog program. The semantics of ProbLog is defined by the success probability of a query, which is the probability that the queried tuple succeeds among these subprograms.

```
rid p1: H() :- B1(),B2(),...,Bn().
tid p2: B1().
```

Figure 1: ProbLog syntax

Figure 1 summarizes the ProbLog syntax. There are two types of clauses: a weighted conjunctive rule (first line in Figure 1 where `H()` means the rule head, and `B1(),B2(),..., Bn()` are relations in the rule body), and a probabilistic base tuple (second line in Figure 1). Each conjunctive rule has a `rid`, labeled with a probability (p1 in Figure 1) of being true. Each base tuple has a `tid`, labeled with a probability score of existence (p2 in Figure 1).

2.1 Running Example

As a running example used throughout the paper, we consider the `Acquaintance` ProbLog program shown in Figure 2. The program computes all pairs of people who may know each other.

```
r1 0.8: know(P1,P2) :-
        live(P1,C), live(P2,C), P1!=P2.
r2 0.4: know(P1,P2) :-
        like(P1,L), like(P2,L), P1!=P2.
r3 0.2: know(P1,P3) :-
        know(P1,P2), know(P2,P3), P1!=P3.
t1 1.0: live("Steve","DC").
t2 1.0: live("Elena","DC").
t3 1.0: live("Mary","NYC").
t4 0.4: like("Steve","Veggies").
t5 0.6: like("Elena","Veggies").
t6 1.0: know("Ben","Steve").
```

Figure 2: `Acquaintance` ProbLog rules and base tuples

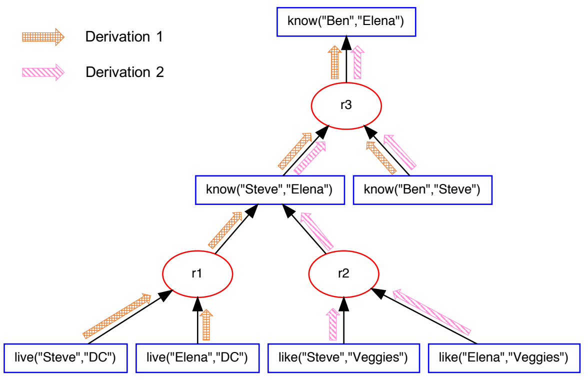


Figure 3: Provenance graph of `know("Ben","Elena")` with annotated derivations

In the `Acquaintance` program, people who live in the same place (`r1`) or like the same item (`r2`) may know each other with some probability. The `Acquaintance` relationship can also be transitive with some probability (`r3`). Given the program in Figure 2, our P3 system demonstrates how we can derive more `Acquaintance` relations based on existing relations. For example, we can infer whether `know("Ben","Elena")` can be derived by querying the tuple, and also derive the probability that Ben knows Elena.

Beyond inferring probability values, P3 will enable the following classes of queries:

- **Probability explanations.** Figure 3 is a provenance graph which explains how `know("Ben","Elena")` is derived. There are two derivations in Figure 3, annotated by two types of arrows. They share some paths to derive the tuple. We can use provenance queries to answer which derivation contributes more to the tuple probability.
- **Change modification.** After getting the probability score of `know("Ben","Elena")`, we may be dissatisfied with its score, and would like to increase or decrease it, either by changing base tuple values or rule weights. To minimize disruption, we would like make the fewest possible changes. A provenance query may be used to find out the variable that influences the derivation of `know("Ben", "Elena")` most, so that if we would like to modify its probability values with the fewest number of variable changes, we can start with the most influential variables. Likewise, we can also use provenance queries to determine the fewest number of modifications to rules.

2.2 ProbLog Semantics

As is shown in Figure 1, each clause c_i (whether a base tuple or rule) is labeled with a probability p_i. These probabilities are mutually independent. A ProbLog program $T = \{p_1 : c_1, \cdots, p_n : c_n\}$ defines a probability distribution over logic programs $L \subseteq L_T = \{c_1, \cdots, c_n\}$ as follows:

$$P(L|T) = \prod_{c_i \in L} p_i \prod_{c_i \in L_T \setminus L} (1 - p_i) \quad (1)$$

where L denotes one non-probabilistic subprogram, also called one possible world. The success probability $P(q|T)$ of a query q in a ProbLog program T is defined as:

$$P(q|L) = \begin{cases} 1, & \exists\theta : L \models q\theta \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

$$P(q, L|T) = P(q|L) \cdot P(L|T) \tag{3}$$

$$P(q|T) = \sum_{L \subseteq L_T} P(q, L|T) \tag{4}$$

The success probability of query q corresponds to the probability that the query q has a proof, given the probability distribution over logic programs. As mentioned in [24], the success probability of a ProbLog query can be computed as the probability of a Boolean monotone DNF (Disjunctive Normal Form) formula of binary variables being true, which is an NP-hard problem [29]. The DNF formula is obtained by SLD-resolution [8], and then represented by binary decision diagrams (BDDs) [4] in order to compute its probability efficiently.

The reason we choose ProbLog semantics to be the focused of this paper is as follows:

- Sato's distribution semantics or possible worlds semantics [5] is widely used in probabilistic databases, where the base tuple score has a meaningful semantics - probability. In ProbLog, the rule weight also means the probability that the rule is true.
- Provenance in probabilistic databases is intuitively easy to be extended in ProbLog's syntax and semantics.
- Alternative PLP languages (such as PRISM, ICL, SLP, and Datalog$_P$) that adhere to the Sato's distribution semantics impose extra constraints such as mutual exclusion constraint for rule bodies of the same rule head, and do not allow for recursion. However, ProbLog is more general and expressive, and it supports recursive rules which is necessary in network data applications. Hence, provenance support for ProbLog naturally carries forward to these alternative languages.

Note that while we focus on ProbLog, given ProbLog's generality, the concepts that we introduce in this paper can be generally applied to any declarative probabilistic programs that are based on union of conjunctive rules.

3 PROVENANCE MODEL

In this section, we introduce the provenance model used in P3. Traditionally, provenance can have both graph representation (provenance graph) and algebraic representation (provenance polynomials). P3 provides both types of representations, and as we will see later, different representations support different types of provenance queries.

3.1 Provenance Graph

Given a ProbLog-like PLP program, we model its provenance as a directed graph $G(V, E)$, which describes the data dependencies (see Figure 3 as an example).

The vertex set V in G consists of tuple vertices and rule execution vertices. An oval denotes a rule execution vertex, and a rectangle denotes a tuple vertex. We also annotate the associated probability for each tuple vertex and rule execution vertex. For a concise graph representation, we omit the probability values in the example provenance graph, but they are maintained as an associated attribute and can be queried.

The edge set E consists of unidirectional edges that represent data dependencies between tuple vertices and rule execution vertices. An edge is always pointing from a tuple vertex t to a rule execution vertex r, or from a rule execution vertex r to a tuple vertex t. The former indicates t is an input of r, and the latter means t is derived through r. The provenance graph G shows the complete derivation history of teh PLP program. For a queried tuple q (derivable from the PLP program), it provenance is the a subgraph of G rooted by q's corresponding tuple vertex.

For PLP programs containing recursive rules, it may generate cycles during the derivations. A cycle appears when a derived tuple can also be an input tuple in one of its own derivations. However, we prove that the cycles in the graph can be removed without affecting computing the success probability of the queried tuple q (See Section 3.3 for details).

Figure 3 shows a simple provenance graph for `Acquaintance` example. Intuitively, we can find two derivations of tuple `know("Ben","Elena")` in the graph. `know("Ben","Elena")` is derived through rule `r3` when both `know("Ben","Steve")` and `know("Steve","Elena")` are true. Then there are two ways to derive `know("Steve","Elena")`: through `r1` or through `r2`. The two derivations also share some paths.

3.2 Provenance Maintenance

The provenance maintenance for PLP shares similarities with that of ExSPAN engine [31]: we perform an automatic rule rewrite of the PLP program to create and maintain provenance information at runtime as a side-computation along with the evaluation of the original PLP program. During this process, our system naturally maintains provenance as a graph: the *direct* dependencies of the tuple and rule executions (i.e., the edges in the provenance graph) are captured and stored in relational tables.

More concretely, each rule `rid p H() :- B1(),...,Bn().` is rewritten into three rules at compile time [1]:

```
H() :- B1(),...,Bn().
prov(H(),p,rid) :- B1(),...,Bn().
rule(rid,(B1(),...,Bn())) :- B1(),...,Bn().
```

The first rule performs the original tuple derivation; the second rule records the dependency between the rule execution and its input tuples (i.e., `B1(), ..., Bn()`); the final rule records that the derived tuple `H()` has a derivation from this particular rule execution. In addition, we further identify rules labeled with probabilities and associate the rule execution vertices with their corresponding probabilities. For example, consider rule `r1` in Figure 2, its execution results in two dependencies (captured in the `prov` and `rule` table respectively): $\texttt{know(P1,P2)}\xleftarrow{0.8}\texttt{r1}$ and $\texttt{r1}\leftarrow\texttt{(live(P1,C), live(P2,C))}$. These are reflected in Derivation 1 in Figure 3.

The maintenance of provenance only adds a reasonable constant cost for each rule evaluation, and is expected to have limited impacted on the scalability of the program or application.

3.3 Provenance Querying

With the captured direct dependencies (i.e., the edges in the provenance graph), we can extract the complete provenance of a queried tuple y by recursively traversing the graph: starting from y, we traverse the graph by following the edges until we reach base tuples. The returned provenance is presented in a pre-determined representation. Generally, the graph traversal allows us to extract any provenance representation defined as a provenance semiring [11].

Provenance polynomials. In this paper, provenance polynomial is adopted as the basis for answering more complex queries (such as identifying the most influential base tuples, etc), for

[1]For performance considerations, additional optimization is adopted in the actual implementation to ensure that the rule body, which is the same for all the three rules, only needs to be evaluated once.

its close connection to probability calculation. The provenance polynomial for a queried tuple q in ProbLog is a Boolean formula $\lambda(q)$, where each literal denotes one individual tuple or rule. The literals are Boolean variables and each has some probability of being true. There are two binary operators "$\cdot$" and "+" in $\lambda(q)$. Specifically, "$\cdot$" denotes conjunctive use of multiple tuple or rules for the derivation of a derived tuple, and "+" denotes union of alternative derivations for the same derived tuple.

For example, if tuple q is derived from a conjunctive rule r_1 which takes tuple t_1 and t_2 as input, then the provenance polynomial of this derivation is $r_1 \cdot t_1 \cdot t_2$. Suppose that q has another derivation from rule r_2 which takes tuple t_3 and t_4 as input, then the complete provenance polynomial $\lambda(q)$ is $r_1 \cdot t_1 \cdot t_2 + r_2 \cdot t_3 \cdot t_4$.

Consider the `Acquaintance` example (see Figure 2), the provenance polynomial of the derived tuple `know("Ben","Elena")` is r_3 · `know("Ben","Steve")` · `know("Steve","Elena")`, where `know("Steve","Elena")` is a derived tuple that can be further expanded. The complete provenance polynomial of tuple `know("Ben","Elena")` is the following:

r_3 · `know("Ben","Steve")` ·
(r_1 · `live("Steve","DC")` · `live("Elena","DC")` +
r_2 · `like("Steve","Veggie")` · `like("Elena","Veggie")`)

This provenance polynomial represents the two derivations of `know("Ben","Elena")`.

Success probability of provenance polynomial. Noting that this example provenance polynomial consists of only base tuple literals and rule literals, we can, at least in theory, compute the success probability of the provenance polynomial. We simply treat the provenance polynomial as a formula of random variables; we can then calculate its success probability by plugging in the probabilities of the base tuples and the rules, which are provided as input to a ProbLog program. More formally,

$$\mathbf{P}[\lambda(q)] = \mathbf{E}[\widetilde{\lambda}(q)] \tag{5}$$

where $\widetilde{\lambda}$ denotes the arithmetization form of λ. $\mathbf{P}[]$ denotes success probability, and $\mathbf{E}[]$ denotes expectation.

However, computing the success probability of an arbitrary provenance polynomial is an NP-hard problem [29]. For instance, the success probability of formula $a+b$ is not the sum of the probability of a and the probability of b because of Inclusion–Exclusion principle. Therefore, in practice, we use Monte-Carlo sampling [14] approach to estimate the probability value.

Handling cycles. For a ProbLog program containing recursive rules, a cycle may appear in the provenance graph where a derived tuple can also be an input tuple for one of its own derivation. This raises issues when we retrieve provenance polynomials from the provenance graph – the provenance polynomials may potentially be arbitrarily long (by infinitely expanding the graph traversal through cycles) or contain literals that correspond to derived tuples (by stopping the graph traversal at these derived tuples). This would greatly affect the calculation of the success probability of the provenance polynomial.

However, we show that any such cycles can be removed from the provenance graph without affecting the success probability of the provenance polynomial. Generally, if the queried tuple q has cycles in its provenance graph, its provenance can be written as a polynomial in the following form:

$$\lambda(q) = (P_E + P_I) \cdot \lambda(q) + P'_E + P'_I \tag{6}$$

where P_E and P'_E each denotes a polynomial that only contains base tuples and rule literals. P_I and P'_I each denotes a polynomial containing other derived tuple literals (e.g., the provenance graph contains other cycles that do not involve q). The polynomial $(P_E + P_I) \cdot \lambda(q)$ indicates that some derivations of q depend on the existence of q itself.

Given the existence of cycles, q has infinitely many derivations: some derivations can traverse around a cycle multiple times. We define a series of formula that progressively include derivations with multiple-round cycles:

$\lambda^0(q)$ includes derivations containing no cycles. We have

$$\lambda^0(q) = P'_E + P'_I \tag{7}$$

$\lambda^1(q)$ includes derivations containing at-most-one-round cycles, that is, it includes all derivations in $\lambda^0(q)$ but also derivations containing exact one-round cycles. We have

$$\begin{aligned}
\mathbf{P}[\,\lambda^1(q)\,] &= \mathbf{P}[\,(P_E + P_I) \cdot \lambda^0(q) + P'_E + P'_I\,] && (8)\\
&= \mathbf{P}[\,(P_E + P_I) \cdot (P'_E + P'_I) + P'_E + P'_I\,] && (9)\\
&= \mathbf{P}[\,(1 + P_E + P_I) \cdot (P'_E + P'_I)\,] && (10)\\
&= \mathbf{P}[\,P'_E + P'_I\,] \text{ (Absorption Law)} && (11)
\end{aligned}$$

Thus, we have $\mathbf{P}[\,\lambda^0(q)\,] = \mathbf{P}[\,\lambda^1(q)\,]$. Likewise, we have $\mathbf{P}[\,\lambda^0(q)\,] = \mathbf{P}[\,\lambda^1(q)\,] = \mathbf{P}[\,\lambda^2(q)\,] = \cdots = \mathbf{P}[\,\lambda^\infty(q)\,] = \mathbf{P}[\,P'_E + P'_I\,]$. Therefore, if we are only concerned about calculating its success probability, we can simplify provenance polynomial for q as:

$$\lambda(q) = P'_E + P'_I \tag{12}$$

Furthermore, to simplify P'_I, if any derived tuple q_k in P'_I that has cycles in its derivation graph, we can replace $\lambda(q_k)$ with $\lambda^0(q_k)$. Then we have

$$\lambda(q) = P'_E + P'_I \mid (\,\forall \lambda(q_k), \lambda(q_k) = \lambda^0(q_k)\,) \tag{13}$$

In cases where $\lambda^0(q_k)$ also includes $\lambda(q)$, we can replace $\lambda(q)$ with $P'_E + P'_I \mid (\forall \lambda(q_k), \lambda(q_k) = \lambda^0(q_k))$ in $\lambda^0(q_k)$, and then remove $\lambda^0(q_k)$ following the same process (Equation (6) to Equation (12)). Recursively, the final provenance polynomial for q consists of only base tuple literals and rule literals.

4 REASONING PLP WITH PROVENANCE

The provenance of a queried tuple can be used in a variety of ways to reason a PLP program. For instance, users may want to know how a tuple is derived (Explanation Query), which derivation contributes most to achieving the probability of the derived tuple (Derivation Query), which base tuple has the most influence on the derivation of the queried tuple (Influence Query), and how to change the base tuples to achieve a given target value (Modification Query). These questions can help users reason and debug a PLP program. A summary of the provenance query types discussed in this paper is shown in Table 1.

4.1 Explanation Query

An *Explanation Query* returns the complete derivations of the queried tuple. The success probability of the queried tuple as well as any intermediate derived tuples can be computed efficiently using Monte-Carlo simulation. Consider the `Acquaintance` program (see Figure 2), an example Explanation Query is:

Query 1: Show the derivations of the derived tuple `know("Ben","Elena")`.

By querying the provenance of `know("Ben","Elena")`, we get provenance polynomial λ(`know("Ben","Elena")`) =

Table 1: Summary of provenance query types

Query Type	Operation
Explanation Query	Illustrate the derivations graph of the queried tuple for explanation
Derivation Query	Find the set of most important derivations of the queried tuple given some approximation error
Influence Query	Show the (top-K) most influential variables (base tuples and rules) of the queried tuple
Modification Query	Modify the variables to achieve a target probability value with minimal change

r_3 · `know("Ben","Steve")` ·
(r_1 · `live("Steve","DC")` · `live("Elena","DC")` +
r_2 · `like("Steve","Veggie")` · `like("Elena","Veggie")`)

Its success probability $\mathbf{P}[\,\lambda($`know("Ben","Elena")`$)\,] = 0.18$, meaning there is a 18% probability `Ben` knows `Elena`. The provenance polynomial also corresponds to the visual provenance graph shown in Figure 3 and explained in Section 3.1: tuple `know("Ben","Elena")` has two derivations that share parts of their paths.

4.2 Derivation Query

Sometimes, the complete explanation can be too long or too complex for users to (intuitively) understand; instead, users may be interested in a more compact explanation that still retains a *close-enough* success probability. For example, in the `Acquaintance` scenario, suppose two person `Alice` and `Bob` share many common hobbies (and thus may know each other from the same hobby groups). But if `Alice`'s and `Bob`'s addresses show that they are next-door neighbors, then this serves as a strong explanation for why `Alice` knows `Bob`; it easily trumps the share-similar-hobbies explanation which can be long and tedious.

More generally, after computing the success probability of a queried tuple, it is intuitive to ask: (a) which derivations contributed most to the success probability? (b) can we find a small set of important derivations to achieve an approximate probability? The *Derivation Query* is used to answer these types of questions. Formally, given a provenance polynomial λ, the Derivation Query returns a *sufficient provenance* λ^S:

$$|\,\mathbf{P}[\,\lambda\,] - \mathbf{P}[\,\lambda^S\,]\,| \le \epsilon \qquad (14)$$

where λ^S consists of a subset of the monomials in λ, and ϵ is a user-specified error limit. As an example, consider the `Acquaintance` program. An example Derivation Query is:

Query 2: What are the most important derivations of `know("Ben","Elena")`, assuming an error limit of ϵ?

The original provenance polynomial is given by Query 1, which consists of two monomials, corresponding to the two derivations of `know("Ben","Elena")`. The returned sufficient provenance varies with the value of ϵ: When ϵ is set to 0.001, the sufficient provenance remains the same, as removing either of the monomials would yield a success probability change greater than ϵ. After we increase ϵ to 0.01, the returned sufficient provenance λ^S(`know("Ben","Elena")`) =
r_3 · `know("Ben","Steve")` ·
r_1 · `live("Steve","DC")` · `live("Elena","DC")`
It removes one derivation and presents the most important derivation: The fact that `Steve` and `Elena` live in the same city contributes more to the derivation of `know("Steven","Elena")` (and then, in turn, the derivation of `know("Ben","Elena")`, since rule `r1` has a significantly higher probability than `r2`.

Compute sufficient provenance. For a queried tuple q, each monomial in the provenance DNF formula $\lambda(q)$ corresponds to one derivation path of q. The probability of each monomial is computed by multiplying all the probabilities of literals in the monomial. So it is easy to find the most important derivation of q. However, the probability of λ is not the sum of the probability of each monomial in λ, since these monomials can be correlated. In fact, finding the smallest sufficient provenance of λ, i.e., the $\epsilon-$approximate polynomial with the minimal number of monomials, is NP-hard [25].

A naïve way to compute sufficient provenance is to sort the monomials in the provenance polynomial according to their probabilities in a descending order. We can then progressively remove monomials that have the lowest probabilities, until the error limit ϵ is reached. This doesn't guarantee the smallest sufficient provenance but it provides an approximation. In our evaluation in Section 6, this naïve approach performs surprisingly well.

Alternatively, prior work [25] on approximate lineage for probabilistic databases proposed an algorithm to efficiently find an ϵ-approximation of provenance polynomial. Our system extends it for PLP programs. Briefly, the algorithm finds a sufficient provenance of a $k-$literal polynomial λ in the following steps:

- **Step 1.** It first finds an arbitrary match of λ: A match of λ consists of a set of *independent* monomials in λ. Since these monomials are independent, the success probability of the match can be efficiently computed.
- **Step 2.** If the match is already an $\epsilon-$approximation of λ, then the match is returned as the final result.
- **Step 3.** Otherwise, the monomials in λ are partitioned into groups, where the monomials in each group share at least one literal. Therefore, each group can be rewritten into the form $l \cdot (m_1 + m_2 + ... + m_k)$, where l is the literal shared by all monomials in the group.
- **Step 4.** The algorithm can then recursively find the sufficient provenance for $m_1 + m_2 + ... + m_k$, which is a $(k\text{-}1)-$literal polynomial. The algorithm is guaranteed to terminate at 1-literal polynomial.

Although this algorithm is more efficient than the naïve approach, it relies heavily on the choices of the match (in Step 1) and the groups (in Step 3). In some cases, it provides little reduction in the size of the provenance polynomial.

4.3 Influence Query

In addition to derivations of a queried tuple, users may also be interested in the influence of each literal on the queried tuple. An *Influence Query* returns the most influential literals (i.e., rule weight or base tuple probability) of a given derived tuple.

Intuitively, the influence of a literal x_i on a provenance polynomial λ measures the impact on the success probability of λ when the value of x_i changes. For example, a counterfactual base literal would have a large influence, because setting it to be false would invalidate the derived tuple. We adopt the definition proposed in a prior work [13]: Consider λ as a multiple-variable function, the influence of x_i is the partial derivative $\frac{\partial \lambda}{\partial x_i}$.

Definition 4.1. **Literal influence [13].** The influence of a literal x_i on the provenance polynomial λ, denoted $\text{Inf}_{x_i}(\lambda)$, is:

$$\text{Inf}_{x_i}(\lambda) = \frac{\partial \lambda}{\partial x_i} = \mathbf{P}[\lambda|_{x_i=1}] - \mathbf{P}[\lambda|_{x_i=0}] = \mathbf{E}[\widetilde{\lambda}|_{x_i=1} - \widetilde{\lambda}|_{x_i=0}] \quad (15)$$

where $\widetilde{\lambda}$ denotes the arithmetization form of λ. $\mathbf{P}[]$ denotes probability, and $\mathbf{E}[]$ denotes expectation. $\mathbf{P}[\lambda] = \mathbf{E}[\widetilde{\lambda}]$.

Based on this definition, P3 can answer Influence Query efficiently by using Monte-Carlo sampling approach to estimate $\mathbf{E}[\widetilde{\lambda}|_{x_i=1} - \widetilde{\lambda}|_{x_i=0}]$. Consider the `Acquaintance` program. An example Influence Query is:

Query 3: What are the most influential literals to the success probability of `know("Ben","Elena")`?

Table 2: Results of Influence Query

Variable	Influence Value
r_3	0.896
r_1	0.2
t_6	0.1792

Table 2 lists the top-3 most influential literal. It shows that rule `r3` is the most influential. It makes intuitive sense: Rule `r3` is the critical recursive rule that allows for the generation of multi-hop `know` relationships, including the case for `Ben` and `Elena` who know each other through their direct relationships with `Steve`.

4.4 Modification Query

For a queried tuple, our system can answer which derivations contribute most, and which individual variables are the most influential ones. Users may further explore how to effectively modify the base variables to adjust the queried tuple's success probability, for example, when the success probability of the queried tuple is suspiciously low (or high). The *Modification Query* aims to answer this type of question. We use the results from the Influence Query as a basis to answer Modification Queries.

Based on Equation (5), we can easily get

$$\mathbf{P}[\lambda_t] = \text{Inf}_{x_i}(\lambda) \cdot p(x_i) + \mathbf{P}[\lambda|_{x_i=0}] \quad (16)$$

It means that for any variable x_i, the success probability of the derived tuple is positively correlated to $p(x_i)$, and the positive coefficient is the influence value of x_i. Equation (16) effectively considers $\mathbf{P}[\lambda]$ as a function of $p(x_i)$.

Query 4: Given a target success probability of the queried tuple, how should we modify the base literals to achieve the targe with minimal cost?

Here, we consider the cost is defined as:

$$\text{Cost} = \sum_i |\Delta p(x_i)| \quad (17)$$

which is the summation of the probability change of each modified literal. To find a good solution for Query 5, We follow a heuristic-based greedy algorithm. The algorithm selects the most influential variable, and change its value to reach the target probability. Sometimes, even changing the most influential variable to the maximum value of 1 (or minimum value of 0) is still not enough. In this case, we continue to find the most influential variable in the remaining variables, until the target probability is reached. The most influential variable in each iteration corresponds to the highest slope to change, so the total cost of modification of variables is minimized. The strategy returned by this heuristic-based algorithm is not guaranteed to be an optimal solution of minimal cost, but, by leveraging the result of Influence Query, it works well empirically.

Take the `Acquaintance` program as an example. The original success probability of `know("Ben","Elena")` is 0.18. If we hope to modify the value to be above 0.5, the answer returned by P3 is that we should change variable r_3 to 0.56. The total cost of the change is 0.36. Further evaluation of the effectiveness of the heuristic-based algorithm is elaborated in Section 5.2.

5 CASE STUDY

The `Acquaintance` program is a simple example to illustrate how provenance is utilized to reason the evaluation results of ProbLog programs. In this section, we present two case studies to showcase how users can benefit from P3's practical reasoning capabilities. Performance evaluations will follow in the next section.

Visual Question Answering. Our first use case describes a scenario from multi-modal learning in the machine learning community, called Visual Question Answering (VQA). The learning task is to answer user's questions regarding a presented photo, e.g., identifying a (partially blocked) object in the photo. In this scenario, we use provenance to provide some insights from human-explainable perspective, e.g., to identify the most influential features that lead to the learning result.

Mutual Trust in Social Network. Our second use case is similar to the `Acquaintance` example, which, at the core, computes probabilistic recursive rules. In this scenario, we use sampled data from real-world trust network, and computes pair-wise mutual trust between each pair of peers in the network. We use provenance to identify critical *direct* trust relationships.

5.1 Visual Question Answering

This use case involves supporting the Visual Question Answering (VQA) [2] task using a probabilistic logic program. In VQA, the system answers a natural language question about an image. This task comgines natural language, image processing, and logical reasoning. Prior work on PSL based VQA program [1] can decompose and reason VQA task in terms of logic rules, but lacks the ability to explain the derivation procedure for the answer. Here, we rewrite the VQA-PSL program in ProbLog syntax, and provides provenance queries to explain VQA answers.

The input of this use case is a set of tuples parsed from both the image (`hasImg` relation) and the question (`hasQ` relation) with some probability. The similarity between words are also base tuples (`sim` relation). The input also includes `word` relation tuples. It is the set of answers with prior confidence scores as probability. The scores can come from some sources, e.g. a dictionary with word frequency.

The VQA-PSL program [1] is written into an equivalent VQA ProbLog program shown in Figure 5. Each rule is associated with a weight probability, which can be assigned any reasonable values. The four rules in Figure 5 explain how the final answers (`ans` relation) can be derived step-by-step combining image information, question parsing information, and words similarity knowledge.

In Figure 5, rule `r1` extends the `hasImg` relation by replacing items with their synonyms (but with a diminishing score). For example, `hasImg(V,"apple","in","background")` can be extended to `hasImg(V,"fruit","in","background")`, as `apple` is similar to `fruit` (indicated by `sim("apple","fruit")`).

Rule `r2` states that a word in the dictionary automatically becomes a candidate answer. However, it may then be out-weighted

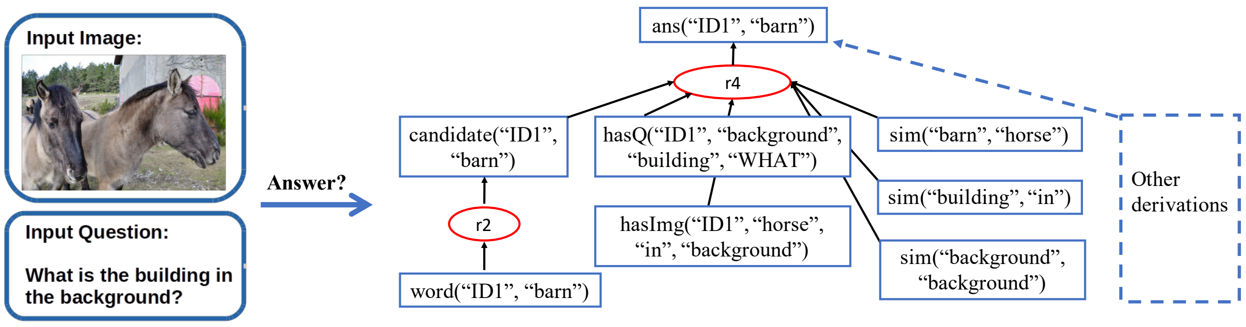

Input Image:
Input Question:
What is the building in the background?
Answer?
ans("ID1", "barn")
r4
candidate("ID1", "barn")
hasQ("ID1", "background", "building", "WHAT")
sim("barn", "horse")
sim("building", "in")
sim("background", "background")
hasImg("ID1", "horse", "in", "background")
r2
word("ID1", "barn")
Other derivations

Figure 4: Abstract provenance graph of `ans("ID1","barn")`. The complete most important derivation is shown.

by results generated by rule `r3` which further considers how the candidate answer is correlated with user's question and the objects appeared in the photo.

Finally, rule `r4` takes all the candidate answers and evaluates their correlation with the question and the objects in the photo. Stronger correlation will have higher scores. Note that we assigned equal weight to all words in the dictionary such that the predicted result is unbiased.

```
r1 w1:hasImgAns(V,Z,X1,R1,Y1) :-
      word(V,Z), hasImg(V,X1,R1,Y1),
      sim(Z,X1), sim(Z,Y1).
r2 w2:candidate(V,Z) :- word(V,Z).
r3 w3:candidate(V,Z) :- word(V,Z),
      hasQ(V,X,R,Y), hasImgAns(V,Z,X1,R1,Y1),
      sim(R,R1), sim(Y,Y1), sim(X,X1).
r4 w4:ans(V,Z) :- candidate(V,Z),
      hasQ(V,X,R,"WHAT"), hasImg(V,Z1,R1,X1),
      sim(Z,Z1), sim(R,R1), sim(X,X1).
```

Figure 5: VQA ProbLog program

Given an image and a question shown in Figure 4, the input tuples can be obtained using a image captioning system and natural language processing system. The word similarity inputs are obtained from Word2Vec library. The evaluation of the ProbLog program returns the tuple `ans("ID1","barn")` as the the result with the highest confidence, meaning that the building in the presented photo (identified by "ID1") is determined as a barn. We query the provenance of `ans("ID1","barn")` to check whether the returned provenance can give meaningful explanations.

Query 1A: Show the derivations of `ans("ID1", "barn")`. Since the provenance graph for the queried tuple has many branches, to enhance readability, we only display a condensed graph shown in Figure 4 instead of the complete fine-grained provenance graph. The actual complete graph can be generated by our system if the user hopes to take a closer look at it.

In Figure 4, we show the most important derivation to `ans("ID1", "barn")`, and leave out other derivations. In the most important derivation, word "barn" is selected as a candidate (rule `r2`) given that "barn" is included in the dictionary. Besides, "barn" is strongly correlated to the horse appeared in the photo, adding some other words similarity relation, `ans("ID1", "barn")` is derived through rule `r4`. There are other derivations that can also derive `ans("ID1", "barn")`, but they contribute less to the final probability of `ans("ID1", "barn")`.

Query 1B: Show the most influential base tuple to `ans("ID1", "barn")`. Our next query concerns the base tuple that influences the final learning result the most. In other words, we would like to understand which factor would affect

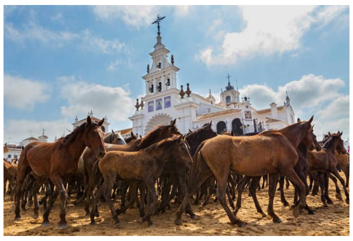


Figure 6: Image of horses in front of a church

the predicted result the most if its value was changed. We performed the influence query on the returned provenance result, then identified that the base tuple `word("ID1","barn")` is the most influential one. This is reasonable, however, less interesting as it is simply stating the obvious truth that a word must be considered as a potential answer to affect the result. As the program mainly depended on the information from `hasImg()` and `sim()` for prediction, we decided to find the most influential tuples in these two relations respectively.

We further identified the base tuple `hasImg("ID1", "horse", "in", "background")` as the most influential one in `hasImg()`. Its influence value was approximately 0.005. Effectively, this result states that the building being identified as a barn is largely influenced by the fact that there is a horse in the background. In `sim()`, the most influential tuple was `sim("barn", "horse")` with 0.03 influence value. Again, it confirmed that horse was the key factor that would affect the result the most. Both tuples coincided with human intuition while supporting the feasibility of our approach as well.

Query 1C: Show the most influential base tuple after making modifications to the input image. Our next query demonstrates the use of provenance queries for debugging. Intuitively, we expected that `ans("ID1","church")` would have the highest probability if we replaced the horses in the photo with a cross. We replaced `hasImg("ID1","horse","in","background")` with `hasImg("ID1","cross","in","background")` to mimic the modified photo aforementioned. However, the result was not as what we expected. We observed that `ans("ID1","barn")` still appeared in the evaluation result with the highest probability value. For debugging purposes, we ran the query on Figure 6 and captured the image information shown in Table 3.

We then updated the word similarity using Word2Vec. However, `ans("ID1","barn")` was still predicted as the most probable answer (i.e., the answer with the high probability). Demonstrating the generality of our approach, we use the previous queries to debug this issue. First, we ran the Derivation Query to show the most important derivations of `ans("ID1", "barn")` and `ans("ID1", "church")`. We found out that "barn" had

Table 3: Captured image information

Information	Prob.
`horse color brown`	1
`horse in field`	0.88
`cloud in sky`	0.85
`building with roof`	0.5
`cross on building`	1

very high similarities with other objects in the image ("cross": 0.30, "horse": 0.35, "cloud": 0.33), while "church" had much lower values ("cross": 0.09, "horse": 0.19, "cloud": 0.01). Surprisingly, `sim("church", "cross")` was much lower than `sim("barn", "cross")`. This explains exactly why the program did not predict the answer correctly.

Next, we would like to see how to increase the probability of `ans("ID1", "church")` with minimum cost. Our goal was to make "Church" the answer with the highest probability. So we ran the Influence Query, and selected the unique tuples that only appeared in the provenance of `ans("ID1", "church")`. The top 3 unique most influential tuples are shown in the following table:

Table 4: Top 3 unique influential tuple for `ans("ID1", "church")`

Tuple	Influence
`sim("church", "cross")`	0.04
`sim("church", "horse")`	0.02
`sim("church", "cloud")`	0.01

It is reasonable to have `sim("church", "cross")` as the most influential tuple. We set the probability of `ans("ID1", "barn")` as the target probability value for `sim("church", "cross")` and further computed the increment using the Modification Query, which returns a result value of 0.42. The value of `sim("church", "cross")` is updated to 0.51. Again, this met with our expectation that `sim("church", "cross")` should be greater than `sim("barn", "cross")`.

This use case demonstrates how multiple queries can be used in tandem to explain and debug unexpected answers caused by input data errors in probabilistic logic programs.

5.2 Mutual Trust in Social Network

For this graph network reachability use case, we use the Bitcoin OTC trust weighted signed network dataset[2]. This is a who-trusts-whom network of people who trade using Bitcoin on a platform called Bitcoin OTC [15, 16]. Each weighted edge in the network graph represents the trust between two users with the degree of trust as weight value. To fit the data in our probabilistic setting, we re-scale the weights of edges from [-10,10] to [0,1] to represent the probability score of trust.

In this use case, the input tuples are `trust` relations between people. For example, tuple `trust(1,2)` of score 0.7 means Person 1 trusts Person 2 with probability 0.7. We introduce the ProbLog program shown in Figure 7 to find `trustPath` and `mutualTrustPath` tuples that can be derived. Rule `r1` is a base case that each `trust` relation tuple is also a one-hop `trustPath` relation tuple. Rule `r2` is a recursive rule used to derive all the reachable `trustPath` tuples. Rule `r3` defines `mutualTrustPath` tuples that can be derived when the `trustPath` between any two nodes are bi-directional. The `Trust` ProbLog program can help finding any indirect trust relations between people, which is of significant interest during actual transactions.

```
r1 1.0: trustPath(P1,P2) :- trust(P1,P2).
r2 1.0: trustPath(P1,P3) :-
        trust(P1,P2), trustPath(P2,P3),
        P1!=P3.
r3 0.8: mutualTrustPath(P1,P2) :-
        trustPath(P1,P2),trustPath(P2,P1).
```

Figure 7: `Trust` ProbLog program

The original network graph is quite large, so for quality check purpose, we use a sample of 30 nodes to evaluate the provenance query results.

Query 2A: Show the derivations of `mutualTrustPath(1,6)`. The answer returned by P3 system is the provenance graph for queried tuple `mutualTrustPath(1,6)`, shown in Figure 8. The provenance graph shows Person 1 and Person 6 mutually trust each other because of the existence of trust paths from 6 to 1 (denoted by `trustPath(1,6)`) and from 1 to 6 (denoted by `trustPath(6,1)`). It further shows that there is one single derivation for `trustPath(6,1)` where Person 2 is the middle man that connects this trust path; on the other hand, `trustPath(1,6)` has two derivations, namely through path $1 \rightarrow 2 \rightarrow 6$ or path $1 \rightarrow 13 \rightarrow 2 \rightarrow 6$.

Query 2B: Show the most influential base tuple for `mutualTrustPath(1,6)`.
Next, we perform a query to understand which base tuples, i.e., the `trust` tuples, are most critical to the mutual trust between Person 1 and Perosn 6. We initialize the base tuples with probability values shown as follows (we omit the ones that are not involved in the derivation of `mutualTrustPath(1,6)`):

Table 5: Initial probability values of base tuples

Literal	Prob.	Literal	Prob.
`trust(1,2)`	0.9	`trust(1,13)`	0.65
`trust(2,1)`	0.9	`trust(2,6)`	0.75
`trust(6,2)`	0.7	`trust(13,2)`	0.6

Given this initialization, the P3 system returns that `trust(6,2)` is the most influential literal with an influence value of 0.51. The second most influential literal is `trust(2,6)` with an influence value of 0.48. Observing the provenance structure in Figure 8, we find that this result comply with human's intuition: `trust(6,2)` is influential[3] because its existence is the basis of the trust path from Person 6 to Person 1; `trust(2,6)` is also influential, more so than `trust(1,2)` and `trust(1,13)`, for the existence of the trust path from Person 1 to Person 6.

More intuitively, the result indicates that if Person 6 wants to increase the mutual trust to Person 1 without directly reaching him, strengthening the trust to Person 2 might be the most effective approach.

Query 2C: Show the optimal way to increase the probability of `mutualTrustPath(1,6)`. Here, we assume that each literal's probability can reach to 1.0 as maximum. The original **P**[mutualTrustPath(1, 6)] is 0.3524. Now we want to approximately double it to a target value 0.7. The strategy returned by P3 is as follows:

[2]https://snap.stanford.edu/data/soc-sign-bitcoin-otc.html

[3]`trust(6,2)` is considered to have a higher influence value than `trust(2,1)` due to the initial probability assignment: whether `trust(6,2)` exist or not has a higher influence because its existence would *almost* imply the existence of the trust path from 6 to 1 as **P**[`trust(2,1)`] is very close to 1.

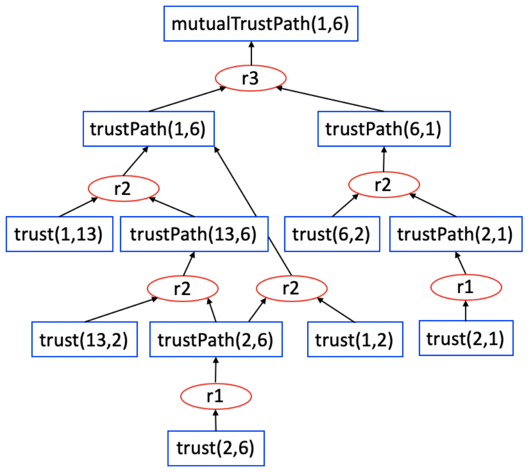


Figure 8: Provenance graph of queried tuple `mutualTrustPath(1,6)`

Table 6: Optimal strategy (total change = 0.58)

	Literal	Change	Overall prob.
Step 1	`trust(6,2)`	0.7 ⟶ 1.0	0.51
Step 2	`trust(2,6)`	0.75 ⟶ 1.0	0.68
Step 3	`trust(2,1)`	0.9 ⟶ 0.93	0.7

To show the effectiveness of our heuristic-based algorithm, we compare the cost, in terms of total change in base tuple's probability values, of its strategy with another randomly generated strategy where a random base tuple is chosen to update in each step. The result of a random strategy is shown as follows:

Table 7: Random strategy (total change = 1.36)

	Literal	Change	Overall prob.
Step 1	`trust(1,13)`	0.65 ⟶ 1.0	0.37
Step 2	`trust(13,2)`	0.6 ⟶ 1.0	0.38
Step 3	`trust(6,2)`	0.7 ⟶ 1.0	0.54
Step 4	`trust(1,2)`	0.9 ⟶ 1.0	0.55
Step 5	`trust(2,6)`	0.75 ⟶ 0.96	0.7

We find that the strategy provided by P3 significantly outperforms the random strategy (0.58 vs 1.36 in total change). We make similar observations for other Modification Queries.

6 EVALUATION

We developed a prototype of P3 based on enhancements to the ExSPAN provenance engine [31]. While ExSPAN was a distributed provenance engine designed for networks, we focused on using ExSPAN primarily in a single-node centralized setting. ExSPAN provides provenance for Datalog programs and provides basic provenance explanations. We enhanced the system to implement ProbLog, collect provenance, and support the three additional types of P3 queries (derivation, influence, and modification).

Experimental setup. Our experiments were performed on a Dell PowerEdge R730 server equipped with dual Intel Xeon E5-2640 CPUs and 32GB memory running Ubuntu 16.04 LTS 64-bit operating system.

As our workload, we evaluated the program shown in Figure 7. The output derivation of interest was the `mutualTrustPath` relation, over which we would run provenance queries. We selected this program as it had all the features of our use case. Meanwhile, there existed real-world data that we could use for our experiments. Specifically, we used the Bitcoin OTC data set described in Section 5.2. The data set consisted of a directed graph network with 5,881 nodes (bitcoin users) and 35,592 edges (trust relations). We sampled the graph with different sizes of nodes, and evaluated the performance of provenance queries elaborated in Section 4 for each sample.

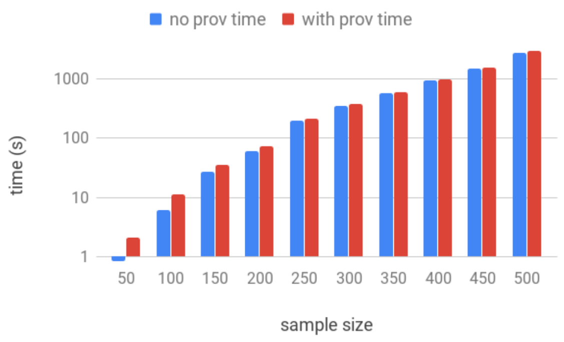


Figure 9: Running time with and without provenance

6.1 Provenance Maintenance and Querying

To evaluate the provenance maintenance and querying performance, we sampled subgraphs from the original trust network with 50, 100, 150, . . . , 500 nodes. For each sample, we randomly chose a small set of seed nodes, and expanded the graph by performing a breadth-first search within the trust network from these seed nodes, until the graph reaches a given number of nodes. We then collected all traversed edges. For each size, we repeated our experiments 10 times with different samples and calculated the average running time.

Maintenance. Our first set of experiments aimed to measure the overhead of provenance maintenance. We compared the running time between running the `Trust` program with and without provenance maintenance. The overhead is shown in Figure 9. We observe that the evaluation time, both with and without provenance, increases exponentially as the sample size grows, which complies with our expectation. In addition, as provenance is maintained at the runtime as a side-computation along each rule evaluation, the provenance maintenance incurs a small overhead. We observed that, in average, the maintenance time is less than 10% of the total running time, and will not impact the asymptotic scalability of PLP programs. Thus, the provenance maintenance is efficient and its overhead in P3 for ProbLog-like PLP programs is low enough to be accepted.

Query. We next measured the overhead to obtain the provenance of a queried tuple, as a generic **explanation query**. Figure 10 summarizes our evaluation results. We fixed the hop limit to 4, which means we only retrieved the mutual trust paths whose length are no greater than 4 hops. Limiting the path length can remove overly long derivation paths from the provenance and expedite the querying process: The number of derivation paths grows exponentially with the path length, however, the long paths contribute little to the derivability of the queried tuple (intuitively, people are less likely to trust a relationship that has many hops). As shown on the figure, the query time is roughly on the same order of magnitude compared to the maintenance time, but grows slower for larger-sized graphs owing to the use of hop limits. Overall, the provenance querying is reasonably efficient

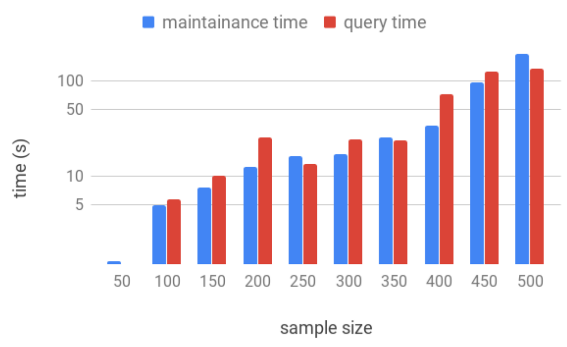


Figure 10: Provenance query time compared with maintenance time. Hop limit is set to 4

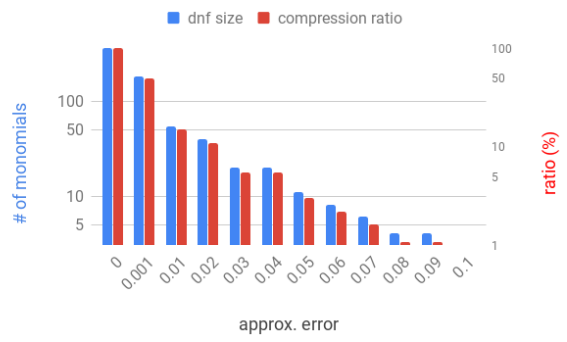


Figure 11: Compression ratio of sufficient provenance

to be practical, and can be further optimized by parallelizing the provenance graph traversal.

6.2 Performance for Different Queries

We have evaluated the overhead of provenance maintenance for a PLP program, and the execution time of generic **explanation query** (i.e., to obtain the provenance for a queried tuple). In this section, we perform a set of experiments to measure the performance of answering other types of provenance queries elaborated in Section 4.

Derivation Query. To evaluate this type of query, we sampled graphs consisting of 150 nodes and 150 edges from the trust network (we repeated the experiment 10 times with different sampled graphs). We queried all possible mutual paths between two specific users and set the hop limit to 6. We ran the query on each sampled graph and calculated the average. For each returned provenance polynomial λ, varying the approximation error ϵ leads to sufficient provenance of different sizes – a more forgiving approximation error (i.e., larger ϵ) leads to a smaller-sized sufficient provenance. We evaluated the compression ratio of sufficient provenance with approximation error from 0.1% to 10% (X% means X percent of $\mathbf{P}[\lambda]$). Compression ratio is defined as the number of monomials in the sufficient provenance divided by the number of monomials in the original provenance polynomials. Our evaluation result is shown in Figure 11. We observed that, as expected, sufficient provenance leads to significant compression: it warrants a 50% deduction in provenance size even with merely 0.1% approximation error; 10% approximation error allows approximately 99.8% deduction.

We further evaluated the computation time of our naïve approach; we observe that the computation time was consistently under 1 second. In fact, it decreased tremendously as we increased the approximation error. This is because the computation of Derivation Queries heavily relies on Monte-Carlo simulation (for evaluating whether more monomials should be removed), and larger approximation error shifts the search for optimal sufficient provenance towards shorter polynomials and therefore a shorter overall query time.

Influence Query. Our next set of experiments evaluates the performance of Influence Queries. We used the same set of sampled graphs as the Derivation Query, that is, each sampled graph consists of 150 nodes and 150 edges. We observe that the average time for computing the influence of all literals (and identifying the most influential literal) is 9.6 seconds; according to the definition of influence presented in Section 4, the computation time of influence queries highly depends on the size of the provenance, more specifically, the number of monomials in the provenance polynomial and the number of distinct literals. We consider two optimizations that significantly reduce the computation time of influence queries.

Parallelize Monte-Carlo simulation. Monte-Carlo simulation repeatedly evaluates the truth value of a Boolean polynomial given a random value assignment of the variables (which can be either true or false). This process is embarrassingly parallel and can greatly benefit from using hardwares such as GPUs. We therefore evaluated the computation time of the Influence Query using a parallel implementation of Monte-Carlo simulation. The experiment was run on a workstation equipped with an Intel i7 9800X CPU, 64GB memory and four Nvidia GTX 1080 Ti GPUs. Table 8 summarizes the time required to compute the influence value sequentially or in parallel.

Table 8: Comparison of influence query time

Seq total	Seq per-literal	Para total	Para per-literal
9.60	0.14	0.85	0.01

We observe that the parallel implementation provides a 10x speed improvement owing to the high degree of parallelization, and reduces the total computation time to under 1 second.

Preprocess using sufficient provenance. We noticed that most literals have a negligibly small influence value. If our goal is to identify the most influential literal, we might be able to avoid computing the influences of most literals. Our approach is to use sufficient provenance as a preprocessing step, such that the influence query runs on a much small provenance polynomial. To evaluate the effectiveness of this method, we first examined whether the returned sufficient provenance still retains the most influential literals. We computed the sufficient provenance when allowing different approximation errors, and compared the top-5 most influential literals to the ones computed from the original provenance. As shown in Figure 12, the rank of the top-5 most influential literals remained the same when the error limit was less than 2%, and then started to fluctuate as the error limit increased. However, the most influential tuple remained the same even the error limit was as high as 10%.

Second, we evaluated the computation time of Influence Query after the preprocessing step retrieves the sufficient provenance. Figure 13 shows the evaluation results with varying approximation error limits. We observe that, since the number of monomials decreased exponentially as we increased the error limit, the computation time of Influence Query also decreased exponentially.

We further measured the total execution time of computing influence queries with sufficient provenance. The result is shown

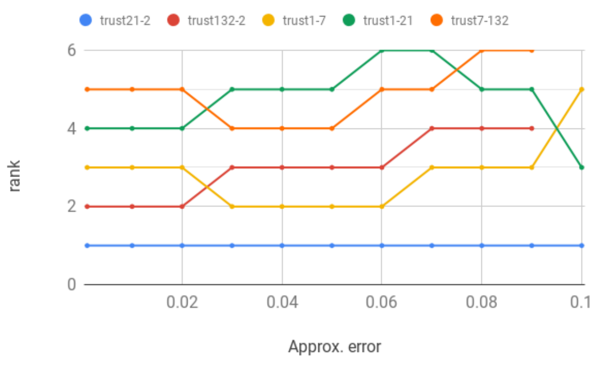


Figure 12: Rank change of top-5 most influential literals

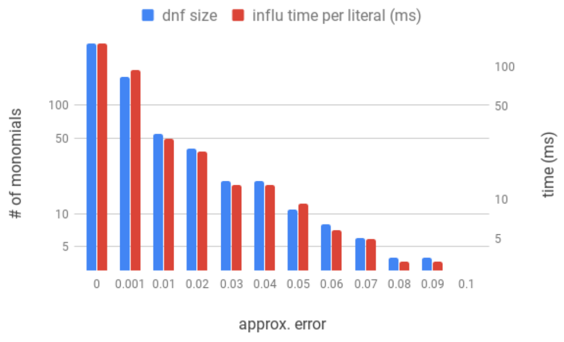


Figure 13: Influence query time for a single literal

in Figure 14. We observed that, for large provenance polynomial, allowing even a small approximation error would reduce the query time tremendously. We compared it with Figure 12 and observed that, when the approximation error limit was set appropriately (around 2% in this case), the returned top influential literals remained unchanged while the computation time could be reduced substantially. For example, we observed an order of magnitude reduction in computation time when setting the error limit to 2%.

Modification Query In our last set of experiments, we evaluated the performance of modification query where we change the provenance probability to a target value with minimum cost. Recall that the changed tuple in each step is the most influential one to the provenance. We tested the Modification query on a given provenance polynomial that consists of 366 monomials and 65 distinct literals. The probability of the provenance was 0.873, and we wanted to reduce it to 0.373. We conducted the experiment using three methods: sequential without sufficient provenance, parallel without sufficient provenance, and sequential with sufficient provenance (with an error limit of 0.01). All the three method returned the same change sequence: a) modify the probability of `trust21-2` by -0.75 (i.e., set its probability to 0), and b) modify the probability of `trust132-2` by -0.196. The computation time are shown in Table 9.

Table 9: Compare running times of modification query

Sequential	Parallel	Seq. with suff. prov.
20.66	1.55	2.44

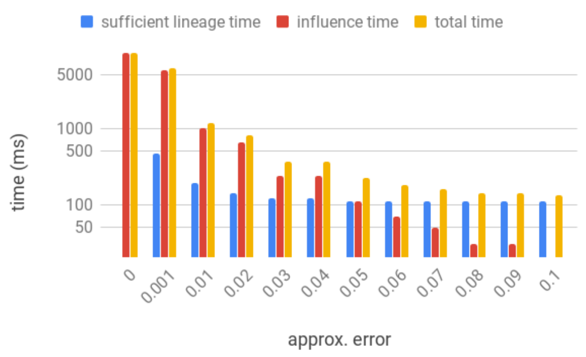


Figure 14: Influence query on sufficient provenance

We observed that sufficient provenance can give the same result while its computation time is in the same order of magnitude compared to the hardware-aided parallel processing.

7 RELATED WORK

We briefly summarize related works in provenance for relational databases (regular and probabilistic), as well as a reference to other probabilistic logic programs.

7.1 Data Provenance

Data provenance is widely used in declarative logic programs like Datalog and NDlog (Network Datalog) [17–19]. Provenance information is stored using *graph representation* and *algebraic representation* [11, 31]. The graph representation is also called provenance graph, and the algebraic representation is commonly encoded as provenance polynomials.

Tuple-level provenance graph is a fine-grained derivation graph. In ExSPAN [31], the provenance graph is acyclic. There are two types of vertices in the graph. One is the tuple vertex, and the other is the rule execution vertex. Each tuple vertex is either a base tuple or a derived tuple; each rule execution vertex denotes an instance of a rule execution. The edges in the provenance graph are unidirectional, and represent data flows between tuple vertices and rule execution vertices.

Provenance can also be represented as *provenance polynomials* [11] which is encoded as an algebraic expression with two binary operations: addition "+" and multiplication "·". Each base tuple is encoded as one literal in the polynomials. Specifically, "+" indicates the combination of tuples with union and projection operations, and "·" denotes a natural join over tuples.

As we will describe later in our paper, P3 also uses ExSPAN style execution to maintain and process provenance, with several extensions in order to support a new provenance model.

7.2 Provenance in Probabilistic Databases

In probabilistic databases community, lineage is the synonym of provenance. Each tuple has an associated probability score in probabilistic databases. For SQL-like queries in probabilistic databases, provenance can support explanations for the queried tuple probabilities. For instance, Trio [30] is an innovative database management system (DBMS) based on an extended relational model called *Uncertainty-Lineage Databases* (ULDBs) to handle the uncertainty of data and data lineage. It extends the traditional model by adding a confidence value (probability of being true) for each tuple. Trio introduces a SQL-based language called *TriQL* for querying confidences and lineage in ULDBs. In our paper, for PLP programs, we do not consider SQL-like queries.

In addition, for probabilistic databases, approximation of lineage such as *sufficient lineage* [25] is used for only keeping track of the most important derivations for the derived tuple given some approximation error ϵ. Kanagal et al. presented a further discussion and provided an efficient approach for sensitivity and explanation analysis [13]. This approach works on read-once lineages from conjunctive queries without self-joins. However, read-once is not a universal property of the provenance polynomials extracted from PLP programs.

7.3 Probabilistic Logic Programs

Milch et al. introduced the BLOG language [20] to define probabilistic models with unknown objects and identity uncertainty. Unlike BLOG based on first-order logic, Goodman et al. developed Church[10], a LISP-like language based on lambda calculus to describe stochastic generative processes. Furthermore, Pfeffer introduced an object-oriented language called Figaro for probabilistic programming. Beside these languages, for probabilistic inference, Alchemy[26] is a system based on Markov logic representation to construct knowledge bases. Niu et al. and Gribkoff et al. introduced Tuffy[22] and SlimShot[12] respectively such that MLN inference can perform on large scale data sets. Compared to the aforementioned studies, our contribution is on providing a feasible solution to perform quantitative query evaluations for debugging purposes.

8 CONCLUSIONS AND FUTURE WORK

This paper proposes P3, a platform and system for capturing provenance in probabilistic logic programs. P3 maintains the provenance information using both graph representation (directed acyclic provenance graph) and algebraic representation (provenance polynomials as Boolean DNF formulas). P3 enables a wide range of novel query types, including explanation query, derivation query, influence query, and modification query. We conduct the theoretical analysis of P3's provenance model and queries. Our evaluation on a P3 prototype demonstrate the feasibility of P3 across multiple use cases with low overhead.

Moving forward, we are working on expanding the scope of PLP programs supported by P3. We plan to extend provenance model and system to support first-order PLP programs with negation, and machine-learning style inference [3, 6, 26]. As we support more language features, we would also like to broaden our use cases to include learning-based applications, such as explainable recommendation through relational learning. Finally, we are exploring ways to improve our performance further, leveraging parallel query execution across multiple machines in a cluster.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful comments. This paper is partially funded from NSF grants CNS-1453392, CNS-1513734, CNS-1513679 and CNS-1704189. This research was additionally supported by ONR under grant N00014-18-1-2618 and by DARPA under grants HR0011-17-C-0047 and HR0011-16-C-0056. The findings and opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Somak Aditya, Yezhou Yang, and Chitta Baral. 2018. Explicit Reasoning over End-to-End Neural Architectures for Visual Question Answering. In *AAAI*.
[2] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, and Devi Parikh. 2015. VQA: Visual Question Answering. In *ICCV*. 2425–2433.
[3] Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. 2015. Hinge-Loss Markov Random Fields and Probabilistic Soft Logic. *Journal of Machine Learning Research* 18 (2015), 109:1–109:67.
[4] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (Aug. 1986), 677–691.
[5] Nilesh N. Dalvi and Dan Suciu. 2004. Efficient query evaluation on probabilistic databases. *PVLDB* (2004), 523–544.
[6] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *TPLP* 15 (2015), 358–401.
[7] Norbert Fuhr. 1995. Probabilistic Datalog&Mdash;a Logic for Powerful Retrieval Methods. In *SIGIR*. ACM, New York, NY, USA, 282–290.
[8] Jean H Gallier. 1988. Logic for Computer Science Foundations of Automatic Theorem Proving. (1988).
[9] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: Probabilistic Inference for Networks. In *PLDI*. ACM, New York, NY, USA, 586–602. https://doi.org/10.1145/3192366.3192400
[10] Noah Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua Tenenbaum. 2008. Church: A language for generative models. *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence, UAI 2008* 2008, 220–229.
[11] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *PODS*. 31–40.
[12] Eric Gribkoff and Dan Suciu. 2016. SlimShot: in-database probabilistic inference for knowledge bases. *Proceedings of the VLDB Endowment* 9 (03 2016), 552–563.
[13] Bhargav Kanagal, Jian Li, and Amol Deshpande. 2011. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *SIGMOD*. 841–852.
[14] Richard M. Karp and Michael Luby. 1983. Monte-Carlo Algorithms for Enumeration and Reliability Problems. In *SFCS*. IEEE Computer Society, 56–64.
[15] Srijan Kumar, Bryan Hooi, Disha Makhija, Mohit Kumar, Christos Faloutsos, and VS Subrahmanian. 2018. Rev2: Fraudulent user prediction in rating platforms. In *WSDM*. ACM, 333–341.
[16] Srijan Kumar, Francesca Spezzano, VS Subrahmanian, and Christos Faloutsos. 2016. Edge weight prediction in weighted signed networks. In *ICDM*. IEEE, 221–230.
[17] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2006. Declarative Networking: Language, Execution and Optimization. In *SIGMOD*. 97–108.
[18] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. 2005. Implementing Declarative Overlays. In *SOSP*. ACM, 75–90.
[19] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. 2005. Declarative Routing: Extensible Routing with Declarative Queries. In *SIGCOMM*.
[20] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel Ong, and Andrey Kolobov. 2005. BLOG: Probabilistic Models with Unknown Objects. *IJCAI* 2005 (01 2005).
[21] Stephen Muggleton et al. 1996. Stochastic logic programs. *ILP* 32 (1996), 254–264.
[22] Feng Niu, Christopher Ré, AnHai Doan, and Jude Shavlik. 2011. Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. *VLDB Endowment* 4, 6 (2011), 373–384.
[23] David Poole. 2008. The independent choice logic and beyond. In *Probabilistic Inductive Logic Programming*. Springer, 222–243.
[24] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *IJCAI*.
[25] Christopher Ré and Dan Suciu. 2008. Approximate lineage for probabilistic databases. In *PVLDB*. 797–808.
[26] Matthew Richardson and Pedro Domingos. 2006. Markov Logic Networks. *Mach. Learn.* 62, 1-2 (Feb. 2006), 107–136.
[27] Taisuke Sato. 1995. A Statistical Learning Method for Logic Programs with Distribution Semantics. In *ICLP*.
[28] Taisuke Sato and Yoshitaka Kameya. 2008. New Advances in Logic-based Probabilistic Modeling by PRISM. In *Probabilistic Inductive Logic Programming*, Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton (Eds.). Springer-Verlag, Berlin, Heidelberg, 118–155.
[29] Leslie G Valiant. 1979. The complexity of enumeration and reliability problems. *SIAM J. Comput.* 8, 3 (1979), 410–421.
[30] Jennifer Widom. 2008. Trio: A System for Data, Uncertainty, and Lineage. In *Managing and Mining Uncertain Data*. Springer.
[31] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. 2010. Efficient Querying and Maintenance of Network Provenance at Internet-scale. In *SIGMOD*. ACM, 615–626.

Improved Cardinality Estimation by Learning Queries Containment Rates

Rojeh Hayek
CS Department, Technion
Haifa 3200003, Israel
srojeh@cs.technion.ac.il

Oded Shmueli
CS Department, Technion
Haifa 3200003, Israel
oshmu@cs.technion.ac.il

ABSTRACT

The *containment rate* of query $Q1$ in query $Q2$ over database D is the percentage of $Q1$'s result tuples over D that are also in $Q2$'s result over D. We directly estimate containment rates between pairs of queries over a specific database. For this, we use a specialized deep learning scheme, Containment Rate Network (CRN), which is tailored to representing pairs of SQL queries (inspired by the MSCN model [22]). Result-cardinality estimation is a core component of query optimization. We describe a novel approach for estimating queries' result-cardinalities using estimated containment rates among queries. This containment rate estimation may rely on CRN or embed, unchanged, known *cardinality* estimation methods. Experimentally, our novel approach for estimating cardinalities, using containment rates between queries, on a challenging real-world database, realizes significant improvements to state of the art cardinality estimation methods.

1 INTRODUCTION

Query $Q1$ is contained in (resp. equivalent to), query $Q2$, analytically, if for all database states D, $Q1$'s result over D is contained in (resp., equals) $Q2$'s result over D. Query containment is a well-known concept that has applications in query optimization. It has been extensively researched in database theory, and many algorithms were proposed for determining containment under different assumptions [8, 9, 16, 40]. However, determining query containment analytically is not practically sufficient. Two queries may be analytically unrelated by containment, although, the execution result on a *specific* database of one query may actually be contained in the other. For example, consider the queries:
Q1: *select * from movies where title = 'Titanic'*
Q2: *select * from movies where release = 1997 and director = 'James Cameron'*
Both queries execution results are identical since there is only one movie called Titanic that was released in 1997 and directed by James Cameron (he has not directed any other movie in 1997). Yet, using the analytic criterion, the queries are unrelated at all by containment.

To our knowledge, while query containment and equivalence have been well researched in past decades, determining the containment rate between two queries on a *specific* database, has not been considered by past research.

By definition, the containment rate of query $Q1$ in query $Q2$ on database D is the percentage of rows (tuples) in $Q1$'s execution result over D that are also in $Q2$'s execution result over D. Determining containment rates allows us to solve other problems, such as determining equivalence between two queries, or whether one query is fully contained in another, on the same *specific* database. In addition, containment rates can be used in many practical applications, for instance, query clustering, query recommendation [11, 15], and in cardinality estimation as will be described subsequently.

Our approach for estimating containment rates is based on a specialized deep learning model, CRN, which enables us to express query features using sets and vectors. An input query is converted into three sets, T, J and P representing the query's tables, joins and column predicates, respectively. Each element of these sets is represented by a vector. Using these vectors, CRN generates a single vector that represents the whole input query. Finally, CRN estimates the containment rate of two input queries by using their representative vectors as input to another specialized neural network. Thus, the CRN model relies on the ability of the neural network to learn the vector representation of queries relative to the *specific* database. As a result, we obtain a small and accurate model for estimating containment rates.

In addition to the CRN model, we introduce a novel technique for estimating queries' cardinalities using estimated query containment rates. We show that using the proposed technique we improve current cardinality estimation techniques significantly. This is especially the case when there are multiple joins, where the known cardinality estimation techniques suffer from under-estimated results and errors that grow exponentially as the number of joins increases [14]. Our technique estimates the cardinalities more robustly (x150/x175 with 4 joins queries, and x1650/x120 with 5 joins queries, compared with PostgreSQL and MSCN, respectively).

As shown in [26], to obtain an efficient query plan, the query optimizer chooses the cheapest alternative from semantically equivalent plan alternatives. Since the cost model uses the cardinality estimates as a principal input, the more accurate the cardinality estimates are, the more accurate the predicted plans costs are. Thus, by using the more accurate cardinality estimates obtained from our technique, the query optimizer can generate better query plans, resulting in faster query execution time.

We compare our technique with PostgreSQL [1], and the pioneering multi-set convolutional network (MSCN) model [22], by examining, on the real-world IMDb database [26], join crossing correlations queries which are known to present a tough challenge to cardinality estimation methods [26, 28, 35].

We show that by employing known existing cardinality estimation methods for containment estimation, we can improve on their cardinality estimates as well, without changing the methods themselves. Thus, our novel approach is highly promising for solving the cardinality estimation problem, the "Achilles heel" of query optimization [30], a cause of many performance issues [26].

The rest of this paper is organized as follows. In Section 2 we define the containment rate problem and in Sections 3-4 we describe and evaluate the CRN model for solving this problem. In Sections 5-6 we describe and evaluate our new approach for estimating cardinalities using containment rates. In Section 7 we show how one can adapt the new ideas to improve existing cardinality estimation models, and in Section 8 we compare the prediction time among the different approaches. Finally, Sections 9-10 present related work, conclusions and future work.

2 CONTAINMENT RATE DEFINITION

We define the containment rate between two queries $Q1$, and $Q2$ on a *specific* database D. *Query $Q1$ is $x\%$-contained in query $Q2$ on database D if precisely $x\%$ of $Q1$'s execution result rows on database D are also in $Q2$'s execution result on database D.* The containment rate is formally a function from **QxQxD** to **R**, where **Q, D** and **R** are the set of all queries, all databases, and the Real numbers, respectively. This function can be directly calculated as follows. Let $Q1(D) = (A, m_1)$ and $Q2(D) = (B, m_2)$ be multisets[1] representing queries $Q1$ and $Q2$ execution results on database D, respectively, then:

$$x\% = \frac{\sum_{x \in (A \cap B)} m_1(x)}{\sum_{x \in A} m_1(x)} * 100$$

Where operator $\cap$ is the regular set intersection operator (in case $Q1$'s execution result is empty, then $Q1$ is 0%-contained in $Q2$). Note that the containment rate is defined only on pairs of queries whose SELECT and FROM clauses are *identical.*

Since we aim to estimate cardinalities using containment rates, we consider only queries with SELECT * clauses, then, given a query Q whose SELECT clause includes specific columns, Q's cardinality is identical to the cardinality of the query with a SELECT * clause instead (as long as the *DISTINCT* keyword is not used). Therefore, in practice, the requirement that the clauses need to be *identical* applies only to the FROM clauses.

2.1 Containment Rate Operator

We denote the containment rate *operator* between queries $Q1$ and $Q2$ on database D as:

$$Q1 \subset^{D}_{\%} Q2$$

Operator $\subset^{D}_{\%}$ returns the containment rate between the given input queries on database D. That is, $Q1 \subset^{D}_{\%} Q2$ returns $x\%$, if $Q1$ is $x\%$-contained in query $Q2$ on database D. For simplicity, we do not mention the *specific* database, as it is usually clear from context. Hence, we write the containment rate operator as $\subset_{\%}$.

3 LEARNED CONTAINMENT RATES

From a high-level perspective, applying machine learning to the containment rate estimation problem is straightforward. Following the training of the CRN model with pairs of queries $(Q1, Q2)$ and the actual containment rates $Q1 \subset_{\%} Q2$, the model is used as an estimator for other, unseen pairs of queries. (Later on, as described in Section 5, we will make use of this model to estimate cardinalities of single queries). There are, however, several questions whose answers determine whether the machine learning model (CRN) will be successful. (1) Which supervised learning algorithm/model should be used. (2) How to represent queries as input and the containment rates as output to the model ("featurization"). (3) How to obtain the initial training dataset ("cold start problem"). Next, we describe how we address each one of these questions.

[1]From Wikipedia: A multiset may be formally defined as a 2-tuple $(S,\ m)$ where S is the underlying set of the multiset, formed from its distinct elements, and $m : S \rightarrow N_{\geq 1}$ is a function from S to the set of the positive integers, giving the multiplicity. The number of occurrences of element x in the multiset is $m(x)$.

3.1 Cold Start Problem

3.1.1 Defining the Database. We generated a training-set, and later on evaluated our model on it, using the IMDb database. IMDb contains many correlations and has been shown to be very challenging for cardinality estimators [26]. This database contains a plethora of information about movies and related facts about actors, directors, and production companies, with more than 2.5M movie titles produced over 130 years (starting from 1880) by 235,000 different companies with over 4M actors.

3.1.2 Generating the Development Dataset. Our approach for solving the "cold start problem" is to obtain an initial training corpus using a specialized queries generator that randomly generates queries based on the IMDB schema and the actual columns values. Our queries generator generates the dataset in three main steps. In the first step (similarly to MSCN's queries generator), it repeatedly generates multiple SQL queries as follows. It randomly chooses a set of tables t ($t = \{bt_1, bt_2, ..., bt_{|t|}\}$). Then, it adds $|t| - 1$ join edges to the query, $bt_i.col_a = bt_{i+1}.col_b$, $1 \leq i < |t|$. Each of these joins is on a column containing the ID of movies (each table in IMDB has such a column). Note that when $|t| = 1$, there are no joins in the query.

For each base table bt in t, it uniformly draws the number of query predicates p_{bt} ($0 \leq p_{bt} \leq$ number of columns in table bt). Subsequently, for each predicate it uniformly draws a column from the relevant table bt, a predicate type ($<$, $=$, *or* $>$), and a value from the corresponding column values range in the database. To avoid a combinatorial explosion, and to simplify the problem that the model needs to learn, we force the queries generator to create queries with up to two joins and let the model generalize to a larger number of joins (that is, the maximum cardinality of set t is 3). Note that all the generated queries include a SELECT * clause. They are denoted as *initial-queries.*

To create pairs of queries that are contained in each other with different containment rates, we generate, in the second step, queries that are "similar" to the *initial-queries*, but still, different from them, as follows. For each query Q in *initial-queries*, the generator repeatedly creates multiple queries by randomly changing query Q's predicates' types, or the predicates' values, and by randomly adding additional predicates to the original query Q. This way, we create a "hard" dataset, which includes pairs of queries that look "similar", but having mutual containment rates that vary significantly. Finally, in the third and last step, using the queries obtained from both previous steps, the queries generator generates pairs of queries whose FROM clauses are identical.

After generating the dataset, we execute the dataset queries on the IMDb database, to obtain their true containment rates and skip query pairs that include a query with an empty result set. Using this process, we obtain an initial training set of 100,000 pairs of queries with zero to two joins. We split the training samples into 80% training samples and 20% validation samples.

3.2 Model

Featurizing all the queries' literals and predicates as one "big hot vector", over all the possible words that may appear in the queries, is impractical. Also, serializing the queries' SELECT, FROM, and WHERE clauses elements into an ordered sequence of elements, is not practical, since the order in these clauses is arbitrary. Thus, standard deep neural network architectures such as simple multi-layer perceptrons [6], convolutional neural networks [6], or recurrent neural networks [6], are not directly applicable to our problem.

Our *Containment Rate Network* (CRN) model uses a specialized vector representation for representing the input queries and the output containment rates. As depicted in Figure 1, the CRN model runs in three main stages. Consider an input queries pair $(Q1, Q2)$. In the first stage, we convert $Q1$ (resp., $Q2$) into a set of vectors $V1$ (resp., $V2$). Thus $(Q1, Q2)$ is represented by $(V1, V2)$. In the second stage, we convert set $V1$ (resp., $V2$) into a unique single representative vector $Qvec1$ (resp., $Qvec2$), using a specialized neural network, MLP_i, for each set separately. In the third stage, we estimate the containment rate $Q1 \subset_{\%} Q2$, using the representative vectors $Qvec1$ and $Qvec2$, and another specialized neural network, MLP_{out}.

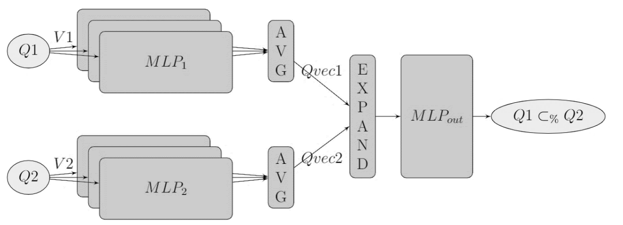


Figure 1: CRN Model Archeticture.

3.2.1 *First Stage, from $(Q1, Q2)$ to $(V1, V2)$.* In the same way as MSCN model [22], we represent each query Q as a collection of three sets (T, J, P). T is the set of all the tables in Q's FROM clause. J is the set of all the joins (i.e., join clauses) in Q's WHERE clause. P is the set of all the (column) predicates in Q's WHERE clause. Using sets T, J, and P, we obtain a set of vectors V representing the query, as described later. Unlike MSCN, in our model all the vectors of set V have the same dimension and the same segmentation as depicted in Table 1, where $\#T$ is the number of all the tables in the database, $\#C$ is the number of all the columns in all the database tables, and $\#O$ is the number of possible predicates operators. In total, the vector dimension is $\#T + 3 * \#C + \#O + 1$, denoted as L.

The queries tables, joins and column predicates (sets T, J and P) are inseparable, hence, treating each set individually using different neural networks may disorientate the model. Therefore, we choose to featurize these sets using the same vector format in order to ease learning.

Type	Table	Join		Column Predicate		
Segment	T-seg	J1-seg	J2-seg	C-seg	O-seg	V-seg
Segment size	$\#T$	$\#C$	$\#C$	$\#C$	$\#O$	1
Featurization	*one hot*	*one hot*	*one hot*	*one hot*	*one hot*	*norm*

Table 1: Vector Segmentation.

Element of sets T, J, and P, are represented by vectors as follows (see a simple example in Figure 2). All the vectors have the same dimension L. Each table $t \in T$ is represented by a unique one-hot vector (a binary vector of length $\#T$ with a single non-zero entry, uniquely identifying a specific table) placed in the T-seg segment. Each join clause of the form $(col1, =, col2) \in J$ is represented as follows. $col1$ and $col2$ are represented by a unique one-hot vectors placed in J1-seg and J2-seg segments, respectively. Each predicate of the form $(col, op, val) \in P$ is represented as follows. col and op are represented by a unique one-hot vectors placed in the C-seg and V-seg segments, respectively. val is represented as a normalized value $\in [0, 1]$, normalized using the minimum and maximum values of the respective column, placed in the V-seg segment. For each vector, all the other unmentioned segments are zeroed. Given input queries pair, $(Q1, Q2)$, we convert query $Q1$ (resp., $Q2$) into sets T, J and P, and *each* element of these sets is represented by a vector as described above, together generating set $V1$ (resp., $V2$).

3.2.2 *Second Stage, from $(V1, V2)$ to $(Qvec1, Qvec2)$.* Given set of vectors V_i, we present each vector of the set into a fully-connected one-layer neural network, denoted as MLP_i, converting each vector into a vector of dimension H. The final representation $Qvec_i$ for this set is then given by the average over the individual transformed representations of its elements, i.e.,

$$Qvec_i = \frac{1}{|V_i|} \sum_{v \in V_i} MLP_i(v)$$

$$MLP_i(v) = Relu(vU_i + b_i)$$

Where $U_i \in R^{LxH}$, $b_i \in R^H$ are the learned weights and bias, and $v \in R^L$ is the input row vector. We choose an average (instead of, e.g., sum) to ease generalization to different numbers of elements in the sets, as otherwise the overall magnitude of $Qvec$ would vary depending on the number of elements in the set V_i.

3.2.3 *Third Stage, from $(Qvec1, Qvec2)$ to $Q1 \subset_{\%} Q2$.* Given the representative vectors of the input queries,$(Qvec1, Qvec2)$, we aim to predict the containment rate $Q1 \subset_{\%} Q2$ as accurately as possible. Since we do not know what a "natural" containment rate measure is in the representative queries vector space, encoded by the neural networks of the second step, we use a fully-connected two-layer neural network, denoted as MLP_{out}, to compute the estimated containment rate of the input queries, leaving it up to this neural network to learn the correct containment rate measure.

MLP_{out} takes as input a vector of size $4H$ which is constructed using function *ExpandFunction* that creates a row of concatenated vectors of size $4H$ using vectors $Qvec1$ and $Qvec2$. We use this function in order to provide the final network, MLP_{out}, with additional information that may enhance its learning and thus obtain more accurate containment rates estimations.

The first layer in MLP_{out} converts the input vector into a vector of size $2H$. The second layer converts the obtained vector of size $2H$, into a single value representing the containment rate.

$$\hat{y} = MLP_{out}(Expand(Qvec1, Qvec2))$$

$$MLP_{out}(v) = Sigmoid(ReLU(vU_{out1} + b_{out1})U_{out2} + b_{out2})$$

$$Expand(v_1, v_2) = [v_1,\ v_2,\ abs(v_1 - v_2),\ v_1 \odot v_2]$$

Here, $\hat{y}$ is the estimated containment rate (a number in $[0, 1]$), $U_{out1} \in R^{4Hx2H}$, $b_{out1} \in R^{2H}$ and $U_{out2} \in R^{2Hx1}$, $b_{out2} \in R^1$ are the learned weights and bias, abs is the absolute value function, and $\odot$ is the dot-product function.

We use the $ReLU$[2] activation function for hidden layers in all the neural networks, as they show strong empirical performance advantages and are fast to evaluate.

In the final step, we apply the $Sigmoid$[3] activation function in the second layer to output a float value in the range [0,1], as the containment rate values are within this interval. Therefore, we do not apply any featurization on the containment rates (the output of the model) and the model is trained with the actual containment rate values without any featurization steps.

[2]ReLU(x) = max(0,x); see [36].
[3]Sigmoid(x) = $1/(1 + e^{-x})$; see [36].

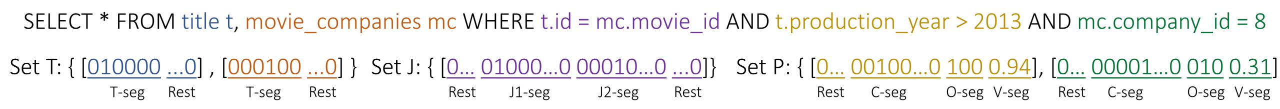

SELECT * FROM title t, movie_companies mc WHERE t.id = mc.movie_id AND t.production_year > 2013 AND mc.company_id = 8
Set T: { [010000 ...0] , [000100 ...0] }
T-seg
Rest
T-seg
Rest
Set J: { [0... 01000...0 00010...0 ...0]}
Rest
J1-seg
J2-seg
Rest
Set P: { [0... 00100...0 100 0.94], [0... 00001...0 010 0.31] }
Rest
C-seg
O-seg
V-seg
Rest
C-seg
O-seg
V-seg

Figure 2: Query featurization as sets of feature vectors obtained from sets T, J and P (Rest denotes zeroed segments).

3.2.4 *Loss Function.* Since we are interested in minimizing the ratio between the predicted and the actual containment rates, we use the q-error metric in our evaluation. We train our model to minimize the mean q-error [33], which is the ratio between an estimated and the actual containment rate (or vice versa). Let y be the true containment rate, and $\hat{y}$ the estimated rate, then the q-error is defined as follows.

$$q - error(y, \hat{y}) \;=\; \hat{y} > y \;?\; \frac{\hat{y}}{y} \;:\; \frac{y}{\hat{y}}$$

The q-error is not defined when y (or y') equals zero. Therefore, in creating the training and testing datasets we skip query pairs that include a query with an empty results set (see Section 3.1.2).

In addition to optimizing the mean q-error, we also examined the mean squared error (MSE) and the mean absolute error (MAE) as optimization goals. MSE and MAE would optimize the squared/absolute differences between the predicted and the actual containment rates. Optimizing with theses metrics makes the model put less emphasis on heavy outliers (that lead to large errors). Therefore, we decided to optimize our model using the q-error metric which yielded better results.

3.3 Training and Testing Interface

Building CRN involves two main steps. (1) Generating a random training set using the schema and data information as described in Section 3.1. (2) Repeatedly using this training data, we train the CRN model as described in Section 3.2 until the mean q-error of the validation test starts to converges to its best absolute value. That is, we use the early stopping technique [39] and stop the training before convergence to avoid over-fitting. Both steps are performed on an immutable snapshot of the database.

After the training phase, to predict the containment rate of an input query pair, the queries first need to be transformed into their feature representation, and then they are presented as input to the model, and the model outputs the estimated containment rate (Section 3.2). We train and test our model using the Tensor-Flow framework [34], and make use of the efficient Adam optimizer [21] for training the model.

3.4 Hyperparameter Search

To optimize our model's performance, we conducted a search over its hyperparameter space. In particular, we focused on tuning the neural networks hidden layer size (H) as we found out that this hyperparameter has the most impact on the results.

Note that the same H value is shared in all the neural networks of the CRN model, as described in section 3.2. During the tuning of the size hyperparameter of the neural network hidden layer, we found that increasing the size of our hidden layer generally led to an increase in the model accuracy, till it reached the best mean q-error on the validation test. Afterwards, the results began to decline in quality because of over-fitting (see Figure 3). Hence, we choose a hidden layer of size 512, as a good balance between accuracy and training time.

Overall, we found that our model performs uniformly well across a wide range of settings when considering different batch sizes and learning rates. We use a learning rate of 0.001, and batch size of 128, as these settings lead to the best results on the validation test.

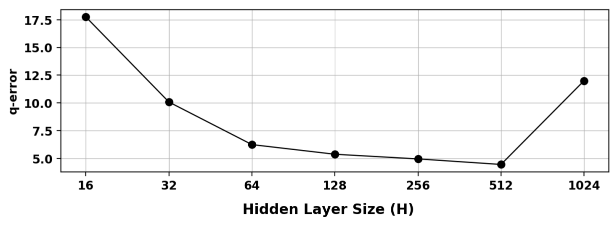


Figure 3: The mean q-error on the validation set with different hidden layer sizes.

3.5 Model Computational Costs

We analyze the training, prediction, and space costs of the CRN model with the default hyperparameters (H=512, batch size=128, learning rate=0.001).

3.5.1 *Training Time.* Figure 4 shows how the mean q-error of the validation set decreases with additional epochs, until convergence to a mean q-error of around 4.5. The CRN model requires almost 120 passes on the training set to converge. On average, measured across six runs, a training run with 120 epochs takes almost 200 minutes.

3.5.2 *Prediction Time.* The prediction process is dominated by converting the input queries into the corresponding vectors, and presenting these vectors as input to the CRN model. On average, the prediction time is 0.5ms per single pair of queries, including the overhead introduced by the Tensor-Flow framework.

3.5.3 *Model Size.* The CRN model includes all the learned parameters mentioned in Section 3.2 (U_1, U_2, U_{out1}, U_{out2}, b_1, b_2, b_{out1}, b_{out2}). In total, there are $2*L*H+8*H^2+6*H+1$ learned parameters. In practice, the size of the model, when serialized to disk, is roughly $1.5MB$.

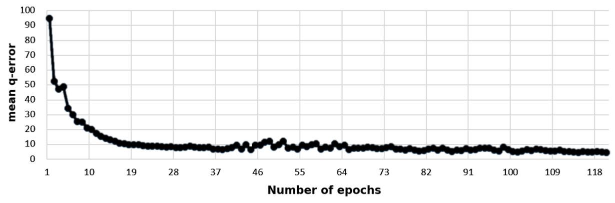


Figure 4: Convergence of the mean q-error on the validation set.

4 CONTAINMENT EVALUATION

Since the focus of this paper is on cardinality estimation using containment rates, in this section we only briefly present the containment evaluation results of the CRN model when compared to other (baseline) methods. In the following sections, we present in detail the experiments of the cardinality estimation technique.

Since to the best of our knowledge, the problem of determining containment rate has not been addressed till now, we use a transformation as described in Section 4.1 below.

4.1 From Cardinality to Containment

To our knowledge, this is the first work to address the problem of containment rate estimation. In order to compare our results with different baseline methods, we used existing cardinality estimation methods to predict the containment rates, using the Crd2Cnt transformation, as depicted in the middle part diagram in Figure 5. (This transformation will be used also in our technique to improve existing cardinality estimation models in Section 7).

4.1.1 The Crd2Cnt Transformation. Given a cardinality estimation model M, we can convert it to a containment rate estimation model using the Crd2Cnt transformation which returns a model M' for estimating containment rates. The obtained model M' functions as follows. Given input queries $Q1$ and $Q2$, whose containment rate $Q1 \subset_{\%} Q2$ needs to be estimated:

- Calculate the cardinality of query $Q1 \cap Q2$ using M.
- Calculate the cardinality of query $Q1$ using M.
- Then, the containment rate estimate is:

$$Q1 \subset_{\%} Q2 = \frac{|Q1 \cap Q2|}{|Q1|}$$

Here, $Q1 \cap Q2$ is the intersection query of $Q1$ and $Q2$ whose SELECT and FROM clauses are identical to $Q1$'s (or $Q2$'s) clauses, and whose WHERE clause is $Q1$'s AND $Q2$'s WHERE clauses. Note that, by definition, if $|Q1| = 0$ then $Q1 \subset_{\%} Q2 = 0$.

Given model M, we denote the obtained model M', via the Crd2Cnt transformation, as Crd2Cnt(M).

4.2 Experimental Results

We compared the CRN model predictions to those based on the other examined cardinality estimation models, using the Crd2Cnt transformation. We evaluated the models with several workloads, that included over 2000 queries with zero to five joins, on the challenging real-world IMDB database [26]. In terms of mean q-error [33], the CRN model reduced the mean q-errors by a factor of roughly 8 compared with the estimates obtained from Crd2Cnt(PostgreSQL) and Crd2Cnt(MSCN).

To provide a fuller picture, in Table 2 we show the percentiles, maximum, and mean q-errors, on one of the examined evaluation workloads. Additional details may be found in *arXiv* [18].

	50th	75th	90th	95th	99th	max	mean
Crd2Cnt(PostgreSQL)	4.5	46.22	322	1330	39051	316122	1345
Crd2Cnt(MSCN)	4.1	17.85	157	754	14197	768051	1238
CRN	**3.64**	**13.19**	**96.6**	**255**	**2779**	**56965**	**161**

Table 2: Estimation errors on 1200 examined queries with zero to five joins, equally distributed in the number of joins. In all the similar tables presented in this paper, we provide the percentiles, maximum, and the mean q-errors of the tests. The *p*'th percentile, is the q-error value below which *p*% of the test q-errors are found. For example, 50% of the CRN test q-errors are smaller than 3.64.

5 CARDINALITY ESTIMATION USING CONTAINMENT RATES

In this section we consider one application of the proposed containment rate estimation model: cardinality estimation. We introduce a novel approach for estimating cardinalities using query containment rates, and we show that using the proposed approach, we improve cardinality estimations significantly, especially in the case when there are multiple joins.

A traditional query optimizer is crucially dependent on cardinality estimation, which enables choosing among different plan alternatives by using the cardinality estimation of intermediate results within query execution plans. Therefore, the query optimizer must use reasonably good estimates. However, estimates produced by all widely-used database cardinality estimation models are routinely significantly wrong (under/over-estimated), resulting in not choosing the best plans, leading to slow executions [26].

Three principal approaches for estimating cardinalities have emerged. (1) Using database profiling [1]. (2) Using histograms [3, 7]. (3) Using sampling techniques [5, 27, 37]. Recently, deep learning neural networks were also used for solving this problem [22, 45]. However, all these approaches, with all the many attempts to improve them, have conceptually addressed the problem *directly* in the same way, as a black box, where the input is a query, and the output is its cardinality estimation, as described in the leftmost diagram in Figure 5. In our proposed approach, we address the problem differently, and we obtain better estimates as described in Section 6.

In prior works, the answers to previous queries were used for speeding up new queries, by incrementally updating histograms, and in the context of query re-optimization [3, 7, 13, 20]. Similarly, using the CRN model for predicting containment rates, we are making use of these previous answers to reveal the underlying relations between the new queries and the previous ones.

Our new technique for estimating cardinalities mainly relies on two key ideas. The first one is the new framework in which we solve the problem. The second is the use of a *queries pool* that maintains multiple previously executed queries along with their actual *cardinalities*, as part of the database meta information. The queries pool provides new information that enables our technique to achieve better estimates. Using a containment rate estimation model, we make use of previously executed queries along with their actual cardinalities to estimate the result-cardinality of a new query. This is done with the help of a simple transformation from the problem of containment rate estimation to the problem of cardinality estimation (see Section 5.1).

5.1 From Containment to Cardinality

Using a containment rate estimation models, we can obtain cardinality estimates using the Cnt2Crd transformation, as depicted in the rightmost diagram in Figure 5.

5.1.1 The Cnt2Crd Transformation. Given a containment rate estimation model[4] M, we convert it to a cardinality estimation model using the Cnt2Crd transformation which returns a model M' for estimating cardinalities. The obtained model M' functions as follows. We are given a "new" query, denoted as Q_{new}, as input to cardinality estimation. Assume that there is an "old" query, denoted as Q_{old}, whose FROM clause is the same as Q_{new}'s

[4]The term "model" may refer to an ML model or simply to a method.

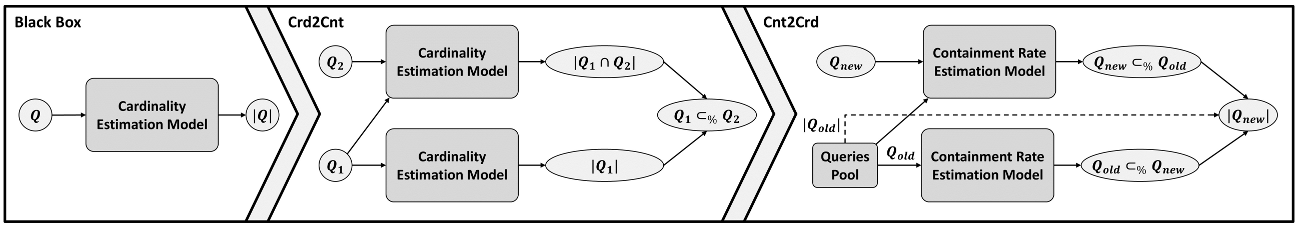

Black Box
Q
Cardinality Estimation Model
|Q|
Crd2Cnt
Q₂
Q₁
Cardinality Estimation Model
Cardinality Estimation Model
|Q₁ ∩ Q₂|
|Q₁|
Q₁ ⊂% Q₂
Cnt2Crd
Qnew
Containment Rate Estimation Model
Qnew ⊂% Qold
|Qold|
Queries Pool
Qold
Containment Rate Estimation Model
Qold ⊂% Qnew
|Qnew|

Figure 5: A novel approach, from cardinality estimation to containment rate estimation, and back to cardinality estimation by using a queries pool.

FROM clause, that has already been executed over the database, and therefore $|Q_{old}|$ is known, then M' functions as follows:

- Calculate $x\_rate = Q_{old} \subset_{\%} Q_{new}$ using M.
- Calculate $y\_rate = Q_{new} \subset_{\%} Q_{old}$ using M.
- Then, the cardinality estimate equals to:

$$|Q_{new}| = \frac{x\_rate}{y\_rate} * |Q_{old}|$$

provided that $y\_rate = Q_{new} \subset_{\%} Q_{old} \neq 0$. This is true, since:

$$x\_rate = \frac{|Q_{new} \cap Q_{old}|}{|Q_{old}|}, \;\; y\_rate = \frac{|Q_{new} \cap Q_{old}|}{|Q_{new}|}$$

And therefore,

$$\frac{x\_rate}{y\_rate} = \frac{|Q_{new} \cap Q_{old}|}{|Q_{old}|} * \frac{|Q_{new}|}{|Q_{new} \cap Q_{old}|} = \frac{|Q_{new}|}{|Q_{old}|}$$

where the query intersection operator, ∩, is as defined in Section 4.1.1. Given model M, we denote the obtained model M', via the Cnt2Crd transformation, as Cnt2Crd(M).

5.2 Queries Pool

Our technique for estimating cardinality relies mainly on a queries pool that includes records of multiple queries.

The queries pool is envisioned to be an additional component of the DBMS, along with all the other customary components. It includes multiple queries with their actual cardinalities[5], but without the queries execution results. Therefore, holding such a pool in the DBMS as part of its meta information does not require significant storage space or other computing resources. Maintaining a queries pool in the DBMS is thus a reasonable expectation. The DBMS continuously executes queries, and therefore, we can easily configure the DBMS to store these queries along with their actual cardinalities in the queries pool.

In addition, we may construct in advance a queries pool using a queries generator that randomly creates multiple queries with many of the possible joins, and with different column predicates. We then execute these queries on the database to obtain and save their actual cardinalities in the queries pool.

Notice that we can combine both approaches (actual computing and a generator) to create the queries pool. The advantage of the first approach is that in a real-world situation, queries that are posed in sequence by the same user, may be similar and therefore we can get more accurate cardinality estimates. The second approach helps in cases where the queries posed by users are diverse (e.g., different FROM clauses). Therefore, in such cases, we need to make sure, in advance, that the queries pool contains sufficiently many queries that cover all the possible cases.

[5]Due to limited space, we do not detail the efficient hash-based data structures used to implement the queries pool.

Given a query Q whose cardinality is to be estimated , it is possible that we fail to find any appropriate query, in the queries pool, to match with query Q. This happens when all the queries in the queries pool have a different FROM clause than that of query Q, or that they are not contained at all in query Q. In such cases we can always use the *known* basic cardinality estimation models. In addition, we can make sure that the queries pool includes queries with the most frequently used FROM clauses, with empty column predicates. That is, queries of the following form:

$$SELECT\ *\ FROM\ \ -set\ of\ tables-\ \ WHERE\ TRUE$$

In this case, for most of the queries posed in the database, there is at least one query that matches in the queries pool with the given query, and hence, we can estimate the cardinality (perhaps less accurately) without resorting to the basic cardinality estimation models.

5.3 A Cardinality Estimation Technique

Consider a new query Q_{new}, and assume that the DBMS includes a queries pool as previously described. To estimate the cardinality of Q_{new} accurately, we use *multiple* "old" queries instead of *one* query, using the same Cnt2Crd transformation of Section 5.1.1, as described in Figure 6.

```
EstimateCardinality(Query $Q_{new}$, Queries Pool $QP$):
    $results$ = empty list

    For every pair ($Q_{old}$, $|Q_{old}|$) in $QP$:
        if $Q_{old}$'s FROM clause ≠ $Q_{new}$'s FROM clause:
            continue
        Calculate $x\_rate = Q_{old} \subset_{\%} Q_{new}$
        Calculate $y\_rate = Q_{new} \subset_{\%} Q_{old}$
        if $y\_rate <= epsilon$: /* y essentially zero */
            continue
        $results$.append($x\_rate/y\_rate * |Q_{old}|$)

    return F($results$)
```

Figure 6: Cardinality Estimation Technique.

Algorithm EstimateCardinality considers all the *matching* queries whose FROM clauses are identical to Q_{new}'s FROM clause. For each matching query, we estimate Q_{new}'s cardinality using the Cnt2Crd transformation and save the estimated result in the *results* list. The final cardinality is obtained by applying the final function, F, that converts all the estimated results recorded in the *results* list, into a single final estimation value. Note that the technique can be easily parallelized since each iteration in the For loop is independent, and thus can be calculated in parallel.

5.3.1 *Comparing Different Final Functions.*
We examined various final functions (F), including:

- Median, returning the median value of the *results* list.
- Mean, returning the mean value of the *results* list.
- Trimmed mean, returning the trimmed mean of the *results* list without the 25% outliers (trimmed removes a designated percentage of the largest and smallest values before calculating the mean).

Experimentally, the cardinality estimates using the various functions were very similar in terms of q-error. But the Median function yielded the best estimates as it is more stable to outliers (we do not detail these experiments due to limited space).

5.3.2 *Early Stopping.*
The described cardinality estimation technique considers all the matching queries to the given input query on the queries pool. However, we can configure the technique for early stopping. That is, taking into account all the matching queries in the pool is not always necessary. We can set a limit on the number of matching queries that are used to estimate the input query cardinality, and thus obtain predictions faster by considering only a subset of the matching queries.

In the reported experiments we consider all the queries in the pool since the pool size is limited as described in Section 6.2.

6 CARDINALITY EVALUATION

We evaluate our proposed technique for estimating cardinality, with different test sets, while using the CRN model as defined in Section 3.2 for estimating containment rates.

We compare our cardinality estimates with those of the PostgreSQL version 11 cardinality estimation component [1], a simple and commonly used method for cardinality estimation. In addition, we compare our cardinality estimates with those of the MSCN model [22]. MSCN was shown to be superior to the best methods for estimating cardinalities such as Random Sampling (RS) [5, 37] and the state-of-the-art Index-Based Join Sampling (IBJS) [27].

In order to make a fair comparison between the CRN model and the MSCN model, we train the MSCN model with the *same* data that was used to train the CRN model. The CRN model takes two queries as input, whereas the MSCN model takes one query as input. Therefore, to even the playing field, we created the training dataset for the MSCN model as follows. For each pair of queries ($Q1, Q2$) used in training the CRN model, we added the following two input queries to the MSCN training set:

- $Q1 \cap Q2$, along with its actual cardinality.
- $Q1$, along with its actual cardinality.

Finally, we ensure that the training set includes only unique queries without repetition. This way, both models, MSCN and CRN, are trained with the *same* information. Note that comparing with the profiling and histograms-based PostgreSQL does not require generating training set.

We create the test workloads using the same queries generator used for creating the training set of the CRN and the MSCN models (described in Section 3.1.2), while skipping its last step. That is, we only run the first two steps of the generator. The third step creates query pairs which are irrelevant for the cardinality estimation task.

6.1 Evaluation Workloads

We evaluate our approach on the (challenging) IMDb dataset, using three different query workloads:

- crd_test1, a synthetic workload generated by the same queries generator that was used for creating the training data of the CRN model, as described in Section 3.1 (using a different random seed) with 450 unique queries, with zero to two joins.
- crd_test2, a synthetic workload generated by the same queries generator as the training data of the CRN model, as described in Section 3.1 (using a different random seed) with 450 unique queries, with zero to *five* joins. This dataset is designed to examine how the technique generalizes to additional joins.
- scale, another synthetic workload, with 500 unique queries, derived from the MSCN test set as introduced in [22]. This dataset is designed to examine how the technique generalizes to queries that were *not* created with the same queries generator used for training.

number of joins	**0**	**1**	**2**	**3**	**4**	**5**	**overall**
crd_test1	150	150	150	0	0	0	450
crd_test2	75	75	75	75	75	75	450
scale	115	115	107	88	75	0	500

Table 3: Distribution of joins.

6.2 Queries Pool

Our technique relies on a queries pool, we thus created a synthetic queries pool, QP, generated by the same queries generator as the training data of the containment rate estimation model, as described in Section 3.1 (using a different random seed) with 300 queries, equally distributed among all the possible FROM clauses over the database. In particular, QP, covers all the possible FROM clauses that are used in the test workloads. Note that, there are no shared queries between the QP queries and the test workloads queries.

Consider a query Q whose cardinality needs to be estimated. On the one hand, the generated QP contains "similar" queries to query Q. These can help the machine in predicting the cardinality. On the other hand, it also includes queries that are not similar at all to query Q. These may cause erroneous cardinality estimates. Therefore, the generated queries pool QP, faithfully represents a real-world situation.

6.3 Experimental Environment

In all the following cardinality estimation experiments, for predicting the cardinality of a given query Q in a workload W, we use the whole queries pool QP as described in Section 6.2 with all its 300 queries. That is, the "old" queries used for predicting cardinalities, are the queries of QP. In addition, in all the experiments we use the Median function as the final F function.

6.4 The Quality of Estimates

Figure 7 depicts the q-error of the Cnt2Crd(CRN) model as compared to MSCN and PostgreSQL on the crd_test1 workload. While PostgreSQL's errors are more skewed towards the positive domain, MSCN is competitive with Cnt2Crd(CRN) in all the described values. As can be seen in Table 4, while MSCN provides the best results in the margins, the Cnt2Crd(CRN) model is more accurate in 75% of the tests (as it is less accurate, in the margins, than MSCN with queries that have up to two joins). In addition, we show in the next section (Section 6.5) that the Cnt2Crd(CRN) model is more robust when considering queries with more joins than in the training dataset.

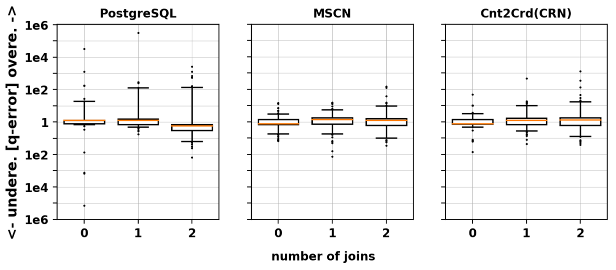


Figure 7: Estimation errors on the crd_test1 workload. In all the similar plots presented in this paper, the box boundaries are at the 25th/75th percentiles and the horizontal lines mark the 5th/95th percentiles. Hence, 50% of the tests results are located within the box boundaries, and 90% are located between the horizontal lines. The orange horizontal line mark the 50th percentile.

	50th	75th	90th	95th	99th	max	mean
PostgreSQL	1.74	3.72	22.46	149	1372	499266	1623
MSCN	2.11	4.13	**7.79**	**12.24**	**51.04**	**184**	**4.66**
Cnt2Crd(CRN)	**1.83**	**3.71**	10.01	18.16	76.54	1106	9.63

Table 4: Estimation errors on the crd_test1 workload.

6.5 Generalizing to Additional Joins

We examine how our technique generalizes to queries with additional joins, without having seen such queries during training. To do so, we use the crd_test2 workload which includes queries with zero to *five* joins. Recall that we trained both the CRN model and the MSCN model only with query pairs that have between zero and two joins.

From Tables 5 and 6, and Figure 8, it is clear that Cnt2Crd(CRN) model is significantly more robust in generalizing to queries with additional joins. This is clearly illustrated in the Cnt2Crd(CRN) box plot. The boxes are almost within the same q-error interval, close to q-error 1, which is the best q-error (obtained when an estimate is 100% accurate). In terms of mean q-error, the Cnt2Crd(CRN) model reduces the mean by a factor x100 and x1000 compared with MSCN and PostgreSQL, respectively.

	50th	75th	90th	95th	99th	max	mean
PostgreSQL	9.22	289	5189	21202	576147	4573136	35169
MSCN	4.49	119	3018	6880	61479	388328	3402
Cnt2Crd(CRN)	**2.66**	**6.50**	**18.72**	**72.74**	**528**	**6004**	**34.42**

Table 5: Estimation errors on the crd_test2 workload.

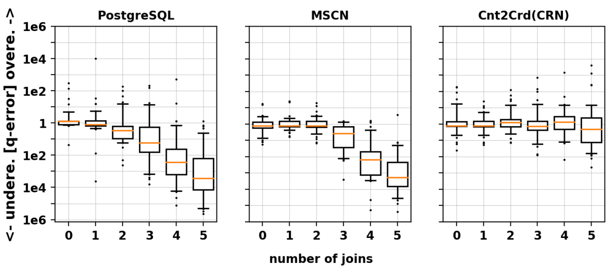


Figure 8: Estimation errors on the crd_test2 workload.

	50th	75th	90th	95th	99th	max	mean
PostgreSQL	229	3326	22249	166118	2069214	4573136	70569
MSCN	121	1810	6900	25884	83809	388328	6801
Cnt2Crd(CRN)	**4.28**	**10.84**	**43.71**	**93.11**	**1103**	**6004**	**61.26**

Table 6: Estimation errors on the crd_test2 workload considering only queries with three to five joins.

To highlight these improvements, we describe, in Table 7 and Figure 9, the mean and median q-error for each possible number of joins separately (note the logarithmic y-axis scale in Figure 9).

The known cardinality estimation models suffer from producing under-estimated results and errors that grow exponentially as the number of joins increases [14]. This also happens in the cases we examined. The Cnt2Crd(CRN) model was better at handling additional joins (even though CRN was trained only with queries with up to two joins, as was MSCN). The reason why the Cnt2Crd(CRN) model successfully generalizes to additional joins lies in its use of the queries pool. The queries pool contains queries with a *similar number of joins* as the input queries, along with their true cardinalities. The underlying CRN model estimates the containment rates accurately even when considering a high number of joins. As a result, the Cnt2Crd(CRN) cardinality estimates are accurate as well.

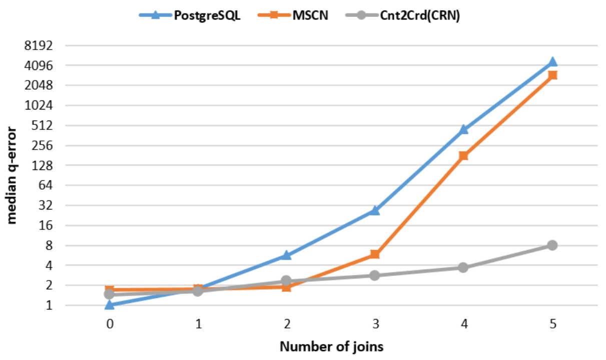


Figure 9: Q-error medians for each number of joins.

number of joins	0	1	2	3	4	5
PostgreSQL	10.41	216	25.38	355	4430	210657
MSCN	**3.44**	3.56	**3.31**	81.95	5427	14895
Cnt2Crd(CRN)	12.43	**3.54**	6.77	**23.24**	**30.51**	**129**

Table 7: Q-error means for each number of joins.

6.6 Generalizing to Different Kinds of Queries

In this experiment, we explore how the Cnt2Crd(CRN) model generalizes to a workload that was not generated by the same queries generator that was used for creating the CRN model training set. To do so, we examine the *scale workload* that is generated using another queries generator in [22]. As shown in Table 8, Cnt2Crd(CRN) is clearly more robust than MSCN and PostgreSQL in all the described percentiles. Examining Figure 10, it is clear that the Cnt2Crd(CRN) model is significantly more robust with queries with 3 and 4 joins. Recall that the *QP* queries pool in this experiment was not changed, while the scale workload is derived from *another* queries generator. In summary, this experiment shows that Cnt2Crd(CRN) generalizes well to workloads that were created with a different generator than the one used to create the training data.

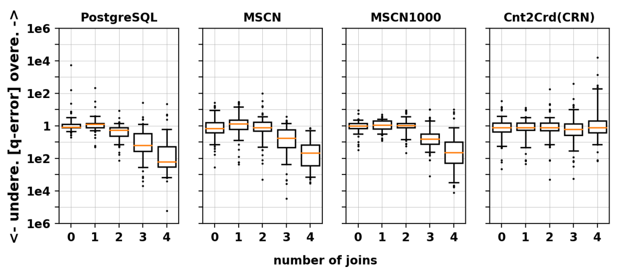


Figure 10: Estimation errors on the scale workload.

	50th	75th	90th	95th	99th	max	mean
PostgreSQL	2.62	15.42	183	551	2069	233863	586
MSCN	3.76	16.84	100	448	3467	47847	204
Cnt2Crd(CRN)	**2.53**	**5.88**	**24.02**	**95.26**	**598**	**19632**	**69.85**

Table 8: Estimation errors on the scale workload.

To further examine how Cnt2Crd(CRN) generalizes, we conducted the following experiment. We compared the Cnt2Crd(CRN) model with an improved version of the MSCN model that combines the deep learning approach and sampling techniques by using samples of 1000 materialized base tables, as described in [22]. We denote this model as MSCN1000.

We make the test "easier" for MSCN1000 model by training the MSCN1000 model with a training set that was created with the *same* queries generator that was used for generating the scale workload. As depicted in Figure 10, the MSCN1000 model is more robust in queries with zero to two joins, still, the Cnt2Crd(CRN) model is superior on queries with additional joins. Recall that the CRN model training set *was not changed*, while the MSCN1000 model was trained with queries obtained from the *same* queries generator that was used for creating the test (i.e., scale) workload. In addition, note that the MSCN1000 model uses sampling techniques whereas Cnt2Crd(CRN) does not. Thus, this experiment further demonstrates the superiority of Cnt2Crd(CRN) in generalizing to additional joins.

We obtain these improvements for the same reason described in Section 6.5. The CRN model is more robust in generalizing for additional unseen (during training) joins. As a result, the Cnt2Crd(CRN) model generalizes well for cardinality estimation.

7 IMPROVING EXISTING CARDINALITY ESTIMATION MODELS

In this section we describe how existing cardinality estimation models can be improved using the idea underlining our proposed technique. The proposed technique for improving existing cardinality estimation models relies on the same technique for predicting cardinalities using a containment rate estimation model, as described in Section 5.3.

In the previous section we used the CRN model in predicting containment rates. CRN can be replaced with *any* other method for predicting containment rates. In particular, it can be replaced with any existing cardinality estimation model after "converting" it to estimating containment rates using the Crd2Cnt transformation, as described in Section 4.1.

At first glance, our proposed technique seems to be a more complicated method for solving the problem of estimating cardinalities. However, we show that by applying it to known existing models, we improve their estimates, without changing the models themselves. These results indicate that the traditional approach, which directly addressed this problem, straightforwardly, using models to predict cardinalities, can be improved upon.

In the remainder of this section, we described the proposed approach, and show how existing cardinality estimation methods are significantly improved upon, by using this technique.

7.1 Approach Demonstration

Given an existing cardinality estimation model M, we first convert M to a model M' for estimating containment rates, using the Crd2Cnt transformation, as described in Section 4.1. Afterwards, given the obtained containment rate estimation model M', we convert it to a model M'' for estimating cardinalities, using the Cnt2Crd transformation, as described in Section 5.3, which uses a queries pool.

To summarize, our technique converts an existing cardinality estimation model M to an intermediate model M' for estimating containment rates, and then, using M' we create a model M'' for estimating cardinalities with the help of the queries pool, as depicted in Figure 5 from left to right.

For clarity, given cardinality estimation model M, we denote the model M'' described above, i.e., model Cnt2Crd(Crd2Cnt(M)), as *Improved M* model.

7.2 Existing Models vs. Improved Models

We examine how our proposed technique improves the PostgreSQL and the MSCN models, by using the crd_test2 workload as defined in Section 6.1, as it includes the most number of joins. Table 9 depicts the estimates when using directly the PostgreSQL or MSCN models, compared with the estimates when adopting our technique with each one of these models (i.e., the Improved PostgreSQL model and the Improved MSCN model). Examining the results, it is clear that the proposed technique significantly improves the estimates (by a factor x7 for PostgreSQL and x122 for MSCN in terms of mean q-error) without changing the models themselves (embedded within the Improved version).

The reason why the existing cardinality estimation models obtain better estimates (when adopting our technique) stems from the fact that these models are generalizing better when they are converted to estimate containment rates. Thus, along with the use of the queries pool, when these models are converted back to estimate cardinalities, they obtain better estimates.

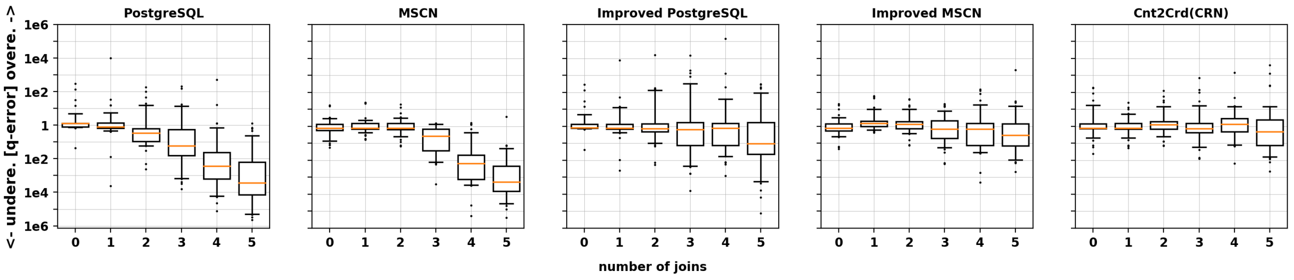


Figure 11: Estimation errors on the crd_test2 workload, compared with all models.

These results highlight the power of our proposed approach. The approach provides an effective and simple technique for improving existing cardinality estimation models. By adopting our approach and crating a queries pool in the database, cardinality estimates can be improved significantly.

	50th	75th	90th	95th	99th	max	mean
PostgreSQL	9.22	289	5189	21202	576147	4573136	35169
Improved PostgreSQL	**2.61**	**19.3**	**155**	**538**	**17697**	**1892732**	**5081**
MSCN	4.49	119	3018	6880	61479	388328	3402
Improved MSCN	**2.89**	**7.43**	**25.26**	**55.73**	**196**	**3184**	**27.78**

Table 9: Estimation errors on the crd_test2 workload.

7.3 Improved Models vs. Cnt2Crd(CRN)

Using the crd_test2 workloade, we examine how our technique improves PostgreSQL and MSCN, compared with Cnt2Crd(CRN). Examining Table 10, it is clear that in 90% of the tests, the best estimates are those obtained when directly using the CRN model to estimate the containment rates, instead of converting existing cardinality estimation models to obtain containment rates (Improved MSCN and Improved PostgreSQL). It seems that the CRN model is more accurate in estimating containment rates since it is directly designed for performing this task, whereas existing cardinality estimation models need to first be converted in order to estimate containment rates using the Crd2Cnt transformation.

	50th	75th	90th	95th	99th	max	mean
Improved PostgreSQL	2.61	19.3	155	538	17697	1892732	5081
Improved MSCN	2.89	7.43	25.26	**55.73**	**196**	**3184**	**27.78**
Cnt2Crd(CRN)	**2.66**	**6.50**	**18.72**	72.74	528	6004	34.42

Table 10: Estimation errors on the crd_test2 workload.

8 CARDINALITY PREDICTION TIME

Using the idea of using containment rates estimations to predict cardinalities, the cardinality prediction process is dominated by calculating the containment rates of the given input query with the relevant queries in the queries pool, and calculating the final function F on these results to obtain the predicted cardinality, as described in Section 5.3. Therefore, the larger the queries pool is, the more accurate the predictions are, and the longer the prediction time is. Table 11, shows the medians and the means estimation errors on the crd_test2 workload, along with the average prediction time for a single query, when using the Cnt2Crd(CRN) model for estimating cardinalities, with different sizes of QP (equally distributed over all the possible FROM clauses in the database) while using the same final function F (the Median function).

QP Size	50	100	150	200	250	300
Median	3.68	2.55	2.63	2.55	2.61	2.66
Mean	1894	90	41	40	35	34
Prediction Time	3.2ms	7.1ms	9.8ms	11.3ms	14.5ms	16.1ms

Table 11: Median and mean estimation errors on the crd_test2 workload, and the average prediction time, considering different queries pool (QP) sizes.

In table 12, we compare the average prediction time for estimating the cardinality of a single query using all the examined models (when using the whole QP queries pool of size 300). The default MSCN model is the fastest model, since it directly estimates the cardinalities without using a queries pool. The Cnt2Crd(CRN) model is the fastest among all the models that use a queries pool. That is, the Cnt2Crd(CRN) model is faster than the Improved MSCN model and the Improved PostgreSQL model. This is the case, since in the Improved MSCN model or the Improved PostgreSQL model, to obtain the containment rates, both models need to estimate cardinalities of two different queries as described in Section 4.1, whereas the CRN model directly obtains a containment rate in one pass within 0.5ms (see Section 3.5).

Although the prediction time of the models that use queries pools is higher than the most common cardinality estimation model (PostgreSQL), the prediction time is still in the order of a few tens milliseconds. In particular, it is similar to the average prediction time of models that use sampling techniques, such as the MSCN version with 1000 base tables samples.

For the results in Table 12, we used a queries pool (QP) of size 300. We could have used a smaller pool (or adapt the early stopping technique as mentioned in Section 5.3.2), resulting in faster prediction time, and still obtaining better results, as depicted in Table 11. Furthermore, all the the models that use queries pools may be easily parallelized as discussed in Section 5.3, and thus, reducing the prediction time (we ran these models serially in the reported tests).

Model	Prediction Time
PostgreSQL	1.75ms
MSCN	0.5ms
MSCN with 1000 samples	33ms
Improved PostgreSQL	70ms
Improved MSCN	35ms
Cnt2Crd(CRN)	16ms

Table 12: Average prediction time of a single query.

9 RELATED WORK

Over the past five decades, conjunctive queries have been studied in the contexts of database theory and database systems. Conjunctive queries constitute a broad class of frequently used queries. Their expressive power is roughly equivalent to that of the Select-Join-Project queries of relational algebra. Numerous problems and associated algorithms have been researched in depth in this context. Chandra and Merlin [10] showed that determining (analytic) containment of conjunctive queries is an NP-complete problem. Finding the minimal number of conditions that need to be added to a query in order to ensure containment in another query is also an NP-complete problem [44]. This also holds in additional settings involving inclusion and functional dependencies [2, 19, 44].

Although determining whether query $Q1$ is contained in query $Q2$ (analytically) in the case of conjunctive queries is an intractable problem in its full generality, there are many tractable cases. For instance, in [41, 42] it was shown that query containment of conjunctive queries could be solved in linear time, if every database (edb) predicate occurs at most twice in the body of $Q1$. In [12] it was proved that for every $k \geq 1$, conjunctive query containment could be solved in polynomial time, if $Q2$ has querywidth smaller than $k + 1$. In addition to the mentioned cases, there are many other tractable cases [8, 9, 16, 40]. Such cases result from imposing syntactic or structural restrictions on the input queries $Q1$ and $Q2$.

Whereas analytic containment was well researched in the past, to our knowledge, the problem of determining the containment rate on a *specific* database has not been investigated. In this paper, we address this problem using ML techniques.

Lately, we have witnessed extensive adoption of machine learning, and deep neural networks in particular, in many different areas and systems, and in particular in databases. Recent research investigates machine learning for classical database problems such as join ordering [31], index structures [23], query optimization [24, 38], concurrency control [4], and recently in cardinality estimation [22, 45]. MSCN, a recently conceived sophisticated NN model, estimates cardinalities [22]. MSCN has been shown to be superior in estimating cardinalities for queries that have the same number of joins as that in the queries training dataset. However, MSCN proved less effective when considering queries with more joins. In this paper, we propose a deep learning-based approach, inspired by the MSCN model, for predicting containment rates on a *specific* database. Additionally, we show how containment rates can be used to predict cardinalities more accurately.

There were many attempts to tackle the problem of cardinality estimation; for example, Random Sampling techniques [5, 37], Index based Sampling [27], and recently deep learning [22, 45]. However, all these attempts have addressed, conceptually, the problem directly in the same way, as a black box, where the input is a query, and the output is the cardinality estimate. In this paper, we address this problem differently by using information (the actual cardinalities) about queries that have already been executed in the database.

A similar idea of using the information contained in the execution results of queries was used to refine and update columns of histograms. In this approach, histograms are incrementally refined every time they are used, by comparing the histogram estimated selectivity to the actual selectivity. This leads to more accurate histograms, and to better cardinality estimates [3, 7, 13, 20].

10 CONCLUSIONS AND FUTURE WORK

We introduced a new problem, that of estimating containment rates between queries over a *specific* database, and introduced the CRN model, a new deep learning model for solving it (inspired by MSCN [22]). We trained CRN with generated queries, uniformly distributed within a constrained space, and showed that CRN usually obtains the best results in estimating containment rates as compared with other examined models.

We introduced a novel approach for cardinality estimation, based on the CRN-based containment rate estimation model, and with the help of a queries pool. We showed the superiority of our new approach in estimating cardinalities more accurately than state-of-the-art approaches. Further, we showed that our approach addresses the weak spot of existing cardinality estimation models, which is handling multiple joins.

In addition, we proposed a technique for improving *any* existing cardinality estimation model (M) without the need to change the model itself, by embedding it within a three step method ($Cnt2Crd(Crd2Cnt(M))$). Observe that it is possible to further improve the estimation by using the obtained improved model $Cnt2Crd(Crd2Cnt(M))$, and generating models (repeatedly), e.g., $Cnt2Crd(Crd2Cnt(Cnt2Crd(Crd2Cnt(M))))$[6]. Given that the estimates of state-of-the-art models are quite fragile, and that our technique for estimating cardinalities is simple, has low overhead, and is quite effective, we believe that it is highly promising and practical for solving the cardinality estimation problem.

We considered cardinality estimation for SQL queries *not* using the *DISTINCT* keyword. For various intermediate results, a query planner requires the set-theoretic cardinality (without duplicates). For example, employing counting techniques for handling duplicates, considering sorting, creating an index or a hash table, and more. This requirement may therefore limit our techniques' usability. One may use our (inaccurate for this case) predictions as proxies. However, a better technique is needed and we are currently evaluating a promising extension of our machine learning approach for predicting set-theoretic cardinalities (i.e., queries with the *DISTINCT* keyword).

To make our containment based approach suitable for more general queries, the CRN model for estimating containment rates can be extended to support other types of queries, such as queries that include complex predicates. In addition, the CRN model can be configured to support databases that are updated from time to time. Next, we discuss some of these extensions, and sketch possible future research directions.

Strings. A simple addition to our current implementation may support equality predicates on strings. To do so, we could hash all the possible string literals in the database into the integer domain (similarly to MSCN). This way, an equality predicate on strings can be converted to an equality predicate on integers, which the CRN model can handle.

Complex predicates. Complex predicates, such as LIKE, are not supported since they are not represented in the CRN model. To support such predicates we need to change the model architecture to handle such predicates. Note that predicates such as BETWEEN and IN, may be converted to ordinary predicates.

EXCEPT Operator. Given a query Q of the form $Q1$ EXCEPT $Q2$, we can estimate its cardinality using our technique as follows:

$$|Q1\ EXCEPT\ Q2| = |Q1| - |Q1 \cap Q2|$$

[6]This observation is due to one of the referee.

UNION Operator. Given a query Q of the from $Q1$ UNION $Q2$, we can estimate its cardinality using our technique as follows:

$$|Q1\ UNION\ Q2| = |Q1| + |Q2|$$

Observe that for handling both the EXCEPT and the UNION operators, the cardinality of queries $Q1$, $Q2$ and $Q1 \cap Q2$ can be estimated using our technique, as they are conjunctive queries.

The OR operator. Given queries that include the OR operator in their WHERE clause, the CRN model does not handle such queries straightforwardly. But, we can handle such queries using a promising recursive algorithm that we are currently evaluating.

Database updates. Thus far, we assumed that the database is static (read-only database). However, in many real world databases, updates occur frequently. In addition, the database schema itself may be changed. To handle updates we can use one of the following approaches:

(1) We can always completely re-train the CRN model with a new updated training set. This comes with a considerable compute cost for re-executing queries pairs to obtain up-to-date containment rates and the cost for re-training the model itself. In this approach, we can easily handle changes in the database schema, since we can change the model encodings prior to re-training it.

(2) We can *incrementally* train the model starting from its current state, by applying new updated training samples, instead of re-training the model from scratch. While this approach is more practical, a key challenge here is to accommodate changes in the database schema. To handle this issue, we could hold, in advance, additional place holders in our model to be used for future added columns or tables. In addition, the values ranges of each column may change when updating the database, and thus, the normalized values may be modified as well. Ways to handle this problem are the subject of current research.

REFERENCES

[1] . .. PostgreSQL, The World's Most Advanced Open Source Relational Database. *https://www.postgresql.org/* (.).

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

[3] Ashraf Aboulnaga and Surajit Chaudhuri. 1999. Self-tuning Histograms: Building Histograms Without Looking at Data. In *SIGMOD.* 181–192.

[4] Rajesh Bordawekar and Oded Shmueli (Eds.). 2019. *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD.* ACM.

[5] Marco Bressan, Enoch Peserico, and Luca Pretto. 2015. Simple set cardinality estimation through random sampling. *CoRR* abs/1512.07901 (2015).

[6] Jason Brownlee. 2018. When to Use MLP, CNN, and RNN Neural Networks. *https://machinelearningmastery.com/when-to-use-mlp-cnn-and-rnn-neural-networks/* (2018).

[7] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2001. STHoles: A Multidimensional Workload-Aware Histogram. In *Proceedings SIGMOD.* 211–222.

[8] Andrea Calì. 2006. Containment the Conjunctive Queries over Conceptual Schemata. In *Proceedings the Database Systems for Advanced Applications conference,DASFAA.* 628–643.

[9] Edward P. F. Chan. 1992. Containment and Minimization of Positive Conjunctive Queries in OODB's. In *Proceedings SIGACT-SIGMOD-SIGART.* 202–211.

[10] Ashok K. Chandra and Philip M. Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing.* 77–90.

[11] Gloria Chatzopoulou, Magdalini Eirinaki, and Neoklis Polyzotis. 2009. Query Recommendations for Interactive Database Exploration. In *Scientific and Statistical Database Management, 21st International Conference, SSDBM.* 3–18.

[12] Chandra Chekuri and Anand Rajaraman. 2000. Conjunctive query containment revisited. *Theor. Comput. Sci.* 239, 2 (2000), 211–229.

[13] Chung-Min Chen and Nick Roussopoulos. 1994. Adaptive Selectivity Estimation Using Query Feedback. In *Proceedings ACM SIGMOD.* 161–172.

[14] James Clifford and Roger King (Eds.). 1991. *Proceedings the 1991 ACM SIGMOD International Conference on Management of Data.* ACM Press.

[15] Magdalini Eirinaki, Suju Abraham, Neoklis Polyzotis, and Naushin Shaikh. 2014. QueRIE: Collaborative Database Exploration. *IEEE Trans. Knowl. Data Eng.* 26, 7 (2014), 1778–1790.

[16] Carles Farré, Werner Nutt, Ernest Teniente, and Toni Urpí. 2007. Containment of Conjunctive Queries over Databases with Null Values. In *Proceedings ICDT.* 389–403.

[17] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, and David A. Patterson. 2009. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *Proceedings ICDE.* 592–603.

[18] Rojeh Hayek and Oded Shmueli. 2019. Improved Cardinality Estimation by Learning Queries Containment Rates. *CoRR* abs/1908.07723 (2019). arXiv:1908.07723 http://arxiv.org/abs/1908.07723

[19] David S. Johnson and Anthony C. Klug. 1984. Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. *J. Comput. Syst. Sci.* 28, 1 (1984), 167–189.

[20] Navin Kabra and David J. DeWitt. 1998. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD.* 106–117.

[21] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings ICLR.* http://arxiv.org/abs/1412.6980

[22] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *Proceedings CIDR.*

[23] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings SIGMOD.* 489–504.

[24] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR* abs/1808.03196 (2018).

[25] M. Seetha Lakshmi and Shaoyu Zhou. 1998. Selectivity Estimation in Extensible Databases - A Neural Network Approach. In *Proceedings VLDB.* 623–627.

[26] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

[27] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *Proceedings CIDR.*

[28] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668.

[29] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. Cardinality estimation using neural networks. In *Proceedings the 25th Annual International Conference on Computer Science and Software Engineering,CASCON.* 53–59.

[30] Guy Lohman. 2014. IS QUERY OPTIMIZATION A "SOLVED" PROBLEM ? *https://wp.sigmod.org/?p=1075* (2014).

[31] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings aiDM@SIGMOD.* 3:1–3:4.

[32] Alberto O. Mendelzon and Jan Paredaens (Eds.). 1998. *Proceedings the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* ACM Press.

[33] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *PVLDB* 2, 1 (2009), 982–993.

[34] R. Monga, M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *OSDI 16.* 265–283.

[35] Raghunath Nambiar and Meikel Poess (Eds.). 2018. *Performance Evaluation and Benchmarking for the Analytics Era - 9th TPC Technology Conference, TPCTC.* Lecture Notes in Computer Science, Vol. 10661. Springer.

[36] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. 2018. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *CoRR* abs/1811.03378 (2018).

[37] Frank Olken and Doron Rotem. 1990. Random Sampling from Database Files: A Survey. In *Proccedings SSDBM.* 92–111.

[38] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *Proceedings, DEEM@SIGMOD.* 4:1–4:4.

[39] Lutz Prechelt. 2012. Early Stopping - But When? In *Neural Networks: Tricks of the Trade - Second Edition.* 53–67.

[40] Guillem Rull, Philip A. Bernstein, Ivo Garcia dos Santos, Yannis Katsis, Sergey Melnik, and Ernest Teniente. 2013. Query containment in entity SQL. In *Proceedings ACM SIGMOD.* 1169–1172.

[41] Y. Saraiya. 1991. Subtree elimination algorithms in deductive databases. *PhD thesis, Department of Computer Science, Stanford University.* (1991).

[42] Yatin P. Saraiya. 1990. Polynomial-time Program Transformations in Deductive Databases. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '90).* ACM, 132–144.

[43] Cristina Sirangelo. 2018. Positive Relational Algebra. In *Encyclopedia of Database Systems, Second Edition.* Computer Science Press.

[44] Jeffrey D. Ullman. 1989. *Principles the Database and Knowledge-Base Systems, Volume II.* Computer Science Press.

[45] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. 2019. Cardinality estimation with local deep learning models. In *Proceedings aiDM@SIGMOD.* 5:1–5:8.

Automated Management of Indexes for Dataflow Processing Engines in IaaS Clouds

Herald Kllapi
Department of Informatics and Telecommunications,
University of Athens, Greece
herald@di.uoa.gr

Ilia Pietri
Department of Informatics and Telecommunications,
University of Athens, Greece
ipietri@di.uoa.gr

Verena Kantere
School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
verena@dblab.ece.ntua.gr

Yannis Ioannidis
ATHENA Research and Innovation Center, Greece
Department of Informatics and Telecommunications,
University of Athens, Greece
yannis@di.uoa.gr

ABSTRACT

Data structures like indexes are typically used to accelerate dataflow execution locating and accessing data more efficiently. The automated management of data structures has been a challenging problem, traditionally constrained by the time and storage required to build and maintain them. As cloud computing is becoming an attractive platform for the execution of dataflows with the usage of compute and storage resources being charged by cloud providers, monetary cost is becoming an equally important factor for the user to consider. In this work, we identify the opportunity of interleaving dataflow and index build operators in the execution schedule to utilize idle slots for the creation of indexes which may be beneficial for future dataflows. In that way, the cost of building indexes can be eliminated without impacting dataflow execution. We propose an online auto-tuning approach to assess the importance of indexes for the workload based on historical data taking into account the trade-off between the dataflow speed-up they offer and the monetary cost needed to maintain them. The results show that the proposed approach can dynamically adapt to the workload and significantly reduce the average execution time and cost spent per dataflow building and maintaining a proper set of indexes.

1 INTRODUCTION

Modern applications face the need to process large amounts of data using complex functions for analysis [40], data mining [32], Extract-Transform-Load processes (ETL) [45], and more. Such rich tasks are typically expressed in high-level languages like Pig Latin [39], optimized and transformed into data processing flows, or simply dataflows, that describe computations (operators) and flow dependencies between them [34],[33],[48].

Dataflows are usually executed on distributed systems to process independent operators in parallel and reduce overall execution time. Among these, clouds have evolved to a popular platform for large-scale data processing, mainly due to the lack of any upfront investment and *elasticity* (the ability to lease resources on demand for as long as needed). Cloud providers offer compute resources in the form of virtual machines (VMs) which are typically charged based on a per quantum pricing scheme (e.g. one hour) such as Amazon EC2 [3], and storage resources which are usually charged per GB per month [5].

Data structures like indexes and materialized views are additionally used to improve the performance of dataflows, encapsulating prior computations to access data more quickly and avoid unnecessarily large data movements during dataflow execution [36]. Building and maintaining indexes may be costly in terms of computation and storage, often exceeding the gain in performance [30]. However, in several cases the costs can be amortized. For example, the time overhead required for the creation of indexes may be reduced by building them in parallel. Also, indexes are usually associated to multiple dataflows and can thus be exploited for the execution of future dataflows. As the existence of indexes may improve application performance, but may also affect the monetary cost incurred [22, 41], it is important to find a good trade-off between these two conflicting objectives. Hence, index tuning (the selection of indexes based on their usefulness) is required to avoid uncontrolled creation and maintenance of data structures. This task may become even more challenging, when the workload is not known a-priori and the set of indexes may change dynamically over time.

We envision a Query-as-a-Service (QaaS) platform to manage the execution of complex dataflow workloads on clouds, like Google's BigQuery[1]. Dataflows, such as exploratory data-intensive queries, are issued sequentially by the user, e.g. a data scientist, to extract knowledge from data. Each dataflow is associated with a set of indexes that can benefit its execution. The service incorporates automated management of suggested indexes by creating and deleting them based on their usefulness on the dataflow workload. These indexes can either be computed automatically or incorporate feedback from administrators to generate useful recommendations [16, 29, 43]. This is an orthogonal problem and the integration of already proposed solutions would easily work with our approach. For example, most index advisors can output a set of indexes that might be useful (e.g., by doing a what-if analysis). This would be the input to our system.

Building a generic model that captures dataflows and indexes is an open research problem, mainly because operators may have arbitrary user code that is often impossible to analyze, and the usefulness of an index may be specific to each dataflow. However, this is beyond the scope of this work. We identify five generic categories of dataflow operators where indexes can be useful:

- *Lookup*. The complexity of finding a particular record from an input table of size n is $O(n)$ when no data structure is used and can be reduced to $O(log\ n)$ using a B-tree index or $O(1)$ using a hash index.

[1]Google Big Query, https://cloud.google.com/products/big-query

- *R*ange select. Selecting records in a particular range from the input can be efficiently performed using a B-Tree index because the leaves of the tree are sorted. The complexity is $O(log\ n) + k$ where k is the number of records in the range.
- *S*orting. The complexity of operators that perform sorting is $O(n \cdot log\ n)$ and can be reduced to $O(n)$ using a B-Tree index.
- *G*rouping. Grouping can be efficiently performed using sorting, as described above.
- *J*oin. Several algorithms, such as nested loops join, hash join, sort-merge join, can be used. Such algorithms are faster when an appropriate index is provided. For example, the complexity of sort-merge join is $O(n + m)$ if the inputs (of size n and m) are sorted.

In this work, we propose an online auto-tuning approach to assess the usefulness of indexes for the execution of dataflows taking into account the trade-off between the dataflow speed-up they offer and the monetary cost needed to maintain them (the storage cost in the cloud). We identify the opportunity to build indexes and eliminate their cost using slots of *idle time* on compute resources. These may be created due to data dependency constraints between the execution of dataflow operators but also the provider's quantized pricing policy. Building entire indexes sequentially using idle compute resources may not be feasible due to the large data volume [41]. Hence, indexes on partitions of tables or files are built independently. In this way, indexes can be built in parallel and, most importantly, may fit inside idle slots. The approach proposed in this work is generic and can be used in several large-scale data processing platforms, like Hadoop [7], Hive [46], or Pig [39]. Several systems like [26, 35, 46] have been developed to provide highly scalable distributed architectures for data processing on the Cloud; however, the monetary cost and the quantized pricing of resources need to be considered [23]. To the best of our knowledge, there is no index management solution that takes into account the monetary cost of using cloud resources, while related work on execution time and cost optimization of dataflows does not consider building and maintaining indexes.

The main contributions of this work are the following:

- We identify the opportunity to use idle slots on compute resources created when executing data-intensive flows due to data dependency constraints between operators but also the quantum-based pricing policy of compute resources.
- We propose an online auto-tuning approach to assess the importance of indexes based on the trade-offs between the dataflow execution speed-up they offer and the monetary cost needed to maintain them.
- We develop two index interleaving algorithms, namely linear program based interleaving and online interleaving algorithms, to utilize idle slots in the dataflow execution schedule and build indexes in parallel without increasing the monetary cost and the time required for the execution of each dataflow.
- We provide an experimental evaluation to show the effectiveness of the proposed approach to accelerate dataflow execution and eliminate the related monetary costs.

The rest of the paper is organized as follows. Related work is discussed in Section 2. The problem description follows in Section 3, while the optimization problem is defined in Section 4. The online auto-tuning approach and interleaving algorithms proposed are described in Section 5. The experimental evaluation and its results follow in Section 6, while Section 7 concludes the paper.

2 RELATED WORK

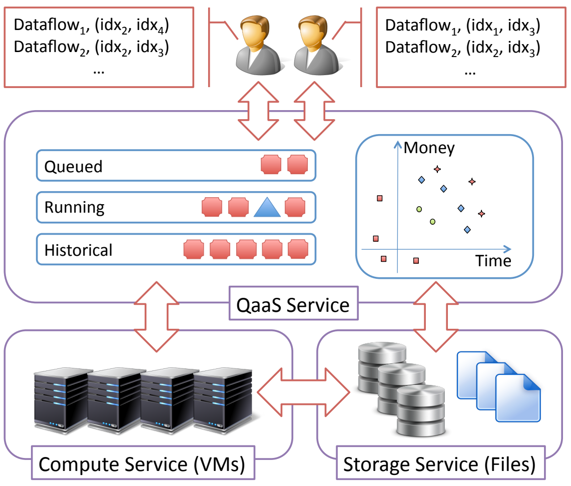
A considerable body of work focuses on VM consolidation to exploit underutilized resources for the execution of multiple workloads [14, 51]. However, consolidating different workloads may greatly affect application performance due to interference, as consolidated VMs may compete for resources [53]. In contrast, the idea of this work is to interleave dataflow and index build operators in the execution schedule to accelerate dataflow execution while eliminating the cost of building indexes.

Offline algorithms for index tuning on centralized systems like [10, 16] do not consider a dynamic environment where the service is unaware of the dataflows and a priori predictions of how long to keep and when to delete indexes cannot be made. Our approach is closer to online algorithms like [9, 38, 52]. However, we target a distributed and elastic environment where VMs are allocated dynamically and compute resources are prepaid for the whole time quanta. Also, what-if optimizations that improve index tuning [16] are complementary to our work and can be used to accelerate the computation of index usefulness. Approaches that incorporate feedback from administrators to improve index recommendations [29, 43] are also orthogonal to our work, as user feedback can be beneficial for the computation of index usefulness. The problem of index interactions has also been studied [42, 44]. Such efforts could be leveraged in our work to delete indexes that become obsolete when index interactions in the dataflow workload are identified.

Online algorithms for distributed environments like [13, 20, 41, 47] focus on replicated databases, investigating which sets of indexes to build on each replica and how to route queries properly to take advantage of them. Such approaches can be used in combination with our proposed approach since multiple replicas for each partition are typically created in distributed environments to increase efficiency and fault tolerance [24]. Indexing mechanisms on clouds like [11, 15, 36] mainly focus on the optimization of application performance and ignore the monetary cost of using the resources. The monetary cost of data structures has been considered in multi-user environments [30, 49] to distribute the creation and maintenance costs of data structures among multiple users. However, our work focuses on single-user environments where resources allocated to the user are dedicated and data structures built are not shared among multiple users. This way, each user is independent and the provider's pricing policy for compute and storage resources like Amazon Elastic MapReduce [4] can be directly used, without considering complex cost sharing policies that users may or may not agree with. Finally, the work in [21] considers the problem of data structure reuse by future queries, materializing and storing the output of operators of MapReduce jobs. To the best of our knowledge, there is no index management approach for single-user environments that takes into account the monetary cost of using cloud resources.

3 PROBLEM DESCRIPTION

Figure 1 shows the architectural framework envisioned in this work. The typical users of the QaaS service are data scientists that issue exploratory query tasks to extract knowledge from data, such as data intensive transformations that perform processing and aggregations on data read from tables or files. The data can be partitioned for flexibility, performance, and fault tolerance and indexes on each partition of tables or files can be built. Each user interacts with the service by issuing dataflows sequentially,



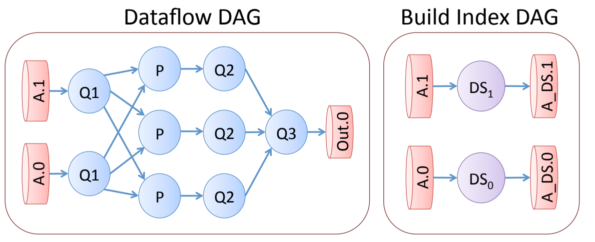
Dataflow$_1$, (idx$_2$, idx$_4$)
Dataflow$_2$, (idx$_2$, idx$_3$)
Dataflow$_1$, (idx$_1$, idx$_3$)
Dataflow$_2$, (idx$_2$, idx$_3$)
Queued
Running
Historical
Money
Time
QaaS Service
Compute Service (VMs)
Storage Service (Files)

Figure 1: The setting of the QaaS service.

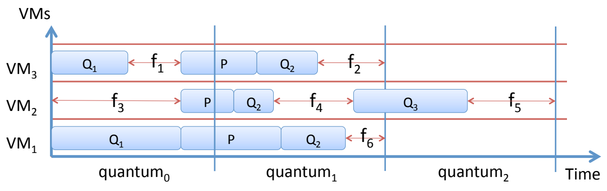
usually observing the results obtained from the execution of a single dataflow before submitting the next one. The service executes the dataflows on top of clouds according to selected execution schedules with desired time-money trade-offs using compute and storage services offered by cloud providers. The execution of the dataflow is interleaved with the execution of build index operators and the indexes created are stored in the cloud storage. Each dataflow submitted for execution has access to currently available indexes and each operator can make use of those associated to partitions it accesses.

Motivation. A dataflow example is shown in Figure 2a. As can be seen in the graph of the DAG (left part of the figure), the dataflow uses two partitions, $A.0$ and $A.1$, of an input table A and performs processing ($Q1$, $Q2$), partitioning (P), and aggregations ($Q3$). The dataflow is associated to a set of indexes ($A\_DS.0$ and $A\_DS.1$) built for the table partitions ($A.0$ and $A.1$, respectively) as shown in the right part of the figure. There are two indexes to be built: A built in three parts $A0$, $A1$ and $A2$ and B built in parts $B0$, $B1$ and $B2$. Parts can be created in parallel using different VMs. Figure 2b shows an execution schedule of the dataflow operators when using 3 VMs ($VM1$, $VM2$, $VM3$). Arrows show the idle slots created due to data dependency constraints and the quantized pricing policy ($f1 - f6$). For example, slot $f4$ on $VM2$ remains idle as $Q3$ cannot be executed until all $Q2$ operators have finished. Such idle slots can be used for the building of indexes without incurring any additional cost, as shown in Figure 2c. Different indexes can be built in parallel such as the case of A_1 and B_0. The execution of the index build operator A_1 at $VM2$ is stopped as it is not completed before the execution of the dataflow operator P starts so that the execution of the dataflow is not delayed. Similarly, B_2 is stopped before the time the leased quantum expires to avoid unnecessary costs for building indexes.

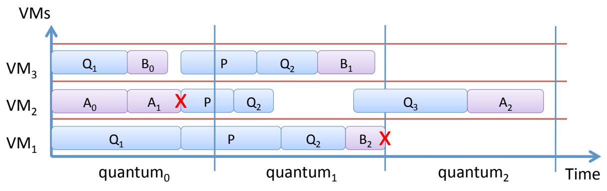
Application Model. A dataflow d is modelled as $d(expr, R, N, t)$, where $expr$ is its definition expressed in an appropriate language, R is the set of input tables, N is the set of indexes that can accelerate its execution, and t is the time point that the dataflow is issued to the service. The dataflow is modelled as a DAG where the nodes correspond to operators and the edges to data dependencies (flows) between them. An *operator* is modelled as $op(cpu, memory, disk, time)$, where cpu is the CPU utilization,



(a) dataflow and build index DAGs.



(b) Execution schedule of the dataflow DAG.



(c) Interleaving of dataflow and build index operators.

Figure 2: Execution of an example dataflow and indexes

mem is the maximum memory needed for its normal operation, $disk$ is the disk resources, and $time$ is its execution time. A flow between two operators is labelled with the size of the data transferred between them. The estimations of operators can be computed analytically or can be collected by the system at runtime [37]. Since we target large datasets, the statistics (e.g., histograms) do not change radically over time (a 10GB update on 1TB dataset is not large enough to change the statistics). The dataflow processing rate is much higher than the rate at which the data is updated. This is the typical case in many settings: updates are done every few days and the datasets are processed much more frequently [27]. Also, operators come from a set that does not change frequently, which is typical for exploratory data analysis [36].

Cloud Model. Compute resources are offered in the form of VMs (or containers) with a fixed capacity of resources, CPU, memory, disk, and network, respectively. Each VM is charged at a fixed price (M_c) per time quantum (Q) and can be dynamically allocated and deallocated based on the workload needs. In this work, homogeneous VMs are assumed. This is typical for many installations; most VMs are of the same size and only few VMs which run critical services are significantly larger (like namenodes of Hadoop [7]). An idle VM (a VM that is not used) is deleted when its currently leased time quantum expires, since the resources are prepaid for whole time quanta [3]. Each VM has a local disk that can be used to store temporary results or data. After deleting a particular VM, the files stored in its local disk cannot be recovered.

A storage service is used to store data persistently; VMs retrieve data from the storage service and cache it to their local disks and transfer data to the storage service after the execution of an operator finishes. This scheme is flexible as compute from storage resources are decoupled. Typically, cloud providers charge a fixed amount of money per GB per month (MC). In the model used, the cost of storing data, M_{st}, is measured in GB per time quantum (Q). As a year has approximately 365.25 days and, assuming that Q is measured in minutes, we compute M_{st} as $(MC \cdot 12 \cdot Q)/(365.25 \cdot 24 \cdot 60)$.

Data Model. Tables can be partitioned and stored to the storage service of the cloud. As mentioned, allocated VMs can cache table partitions to their local disk to avoid network overheads when possible. Data updates are performed in batches periodically (every day or week). Each update creates a new version of the table partitions changed [2], invalidating old versions and indexes built on them. A table t in the database is modelled by its schema (i.e., the names and types of its columns), its ordered set of partitions, and its statistics: $t(schema, P, S)$. A partition $p \in P$ is modelled by its id, its number of records n and a particular path in the storage service where the partition is located: $p(id, n, path)$. The statistics contain the average size of the fields of each column.

An index idx built on table t is modelled as $idx(t, C, T)$, where C is the ordered set of columns based on which the index is built and T is the ordered set with the respective creation time points of its partitions. Each index consists of several index partitions built on different table partitions. Index partitions can be built on different time periods. The *index size* is computed by adding the sizes of its partitions. The size of a partition can be computed based on the type of the index (e.g. Hash, B^+Tree). We assume without loss of generality that B^+Trees are used. The size of partition p of index idx is computed as follows:

$$size(idx, p) = (1 - k^{\log_k(p.n)}) \cdot RecSize/(1 - k),$$

where $RecSize$ is the average size of the record in the index, computed from column statistics, and k is the width of the tree computed from the block size on the disk and the record size $RecSize$. Assuming that the tree is balanced, its size is computed using geometric series as follows: the total number of records including the non-leaf blocks is $\sum_{i=0}^{m} k^i = (1-k^m)/(1-k)$, where m is the height of the tree computed as $m = \log_k N$. Parameter N represents the number of records in the partition. The time to build an index idx, $t_i(idx)$, is computed by adding the time to build all the index partitions of the corresponding table. The time to build the index on a partition p is computed as:

$$t_{ip}(idx, p) = t_{io}(idx, p) + C(idx) \cdot p.n \cdot \log_k(p.n)/T_Q,$$

where $C(idx)$ is a constant calculated using the columns in the index. The time to read and write the partition $t_{io}(p)$ is computed as:

$$t_{io}(idx, p) = (p.n \cdot RecSize + size(idx, p))/cont.net,$$

where $cont$ is the container to which the build index operator is assigned for execution. The building of indexes can be expressed as a DAG with operators that take as input one partition and build the partial index on that partition. Operators are independent to each other (there are no edges between the operators in the DAG) and as a result there is a large degree of parallelism. Hence, indexes can be built incrementally (not all index partitions need to be built in order to use the index) and in parallel (two or more index partitions can be built simultaneously). The storage cost of index idx for a time period W (given in time quanta) is computed by adding the cost $stp(idx, p, W)$ of storing each index partition p for that period, where

$$stp(idx, p, W) = W \cdot size(idx, idx.t.P[p]) \cdot M_{st}.$$

Our approach can work with different pricing models. A pricing model is plugged to the scheduler by using the appropriate pricing formulas for the cost of a VM (M_C) and the cost of storage (M_{st}). Also, although we consider a homogeneous environment with a single VM type, the scheduler can consider slots at different VM types.

Dataflow and Index Management. The dataflows are issued sequentially to the service. Historical dataflows (dataflows that have already been executed) are stored in a list called H_d. An execution schedule S_d of a dataflow graph d is a set of assignments of its operators to containers. An execution schedule is computed taking into account the network communication cost using the model in [33]. The execution time of a dataflow in schedule S_d, $t_d(S_d)$, is defined as the time period from the time the first operator starts executing until the time the execution of the last operator finishes. The monetary cost $m_d(S_d)$ is computed taking into account the sum of the total time quanta of the VMs leased. The monetary cost and execution time are measured in quanta in order to have the same unit. An idle slot $f(id, q, c, S_d)$ in schedule S_d is a continuous time period inside the leased time quantum q of the container, c, that has no operators running. The fragmentation of the schedule is the set of all the idle slots in the leased containers and shows the time the compute resources are not used during the execution schedule, but they are charged by the cloud provider.

Idle slots can be used for the building of indexes. We denote as I the evolving ordered set of indexes built and maintained by the service. The set of indexes available at time t is denoted as $I(t)$ and the set of all indexes created during the operation of the service (independently of whether they are deleted or not) is denoted as I. Potential indexes are indexes that are associated with one or more dataflows, but they are not beneficial to build. Indexes built on table partitions that are updated are deleted and marked as *not built* to support index updates.

4 OPTIMIZATION PROBLEM

This work considers the problem of interleaving indexes with the execution of dataflows so that dataflow execution, in terms of execution time and monetary cost, is not affected. The aim is to determine the set of beneficial indexes required to build and maintain over time to achieve good trade-offs between the dataflow speedup and the monetary cost required taking into account the storage cost needed to maintain the indexes.

The optimization problem is formulated as a weighted single objective problem using a linear function to express the different tradeoffs between the time and money objectives:

$$\max_I \Big[\sum_i M_c \cdot (\alpha \cdot \delta t_d(d_i) + (1-\alpha) \cdot \delta m_d(d_i)) - \sum_j st(I[j]) \Big], \quad (1)$$

where d is the dataflow, $st(I[j])$ is the storage cost of index $I(j)$, $\delta t_d(d_i)$ is the difference (given in quanta) in dataflow execution time without and with the use of indexes and $\delta m_d(d_i)$ is the difference (quanta) in the monetary cost required without and with the use of indexes. Parameter α essentially expresses how much money a time quantum is valued, taking values between 0 and 1 that correspond to scenarios where the optimization

Table 1: Notation used.

Parameter	Meaning
T_Q	Quantum size
M_c	VM price (per quantum)
M_{st}	Storage cost (per GB per quantum)
$I(t)$	The set of indexes available at time t
a	Parameter for time-cost trade-off
$st(idx, W)$	Storage cost of index idx within a time window W
$gt(idx, t)$	Gain in time for index idx at time t
$gm(idx, t)$	Gain in money for index idx at time t
$dc(t)$	Gain fading function
$t_i(idx)$	Time for building index idx
$m_i(idx)$	Monetary cost for building index idx

problem becomes one-dimensional. Small values of α indicate that monetary cost (or money) is more important, while large values of α indicate scenarios where time is more important. The difference in time $\delta t_d(d_i)$ is multiplied with the container price per quantum (M_c) so that the time and money objectives have the same units.

In a dynamic environment where arbitrary dataflows are issued at arbitrary time points using different sets of potential indexes, it is hard to find the optimal sequence of index sets ($I(t)$), i.e., determine when to build and delete indexes using Equation 1. We formulate the optimization problem to a more suitable form (Equation 2) for the computation in an online fashion:

$$I(t) = arg \max_{I} \Big[\sum_{idx \in I} (\alpha \cdot M_c \cdot gt(idx, t) + (1-\alpha) \cdot gm(idx, t)) \Big], \quad (2)$$

where functions $gm(idx, t)$ and $gt(idx, t)$ (described in Equations 4 and 5) compute the gain in money and time, respectively, when using a particular index idx at time point t and within a time window of predefined size W (e.g., two quanta). Table 1 summarizes the notation used to define the optimization problem. Equation 2 can be approximated in an online fashion by computing the gain of each index individually, building and maintaining only those that contribute in a positive way to the summation at any given point in time. More specifically, an index idx is said to be beneficial at time point t if its gain (the weighted summation of time and money gain of Equation 3) is positive.

$$g(idx, t) = \alpha \cdot M_c \cdot gt(idx, t) + (1 - \alpha) \cdot gm(idx, t) \quad (3)$$

Indexes are built as soon as they become beneficial and are deleted as soon as they become non beneficial.

The gain in money $gm(idx, t)$ of index idx is computed by adding the gain in money $gm_d(idx, d_i)$ of the index on each related dataflow d_i (the dataflows that use it and are evaluated inside time window $[t - W, t]$ and the currently running dataflow) and the monetary cost required to build and store the index for time period W, as described in Equation 4:

$$gm(idx, t) = \sum_i \Big(\delta(d_i, t) \cdot dc(\delta T_{d_i}) \cdot M_c \cdot gm_d(idx, d_i)\Big) - (M_c \cdot m_i(idx) + st(idx, W)) \quad (4)$$

where $\delta(d_i, t)$ is 1 if the dataflow f has been executed during time period $[t - W, t]$ or else 0, δT_{d_i} is the number of quanta passed since the dataflow d_i was executed (0 for the ones that are currently running or queued) and $dc(t)$ is a function that reduces with time in order to fade the gain of the historical dataflows. An exponential function is used to fade the gain: $dc(t) = e^{-t/D}$, where parameter D controls the degree the historical dataflows affect it. A small value of D means that $dc(t)$ approaches quickly to 0 and, as a consequence, $gm(idx)$ becomes negative. We assume that D is the same for all indexes. However, different values of the controller D can be used for each individual index. Automatic learning of the controller for each index based on predictions is a direction for future work. Also, $m_i(idx)$ is the monetary cost required to build the index, $st(idx, W)$ the storage cost required to maintain it for a time window W and $gm_d(idx, d_i)$ is the gain in money of dataflow d_i when using index idx which is computed based on the time gain of the index on d_i. The gain in money $gm_d(idx, d_i)$ also includes the monetary cost spent to read the index from the storage service, which is equivalent to the time to read the index, as both of them are measured in quanta. If dataflow d_i does not use index idx, then $gm_d(idx, d_i) = 0$.

Similarly, the time gain $gt(idx, t)$ of index idx at time point t is computed taking into account the gains of index idx on the dataflows executed within the time window W, subtracting the time needed to build it as follows:

$$gt(idx, t) = \sum_i \Big(\delta(d_i, t) \cdot dc(\delta T_{d_i}) \cdot gt_d(idx, d_i)\Big) - t_i(idx) \quad (5)$$

where $gt_d(idx, d_i)$ is the gain in time of dataflow d_i when using index idx.

An example to illustrate the proposed approach is presented. Assume the dataflows shown in Table 2 are issued to the service at the time points specified. The dataflows use two indexes, A (of size 100 MB) and B (of size 500 MB). The time and money gain of the indexes for each dataflow is included in the table. Figure 3 shows the gain of each index computed over time for the case of $\alpha = 0.5$ and $D = 60$. It can be seen that the gain of both indexes, A and B, is negative in the beginning due to their storage cost. As dataflows specify them as useful, the gain becomes positive at some point (the indexes become beneficial) and then decreases over time because of parameter D (that impacts index usefulness). For example, index B becomes beneficial at time point 30 and will be deleted at time point 125 where it stops being useful.

Table 2: Dataflows Issued using Indexes A and B.

Dataflow	Time Gain	Money Gain
$d_1(-, -, \{B\}, 10)$	$gt_d(B, d_1) = 1.0$	$gm_d(B, d_1) = 3.0$
$d_2(-, -, \{B\}, 30)$	$gt_d(B, d_1) = 2.0$	$gm_d(B, d_1) = 5.0$
$d_3(-, -, \{A, B\}, 50)$	$gt_d(A, d_1) = 2.0$ $gt_d(B, d_1) = 3.0$	$gm_d(A, d_1) = 8.0$ $gm_d(B, d_1) = 8.0$
$d_4(-, -, \{A\}, 100)$	$gt_d(A, d_1) = 3.0$	$gm_d(B, d_1) = 5.0$

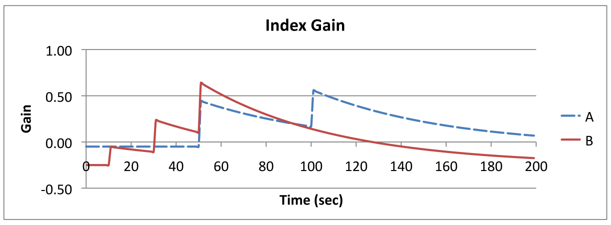


Figure 3: Gain over time of two indexes A and B.

5 AUTO-TUNING APPROACH

In this section, we propose an auto-tuning approach to select and build an optimal set of indexes over time. Statistics from historical (issued) dataflows and their specified indexes are continuously collected and used to make decisions about which indexes to build or delete at each time point. Dataflows to be executed can only use indexes that are currently available, while new indexes to be built are scheduled with the currently issued dataflow. Indexes are built using idle slots in the execution schedule of the issued dataflow so that the dataflow execution is not affected. However, beneficial

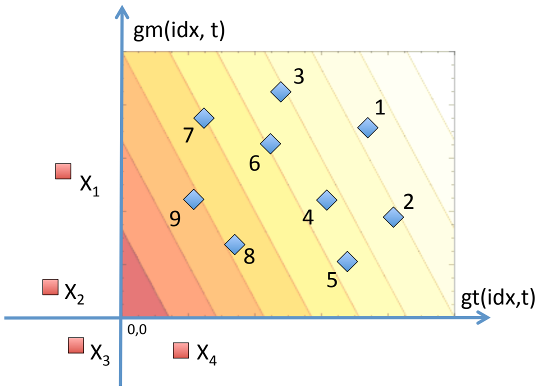


Figure 4: Index ordering based on α at time point t.

indexes to be built may not fit in the currently available slots and selecting which beneficial indexes to build is required. Since the goal is to maximize the optimization objective of Equation 2, indexes are ranked based on their usefulness and the best subset to be built is selected.

5.1 Index Ranking

Equation 3 is used to compute the usefulness or gain of indexes at each time point, as described earlier in Section 4. Non beneficial indexes are not built or are deleted if they are already available. Note that an index that is not used for a long time may become non-beneficial because of the increased storage cost. We only consider beneficial indexes with $gm(idx, t)$ and $gt(idx, t)$ (Equations 4 and 5) larger than 0 and among them higher values of gain (Equation 3) are preferred. Essentially, indexes can be depicted in a bi-dimensional space based on their time and money gain as shown in Figure 4. Indexes at the lighter areas are prioritized. For example, point 1 has the highest priority for a=0.7, while indexes X_1, X_2, X_3, and X_4 are not beneficial.

5.2 Online Index Tuning

The online index tuning approach proposed is shown in Algorithm 1. The algorithm schedules the issued dataflow along with the subset of potential indexes that maximize the total gain. Beneficial indexes are assigned to idle compute resources in the dataflow execution schedule without violating the constraints (i.e, the time and the monetary cost of the dataflow are not affected). Indexes that are not beneficial or cannot fit to the schedule are deleted. Note that partitions of a particular index can be built in the context of several dataflows if there is not enough idle time to build it entirely in the context of one dataflow. The algorithm is triggered every time a new dataflow is issued, the execution of a dataflow finishes or periodically at fixed time intervals to delete indexes that become non beneficial when there is not any new dataflow to be issued. In more detail, the procedure triggered is the following. The gains in time and cost for each index are computed and beneficial indexes are ranked (lines 2-9 of Algorithm 1) as described in Section 5.1. Then, the algorithm calls the index interleaving procedure to compute the skyline of execution schedules of the dataflow df interleaved with build index operators and selects from the skyline the schedule to be executed (lines 10-11). Different methodologies can be used to choose the schedule to be executed. In this work, the fastest schedule is chosen. In lines 13-19 the algorithm identifies index partitions that are not beneficial and need to be deleted.

Algorithm 1 Online Index Tuning

```
                    Input:
      $H_d$: The historical dataflows.
      $A_i$, $B_i$, $P_i$: The index lists.
      $df$: The next dataflow to schedule.
                    Return:
      $S_{df}$: The schedule of the dataflow.
      $S_{BI}$: The schedule of the build indexes.
      $DI$: The indexes that should be deleted.
 1: $GAINS \leftarrow \emptyset$
 2: for $i \in P_i$ do
 3:     $gt \leftarrow gt(i, H_d \cup df)$
 4:     $gm \leftarrow gm(i, H_d \cup df)$                ▷ Compute the index gains
 5:     if $gt > 0$ and $gm > 0$ then
 6:         $GAINS \leftarrow GAINS \cup \{i\}$
 7:     end if
 8: end for
 9: $RANK \leftarrow rank2Dspace(GAINS)$                    ▷ rank the indexes
10: $skyline \leftarrow schedule(df, A_i, RANK)$
                       ▷ Scheduling of both the dataflow and indexes
11: $S_{df}, S_{BI} \leftarrow select(skyline)$       ▷ Select the schedule from skyline
12: $DI \leftarrow \emptyset$
13: for $i \in A_d$ do
14:     $gt \leftarrow gt(i, H_d \cup df)$
15:     $gm \leftarrow gm(i, H_d \cup df)$
16:     if $gt \leq 0$ and $gm \leq 0$ then
17:         $DI \leftarrow DI \cup \{i\}$                      ▷ Indexes to be deleted
18:     end if
19: end for
20: return ($S_{df}, S_{BI}, DI$)
```

5.3 Index interleaving approaches

In this section, we propose two different approaches to schedule dataflows interleaved with build index operators without using additional monetary cost, namely the Linear program based interleaving algorithm (LP) and the online interleaving algorithm. The LP interleaving algorithm initially schedules the currently issued dataflow and finds the idle slots in the compute resources. Then, it uses a linear programming algorithm to determine the subset of potential index partitions and tries to assign them on the idle slots based on their ranking (gain). The online interleaving algorithm schedules the current dataflow and the index build operators together labeling the index build operators as optional operators to be scheduled.

5.3.1 Linear program based interleaving algorithm. The LP interleaving algorithm shown in Algorithm 2 schedules indexes after the dataflows. More specifically, the algorithm initially updates the operator runtimes based on the available index partitions. Estimations of runtimes can be provided based on existing models [50]. The algorithm calls the scheduler described in Algorithm 4 to compute the skyline of the execution schedules (line 6). For each schedule in the skyline, the algorithm finds the set of idle slots and sorts them in decreasing order based on their size (lines 8-10). For each slot, a linear program (line 12) is solved to determine the subset of potential indexes that maximize the total gain. The build index operators in each idle slot are sorted by gain so that the building of less useful indexes is stopped when the time quantum ends or the next assigned operator is scheduled (as shown in Figure 2c) before the build index operator finishes due to runtime estimation errors. The build index operators whose execution has been stopped are queued and scheduled with the next dataflow issued. Overall, the algorithm does not violate the constraints (i.e. index interleaving does not affect dataflow execution in terms of time and money) as indexes are built on slots that are not used for the execution of dataflow operators, but they are charged.

Algorithm 2 Linear program based interleaving algorithm

Input:
df: The current dataflow from the input.
A_i: The available indexes.
I: The ranked list of indexes.
Return:
$skyline$: The skyline of solutions.

```
 1: for $op \in df$ do
 2:     if $op$ uses indexes in $A_i$ then
 3:         update($op$, $A_i$)             ▷ op. runtimes based on available indexes
 4:     end if
 5: end for
 6: skyline ← Skyline($df$)           ▷ generate skyline of execution schedules
 7: for s in skyline do
 8:     idle_time ← FindIdleTime(s)
 9:     ordered_idle_time ← OrderBySize(idle_time)
10:     indexes ← $\cup(I.P)$
11:     for i in ordered_idle_time do
12:         maxset ← SolveLinearProgram(i, indexes)
                                  ▷index set to be built based on linear program
13:         for m in maxset do
14:             schedule(m, i)                   ▷ assign indexes to idle slots
15:         end for
16:         indexes ← indexes - maxset
17:     end for
18:     add $indexes$ to $s$
19: end for
20: return skyline              ▷ schedules of dataflow ops and index build ops
```

Algorithm 3 Linear Program Algorithm

Input:
f: The size of the idle time segment.
p_i: The sizes of all the build index partition operators.
g_i: The gain of all the build index partition operators.
Return:
The subset of the build index operators that maximize Equation 2

```
1: $max\left[\sum_i (w_i * g_i)\right]$
   w.r.t
2: $\sum_i (w_i * p_i) \leq f$
3: $0 \leq w_i \leq 1, \forall i$
4: $integer(w_i), \forall i$
5: return $(w_1, w_2, \ldots w_n)$
```

Linear program approximation algorithm. The problem of assigning build index operators into idle time slots on compute resources is a variation of the Knapsack [31] problem, which is NP-hard. The Linear program approximation algorithm shown in Algorithm 3 is an approximation algorithm to solve a 0/1 knapsack problem for each idle time slot. The algorithm solves the relaxed problem setting the weights of the build index operators between the values 0 and 1 and calls a branch and bound algorithm to find integer values.

Skyline dataflow scheduler. Different execution schedules that vary in the achieved execution time and monetary cost can be created by assigning the dataflow operators to potential slots of the available VMs. Between them, non-dominated solutions (solutions that outperform others in terms of execution time and monetary cost) may be preferred. The set of non-dominated solutions achieved comprises the obtained *skyline* of execution schedules. The algorithm in [12] is used to develop the skyline of execution schedules for each dataflow. An operator is candidate for assignment when all of its predecessors are assigned, starting from operators without data dependencies (entry nodes in the dataflow graph). At each iteration, the algorithm (Algorithm 4) assigns the next available operator to the partial solutions of the current skyline taking into account the communication costs and data dependency constraints between the operators. After the assignment of the new operator to all the possible slots, the new skyline is computed. Between schedules with the same execution

Algorithm 4 Skyline Dataflow Scheduler

Input: df: The dataflow DAG.
C: The maximum number of containers to use.
Output: $skyline$: The solutions in the skyline.

```
 1: $skyline \leftarrow \oslash$
 2: $ready$ ←{operators in $df$ that have no dependencies}
 3: $firstOperator \leftarrow ready.peek()$
 4: $firstSchedule \leftarrow \{assign(firstOperator, 1, -, -)\}$
 5: $skyline$ ←{$firstSchedule$}
 6: while $ready \neq \oslash$ do
 7:     $next \leftarrow ready.peek()$
 8:     $S \leftarrow \oslash$
 9:     for all schedules $s$ in $skyline$ do
10:         for all containers c (c ≤ C) do
11:             $S \leftarrow S \cup \{s + assign(next, c, -, -)\}$
12:         end for
13:     end for
14:     $skyline$ ← skyline of $S$              ▷ new skyline of schedules
15:     $ready \leftarrow ready - \{next\}$ ∪ {operators in $df$ that dependency con-
        straints no longer exist}
16: end while
17: return $skyline$
```

time and monetary cost, the schedule with the most sequential idle compute time is selected, since the aim of our work is to use idle slots where index build operators may fit. The procedure described is repeated for the next available operator. The algorithm terminates when all operators are assigned and the final skyline is generated. Note that the impact of data transfers on the execution of data-intensive dataflows may be significant and overhead may be introduced [18]. Thus, each dataflow is scheduled offline to generate more efficient schedules where the overhead from data transfers is considered.

5.3.2 Online interleaving algorithm. The online interleaving algorithm is a modification of the scheduler in [12] to use optional operators and schedule index build operators along dataflows. To do so, operators are separated to optional and non optional using a boolean variable; the variable is set to $true$ (optional operators for execution) for each index build operator while the variable is $false$ for all dataflow operators. Algorithm 4 is modified so that the schedules in each iteration may vary in the number of assigned operators. The ready operators list (line 2 of Algorithm 4) includes optional index build operators which are candidate for scheduling. If the operator *next* in line 7 is optional, the previous skyline (*skyline*) is kept and unioned with the set of schedules *S* (line 11) before computing the new skyline in line 14. As a result, the newly generated skyline may consist of schedules with different numbers of operators. Between schedules with the same execution time and money, schedules with a larger number of operators are preferred. Also, the schedules kept in the new skyline do not violate the constraints of the optimization problem, as solutions that belong to the initial skyline and have lower execution time or monetary cost will dominate solutions in the unioned set where the assignment of optional operators have affected dataflow execution. Hence, only schedules where the assignment of optional operators does not affect the dataflow execution time and monetary cost will be kept in the newly computed skyline.

6 EXPERIMENTAL EVALUATION

In this section, the proposed approach is evaluated based on simulation. The skyline dataflow scheduler described in Section 5.3.1 (*offline*) is evaluated using an online load balance scheduler (*online*) typically deployed in elastic clouds as baseline. The *online* algorithm examines the dataflow graph in an online greedy

Table 3: Experiment Parameters

Parameter	Values
Quantum size	60 seconds
Quantum cost	\$0.1
Storage Cost	\$$10^{-4}$ per MB per Quantum
Max Containers	100
Dataflow	Montage, Ligo, Cybershake
Operators / Dataflow	100
α	0.5
Index gain fading D	1 quantum
Poisson Generator λ	1 quantum
Total Time	720 quanta

fashion scheduling the operators to the available containers so that load balance is achieved. Finally, the two index interleaving algorithms, the *LP interleaving* algorithm and the *online interleaving* algorithm, are evaluated and compared using two baseline index management approaches: a naive approach that does not create indexes at all (*no indexes*) and an approach that randomly selects indexes from the potential set and randomly assigns them to containers to be built (*random*).

6.1 Experimental Setup

Table 3 summarizes the parameters used in the experiments. Homogeneous containers with similar capacity in resources (CPU, memory, disk, and network) are assumed. Each container has one CPU and one disk. The CPU and memory needs of each operator is specified as a percentage of container's CPU and memory respectively. A disk size of 100 GB and a speed of 250 MB/sec (typical SSD) are assumed. Allocated containers cache table partitions and indexes read from the storage service. A time quantum Q of 60 seconds is assumed. Pricing is based on *Amazon*'s billing policy [6]. The price M_c charged for the provisioning of each container per time quantum is set equal to \$0.1 and the storage cost M_{st} is set equal to \$$10^{-4}$ per MB per quantum. The storage of the cloud is computed by counting the number of bytes transferred and charging appropriately over time.

In the simulator used, user queries are sent to the scheduler, which adds them to a queue. Each query is transformed into an execution graph of operators with data dependencies. Given the execution graph, the scheduler selects a subset of containers and schedules the execution of the graph operators on these containers, respecting the graph dependencies. The set of active containers can be dynamically varied based on the demand. Each operator has a priority specified and each container has a queue with operators that are executed as soon as the memory needed is sufficient. Dataflow operators have priority 1 and build index operators have priority -1. Operators with negative priority are stopped when operators with positive priority arrive to the container or its current time quantum expires. A network bandwidth of 1 Gbps is assumed. The execution of an operator is delayed until its input data are transferred. Also, if an index is available and beneficial, the container reads the index in addition to the input of the operator, depending on the speedup it offers.

In the simulator used, each container has a local disk to cache input files from the storage service. If the data required as input from the operator are already in the cache, data transfer is considered to be 0. If the container cache gets full, LRU policy is used to create empty space. Containers that do not have any dataflow operators scheduled on them are deleted at the end of the leased quantum.

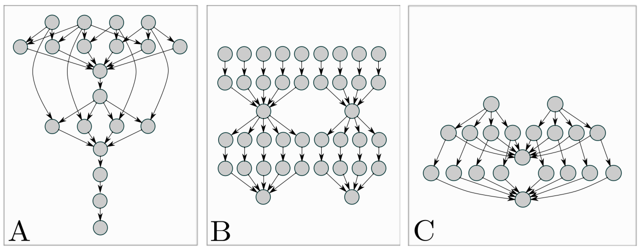


Figure 5: The dataflow graphs Montage(A), Ligo(B), and Cybershake(C).

Table 4: Basic statistics of the scientific dataflows.

Time (sec)	#	Min	Max	Mean	Stdev
Montage	100	3.82	49.32	11.32	2.95
Ligo	100	4.03	689.39	222.33	241.42
Cybershake	100	0.55	199.43	22.97	25.08
Input (MB)	**#**	**Min**	**Max**	**Mean**	**Stdev**
Montage	20	0.01	4.02	3.22	1.65
Ligo	53	0.86	14.91	14.24	2.70
Cybershake	52	1.81	19169.75	1459.08	5091.69

Table 5: Indexes on table lineitem.

Column	Type	Index Size	% Table Size
comment	text	422.30 MB	30.16 %
shipinstruct	20 chars	248.95 MB	17.78 %
commitdate	date	225.91 MB	16.13 %
orderkey	integer	146.99 MB	10.49 %

Synthetic data of three real scientific applications, namely Montage [28], Ligo [19] and Cybershake [17], are used to evaluate the proposed approach. Montage shown in Figure 5A is used to generate image mosaics of the sky, LIGO shown in Figure 5B is used to analyze galactic binary systems and Cybershake shown in Figure 5C is used for the characterization of earthquakes. The dataflows are produced using the generator in [8] which specifies the execution time of each operator, the dependencies between them and the sizes of the input/output files of each operator. The basic statistics of the operators are shown in Table 4.

The input files of the dataflows shown in Table 4 are used as a database of files. The total number of files is 125 and their total size is 76.69 GB. The maximum size of a file partition is set equal to 128 MB, resulting in a total number of 713 file partitions. The TPC-H benchmark [1] is used to compute the sizes of typical indexes and model the speed-ups they provide. Table `lineitem` with scale 2 which has approximately 12 million rows and a size of 1.4 GB is used. Table 5 shows the sizes of indexes on four different columns of the table. To model the speed-up that indexes offer, the following SQL queries were created based on the categories presented in Section 1:

Order by:
```
SELECT orderkey FROM lineitem
ORDER BY orderkey;
```
Select range (large):
```
SELECT orderkey FROM lineitem
WHERE orderkey > 1000000
  AND orderkey < 2000000;
```
Select range (small):

Table 6: Index speedup.

Query	No-Index	Index	Speedup
Order by	44.730 sec	6.010 sec	7.44x
Select range (large)	5.103 sec	0.054 sec	94.44x
Select range (small)	4.921 sec	0.016 sec	307.50x
Lookup	4.393 sec	0.007 sec	627.14x

```
SELECT orderkey FROM lineitem
WHERE orderkey > 10000
  AND orderkey < 20000;
```

Lookup:

```
SELECT orderkey FROM lineitem
where orderkey = 1000000;
```

Table 6 shows the speed-up the index on column orderkey offers. Four potential indexes for each file are used. Each index size is computed using the percentages shown in Table 5 and its speed-up is randomly chosen from the values of Table 6.

A *Dataflow Generator Client* issues dataflows at time points that follow a Poisson distribution. More specifically, the generator implemented computes the arrival time k (in seconds) of the next dataflow as $f(k;\lambda) = \Pr(X = k) = \lambda^k e^{-\lambda}/k!$, with λ equal to 60 seconds. Dataflows are generated using two settings: randomly (*random generator*) and with phases (*phase generator*). The phase generator produces dataflows to measure the adaptability of the proposed approach to workload changes as follows: Cybershake dataflows for 33.3 quanta (10000 sec), Ligo dataflows for 16.6 quanta (5000 sec), Montage dataflows for 66.6 quanta (20000 sec) and Cybershake dataflows for 27.3 quanta (8200 sec) with each generated dataflow having different speed-ups for the indexes it uses.

6.2 Scheduler robustness for estimation errors

In reality, operator runtimes and data sizes may be overestimated or underestimated. In the first set of experiments, the sensitivity of the scheduler to estimation errors is investigated. To do so, the runtime of operators and the data sizes they generate are randomly varied within a certain percentage and the difference between the actual and estimated values for time, money and fragmentation are computed. For example, for an estimation error of 10% a random value in the range of [90 - 110] seconds is selected to modify the runtime of an operator initially estimated at 100 seconds. Figure 6 shows the results for different values of estimation errors added on the CPU time (operator runtime) and data used. As can be seen, the estimations are robust considering that an error of more than 20% in operator runtime and datasize estimations is relatively high. When the estimations are extremely poor (large errors), the performance of the algorithm can be significantly affected. This is because the algorithm makes scheduling decisions offline (before dataflow execution) based on estimations of operator runtimes and datasizes and does not adapt to unpredicted changes. Future work could investigate how to incorporate estimation errors on decision making to account for inaccurate estimates and yield better performance.

6.3 Comparison of dataflow schedulers

In this set of experiments, the skyline dataflow scheduler proposed (*offline*) is compared with the online load balance scheduler typically used in IaaS clouds (*online*). Operator runtimes and data

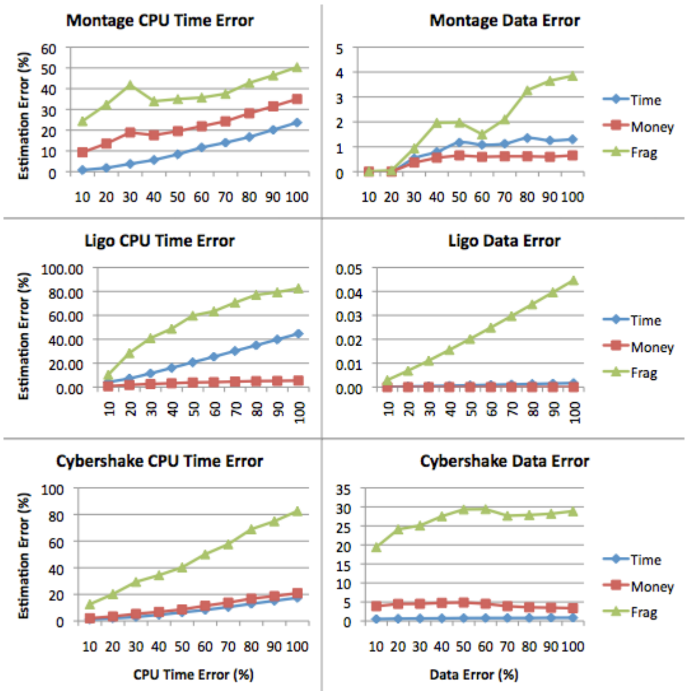
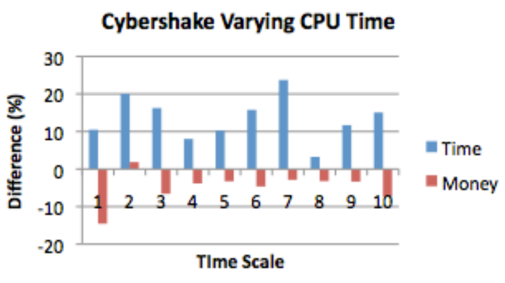


Figure 6: Sensitivity of the offline scheduler to inaccurate estimations.

sizes are scaled to evaluate the efficiency of the proposed scheduler for different scenarios. Since the online scheduler generates a single execution schedule, the fastest schedule from the skyline obtained using the proposed skyline dataflow scheduler (offline) is used for the comparison. The results for Cybershake (the results are similar for the other dataflows) are presented in Figure 7; the y-axis shows the difference (%) between the offline and the online scheduler. The left part of Figure 7 shows the results when scaling the operator runtimes up to 10x (shown in the x-axis) and keeping the data sizes small (scaled to 0.01 of the original size). The online scheduler performs well for these type of dataflows (CPU-intensive) generating faster but slightly more expensive schedules by balancing the load. However, load balancing does not work well for data-intensive dataflows where data placement greatly affects the execution of dataflows. The right part of Figure 7 shows the results when scaling the size of data up to 100x. It can be seen that the schedules generated by the online load balance scheduler are up to 2x slower and up to 4x more expensive compared to the proposed offline scheduler.

6.4 Comparison of index interleaving algorithms

In this experiment, we compare the two index interleaving algorithms proposed; the *LP interleaving* algorithm and the *online* interleaving algorithm. Figure 8 shows the number of indexes built at each schedule in the skylines obtained for Montage using the two index interleaving algorithms (the results are similar for the other two dataflows). The first observation is that the LP interleaving algorithm is able to schedule significantly more build index operators. This is because the information about the fragmented resources is available before the algorithm runs. In contrast, the online algorithm schedules the index build operators and the dataflow operators at the same time. Also, the two skylines obtained are not the same (as can be seen from the monetary cost that corresponds to each point). This is because the



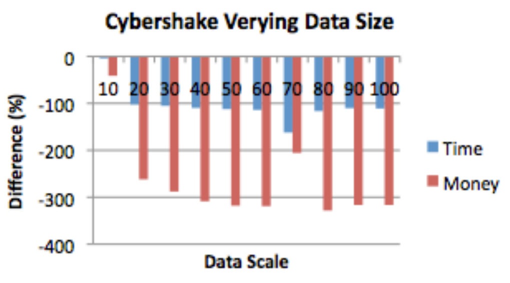


Figure 7: Comparison of the *online* and *offline* scheduler performance.

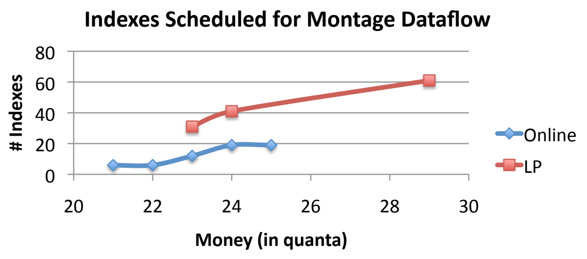


Figure 8: Number of indexes scheduled using different algorithms for Montage dataflow.

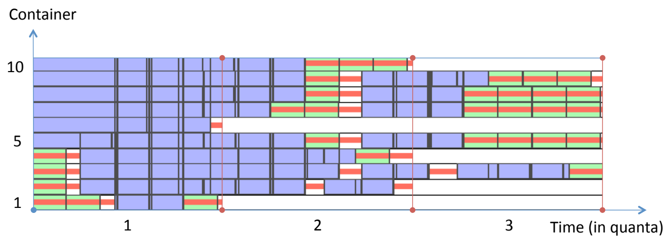


Figure 9: Montage with build index ops (green).

online algorithm interferes with the scheduling of the dataflow operators resulting in cheaper schedules.

Figure 9 shows an example with the timeline of Montage interleaved with build index operators scheduled by the LP interleaving algorithm. Dataflow operators are shown in blue and build index operators are shown in green. The red line indicates idle compute resources. We observe that the LP interleaving algorithm uses a significant amount of idle compute time. The initial idle time is 7.14 quanta and after the assignment of the build index operators, the fragmentation is reduced to 1.6 quanta.

We also compute an upper bound of the quality of the solution found by the LP algorithm by merging all the individual idle time periods and solving the knapsack problem using only one large continuous time segment. We do this using the example of Figure 10, which shows the times of the build index operators and the fragmented resources we used. For simplicity, we set the gain of each operator to be equal to its execution time. As a baseline, we compare with the following greedy algorithm (inspired by Graham [25]): first, we order the operators by descending execution times (and gain in this case) and proceed by assigning each operator to the idle time segment with the most remaining time. A build index operator that does not fit anywhere is not scheduled. Figure 11 shows the results of the LP interleaving algorithm compared to the baseline and the upper bound. We observe that the LP interleaving algorithm is able to find a solution close to the theoretical upper bound (within 5% in this experiment).

6.5 Dynamic Dataflow Workload

In this experiment, the efficiency of the proposed auto-tuning approach (shown in Algorithm 2) is evaluated and compared using the *no-indexes* and *random* approaches as baseline algorithms.

6.5.1 Dataflow Generator with Phases. Initially, the results obtained using the dataflow generator client with phases are presented. Figure 12 shows the number of dataflows finished after 720 time quanta using the different approaches. It can be seen that the number of dataflows executed is doubled when using the proposed approach compared to the baseline where no index is used. Furthermore, the monetary cost spent per dataflow is significantly reduced. It can also be seen that the random approach does not greatly affect the number of finished dataflows compared to the scenario of not using indexes (no index). However the average monetary cost per dataflow is significantly increased due to the storage cost required, which is not taken into account. Finally, the cost per dataflow is increased when non beneficial indexes are maintained, as can be seen by comparing the columns labelled as Gain (no delete) and Gain.

Table 7 shows the total number of operators executed and stopped due to quantum expiration or preemption for the execution of a dataflow operator. It can be seen that the packing achieved by the LP interleaving algorithm is better compared to the random algorithm and fewer build index operators are stopped prematurely.

Table 7: Operators executed.

Algorithm	Total Ops	Killed Ops	Percentage
No Index	22402	0	0
Random	25649	1143	4.4
Gain	49549	1418	2.8

Figure 13 shows the number of indexes built and the total storage cost over time. It can be seen that the proposed approach adapts to the workload by creating and deleting indexes when they become non-beneficial. When Cybershake is re-issued in the final phase, some previously deleted indexes become beneficial again and are recreated.

6.5.2 Random Dataflow Generator. In this experiment, a random dataflow generator client is used. Figure 14 shows the number of dataflows finished after 720 time quanta. The number of dataflows executed is larger using the proposed approach. This is because the average execution time per dataflow is reduced.

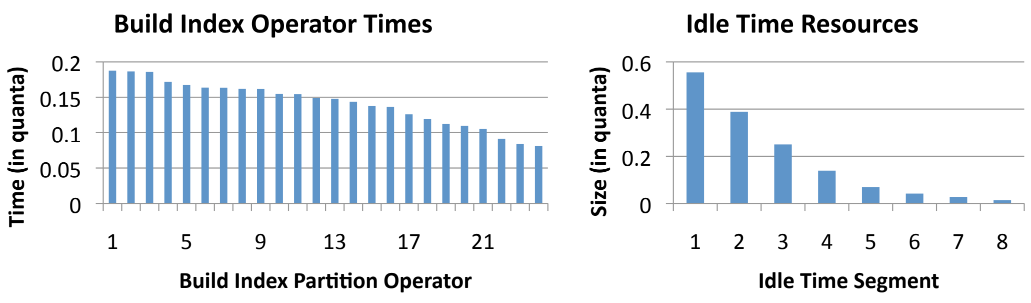


Figure 10: Histogram with execution times of build index operators and idle time resources.

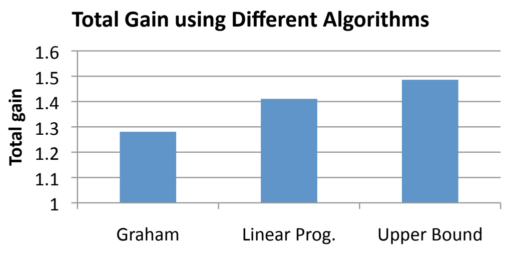


Figure 11: Total gain using different algorithms using the build index operators and idle compute times of Figure 10.

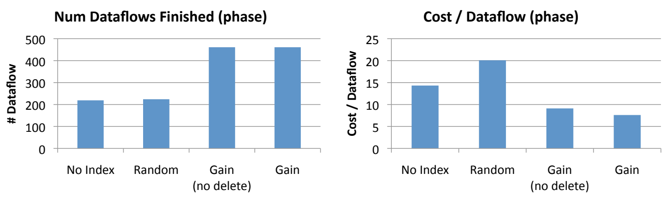


Figure 12: Executed dataflows and average cost/dataflow (phase dataflow generator).

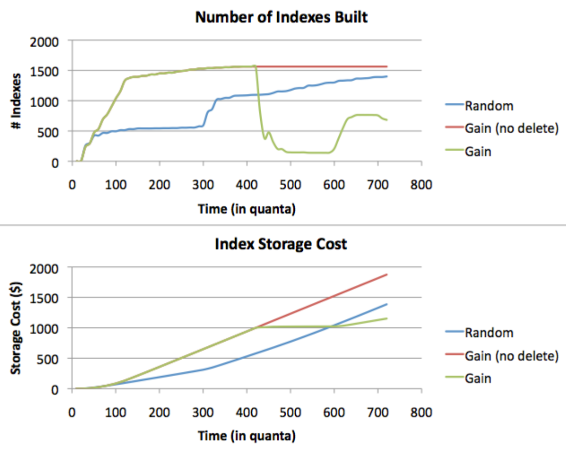


Figure 13: Adaptation of the algorithm to the dataflow workload.

Also, the cost per dataflow is reduced, but not as much as in the previous experiment where the phase dataflow generator client was used. This is because the input is totally at random and, as a result, indexes are stored for a longer period (essentially, they never become non-beneficial). Even in this case, the proposed approach outperforms the baseline approaches.

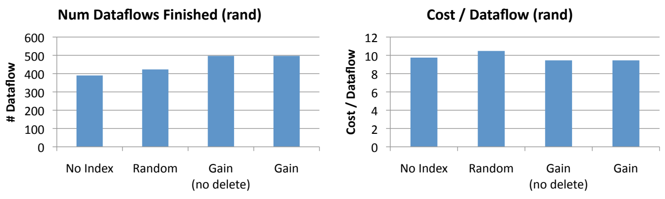


Figure 14: Executed dataflows and average cost per dataflow (random dataflow generator).

7 CONCLUSIONS

In this paper the problem of index management to improve the performance of data-intensive flows on the Cloud is considered. An online auto-tuning approach to assess the usefulness of indexes for the execution of dataflows and utilize idle slots in the execution schedule to build a proper set of indexes is described. The results show that the proposed approach can significantly reduce the average execution time and monetary cost required per dataflow. Future work could evaluate the benefits of index management for scenarios with heterogeneous cloud resources. Also, in this work, we consider a conservative approach to build indexes using idle slots so that they do not interfere with the user workload. Building indexes in a delayed manner for scenarios were idle slots are short is an interesting direction of our future work. Finally, automatic learning of the index gain fading controller to select proper respective values for each index and improve the performance of the proposed approach is another research direction.

ACKNOWLEDGMENT

This work is partially supported by the European Commission under grant agreement 318338, project Optique.

REFERENCES

[1] [n. d.]. TPC-H. http://www.tpc.org/tpch/
[2] 2009. Multi-Version Concurrency Control Algorithms. In *Encyclopedia of Database Systems*. 1870.
[3] Amazon. [n. d.]. Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/
[4] Amazon. [n. d.]. Elastic Map Reduce. http://aws.amazon.com/elasticmapreduce/
[5] Amazon. [n. d.]. Simple Storage Service (S3). http://aws.amazon.com/s3/
[6] Amazon. [n. d.]. Web Services. http://aws.amazon.com/
[7] Apache. [n. d.]. Hadoop. http://hadoop.apache.org/
[8] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, Mei-Hui Su, and K. Vahi. 2008. Characterization of scientific workflows, In WORKS. *"Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on"*, 1 –10. https://doi.org/10.1109/WORKS.2008.4723958
[9] N. Bruno and S. Chaudhuri. 2007. An Online Approach to Physical Design Tuning. In *ICDE*. 826–835. https://doi.org/10.1109/ICDE.2007.367928
[10] Nicolas Bruno and Surajit Chaudhuri. 2010. Constrained physical design tuning. *VLDB J.* 19, 1 (2010), 21–44.

[11] Gang Chen, Hoang Tam Vo, Sai Wu, Beng Chin Ooi, and M. Tamer Özsu. 2011. A Framework for Supporting DBMS-like Indexes in the Cloud. *PVLDB* 4, 11 (2011), 702–713.

[12] Y. Chronis et al. 2016. A relational approach to complex dataflowss. In *EDBT/ICDT Workshops*.

[13] Mariano P. Consens, Kleoni Ioannidou, Jeff LeFevre, and Neoklis Polyzotis. 2012. Divergent physical design tuning for replicated databases. In *SIGMOD Conference*. 49–60.

[14] Carlo Curino, Evan P. C. Jones, Samuel Madden, and Hari Balakrishnan. 2011. Workload-aware database monitoring and consolidation. In *SIGMOD Conference*. 313–324.

[15] 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 the 2019 International Conference on Management of Data (SIGMOD '19)*. ACM, New York, NY, USA, 666–679. https://doi.org/10.1145/3299869.3314035

[16] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *PVLDB* 4, 6 (2011), 362–372.

[17] Ewa Deelman et al. 2006. Managing Large-Scale Workflow Execution from Resource Provisioning to Provenance Tracking: The CyberShake Example. In *e-Science*. 14. https://doi.org/10.1109/E-SCIENCE.2006.99

[18] Ewa Deelman and Ann L. Chervenak. 2008. Data Management Challenges of Data-Intensive Scientific Workflows. In *CCGRID*.

[19] Ewa Deelman, Carl Kesselman, et al. 2002. GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists. In *HPDC*. 225. https://doi.org/10.1109/HPDC.2002.1029922

[20] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. 2012. Only Aggressive Elephants are Fast Elephants. *PVLDB* 5, 11 (2012), 1591–1602.

[21] Iman Elghandour and Ashraf Aboulnaga. 2012. ReStore: Reusing Results of MapReduce Jobs. *Proc. VLDB Endow.* 5, 6 (Feb. 2012), 586–597.

[22] Wenfei Fan, Floris Geerts, and Frank Neven. 2013. Making Queries Tractable on Big Data with Preprocessing. *PVLDB* 6, 9 (2013).

[23] Daniela Florescu and Donald Kossmann. 2009. Rethinking cost and performance of database systems. *SIGMOD Record* 38, 1 (2009), 43–48.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *SOSP*. 29–43.

[25] Ronald L. Graham. 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal of Applied Mathematics* 17, 2 (1969), 416–429.

[26] Ashish Gupta, Fan Yang, Jason Govig, et al. 2014. Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing. *PVLDB* 7, 12 (2014), 1259–1270.

[27] Ramanujam Halasipuram, Prasad M. Deshpande, and Sriram Padmanabhan. 2014. Determining Essential Statistics for Cost Based Optimization of an ETL Workflow. In *EDBT*. 307–318.

[28] Joseph C. Jacob et al. 2009. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *IJCSE* 4, 2 (2009), 73–87. https://doi.org/10.1504/IJCSE.2009.026999

[29] Ivo Jimenez, Huascar Sanchez, Quoc Trung Tran, and Neoklis Polyzotis. 2012. Kaizen: a semi-automatic index advisor. In *SIGMOD Conference*. 685–688.

[30] Verena Kantere, Debabrata Dash, Georgios Gratsias, and Anastasia Ailamaki. 2011. Predicting cost amortization for query services. In *SIGMOD Conference*. 325–336.

[31] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. *Knapsack problems*. Springer. I–XX, 1–546 pages.

[32] Herald Kllapi, Boulos Harb, and Cong Yu. 2014. Near neighbor join. In *ICDE*. 1120–1131.

[33] Herald Kllapi, Eva Sitaridi, Manolis M. Tsangaris, and Yannis E. Ioannidis. 2011. Schedule optimization for data processing flows on the cloud. In *Proc. of SIGMOD*. 289–300.

[34] Donald Kossmann. 2000. The State of the art in distributed query processing. *Comput. Surveys* 32, 4 (2000), 422–469.

[35] Avinash Lakshman and Prashant Malik. 2009. Cassandra: structured storage system on a P2P network. In *PODC*. 5.

[36] Jeff LeFevre et al. 2014. MISO: souping up big data query processing with a multistore system. In *SIGMOD Conference*. 1591–1602.

[37] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. 2012. Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques. *PVLDB* 5, 11 (2012).

[38] Tanu Malik, Xiaodan Wang, Debabrata Dash, Amitabh Chaudhary, Anastasia Ailamaki, and Randal C. Burns. 2009. Adaptive Physical Design for Curated Archives. In *SSDBM*. 148–166.

[39] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*.

[40] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. 2005. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming* 13, 4 (2005), 277–298.

[41] Stefan Richter, Jorge-Arnulfo Quiané-Ruiz, Stefan Schuh, and Jens Dittrich. 2014. Towards zero-overhead static and adaptive indexing in Hadoop. *VLDB J.* 23, 3 (2014), 469–494.

[42] R. Schlosser, J. Kossmann, and M. Boissier. 2019. Efficient Scalable Multi-attribute Index Selection Using Recursive Strategies. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1238–1249. https://doi.org/10.1109/ICDE.2019.00113

[43] Karl Schnaitter and Neoklis Polyzotis. 2012. Semi-Automatic Index Tuning: Keeping DBAs in the Loop. *PVLDB* 5, 5 (2012).

[44] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. 2009. Index Interactions in Physical Design Tuning: Modeling, Analysis, and Applications. *PVLDB* 2, 1 (2009), 1234–1245.

[45] Alkis Simitsis. 2003. Modeling and managing ETL processes. In *VLDB PhD Workshop*.

[46] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*. 996–1005.

[47] Quoc Trung Tran, Ivo Jimenez, Rui Wang, Neoklis Polyzotis, and Anastasia Ailamaki. 2013. RITA: An Index-Tuning Advisor for Replicated Databases. *CoRR* abs/1304.1411 (2013).

[48] Manolis M. Tsangaris et al. 2009. Dataflow Processing and Optimization on Grid and Cloud Infrastructures. *IEEE Data Eng. Bull.* 32, 1 (2009), 67–74.

[49] Prasang Upadhyaya, Magdalena Balazinska, and Dan Suciu. 2012. How to Price Shared Optimizations in the Cloud. *PVLDB* 5, 6 (2012).

[50] Rui Wang, Quoc Trung Tran, Ivo Jimenez, and Neoklis Polyzotis. 2013. INUM+: A leaner, more accurate and more efficient fast what-if optimizer. In *ICDE Workshops*. 50–55.

[51] Petrie Wong, Zhian He, and Eric Lo. 2013. Parallel analytics as a service. In *SIGMOD Conference*. 25–36.

[52] Eugene Wu and Samuel Madden. 2011. Partitioning techniques for fine-grained indexing. In *ICDE*. 1127–1138.

[53] Q. Zhu and T. Tung. 2012. A Performance Interference Model for Managing Consolidated Workloads in QoS-Aware Clouds. In *Proceedings of the 5th IEEE CLOUD*. IEEE, 170–179.

Optimal Histograms with Outliers

Rachel Behar
The School of Computer Science & Engineering
Jerusalem, Israel
rachel.beharharr@mail.huji.ac.il

Sara Cohen
The School of Computer Science & Engineering
Jerusalem, Israel
sara@cs.huji.ac.il

ABSTRACT

Histograms are a well studied and simple way to summarize data. As such, they are used extensively in a variety of applications that require estimates of data frequency values. Significant previous work has studied the problem of finding optimal histograms with respect to an error measure. In this paper we study the classic problem of finding an optimal histogram for a dataset, with a new twist: The histogram must contain at least $n - k$ of the n data points. The k excluded data points are considered outliers.

We consider two notions of excluding data items, by allowing arbitrary items to be excluded, or only removing items while retaining a *consistent* histogram. Polynomial algorithms are presented for these problems. Significant experimentation demonstrates that our algorithms work well in practice to reduce the histogram error.

1 INTRODUCTION

Covering a set of data points with a histogram is a fundamental problem that lays at the heart of many database applications. Problems of this sort appear as optimization problems with various constraints on the type and size of the histogram, as well as an error function that must be minimized. Among these problems, finding an optimal histogram covering all but a few input points is of interest in view of outlier removal. In such a problem, given n data points, we are asked to find the optimal histogram covering at least $n - k$ of the n input points. From the viewpoint of optimization, excluding k points reduces the error and in this sense, the excluded data points can be considered outliers. In computational geometry, variations of this problem have been studied in many different settings.

A histogram provides the user with an estimate of frequencies of data elements within each bucket. For example, consider a histogram over integer exam grades. The bucket $([51, 60]; 40)$ indicates that there are 40 exam grades between 51 and 60, inclusive, in the dataset. Without additional information, a uniform distribution of elements within the bucket will be assumed, i.e., 4 grades each for every number between 51 and 60. In practice, this may be far from the correct frequencies, e.g., it is possible that there are 19 occurrences each of grade 59 and 60 and two occurrences of grade 51.

Using an error metric, such as sum-squared error, it is possible to measure the distance between the histogram's estimation and the actual frequencies. When the error is large, the histogram poorly captures the data. The simplest way to decrease the error is to increase the number of buckets. By using more buckets, the user has a more fine-grained view of the data. This, however, is not always an appropriate solution, and in particular, is inappropriate if the histogram is given to a human to view (and not simply to a computer program to continue manipulating for some purpose). If the user is given a very large number of buckets she loses the ability, to some degree, of seeing "the big picture" of the data. It is more difficult to grasp the meaning of the data when many buckets are displayed. A large number of buckets also degrades the quality of the user interface, particularly on mobile devices.

Histograms with a limited number of buckets can have high error as they must cover all elements within the dataset. However, there can be elements that are outliers, in the sense that it is their inclusion that causes the error to increase. In the above example of a dataset of grades, the occurrences of grade 51 can be seen as outliers. Indeed, if these elements did not have to be covered by a bucket, one could use $([59, 60]; 38)$, for which the uniform distribution perfectly estimates the actual grades in the bucket range.

In this paper our goal is to find optimal histograms, given a bound β on the number of buckets, as well as a bound k on the number of outliers that need not be covered. In other words, the goal is to optimally choose up to k elements and a histogram with up to β buckets, so as to minimize the total error.

The main contributions of the paper are:

- We introduce two intuitive notions for an optimal histogram (called a *summary*) with deletions.
- We present several algorithms for solving this problem, including algorithms that are provably optimal.
- Extensive experimentation demonstrates the quality of our solutions.

This paper is organized as follows. We start by discussing related work in Section 2. In Section 3 we set up the framework by providing the necessary definitions. In Sections 4 and 5 we study the optimal summary with deletion problem when there is only a single bucket, or when multiple buckets are allowed. In Section 6 we consider datasets with multiple columns. Extensive experimentation appears in Section 7, and Section 8 concludes.

2 RELATED WORK

Related work falls into two sub-areas: finding optimal histograms and identifying outliers in data. We discuss each of these aspects below.

Histograms. Histograms summarize data by storing the number of values within a given data range. They are traditionally used for estimating costs of query plans, as well as for approximate query answering. Various types of histograms have been considered in the past. These differ both on their maintenance efficiency over dynamic data, and their effectiveness for approximating query answers. Some of the classical types of histograms considered include equi-width (in which all buckets have the same range size), equi-depth (in which all buckets have the same number of data elements), serial (which group elements according to frequency) and end-biased (in which all buckets except one have a single element range). This paper considers the popular v-optimal histograms [9] which has been proven to be optimal in minimizing the variance of element frequencies within the buckets. A taxonomy of histogram types was presented in [12]

Due to their popularity, there has been much effort to efficiently compute histograms from data. In [11], a polynomial time algorithm was presented to compute v-optimal histograms. This work was later expanded in [10] for the case in which there is a bound on the total number of buckets that can be used to summarize several different columns of data. There has also been work on approximating histograms, e.g., [7, 8], and using sampling for constructing histograms [5].

No previous work has considered the setting of computing histograms when not all of the elements must be covered. Among previous work, our paper is most related to [11] as we also present an optimal algorithm for histogram creation, albeit in a new setting.

Outliers. Intuitively, an outlier is a data element that is distant from other elements. Many different criteria have been considered in the past to determine when a data point is an outlier, and outlier detection algorithms have been presented. In particular, algorithms differ as to whether they compare data elements with the entire set, or only with a small local subset. In our work, we assume that the user provides us with a number k of outliers that can be removed. This differs from most previous work, in which no such number is provided. In [4, 14] algorithms for top-k outliers are presented. However, they may return less than k elements if, fewer are deemed to be outliers. (In addition, they differ in their notion of an outlier, as they do not attempt to minimize histogram error.)

Histograms have been used to efficiently find outliers [6, 13]. Our algorithms can also be viewed as finding outliers using histograms. However, our approach is significantly different from previous work. In [6, 13], histograms are built over the entire data set, and then data elements that significantly differ from their fellow bucket elements are considered outliers. In our work, we find the optimal set of data elements that should be removed in order to generate a histogram with low error.

Several works have considered similar types of problems, i.e., covering all but k data elements in an optimal manner (in some sense). In particular, [2] considers optimality in terms of minimum diameter, minimum area bounding box or minimal area convex hull. In [3], the goal is to optimally remove k data elements so that the remaining have a minimum-width square or rectangular annulus. Covering all but k data elements with disjoint boxes is considered in [1]. While these works have a similar goal, they differ in the type of cover that must be created and in error function that should be minimized. Hence, previous techniques cannot carry over to our framework.

3 DEFINITIONS

Datasets and Frequencies. The goal of this paper is to find a method to effectively summarize a large dataset, such as the contents of a relation in a database, or the result of a query. In the following, a *dataset* P is a multiset of elements. To simplify the presentation, we assume that the domain of P is $\mathbb{Z}$. However, all our results immediately generalize to arbitrary completely ordered discrete domains.[1]

Given a multiset of elements P, for all $q \in \mathbb{Z}$, we use $card(q, P)$ to denote the number of copies of q that appear in P. We use $\vec{f_P}$ to denote the *frequency vector* of P, i.e., $\vec{f_P}[q] = card(q, P)$. Note that $\vec{f_P}$ has finitely many non-zero elements. We differentiate between the *elements* of $\vec{f_P}$ denoted p, q, r, and the *frequency values* (or simply *values* for short) of $\vec{f_P}$, denoted v, w.

Example 3.1. Consider the multiset, used as running example,

$$P = \{\!\{10, 20, 30, 20, 30, 40, 10, 40, 50, 0, 0, 0, 0\}\!\}.$$

The frequency vector of P, when considering only non-zero elements, is

$$\vec{f_P}[q] = \{0{:}4, 10{:}2, 20{:}2, 30{:}2, 40{:}2, 50{:}1\} \quad \square$$

Summaries. A *bucket* is written $b = ([p, q]; n)$, where

- $p \le q$ are integers defining the bucket *endpoints* and
- n is a *positive* natural number indicating the number of elements in the bucket.

We say that $\{p, \dots, q\}$ is the range of b.

Now, consider a multiset P and a bucket $b = ([p, q]; n)$. We say that b is *consistent* with P if $\sum_{r=p}^{q} \vec{f_P}[r] = n$, i.e., n is precisely the sum of frequencies of all elements within the range of b in P.

We say that a set of buckets B is a *summary* of P if

- every two buckets in B have disjoint ranges,
- all buckets in B are consistent with P,
- for every $r \in P$, there is some $b \in B$ such that r is in the range of b.

Thus, intuitively, a summary of P is a set of disjoint consistent buckets that cover all elements appearing in P.

Example 3.2. The following are different summaries of dataset P from Example 3.1, with one and two buckets, respectively:

$$B_1 = \{([0, 50]; 13)\} \qquad B_2 = \{([0, 0]; 4), ([10, 50]; 9)\} \quad \square$$

Remark 1. *A summary of a multiset of elements is very similar to the well-studied notion of a histogram of a multiset of elements. Much of the previous work on histograms assumed that the buckets fully covered the domain. This was important, as data was dynamically added to the dataset and had to be incorporated into the histogram (and therefore, had to have a bucket to which it could be added). In this work, since our goal is to summarize the current contents of a dataset, we only create non-empty buckets, with endpoints corresponding to actual elements in the dataset. Empty buckets are not of interest.*

We associate a summary B with a vector $\vec{e}_B$ used to *estimate* frequencies of elements, as follows. Let r be an integer. If r is within the range of a bucket $([p, q]; n) \in B$, then $\vec{e}_B[r] = \frac{n}{q-p+1}$. Otherwise, $\vec{e}_B[r] = 0$.

Now, the *error* of B w.r.t. P, denoted $err(B, P)$ is the distance of $\vec{e}_B$ from $\vec{f_P}$. In general, different distance metrics can be employed. In this paper, we use the sum-squared-error (SSE) metric as, used by [11]:

$$err(B, P) = \sum_{q \in \mathbb{Z}} (\vec{f_P}[q] - \vec{e}_B[q])^2 .$$

However, the work generalizes to additional metrics. It has been shown [11] that this error metric can be equivalently computed as:

$$err(B, P) = \sum_{([p,q];n) \in B} \left(\sum_{p \le r \le q} (\vec{f_P}[r])^2 - \frac{n^2}{q - p + 1} \right) . \quad (1)$$

Example 3.3. Recall B_1 and B_2 from Example 3.2. Then, $err(B_1, P) = 29.7$ and $err(B_2, P) = 15$. $\square$

[1] This work considers only discrete domains. For attributes over continuous domains, our work can be applied if a discretization function is used, e.g., by rounding to a given decimal place.

Optimal Summaries. For any dataset P, it is possible to find a summary B with $err(B, P) = 0$ by simply creating a singleton bucket for each element appearing in P. However, such a summary will be of size similar to that of P itself, and will be unwieldy for large datasets. Hence, we will be interested in creating summaries of bounded size.

Let β be a positive integer, called the *bucket size bound*. A summary B is size-bounded by β if the number of buckets in B is at most β. Given a dataset P and a bucket size bound β, the *optimal summary problem* is to find a summary B of P that is size-bounded by β, and has minimal error. This problem has been considered in the past, and has been shown to be solvable in polynomial time using dynamic programming [11].

Optimal Summaries with Deletion. In practice, some of the elements in the dataset may be outliers that greatly contribute to the error of the summary, while adding little information of interest. When we are interested in summarizing the data to understand the general trends, such outliers may not be of interest. Indeed, a better understanding of the data may be derived when some elements are removed, to derive a smoother dataset, before creating a summary.

We say that P' is a *k-deletion* of P if P' is derived by removing at most k elements from P. We consider two different types of summaries after deletion. We say that B is a *k-deletion summary* of P if there is a k-deletion P' of P such that B is a summary of P'. A stronger requirement on the summary can be formulated as follows: We say that B is a *consistent k-deletion summary* of P if there is a k-deletion P' of P such that B is a summary of P' *and* every bucket in B is consistent with P.

Example 3.4. Consider the dataset

$$P = \{\!\{1, 1, 2, 2, 2, 3, 3\}\!\}.$$

Suppose that we can only use a single bucket, but we may delete up to $k = 2$ elements.

The optimal choice is to delete a single occurrence of element 2, yielding the dataset $P' = \{\!\{1, 1, 2, 2, 3, 3\}\!\}$, which can be summarized using one bucket with an error of 0. This summary $B' = ([1, 3]; 6)$, however, is not consistent with P.

An optimal consistent 2-deletion summary can be derived by removing all occurrences of the element 1 (or all occurrences of the element 3) yielding the consistent summary $B'' = ([2, 3]; 5)$ with error 0.5.

This example also serves to demonstrate why consistent summaries have an added benefit. The summary B'' accurately reflects the dataset, namely that it contains five elements in the range $[2, 3]$. The summary B' can be seen as being somewhat confusing, as it seems to imply that the dataset has six elements in the range $[1, 3]$, when in fact there are seven such elements (and one occurrence of element 2 has been removed to reduce the error). □

Problems of Interest. In this paper we study the following problem, called the (consistent) optimization problem: *Let P be a dataset, β be a bucket size bound and k be a natural number. Find P', B such that P' is a k-deletion of P and B is a summary of P' (and B is consistent with P) such that $err(B, P')$ is minimized.*

4 SINGLE BUCKET SUMMARIES

Before studying the general optimization problems, we start by assuming that only a single bucket may be used. We note that even in this case there can be exponentially many different choices of k elements to delete. Hence, finding an optimal k-deletion summary with one bucket is not a trivial problem. In addition, the ability to optimize for a single bucket is an important component in a solution for multiple buckets. Therefore, in this section we present algorithms that find an optimal k-deletion summary in polynomial time, when only a single bucket can be used. We start by considering arbitrary single bucket summaries with deletions, and then consider the special case of consistent single bucket summaries with deletions.

4.1 Arbitrary Single Bucket Summaries

We use $minArbErr(\vec{f}, \beta, k)$ to denote the minimal error that can be achieved for a multiset with frequency vector $\vec{f}$, when up to k elements can be deleted from the multiset, and up to β buckets can be used. The algorithms that we will present compute this error value directly. As is standard with dynamic programming, to find the k elements to be deleted from the multiset and the bucket boundaries that should be used, some additional book-keeping is necessary. This is quite straight-forward, but clutters the presentation. Hence, we focus on computing $minArbErr(\vec{f}, \beta, k)$.

In this section, we study the problem of computing the value $minArbErr(\vec{f}, 1, k)$ for a given $\vec{f}$ and k, i.e., our optimization problem for the special case where only one bucket can be used. Intuitively, there are two different ways to choose elements to remove, in order to reduce the error:

(1) we can remove the first or last elements with non-zero frequency in $\vec{f}$, thereby allowing a bucket with smaller range to cover $\vec{f}$;
(2) we can remove elements that have the greatest frequency, thereby reducing the variance of the frequencies within the bucket.

Before presenting our algorithm, we introduce some necessary notion used in the algorithms throughout this paper. Let $\vec{f}$ be a frequency vector. We use $\mathsf{size}(\vec{f})$ to denote the number of different non-zero elements in $\vec{f}$, and $\mathsf{first}(\vec{f})$ and $\mathsf{last}(\vec{f})$ to denote the first and last elements with non-zero frequency in $\vec{f}$, respectively. Finally, $\mathsf{next}(\vec{f}, p)$ and $\mathsf{prev}(\vec{f}, p)$ denote the element immediately after and immediately preceding p in $\vec{f}$ with non-zero frequency, respectively. When $\vec{f}$ is clear from the context, it will be omitted, and we will simply write size, first, last, $\mathsf{prev}(p)$, $\mathsf{next}(p)$.

For elements p, q, let $\vec{f}_{[p,q]}$ be the frequency vector

$$\vec{f}_{[p,q]}[r] = \begin{cases} \vec{f}[r] & p \le r \le q \\ 0 & \text{otherwise} \end{cases}$$

The algorithm ArbSingleBErr in Figure 1 computes the value $minArbErr(\vec{f}, 1, k)$. ArbSingleBErr starts by first checking if all elements in the vector can be removed (Lines 1–2), in which case an error of zero is returned. Next, the algorithm considers all options of removing first or last elements, as well as other elements with high frequency. Intuitively, the values k_{left} and k_{right} in the algorithm are used to specify how many elements can be removed from the beginning and end, respectively, of $\vec{f}$. This leaves $k_{mid} = k - k_{left} - k_{right}$ elements of high frequency to be removed from the remainder of $\vec{f}$.

The algorithm ArbSingleBErr therefore iterates over values of $k_{left} \le k$ (Line 6), and values of $k_{right} \le k - k_{left}$ (Line 9). We note that we always choose values for k_{left} and k_{right} that are sufficient to reduce the frequency of elements at the beginning

```
Algorithm ArbSingleBErr($\vec{f}, k$)
1.  if $\sum_{p=\text{first}}^{\text{last}} \vec{f}[p] \le k$
2.    then return 0
3.  $e_* \leftarrow \infty$
4.  $p \leftarrow$ first
5.  $k_{left} \leftarrow 0$
6.  while $k_{left} \le k$
7.    do $q \leftarrow$ last
8.      $k_{right} \leftarrow 0$
9.      while $k_{right} \le k - k_{left}$
10.       do $k_{mid} \leftarrow k - k_{left} - k_{right}$
11.         $e \leftarrow$ LowerMaxError($\vec{f}_{[p,q]}, k_{mid}$)
12.         $e_* \leftarrow \min\{e_*, e\}$
13.         $k_{right} \leftarrow k_{right} + \vec{f}[q]$
14.         $q \leftarrow$ prev($q$)
15.     $k_{left} \leftarrow k_{left} + \vec{f}[p]$
16.     $p \leftarrow$ next($p$)
17. return $e_*$
```

```
Algorithm LowerMaxError($\vec{f}, k$)
1.  $p_{min}{:}v_{min} \leftarrow$ minFreqVal($\vec{f}$)
2.  $p_{max}{:}v_{max} \leftarrow$ maxFreqVal($\vec{f}$)
3.  while ($k > 0$ and $v_{max} > v_{min}$)
4.    do $\vec{f}[p_{max}] \leftarrow \vec{f}[p_{max}] - 1$
5.      $k \leftarrow k - 1$
6.      $p_{max}{:}v_{max} \leftarrow$ maxFreqVal($\vec{f}$)
7.  $n_{\text{sum}} \leftarrow \sum_{p=\text{first}}^{\text{last}} \vec{f}[p]$
8.  $n_{\text{sqsum}} \leftarrow \sum_{p=\text{first}}^{\text{last}} (\vec{f}[p])^2$
9.  return $n_{\text{sqsum}} - {n_{\text{sum}}}^2/(\text{last} - \text{first} + 1)$
```

Figure 1: Returns *minArbErr*($\vec{f}, 1, k$)**.**

and end of $\vec{f}$ to zero. This is achieved by increasing k_{left} and k_{right} by the actual frequency values appearing in $\vec{f}$ (Lines 15 and 13).

For given values of k_{left} and k_{right} we have p and q that correspond to the first and last elements of $\vec{f}$ whose frequency has not been reduced. They determine the range of values in $\vec{f}$ for which we will attempt to reduce the highest frequency values (to reduce variance) using the algorithm LowerMaxError.

LowerMaxError uses minFreqVal (resp. maxFreqVal) to return the element and corresponding frequency value which is minimal (resp. maximal) in $\vec{f}$. While more elements can be removed ($k > 0$) and there is still non-zero variance within the range ($v_{max} > v_{min}$) the frequency of the most frequent element is reduced by one (Line 4). Finally, Line 9 returns the error for the single bucket that would be created for the updated $\vec{f}$, using Equation 1. ArbSingleBErr keeps track of the lowest error option (Line 12), and returns this value (Line 17).

Correctness and runtime of ArbSingleBErr are stated in the following theorem.

Theorem 4.1. *For a given $\vec{f}$ and k,* ArbSingleBErr($\vec{f}, k$) *computes minArbErr($\vec{f}, 1, k$) in time $O(k^2(\text{size}\log\text{size} + k))$.*

Proof. ArbSingleBErr iterates over all options of removing all occurrences of elements (i.e., reducing to a zero frequency) from the beginning and end of $\vec{f}$. For each such option, LowerMaxError is used to reduce the frequencies of highest frequency elements. Hence, it is sufficient to show that LowerMaxError returns the minimal error that can be derived by k-deletions that do not remove all occurrences of the first and last elements of $\vec{f}$.

In the following, assume that p and q are the first and last elements, respectively, with non-zero frequency in $\vec{f}$. Thus, the single bucket over a k-deletion of $\vec{f}$ will have the form $([p, q]; n)$ where n is the sum of frequencies of the remaining elements of $\vec{f}$.

Recall that LowerMaxError does the following simple action: while there are still elements that can be deleted and not all frequencies in $\vec{f}$ (between p and q) are equal, find the most frequent element, and reduce its frequency by one. We will prove that this strategy achieves minimal error.

Assume, by way of contradiction, that there exist a different strategy to delete up to k elements from $\vec{f}$, that achieves the minimal error e_0. Let $\vec{f}'$ be this k-deletion. Let

$$v_{est} = (\sum_{p \le r \le q} \vec{f}'[r])/(q - p + 1).$$

Note that v_{est} is the estimated frequency value for each element in the bucket. The error of $\vec{f}'$ is

$$e_0 = \sum_{p \le r \le q} (\vec{f}'[r] - v_{est})^2 .$$

We will show that LowerMaxError also returns e_0.

Let $v_- = \min_{p \le r \le q}\{\vec{f}'[r] \mid \vec{f}'[r] \ne \vec{f}[r]\}$, i.e., v_- is the minimal frequency in $\vec{f}'$ changed by the deletion. Let $v_+ = \max_{p \le r \le q}\{\vec{f}'[r]\}$, i.e., v_+ is the maximal frequency value in $\vec{f}'$.

Now we consider the frequency vector derived by reducing the frequency v_+ by one and increasing the frequency v_- by one. The error of this new frequency vector e_0' differs from e_0 only in the summation for frequencies v_+ and v_-. Now we will show that $e_0' - e_0 \le 0$ i.e, this change did not increase the error. Observe that

$$\begin{aligned} e_0' - e_0 = {} & (v_+ - 1 - v_{est})^2 + (v_- + 1 - v_{est})^2 - \qquad (2)\\ & \Big((v_+ - v_{est})^2 + (v_- - v_{est})^2\Big) \\ = {} & 2(v_- - v_{est}) - 2(v_+ - v_{est}) + 2 \qquad (3)\\ \le {} & 2(v_- - v_{est}) - 2(v_- + 1 - v_{est}) + 2 = 0 \qquad (4) \end{aligned}$$

where Equation 4 holds since $v_- + 1 \le v_+$, since otherwise we could have produced $\vec{f}'$ using LowerMaxError.

Now, by repeating this change up to k times, the result of LowerMaxError will be equal to the minimum error.

To prove the runtime, observe that ArbSingleBErr makes at most k^2 calls to LowerMaxError. Each call to LowerMaxError costs size log size + k. To achieve this time, we first sort the vector $\vec{f}_{[p,q]}$ sent to the algorithm. This allows us to efficiently evaluate all calls to minFreqVal and maxFreqVal. Then, we require another k operations to lower the frequencies, giving a total of $O(\text{size}\log\text{size} + k)$ for algorithm LowerMaxError, and $O(k^2(\text{size}\log\text{size} + k))$ for ArbSingleBErr, as required. □

4.2 Consistent Single Bucket Summaries

We use *minConErr*($\vec{f}, \beta, k$) to denote the minimal error that can be achieved for a multiset with frequency vector $\vec{f}$, when a k-deletion is taken and up to β buckets can be used, and the summary created *must be consistent* with $\vec{f}$. In this section we show how to compute this value for the special case where $\beta = 1$.

ConSingleBErr (Figure 2) uses a precomputed matrix SSE of dimensions size($\vec{f}$)×size($\vec{f}$), where $SSE[p, q]$ is the sum-squared error for a single bucket over the range $[p, q]$ of $\vec{f}$, with no

```
Algorithm ConSingleBErr($\vec{f}, k$)
1.  if $\sum_{p=\text{first}}^{\text{last}} \vec{f}[p] \le k$
2.     then return 0
3.  $q \leftarrow$ last
4.  $k_{right} \leftarrow 0$
5.  while $k_{right} + \vec{f}[q] \le k$
6.      do $k_{right} \leftarrow k_{right} + \vec{f}[q]$
7.         $q \leftarrow \text{prev}(q)$
8.  $e \leftarrow SSE[\text{first}, q]$
9.  $p \leftarrow$ first
10. $k_{left} \leftarrow 0$
11. while $k_{left} + \vec{f}[p] \le k$
12.     do $k_{left} \leftarrow k_{left} + \vec{f}[p]$
13.        $p \leftarrow \text{next}(p)$
14.        while $k_{left} + k_{right} > k$
15.            do $q \leftarrow \text{next}(q)$
16.               $k_{right} \leftarrow k_{right} - \vec{f}[q]$
17.        $e \leftarrow \min\{e, SSE[p, q]\}$
18. return $e$
```

Figure 2: Returns *$minConErr(\vec{f}, 1, k)$*.

deletions,[2] i.e., $SSE[p, q] = \sum_{p \le r \le q} (\vec{f}[r] - v_{est})^2$ where $v_{est} = (\sum_{p \le r \le q} \vec{f}[r])/(q - p + 1)$.

The algorithm ConSingleBErr first checks if all elements in $\vec{f}$ can be deleted. In this case, the error is zero (Lines 1–2). Otherwise, it begins by considering the case in which elements on the right-hand side of $\vec{f}$ are maximally deleted and computes the error in this case (Lines 5–8). Then, in Lines 11–17 we slowly increase the number of elements removed on the left-hand side, while reducing the number of elements removed on the right-hand side, so as to remain within the limit of k. For each such option, we compute the error, and keep the minimal value.

We show the following result.

Theorem 4.2. *For a given $\vec{f}$ and k,* ConSingleBError($\vec{f}, k$) *computes $minConErr(\vec{f}, 1, k)$ in time $O(\min\{k, \text{size}\})$.*

Proof. To prove correctness, it is important to note that only maximal deletions from the two sides of $\vec{f}$ need to be considered. Thus, for example, in Lines 5–7 we maximally remove elements from the right-hand side of $\vec{f}$. This follows from the following claim: Let $\vec{f}$ be a frequency vector, and $p < q$ be elements with non-zero frequencies in $\vec{f}$. Let n be the total frequency of all elements in $\vec{f}$ and let n' be the total frequency of all elements between p and q (including) in $\vec{f}$. Then, $err(\{([\text{first}, \text{last}]; n)\}, \vec{f}) \ge err(\{([p, q]; n')\}, \vec{f}_{[p,q]})$. This follows from Lemma 2 in [11].

Finally, to show the runtime, observe that we iterate over $\vec{f}$ twice: once from right to left (Lines 5–7) and once from left to right (Lines 11-17). Thus, the time is at most $O(\text{size})$. Since in both our iterations over $\vec{f}$ we do at most k operations (we increase k_{right} or k_{left} until k), the time is at most $O(k)$. Together, this proves our claim of runtime. □

[2] In practice, it is not necessary to store a matrix of size size × size. Instead this value can be computed in time $O(1)$ given two arrays of size size each, as discussed in [11].

```
Algorithm NoDeletionSummary($\vec{f}, \beta$)
1. for $p \leftarrow$ first to last
2.     do $M[p, 1] \leftarrow SSE[\text{first}, p]$
3. for $\beta' \leftarrow 2$ to $\beta$
4.     do for $q \leftarrow$ next(first) to last
5.            do $M[q, \beta'] \leftarrow \infty$
6.               for $p \leftarrow$ prev($q$) down-to first
7.                   do $e \leftarrow M[p, \beta' - 1] +$
                        $SSE[\text{next}(p), q]$
8.                      $M[q, \beta'] \leftarrow \min\{M[q, \beta'], e\}$
9. return $M[\text{last}, \beta]$
```

Figure 3: Returns *$minArbErr(\vec{f}, \beta, 0)$* **(which is also equal to** *$minConErr(\vec{f}, \beta, 0)$***).**

5 MULTI-BUCKET SUMMARIES

We now consider the problem of finding summaries that can have multiple buckets. The problem of finding an optimal summary, when no deletions are permitted ($k = 0$) is well studied. Hence, as a baseline approach it is natural to try to directly leverage algorithms that find an optimal summary, in order to solve our problem.

5.1 Baseline Approaches

Before discussing how to find summaries with deletions, we review the dynamic algorithm proposed in [11] for building optimal sum-squared-error histograms without deletions.

Optimal Summary without Deletions. Figure 3 presents the algorithm NoDeletionSummary from [11] which computes the optimal error for a multiset with frequency vector $\vec{f}$ when using β buckets but no deletions. Note that in this special case, clearly $minArbErr(\vec{f}, \beta, 0) = minConErr(\vec{f}, \beta, 0)$ will always hold. Hence, the algorithm can be seen as computing either of these values. Intuitively, the algorithm dynamically iterates over all possible divisions of the input into β buckets and returns the minimal sum-square error found.

NoDeletionSummary utilizes two matrices:

- the precomputed matrix SSE discussed earlier;
- a matrix M of dimensions size($\vec{f}$) × β, in which we will update $M[p, \beta']$ to contain $minArbErr(\vec{f}_{[\text{first}(\vec{f}), p]}, \beta', 0)$.

Hence, $M[\text{last}(\vec{f}), \beta]$ is precisely what we are computing.

The algorithm starts by initializing the matrix M with the error elements for the case of a single bucket (Lines 1–2). Note that this loop, (as well as all other loops in this paper that iterate over elements from $\vec{f}$, such as Line 4 and 6) should be understood as implicitly only looping over elements p for which $\vec{f}[p] > 0$. Thus, for each element p with positive frequency, we update $M[p, 1]$.

To compute the remainder of M, the algorithm uses three nested for loops. In the outer loop, it iterates over the number of β' from 2 to β (Line 3). In the first inner loop, it iterates over elements q from next(first) to last (Line 4). In the inner-most loop, the algorithm iterates over all possible elements p for the left side of the right most bucket (Line 6). Using previously computed values, it then calculates the error, under the assumption that the last bucket ranges from just after p to q, (Line 7) and saves the minimal error in $M[q, \beta']$ (Line 8). Finally, the algorithm returns the minimal error that can be achieved with β buckets, i.e., $M[\text{last}, \beta]$

```
Algorithm SumThenDelErr($\vec{f}, \beta, k$)
1.  $B \leftarrow$ NoDeletionSummaryBuckets($\vec{f}, \beta$)
2.  $([p_1, q_1]; n_1) \leftarrow B[1]$
3.  for $k' \leftarrow 0$ to $k$
4.      do $M[1, k'] \leftarrow$ SingleBErr($\vec{f}_{[p_1,q_1]}, k'$)
5.  for $\beta' \leftarrow 2$ to $\beta$
6.      do for $k' \leftarrow 0$ to $k$
7.          do SDUpdateMatrix($\vec{f}, \beta', k', M, B$)
8.  return $M[\beta, k]$
Algorithm SDUpdateMatrix($\vec{f}, \beta, k, M, B$)
1.  $M[\beta, k] \leftarrow \infty$
2.  $([p, q]; n) \leftarrow B[\beta]$
3.  for $k' \leftarrow 0$ to $k$
4.      $e \leftarrow M[\beta - 1, k - k']$
5.      $e' \leftarrow$ SingleBErr($\vec{f}_{[p,q]}, k'$)
6.      $M[\beta, k] \leftarrow \min\{M[\beta, k], e + e'\}$
```

Figure 4: Returns either *minArbSDErr*($\vec{f}, \beta, k$) **or** *minConSDErr*($\vec{f}, \beta, k$)**, depending on the version of** SingleBucketError **used.**

(Line 9). It is not difficult to see that NoDeletionSummary runs in time $O(\beta\,\text{size}^2)$.

Example 5.1. As a running example, to compare algorithms returning multi-bucket summaries, we consider a dataset with the frequency vector $\vec{f}$, defined as

$$\{1{:}2, 2{:}1, 3{:}2, 4{:}1, 5{:}2, 6{:}3, 7{:}2, 8{:}1\}.$$

The optimal 2-bucket summary of $\vec{f}$, without deletions is

$$B = \{([1,7]; 13), ([8,8]; 1)\}$$

with error 2.86. □

Summarize, then Delete. One approach to finding a summary with deletions is to

- *first* compute an optimal summary B without deletions (using NoDeletionSummary) and
- *then* choose the best elements to delete from the buckets, adjusting bucket boundaries accordingly.

We denote the minimal possible error that can be achieved when following this two-step methodology for a dataset with frequency vector $\vec{f}$, bucket bound β and a number k of deletions as *minArbSDErr*($\vec{f}, \beta, k$) and *minConSDErr*($\vec{f}, \beta, k$) for arbitrary and consistent summaries, respectively.

While a-priori it may not be clear how to choose elements to delete, in fact, an optimal choice can be found in polynomial time using dynamic programming. SumThenDelErr in Figure 4 computes the minimal error by filling in a matrix M of dimensions $\beta \times k$. The position $M[\beta', k']$ is updated to be the optimal error of the first β' buckets while allowing up to k' deletions. Hence, the value of interest will be in $M[\beta, k]$ at the end of the algorithm.

Algorithm SumThenDelErr begins by calling an algorithm named NoDeletionSummaryBuckets, which is a version of the previously presented NoDeletionSummary that returns the set of buckets B, for which the error, with no deletions, is minimal. Next the algorithm initializes all entries of M for $\beta = 1$ by calling an algorithm that computes the error for a single bucket (Lines 2–3). In practice, SingleBErr should be replaced with ArbSingleBErr if the goal is to compute *minArbSDErr*($\vec{f}, \beta, k$) and with ConSingleBErr if the goal is to compute *minConSDErr*($\vec{f}, \beta, k$).

Finally, the algorithm considers all choices for $2 \le \beta' \le \beta$, as well as all values of $k' \le k$, by calling SDUpdateMatrix for each combination. Algorithm SDUpdateMatrix is used to update $M[\beta, k]$, by considering all choices k' of how to split the deletions between the first $\beta - 1$ buckets (i.e., $M[\beta - 1, k - k']$) and the final bucket (i.e., SingleBErr($\vec{f}_{[p,q]}, k'$)).

Example 5.2. Recall the frequency vector $\vec{f}$ from Example 5.1. When running SumThenDelErr($\vec{f}, 2, 2$) once with ArbSingleBErr and once with ConSingleBErr we derive the summaries B_1 and B_2, respectively:

$$B_1 = \{([1,7]; 11), ([8,8]; 1)\} \qquad \textit{err}{:}\ 0.18$$
$$B_2 = \{([2,7]; 11), ([8,8]; 1)\} \qquad \textit{err}{:}\ 0.35$$

where the former is derived by deleting two occurrences of element 6, and the latter is derived by removing two occurrences of element 1. □

If we can show that the algorithm returns a value close to *minArbErr*($\vec{f}, \beta, k$) or *minConErr*($\vec{f}, \beta, k$), we can leverage optimizations presented in the past for finding optimal summaries without deletions to solve the problem at hand. Unfortunately, the following theorem states that SumThenDelErr can be arbitrarily bad.

Theorem 5.3. *For any β, k and $e > 0$, there exists a multiset P with frequency vector $\vec{f}$ such that*

$$\textit{minArbSDErr}(\vec{f}, \beta, k) - \textit{minArbErr}(\vec{f}, \beta, k) > e\,,$$
$$\textit{minConSDErr}(\vec{f}, \beta, k) - \textit{minConErr}(\vec{f}, \beta, k) > e\,.$$

Proof. Due to space limitations, we only show the result for $k = 1$ and $\beta = 2$. The more general case, however, can be shown similarly. Let $e > 0$ be an error gap. Let n be a natural number such that $n > max\{10, 5e\}$. Let $\vec{f}$ be the frequency vector $\{0{:}n, 2{:}1, 4{:}n, 7{:}n\}$.

If we can delete elements before choosing the buckets, the optimal strategy is to reduce the frequency of the element 2 in $\vec{f}$, thus deriving the frequency vector $\vec{f}' = \{0{:}n, 4{:}n, 7{:}n\}$. Then the optimal summary would be $\{([0,0]; n), ([4,7]; 2n)\}$. This summary is also consistent. Hence, *minArbErr*($\vec{f}, 2, 1$) = *minConErr*($\vec{f}, 2, 1$) = n^2.

There are three possible no-deletion summaries of $\vec{f}$ with two buckets. These are specified below with their errors.

$$B_1{:}\ \{([0,4]; 2n+1), ([7,7]; n)\} \qquad \textit{err}_1{:}\ \frac{6}{5}n^2 + \frac{4}{5}n + \frac{4}{5}$$
$$B_2{:}\ \{([0,2]; n+1), ([4,7]; 2n)\} \qquad \textit{err}_2{:}\ \frac{5}{3}n^2 + \frac{2}{3}n + \frac{2}{3}$$
$$B_3{:}\ \{([0,0]; n), ([2,7]; 2n+1)\} \qquad \textit{err}_3{:}\ \frac{4}{3}n^2 - \frac{2}{3}n + \frac{5}{6}$$

It is not difficult to see that for a sufficiently large n, the optimal summary without deletions is $B_1 = \{([0,4]; 2n+1), ([7,7]; n)\}$. Thus, this summary would then be returned by the algorithm NoDeletionSummaryBuckets.

No deletion can be made from B_1 if the summary must be consistent. Hence, SumThenDelErr would return precisely err_1 if ConSingleBucketError is used. It easily follows that

$$\textit{minConSDErr}(\vec{f}, \beta, k) - \textit{minConErr}(\vec{f}, \beta, k) > e.$$

For arbitrary summaries, that need not be consistent, the largest decrement in the error of B_1 that can be achieved by deleting one element is derived by reducing the frequency of element 0 (or equivalently of element 4) by one, which gives us the frequency vector $\vec{f}_1' = \{0{:}n-1, 2{:}1, 4{:}n, 7{:}n\}$ and the summary $B_1' = \{([0,4];2n), ([7,7];n)\}$. Thus, we have that $minArbSDErr(\vec{f}, 2, 1) = \frac{6}{5}n^2 - 2n + 2$.

Now, $minArbSDErr(\vec{f}, 2, 1) - minArbErr(\vec{f}, 2, 1)$ equals

$$\frac{6}{5}n^2 - 2n + 2 - n^2 = \frac{1}{5}n^2 - 2n + 2$$

which is greater than e when n is chosen to be greater than $max\{10, 5e\}$, as required. □

The above theorem implies that one cannot create an optimal summary without deletions, and only afterwards delete elements from the pre-chosen buckets without paying an arbitrarily large penalty in error. However, this algorithm was worth discussion as in practice we will show that it often works quite well. See Section 7 for experimental results.

Delete, then Summarize. An alternative approach that also uses previous work on optimal summaries without deletions is to

- *first* choose k outliers in the dataset to delete, thereby deriving a k-deletion P' of the original dataset P
- *then* compute an optimal summary without deletions for P' (using NoDeletionSummary).

This approach may not return consistent summaries, but it can be used to produce arbitrary summaries. It is difficult to formally prove the quality of the results that this approach will derive. In particular, there are many different ways to define the notion of an outlier. Hence in the first step it is not even clear what we are attempting to delete. Second, most outlier detection algorithms return a set of outliers given a dataset, but cannot be tuned to find a specific number of outliers. This poses a significant technical challenge. Thus, instead of trying to formally prove that this approach will not yield optimal outcomes, we defer consideration of this approach to the experimentation. There we experimentally show that the results returned using this methodology are significantly inferior to all other approaches considered in this paper.

5.2 Optimal Multi-Bucket Summaries

We consider the problem of computing $minArbErr(\vec{f}, \beta, k)$ for arbitrary values of β. Algorithm ArbMultiBErr in Figure 5 is inspired by NoDeletionSummary. ArbMultiBErr uses dynamic programming to fill in a matrix M of dimension size $\times\ \beta \times k$. The position $M[p, \beta', k']$ is updated to be the minimal error of a k'-deletion summary of $\vec{f}_{[\text{first},p]}$ with β' buckets, i.e., $minArbErr(\vec{f}_{[\text{first},p]}, \beta', k')$. Hence, the value of interest will be in $M[\text{last}, \beta, k]$ at the end of the algorithm.

ArbMultiBErr begins by initializing all entries of M for $\beta = 1$ by calling ArbSingleBErr (Lines 1–3). Next, the algorithm considers all choices for $2 \le \beta' \le \beta$, as well as all elements p, other than the first, and all values of $k' \le k$. Algorithm ArbUpdateMatrix is used to update $M[p, \beta', k']$.

Given p, β, k, ArbUpdateMatrix computes $M[p, \beta, k]$, by considering the minimal choice among several options:

- If the frequency value $\vec{f}[p]$ of p is at most k, then we can create β buckets for $\vec{f}_{[\text{first},p]}$ by removing all occurrences of p, and then finding the optimal buckets for $\vec{f}_{[\text{first},\text{prev}(p)]}$ with $k - \vec{f}[p]$ deletions, using the precomputed value in M. (Lines 1–3)

```
Algorithm ArbMultiBErr($\vec{f}, \beta, k$)
1. for $p \leftarrow$ first to last
2.     do for $k' \leftarrow 0$ to $k$
3.         do $M[p, 1, k'] \leftarrow$ ArbSingleBErr($\vec{f}_{[\text{first},p]}, k'$)
4. for $\beta' \leftarrow 2$ to $\beta$
5.     do for $p \leftarrow$ next(first) to last
6.         do for $k' \leftarrow 0$ to $k$
7.             do ArbUpdateMatrix($\vec{f}, \beta', k', p, M$)
8. return $M[\text{last}, \beta, k]$
Algorithm ArbUpdateMatrix($\vec{f}, \beta, k, p, M$)
1. if $\vec{f}[p] \le k$
2.    then $M[p, \beta, k] \leftarrow M[\text{prev}(p), \beta, k - \vec{f}[p]]$
3.    else $M[p, \beta, k] \leftarrow \infty$
4. for $q \leftarrow$ first to prev($p$)
5.     do for $k' \leftarrow 0$ to $k$
6.         do $e \leftarrow M[q, \beta - 1, k - k']$
7.            $e' \leftarrow$ LowerMaxError($\vec{f}_{[\text{next}(q),p]}, k'$)
8.            $M[p, \beta', k'] \leftarrow \min\{M[p, \beta, k], e + e'\}$
```

Figure 5: Returns $minArbErr(\vec{f}, \beta, k)$.

- In addition, for every element q preceding p, and for every $k' \le k$ we consider the case that $\beta - 1$ buckets are used to cover $\vec{f}_{[\text{first},q]}$ with $k - k'$ deletions and a final bucket covers the range from next(q) to p, using k' deletions. The error of such a summary is $M[q, \beta - 1, k - k'] +$ LowerMaxError($\vec{f}_{[\text{next}(q),p]}, k'$). (Lines 4–8)

Example 5.4. Recall the frequency vector $\vec{f}$ from Example 5.1. With ArbMultiBErr we will get an error of 0.125 corresponding to the deletion of one occurrence of the element of 6, as well as the element 8, and the buckets $\{([1,4];6), ([5,7];6)\}$. □

Theorem 5.5. *Given $\vec{f}$, β and k,* ArbMultiBErr($\vec{f}, \beta, k$) *computes $minArbErr(\vec{f}, \beta, k)$ in time*

$$O(\beta k^2 \text{size}^2 + \text{size}^2(\text{size} \log \text{size} + k)).$$

Proof. Correctness is easy, and follows from Theorem 4.1, and from the fact that the algorithm considers all possible ways to divide the frequency vector into buckets, and all possible choices of deletions. Note that the algorithm directly considers removals of elements only on the right-hand of buckets. (For the first bucket both sides are considered). However, any removals of elements at the left-hand side of these buckets can equivalently be thought of as removals from the right-hand side of the previous bucket.

To achieve the stated runtime, we pre-compute the values of LowerMaxError($\vec{f}_{[p,q]}, k'$) for all $p < q$ (with non-zero frequencies) and for all $k' \le k$. For a choice of p and q, we can compute LowerMaxError($\vec{f}_{[p,q]}, k'$) for all values of k' in time size log size + k, by sorting $\vec{f}_{[p,q]}$ by frequency values, and then computing the results incrementally starting with $k = 0$. This give a total time of $\text{size}^2(\text{size} \log \text{size} + k)$ for LowerMaxError computations. These values are then read from a pre-computed data structure, in $O(1)$ when running ArbUpdateMatrix.

The remainder of the algorithm runs in time $O(\beta k^2 \text{size}^2)$ due to the nested loops of length β (Line 4 of ArbMultiBErr), of length size (Line 5), of length k (Line 6) and the loop of lengths size (Line 4 of ArbUpdateMatrix) and length k (Line 5). □

```
Algorithm ConUpdateMatrix($\vec{f}, \beta, k, p, M$)
1.  if $\vec{f}[p] \le k$
2.      then $M[p, \beta, k] \leftarrow M[\text{prev}(p), \beta, k - \vec{f}[p]]$
3.      else $M[p, \beta, k] \leftarrow \infty$
4.  for $q \leftarrow \text{prev}(p)$ to first
5.      do $e \leftarrow M[q, \beta - 1, k] + SSE[\text{next}(q), p]$
6.          $M[p, \beta, k] \leftarrow \min\{M[p, \beta, k], e\}$
```

Figure 6: Used as a sub-procedure in order to compute *minConErr*($\vec{f}, \beta, k$).

5.3 Optimal Consistent Multi-Bucket Summaries

We now consider the problem of computing the error for optimal consistent summaries, i.e., *minConErr*($\vec{f}, k, \beta$). Let ConMultiBErr be an algorithm identical to ArbMultiBErr, except that in Line 3 it calls ConSingleBErr and in Line 7 it calls ConUpdateMatrix. Recall that ConSingleBErr appears in Figure 2. ConUpdateMatrix appears in Figure 6. It remains to explain how this algorithm works.

ConUpdateMatrix($\vec{f}, \beta, k, b, M$) is called to update the value in $M[p, \beta, k]$ for $\beta > 1$. It first considers the case that the right-most element (which is now p) can be completely removed (Lines 1–3), in which case the previously computed error can be used. It then (Lines 4–6) considers all splits of $\vec{f}$ into two, such that $\beta - 1$ buckets are used to summarize $\vec{f}_{[\text{first},q]}$ (with error appearing in $M[q, \beta - 1, k]$) and one bucket is used to summarize $\vec{f}_{[\text{next}(q),p]}$ (with error appearing in $SSE[\text{next}(q), p]$). Note that in this case, we can assume that no deletions occur in the rightmost bucket, as:

(1) we already considered the case that occurrences of p are removed and
(2) any removals of elements at the left-hand side of the final bucket can equivalently be thought of as removals from the right-hand side of the previous bucket.

Example 5.6. Recall once again $\vec{f}$ from Example 5.1. With algorithm ConMultiBErr we will get an error of 0.17 corresponding to the deletion of the element of 4, as well as the element 8, and the buckets $\{([1, 3]; 5), ([5, 7]; 7)\}$. □

We can show the following result. Observe that ConMultiBError is only slower than algorithm NoDeletionSummary by a factor of k.

Theorem 5.7. *For a given $\vec{f}$, β and k,* ConMultiBErr($\vec{f}, \beta, k$) *computes minConErr*($\vec{f}, \beta, k$) *in time $O(\beta\, k\, \text{size}^2)$.*

Proof. Correctness is easy to show as the different ways to split $\vec{f}$ to different buckets, while deleting all occurrences of elements not in these buckets, is considered. To show the runtime, observe that ConSingleBErr runs in $O(\text{size})$. Therefore, Lines 1–3 of ConMultiBErr run in time $O(k\,\text{size}^2)$. ConUpdateMatrix runs in time $O(\text{size})$. Therefore, Lines 4–7 of ConMultiBErr run in time $O(\beta\, k\, \text{size}^2)$, as required. □

6 SUMMARIES OF DATASETS WITH MULTIPLE COLUMNS

We now briefly consider the problem of finding an optimal summary for datasets with multiple columns, in the presence of outliers. Let $\mathcal{R}$ be a relation with n columns, $(\beta_1, \dots, \beta_n)$ be a n- tuple indicating the number of buckets per column that can be allocated and let k be a number. We consider two types of outliers:

- *value deletions*: a specific value in some tuple t can be considered an outlier in a specific column (even though the other values in t are not outliers), and thus, be deleted from the column;
- *tuple deletions*: a tuple t can be considered an outlier, and thus, be deleted from the relation.

Our goal is to find a summary with β_i buckets for column i, that minimizes the sum of error over all columns, when there can be at most k value or tuple deletions.

	C	X
t_1^1	2	10
t_2^1	2	20
t_3^1	2	30
t_1^2	4	20
t_2^2	4	30
t_3^2	4	40
t_1^3	6	10
t_2^3	6	40
t_3^3	6	50
t^0	1	0
t^1	3	0
t^2	5	0
t^3	7	0

$\mathcal{R}$

	C	X
t_1^1	2	10
t_2^1	2	20
t_3^1	2	30
t_1^2	4	20
t_2^2	4	30
t_3^2	4	40
t_1^3	6	10
t_2^3	6	40
t_3^3	6	50
t^0	1	0
t^1	3	0
t^2	5	0
t^3	7	0

$\mathcal{R}$

Figure 7: Example multicolumn dataset

Example 6.1. Consider the relation $\mathcal{R}$ repeated twice in Figure 7. (Ignore the first column which names the tuples for convenience.) Suppose $k = 6$. In the version of $\mathcal{R}$ on the left, we have chosen up to k values to be deleted per column, while in the version on the right we have chosen up to k tuples to be deleted. (Both are indicated by the color blue.)

Now, suppose we can allocate one bucket for the first column C and three buckets for the second column X. For both types of deletions we can find a *perfect summary* (i.e., one with no error). For the deletions of the left we would choose buckets:

- $([1, 7]; 7)$ for column C and
- $([0, 0]; 4)$, $([20, 20]; 2)$, $([30, 30]; 2)$ for column X.

For the deletions on the right we would choose buckets:

- $([1, 7]; 7)$ for column C and
- $([0, 0]; 4)$, $([20, 20]; 1)$, $([40, 40]; 2)$ for column X.

The above example demonstrated summaries with $(1, 3)$ buckets for the columns and six value/tuples deletions that are clearly optimal (as they have no error at all). The problem arises as to how hard it is to find such optimal summaries in the general case.

Theorem 6.2. *Let $\mathcal{R}$ be a relation with n columns, let $(\beta_1, \dots, \beta_n)$ be the bucket size bounds and let k be natural number. Then:*

(1) *An optimal summary satisfying the bucket size bounds, with up to k value deletions per column can be found in polynomial time.*
(2) *The problem of determining whether there exists a perfect summary of $\mathcal{R}$ with up to k tuple deletions is NP-complete, even if*
   - *the dataset has two attributes* or
   - *all attributes can have at most two buckets.*

The first claim is easy to show as we can simply apply the algorithms from the previous section to each column separately. For the second claim, in the case in which there are only two attributes we can show hardness by a reduction from the monotone satisfying assignment problem. For the case where there are at most two buckets per attribute we can show hardness by a reduction from 1-in-3 SAT. The proofs are omitted due to space limitations, but the main ideas of the second claim are demonstrated in the following examples.

Example 6.3. We demonstrate hardness when the relation contains only two columns.

Let ψ be a positive 3-SAT formula with m clauses $C_1, \dots, C_m$, over the variables $x_1, \dots, x_n$. We say that ψ is h-monotone satisfiable if ψ can be satisfied by an assignment in which at most h variables are assigned true. This is a well-known NP-complete problem.

Let $\mathcal{R}$ be a relation of cardinality 2, containing two types of tuples:

- For each clause $C_i = x_{i_1} \vee x_{i_2} \vee x_{i_3}$ we add to $\mathcal{D}$ three tuples $t_1^i = (2i, 10i_1)$, $t_2^i = (2i, 10i_2)$, $t_3^i = (2i, 10i_3)$.
- For every $0 \geq j \geq m$ we add the tuple $t^j = (2j + 1, 0)$.

Observe that $\mathcal{R}$ contains $4m + 1$ tuples. For example, the formula

$$(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee x_4 \vee x_5)$$

corresponds to the relation $\mathcal{R}$ in Figure 7.

Now, we choose $k = 2m$ and $(1, h + 1)$ as our bucket bound, i.e., a single bucket for the first column and $h + 1$ buckets for the second column. We can show that ψ is h-monotone if and only if there is a k-tuple deletion for which there is a summary bounded by $(1, h + 1)$ with zero error.

For example, suppose that $h = 2$. Consider the k-tuple deletion $\mathcal{R}'$ derived by removing the blue tuples in the relation on the right. It is easy to see that we can use a single bucket for the first column, and $h + 1 = 3$ buckets for the second column, to derive a summary with error of zero. Observe also that $\mathcal{R}'$ corresponds to a satisfying assignment that assigns at most h variables the value true (in this case x_2 and x_4 as the elements 20 and 40 remain after the k-tuple deletion).

One can show (omitting some precise details due to space limitations) that the opposite holds too, i.e., that any k-deletion that has a summary bounded by $(1, h + 1)$ with an error of zero corresponds to an h-monotone satisfying assignment. □

Example 6.4. We demonstrate why determining whether there exists a k-tuple deletion $\mathcal{R}'$ with a perfect summary S is a difficult problem, even when there are only two buckets per column.

Consider a positive 3-SAT formula with m clauses $C_1, \dots, C_m$, over the variables $x_1, \dots, x_n$. Recall that the 1-in-3 SAT problem is to determine whether there exists an assignment that satisfies precisely one variable in each clause.

We create a relation $\mathcal{R}$ with $n + 1$ columns, and three types of tuples:

	X_1	X_2	X_3	X_4	X_5	C
t_1^1	2	-2	-2	0	0	1
t_2^1	-2	2	-2	0	0	1
t_3^1	-2	-2	2	0	0	1
t_1^2	0	2	-2	-2	0	2
t_2^2	0	-2	-2	-2	0	2
t_3^2	0	-2	-2	2	0	2
t_1^3	2	0	0	-2	-2	3
t_2^3	-2	0	0	2	-2	3
t_3^3	-2	0	0	-2	2	3
$t_1^{c,1}$	0	0	0	0	0	1
$t_2^{c,1}$	0	0	0	0	0	1
$t_1^{c,2}$	0	0	0	0	0	2
$t_2^{c,2}$	0	0	0	0	0	2
$t_1^{c,3}$	0	0	0	0	0	3
$t_2^{c,3}$	0	0	0	0	0	3
t_1^0	0	0	0	0	0	0
t_2^0	0	0	0	0	0	0
t_3^0	0	0	0	0	0	0

Figure 8: Example relation in 1-in-3 SAT reduction.

- For each clause $C_i = x_{i_1} \vee x_{i_2} \vee x_{i_3}$ we add three tuples as follows:
  - for all $j = 1, 2, 3$ we have $t_j^i[n + 1] = i$;
  - for all $j = 1, 2, 3$ and $k \notin \{i_1, i_2, i_3\}$ we have $t_j^i[k] = 0$;
  - for all $j = 1, 2, 3$ and $k = 1, 2, 3$ we have $t_j^i[i_k] = 2$ if $k = j$ and $t_j^i[i_k] = -2$, otherwise.
- For each $i \leq m$, we add two tuples $t_1^{c,i} = t_2^{c,i} = (0, \dots, 0, i)$.
- Finally, we add three tuples $t_1^0 = t_2^0 = t_3^0 = (0, \dots, 0)$.

In total, $\mathcal{R}$ has $5m + 3$ tuples. For example, the formula

$$(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee x_4 \vee x_5)$$

corresponds to the dataset in Figure 8.

Now, suppose that $k = 2m$ and our bucket bound allows two buckets per attribute. For the example in Figure 8, we can remove up to 6 tuples. It is not difficult to see that if we remove the blue tuples, we are left with a data set $\mathcal{R}'$ for which we can create a summary with error of zero. This holds since each column $x_1, \dots, x_5$ has precisely two different elements (which can be placed in different buckets) and column C has three occurrences each of every value from 0 to 3. Observe also that the remaining rows define an assignment that satisfies precisely one variable in each clause, if we assign each x_i true if $\mathcal{R}'[X_i]$ contains element 2, and false otherwise. (In the example given this corresponds to mapping x_2 and x_5 to true.)

Indeed, with a bit of reasoning it is possible to show the opposite direction too, i.e., a k-tuple deletion $\mathcal{R}'$ for which there is a summary with error of zero corresponds to a solution to the 1-in-3 SAT problem. Intuitively, this holds since such a tuple-deletion must retain, for each x_i, only the elements 0 and 2 or only the elements 0 and -2 (thus, corresponding to a truth assignment).

Dataset	size_P	$\text{size}_{\bar{f}}$
ForestAspect	581,012	361
ForestDistHydro	581,012	136
IncomeCapitalGain	48,842	123
IncomeHoursWeek	48,842	96

Figure 9: Real datasets.

In addition, we must retain precisely one tuple from every triple t_1^i, t_2^i, t_3^i so as to have error of zero on the column C, i.e., we will map precisely one variable in each clause to the value true.[3] □

7 EXPERIMENTAL RESULTS

In the following we discuss our experimental results which consider different datasets, and varying β and percentage of elements that can be deleted, denoted ρ. As default values, we use $\beta = 10$ and $\rho = 2\%$. In our experiments, we prefer to consider the percentage of elements to be deleted (ρ) instead of an absolute number (k), as the size of the dataset should determine the number of elements we are willing to delete. We set a time limit of five minutes, i.e., tests that exceeded the time limit were excluded from the results.

All experimentation was run on a standard Win7 desktop with 16GB RAM and an Intel i5-4570 processor. The algorithms were implemented in Java, and experimentation was run with a limit of 1GB of main memory.

Algorithms. We implemented five algorithms for the k-deletion problem. We abbreviate the algorithm names as follows. We use Arb for ArbMultiBErr and Con for ConMultiBErr. We implemented algorithm SumThenDelErr with ArbSingleBErr, and ConSingleBErr, henceforth referred to as SDArb and SDCon, respectively. Finally, we implemented the *delete then summarize* strategy using a well-known distance-based outlier detection algorithm [4] to remove outliers and then summarizing the result using NoDeletionSummary. This algorithm is referred to as DS.

Datasets. We run our algorithms on both synthetic datasets and real datasets. The synthetic datasets were generated by two different distributions—randomly permuted zipf distribution with skew parameter $z = 0.85$ (as in [11]), and normal distribution with variance proportional to 25% of the value range. For each experiment over the synthetic datasets, we generated three instances of the dataset with the same parameters, ran the tests on all three, and computed the average result. Our default synthetic datasets have 50,000 elements and a range of 100, i.e., the elements were sampled from $[1, 100]$.

We also use several real datasets, taken from the UCI KDD Archive[4] and the UCI Machine learning repository[5]. When considering the size of these datasets, there are two factors of importance: the number of elements in the dataset, called size_P and the number of different elements in the dataset (i.e., the number of non-zero elements in the frequency vector), called $\text{size}_{\bar{f}}$. Note that $\text{size}_{\bar{f}}$ corresponds to the value size discussed in the previous sections. The dataset sizes are summarized in Figure 9.

[3]More details are required to make a precise proof, e.g., an assumption that no variable appears in all clauses, but these are omitted due to space limitations.
[4]http://kdd.ics.uci.edu/
[5]http://archive.ics.uci.edu/ml/index.php

The Forest dataset contains data obtained from US Forest Service, where the Aspect column is aspect in degrees azimuth, and DistHydro is the horizontal distance to nearest surface water features. For DistHydro we rounded to the nearest 10 meters. The Income dataset (referred to as Adult in the repository) was originally extracted from the Census Bureau 1994 database. We use the columns CapitalGain and HoursWeek, which is the number of work hours per week.

In all runtime graphs, the y-axis is in log scale and the units are in seconds. In all error graphs, the data points represent the ratio of the error in the current setting to the minimum error of the same settings with no deletions. Intuitively, lower values indicate a larger percentage of reduction in error due to deletions.

Comparing All algorithms. In our first test, we compare the error and runtime of all algorithms over the Income dataset and synthetic dataset, using the default values for β and ρ. The results of this test appear in Figures 10a, 10b, 10c and 10d. Due to space limitations, we omit the Forest dataset from this experiment, but similar trends can be seen there.

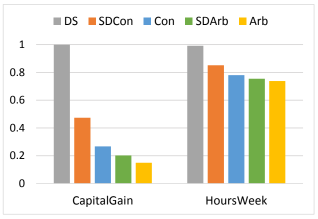
Algorithm DS, which uses distance based outlier detection, is mostly unsuccessful in improving the error. In our testing, there have even been cases in which the error increases using this algorithm. This is not surprising, both due to the fact that outliers are removed before determining bucket boundaries and because the algorithm does not always even produce enough outliers to delete. This result was consistent in all experiments and therefore, we do not consider it in remainder of the experimentation.

For the other algorithms, the graphs show a consistent order of the error reduction—Arb always finds the optimal error, then SDArb, Con and finally SDCon have increasing errors. The runtime performance, on the other hand, is in almost the opposite order, albeit Con and SDCon are very close in runtime. One can observe that Con achieves error that is fairly close to Arb and SDArb with runtime that is better in orders of magnitude.

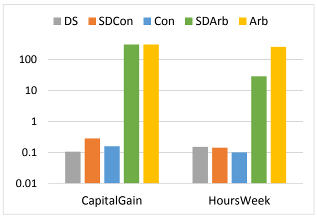
Note that the runtime of Arb is so poor over the permuted zipf dataset, that it timed out, and therefore its runtime and error are missing. This occurs often in the experiments both for Arb and for SDArb, as these algorithms have a runtime that is a function of k^2, even when a single bucket is used.

Varying percentage of Deletions. We tested the effect of changing the percentage of elements deleted by varying the value of ρ, and running our algorithms with k chosen as $\rho \cdot \text{size}_P$. The error and runtime results of these tests appear in Figures 10e, 10g, 11a ,10f, 10h and 11b. Note that for the synthetic datasets, we present the error for both permuted zipf and normal distribution, but the runtime only for the former. Since the runtime is not affected by the data distribution, the graph omitted is almost identical to the one appearing.

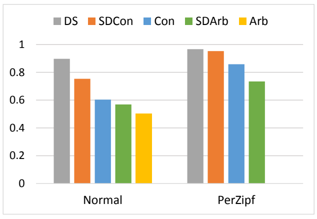
As expected, in all algorithms and datasets, the error is reduced as ρ grows. The degree to which the error is reduced differs between the datasets, due to differences in the data distributions. For the synthetic data, deletions are more significant in normal distribution than in permuted zipf distribution. Almost consistently, Con and SDCon have significantly lower runtimes than SDArb which is much faster than Arb. It is interesting to see that in most tests, on lower values of ρ, SDCon is faster than Con, but as ρ increases, SDCon becomes slower. The only exception in our tests was on the dataset ForestAspect, where the number of distinct values i.e., $\text{size}_{\bar{f}}$, is much higher than in the other



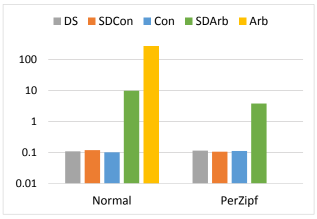
(a) Income Error



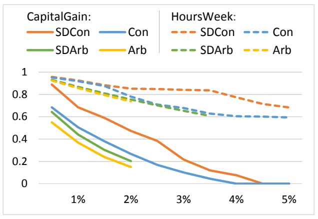
(b) Income Time



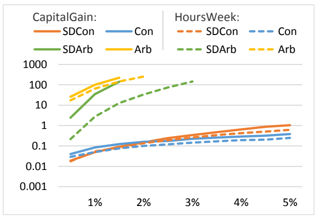
(c) Synthetic Error



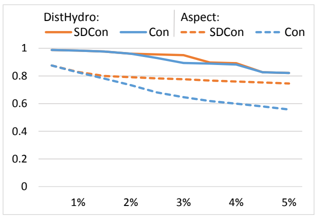
(d) Synthetic Time



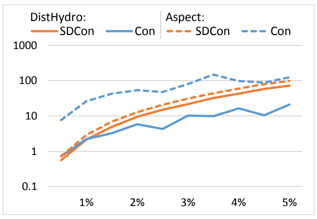
(e) Varying ρ: Income Error



(f) Varying ρ: Income Time

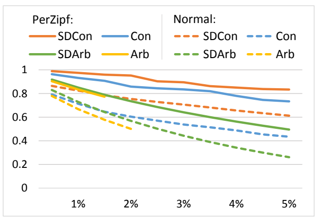


(g) Varying ρ: Forest Error

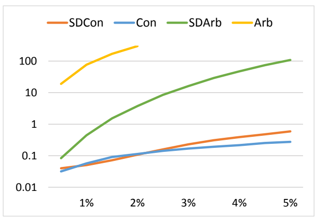


(h) Varying ρ: Forest Time

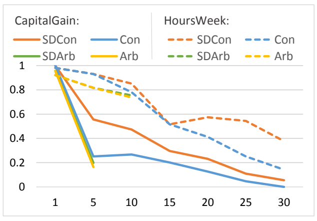
Figure 10: Experiments for default values and varying ρ over real datasets.



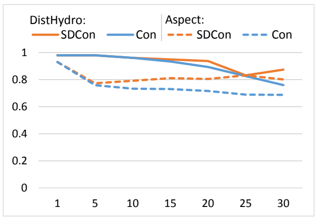
(a) Varying ρ: Synthetic Error



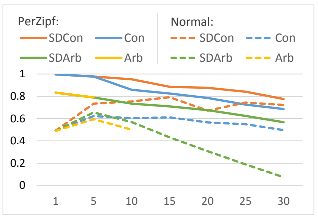
(b) Varying ρ: Synthetic Time



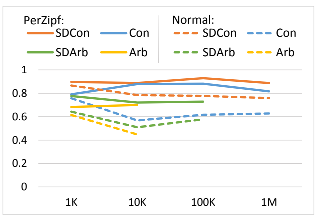
(c) Varying β: Income Error



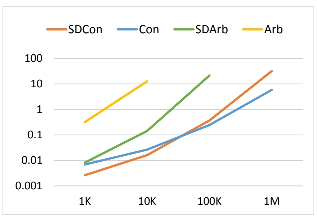
(d) Varying β: Forest Error



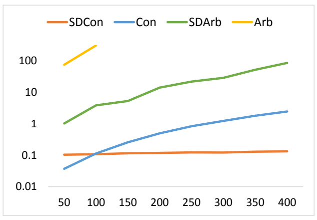
(e) Varying β: Synthetic Error



(f) Varying size_P: Error



(g) Varying size_P: Time



(h) Varying $\text{size}_{\vec{f}}$: Time

Figure 11: Experiments for varying ρ, β, size_P and $\text{size}_{\vec{f}}$

datasets, and larger $\text{size}_{\vec{f}}$ degrades the runtime of Con. This phenomenon will be discussed later in our experiments, when we consider varying the value $\text{size}_{\vec{f}}$.

Varying Number of Buckets. Next, we tested the results of our algorithms when varying the number of buckets. The runtimes, which do not appear in graphs due to space limitations, increase linearly together with β. The error reduction is shown in Figures 11c, 11d, and 11e. In general, as the number of buckets increase, the error decreases. On IncomeCapitalGain for instance, the error goes down to zero with 30 buckets, but On ForestDistHydro on the other hand, the improvement in error with 30 bucket is only a little under 80%. There are also cases where the reduction in error actually increases when buckets are added. This counter-intuitive result is because the error ratio is computed with respect to the *same settings* (and thus, the same number of buckets) without deletions. As the number of buckets increases, the relative gain by removing elements may decrease. This is particularly noticeable for SDCon.

Varying the Values of size_P *and* $\text{size}_{\vec{f}}$. We studied how the total number of elements size_P affects the results. To this end, we created synthetic datasets of increasing size. In Figures 11f and 11g depict the change in error and in runtime over different sizes of multisets of elements. The relative error remains approximately the same even as the dataset size increases. This apparently is the result of the fact that the proportion of elements deleted relative to the entire set remains the same. In contrast, the runtime grows linearly with the size of the dataset. (Note that both axes of Figure 11g are on log scale.) The linear increase in runtime occurs as the constant 2% element deletion translates to bigger absolute numbers of k-deletions as the dataset size increases. Similarly to before, for larger datasets (i.e., larger values of k), SDCon runs slower than Con.

In the next experiment we considered increasing the number of distinct elements $\text{size}_{\vec{f}}$. In this experiment the dataset remained at the default size of 50,000 elements, but we increased the range of values so as to increase $\text{size}_{\vec{f}}$. Due to space limitations, the

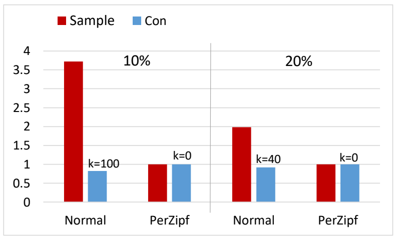


Figure 12: Error of Sampling versus Optimal Deletion

error graph is omitted, as it had the same trends as in Figure 11f. Figure 11h shows the runtime as size$_{\vec{f}}$ increases. As expected, all algorithms have runtime that grow with size$_{\vec{f}}$, except for SDCon whose runtime is independent of size$_{\vec{f}}$.

Comparison with Sampling for Summary Construction. Since histogram construction is costly, in many systems histograms are computed over a sample of the dataset. Intuitively, it might appear that such a sampling based approach will yield similar results to our algorithms that allow for optimal deletions, as perhaps outliers will not be chosen in the sample. In order to check how sampling affects the error in practice, for each dataset P, we constructed a summary B over sampled data using the NoDeletionSummary algorithm. Some of the elements in the original dataset P did not fall into the range of any of the buckets, and were considered to have been "deleted" from the dataset, yielding a new dataset P'. We then computed $err(B, P')$ and compared this value with the error of the optimal consistent summary allowing for $|P - P'|$ deletions.

The results of this experiment on the synthetic datasets, with sampling of 10% and 20% are depicted in Figure 12. (The results on the real datasets were similar.) The size of $|P - P'|$, i.e., the value that was then used as k in the input of ConMultiBErr, appears on top of the bars. The results show two cases. With the permuted Zipf dataset there were no deletions at all as a result of the sampling, and thus, the error is exactly the same as the original error of NoDeletionSummary algorithm. With the normal dataset, there was a larger number of deletions, but the elements deleted by the sampling actually increased the error significantly. When ConMultiBErr was run with the same number of deletions, the error was reduced. We conclude that sampling cannot be used as an effective technique in order to reduce the error of a summary.

8 CONCLUSION

This paper studied the problem of an optimal summary with β buckets, when k outliers need not be covered. We presented the first algorithms for this problem, by taking two different approaches for deleting elements (arbitrarily and consistently), and attempting to determine which elements to delete at different stages (before, during and after finding the optimal summary). We also considered the problem of multi-column datasets. The experimentation shows that algorithm Con has the best balance between low error and low runtime. In addition, on smaller datasets Arb performs very well, providing significantly lower error.

As future work, we intend to consider domains in which there is no natural (useful) ordering over the values, and hence, buckets cannot be described by their endpoints. While our algorithms work well for moderately large datasets, they degrade when the dataset becomes huge. We intend to develop approximation algorithms to deal with this case. Finally, another important direction is finding optimal summaries with outliers over streaming data.

ACKNOWLEDGMENTS

The authors were partially supported by the Israel Science Foundation (Grant 879/16).

REFERENCES

[1] Hee-Kap Ahn, Sang Won Bae, Erik D. Demaine, Martin L. Demaine, Sang-Sub Kim, Matias Korman, Iris Reinbacher, and Wanbin Son. 2011. Covering Points by Disjoint Boxes with Outliers. *Comput. Geom. Theory Appl.* 44, 3 (April 2011), 178–190. https://doi.org/10.1016/j.comgeo.2010.10.002

[2] Rossen Atanassov, Prosenjit Bose, Mathieu Couture, Anil Maheshwari, Pat Morin, Michel Paquette, Michiel Smid, and Stefanie Wuhrer. 2009. Algorithms for optimal outlier removal. *Journal of discrete algorithms* 7, 2 (2009), 239–248.

[3] Sang Won Bae. 2019. Computing a minimum-width square or rectangular annulus with outliers. *Computational Geometry* 76 (2019), 33 – 45. https://doi.org/10.1016/j.comgeo.2018.08.002

[4] Stephen D. Bay and Mark Schwabacher. 2003. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *SIGKDD.* 29–38. https://doi.org/10.1145/956750.956758

[5] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. 1998. Random Sampling for Histogram Construction: How much is enough?. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.* 436–447. https://doi.org/10.1145/276304.276343

[6] Matthew Gebski and Raymond K. Wong. 2007. An Efficient Histogram Method for Outlier Detection. In *DASFAA.* 176–187. https://doi.org/10.1007/978-3-540-71703-4_17

[7] S. Guha and N. Koudas. 2002. Approximating a data stream for querying and estimation: algorithms and performance evaluation. In *Proceedings 18th International Conference on Data Engineering.* 567–576. https://doi.org/10.1109/ICDE.2002.994775

[8] Sudipto Guha, Nick Koudas, and Kyuseok Shim. 2001. Data-streams and Histograms. In *STOC.* ACM, New York, NY, USA, 471–475. https://doi.org/10.1145/380752.380841

[9] Yannis E. Ioannidis and Viswanath Poosala. 1995. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. *SIGMOD Rec.* 24, 2 (May 1995), 233–244. https://doi.org/10.1145/568271.223841

[10] H. V. Jagadish, Hui Jin, Beng Chin Ooi, and Kian-Lee Tan. 2001. Global Optimization of Histograms. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001.* 223–234. https://doi.org/10.1145/375663.375687

[11] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. 1998. Optimal Histograms with Quality Guarantees. In *VLDB.* 275–286. http://www.vldb.org/conf/1998/p275.pdf

[12] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. 1996. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.* 294–305. https://doi.org/10.1145/233269.233342

[13] Saket Sathe and Charu C. Aggarwal. 2018. Subspace Histograms for Outlier Detection in Linear Time. *Knowl. Inf. Syst.* 56, 3 (Sept. 2018), 691–715. https://doi.org/10.1007/s10115-017-1148-8

[14] Yizhou Yan, Lei Cao, and Elke A. Rundensteiner. 2017. Scalable Top-n Local Outlier Detection. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17).* ACM, New York, NY, USA, 1235–1244. https://doi.org/10.1145/3097983.3098191

BalanSiNG: Fast and Scalable Generation of Realistic Signed Networks

Jinhong Jung
Seoul National University
jinhongjung@snu.ac.kr

Ha-Myung Park
Kookmin University
hmpark@kookmin.ac.kr

U Kang
Seoul National University
ukang@snu.ac.kr

ABSTRACT

How can we efficiently generate large-scale signed networks following real-world properties? Due to its rich modeling capability of representing trust relations as positive and negative edges, signed networks have spurred much interests with various applications. Despite its importance, however, existing models for generating signed networks do not correctly reflect properties of real-world signed networks.

In this paper, we propose BalanSiNG, a novel, scalable, and fully parallelizable method for generating large-scale signed networks following realistic properties. We identify a self-similar balanced structure observed from a real-world signed network, and simulate the self-similarity via Kronecker product. Then, we exploit noise and careful weighting of signs such that our resulting network obeys various properties of real-world signed networks. BalanSiNG is easily parallelizable, and we implement it using Spark. Extensive experiments show that BalanSiNG efficiently generates the most realistic signed networks satisfying various desired properties.

KEYWORDS

Signed Network Modeling; Balance Theory; Stochastic Kronecker Signed Graph; Balanced Signed Network Generator

1 INTRODUCTION

Signed networks [26] exhibit relationships between nodes as positive (trust) and negative (distrust) edges, and various online social services such as Epinions [10] have naturally formed signed networks by allowing users to express their trust. Inspired by these interesting trust relationships, many researchers have been recently attracted to mining useful information from signed networks, inducing advanced techniques for diverse applications such as sign prediction [22, 25], link prediction [40, 47], node ranking [15, 16, 28], node embedding [20, 46], node classification [42], anomaly detection [21], and community detection [5, 48].

Even though signed networks are important resources in social network analysis, the understanding of synthetically generating realistic signed networks from scratch was nascent. In unsigned networks, many sophisticated generation models have been proposed, including Barabási-Albert (BA) [1], Forest Fire (FF) [27], Stochastic Kronecker Graph (SKG) [23], and Recursive Matrix (R-MAT) [4]. Among those models, SKG and R-MAT have received significant interest from data mining communities [12, 13, 31, 34, 37] since they well capture various properties of real-world graphs such as power-law degree distributions [1, 8, 9, 23, 31], shrinking effective diameters [3, 9, 27], power-law singular value distribution [4, 9, 23], etc.

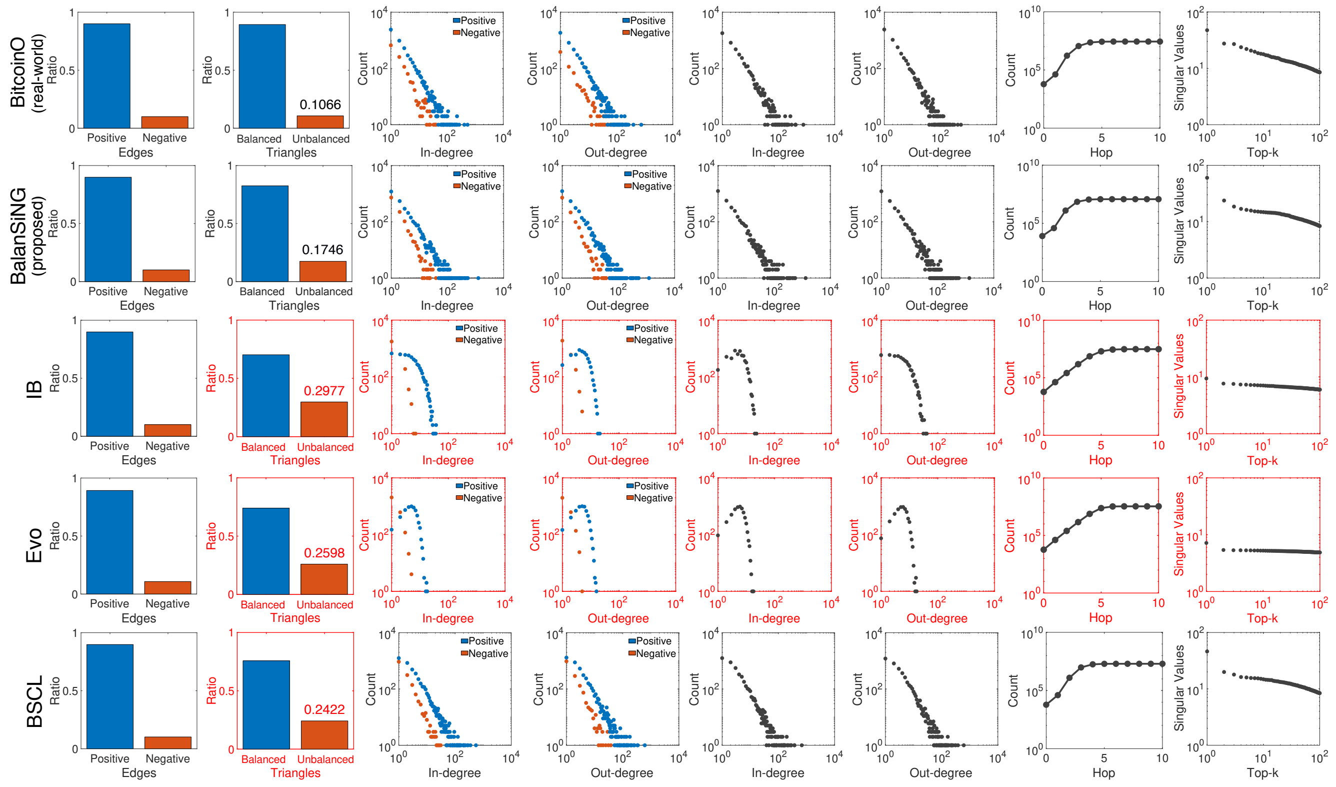
However, the existing models cannot generate realistic signed networks because they do not provide a mechanism for determining signs of edges. Real-world signed networks exhibit not only the traditional properties in unsigned networks, but also distinct characteristics derived from signs (Figure 11). Especially, real-world signed networks are dominated under balance theory [2, 11] that plays a crucial role in the construction of signed networks [6, 26]. According to the balance theory, balanced triangles are more likely to be created than unbalanced ones in real signed networks (details in Section 2). Thus, modeling signed networks demands careful considerations on how to positively or negatively associate three nodes on each triangle.

Motivated by this, several methods have been proposed for signed network generation considering the balance theory. Vukašinović et al. [45] proposed an interaction based model (IB) simulating the generation of signed edges using ant pheromone mechanism and the balance theory. Ludwig et al. [29] suggested an evolutionary model (Evo) that randomly inserts or removes signed edges over time so that the evolving network follows the balance theory. Derr et al. [6] have recently proposed Balanced Signed Chung-Lu (BSCL), the state-of-the-art model imitating an input network based on Transitive Chung-Lu [35] and the balance theory. However, they are limited in generating realistic signed networks (see Figure 1), and computationally inefficient. Furthermore, the scale of existing signed networks remains small; consequently, researchers have suffered from the lack of large-scale signed networks when testing the scalability of their methods. Thus, generating realistic large-scale networks is extremely useful to evaluate the scalability [14, 19, 30–32], simulate their performance depending on various properties of networks [17, 18, 23, 39], and anonymize their data [6, 24].

In this paper, we propose BalanSiNG (Balanced Signed Network Generator), a novel and scalable method for generating synthetic but realistic signed networks. We first identify a self-similar pattern observed from a real signed network. Then, we design Basic Stochastic Kronecker Signed Graph (SKSG-B), a basic model that simulates the self-similarity using Kronecker product and generates fully balanced signed networks. On top of SKSG-B, we propose Stochastic Kronecker Signed Graph (SKSG) by adding random noises to the self-similar pattern and introducing careful weighting to increase the probability of forming positive edges to generate signed networks following real-world properties. From SKSG, we derive BalanSiNG that efficiently creates signed edges fully in parallel. Through extensive experiments, we show that BalanSiNG efficiently generates the most realistic signed networks capturing various properties of real-world signed networks.

Our main contributions are summarized as follows:

- **Novel self-similarity.** We suggest a novel self-similar pattern called *self-similar balanced structure* to be satisfied for generating signed networks (Figure 4).
- **Method.** We propose BalanSiNG, an efficient and parallel method that simulates the suggested self-similarity



(a) Global sign distribution (b) Balanced triangle distribution (c) Signed in-degree distribution (d) Signed out-degree distribution (e) In-degree distribution (f) Out-degree distribution (g) Hop plot (h) Singular value distribution

Figure 1: BalanSiNG generates the most similar network to the real-world network, compared to other methods. The plots show the comparison of properties from real-world signed networks and those from BalanSiNG and competitors. We use BitcoinO dataset [22] for representing the properties of real-world signed networks; other real-world networks give similar results. (a)-(d) illustrate properties derived from edge signs, and (e)-(h) depict traditional properties of real-world networks regardless of edge signs (see Section 2.1). Red colored boxes denote that the corresponding graph does not match the corresponding property.

by using Kronecker product, exploiting noise, and careful weighting (Algorithm 3). BalanSiNG generates signed networks satisfying various desired properties of real-world signed networks listed in Section 2.1.

- **Experiments.** We demonstrate that BalanSiNG generates the most realistic signed networks following real-world properties compared to competitors as shown in Figure 1. We also show that BalanSiNG generates signed networks up to 265× faster than the state-of-the-art method, and near linearly scales up w.r.t. the number of edges on both single and distributed machines (Figure 9).

The source code of BalanSiNG and datasets are available at **https://datalab.snu.ac.kr/balansing**.

2 PRELIMINARIES

We describe the desired properties of real-world signed networks to consider when generating a synthetic signed network in Section 2.1. We formally define the problem addressed in this paper in Section 2.2. We then review Stochastic Kronecker Graph (SKG), a representative generation model for unsigned networks to capture the concept of self-similarity simulation in Section 2.3.

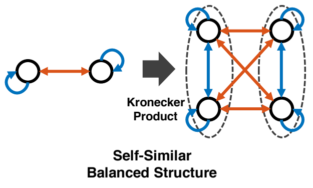
Symbols used in this paper are summarized in Table 1. Throughout the paper, we use a blue arrow and a red arrow to indicate a positive edge and a negative edge, respectively.

2.1 Desired Properties of Signed Networks

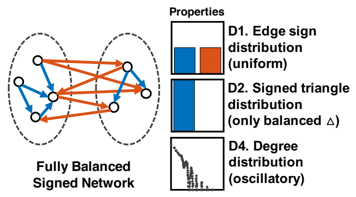
We investigate real-world signed networks to grasp their unique properties to be satisfied when generating signed networks. As shown in the first row of Figure 1, there are not only unique properties derived from signs on edges but also traditional ones studied in unsigned networks. The properties of other real-world networks are in Figure 11. We examine properties induced by signs in Section 2.1.1, and then review the typical ones regardless of signs in Section 2.1.2.

2.1.1 *Properties with respect to signs on edges.*

- **D1) Positively skewed sign proportion [25, 26, 43].** Real-world signed networks contain much more positive edges than negative ones, as demonstrated in the first row of Figure 1(a).
- **D2) Highly balanced triangle proportion [6, 11, 26, 41, 43, 48].** Signed triangles have been extensively studied in signed networks based on balance theory [2, 11] stating that triangles $\triangle_{+++}$ with three positive signs and those $\triangle_{+--}$ with one positive sign are much more plausible than other types of triangles $\triangle_{++-}$ and $\triangle_{---}$. The former are called *balanced triangles*, and the latter are *unbalanced triangles.* Thus, the ratio of balanced triangles is much larger than that of unbalanced triangles as shown in Figure 1(b).
- **D3) Power-law degree distribution for only positive or negative edges [43].** In scale-free networks, in- and out-degree distributions follow a power-law [1]. In real-world signed networks, when we consider only positive (or negative) edges, corresponding degree distributions also follow power-laws as shown in Figures 1(c) and 1(d).



(a) Self-similar balanced structure to be simulated by Kronecker product


Properties
D1. Edge sign distribution (uniform)
D2. Signed triangle distribution (only balanced △)
D4. Degree distribution (oscillatory)
Fully Balanced Signed Network
(b) Signed network following Basic Stochastic Kronecker Signed Graph (SKSG-B)

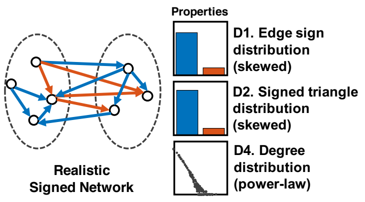

Properties
D1. Edge sign distribution (skewed)
D2. Signed triangle distribution (skewed)
D4. Degree distribution (power-law)
Realistic Signed Network
(c) Signed network following Stochastic Kronecker Signed Graph (SKSG)

Figure 2: Overview of our approach. (a) We suggest *self-similar balanced structure* observed from a real-world signed network (Section 3.1). (b) We design SKSG-B, a basic version of our model, that simulates the self-similarity using Kronecker product, and generates a fully balanced signed network (Section 3.2). (c) We then propose SKSG, an advanced version that produces realistic signed networks by introducing noise and weight splitting (Section 3.3). From SKSG model, we derive BalanSiNG which quickly generates realistic signed networks satisfying the desired properties in parallel (Section 3.4).

2.1.2 Properties without respect to signs on edges.

- **D4) Power-law degree distribution [1, 8, 9, 23, 31].** Real-world networks without signs also show power-law degree distributions as shown in Figures 1(e) and 1(f).
- **D5) Small effective diameter (hop plot) [3, 9, 27].** The hop plot shows the ratio of node pairs reachable from each other within k-hop for each integer k. It is closely related to the effective diameter, the 90 percentile distance in the hop plot. As seen in Figure 1(g), the effective diameters of real-world graphs are small (typically between 4 and 5).
- **D6) Power-law singular value distribution [4, 9, 23].** The singular values in the adjacency matrix of a real graph follow a power-law distribution as shown in Figure 1(h).

2.2 Problem Definition

Problem 1 (Signed Network Generation). *Given the target numbers* $|\mathrm{V}|$ *and* $|\mathrm{E}|$ *of nodes and edges, respectively, we aim to synthetically generate a directed signed network from scratch having* $|\mathrm{V}|$ *nodes and* $|\mathrm{E}|$ *signed edges where the output network should follow the desired properties of real-world signed networks listed in Section 2.1.* ■

2.3 SKG: Stochastic Kronecker Graph Model

SKG [23] is an unsigned network generation model based on Kronecker product. Its motivation is that power-law phenomena in nature occur due to self-similarity, i.e., a self-similar object is approximately similar to a part of itself [36]. SKG stochastically simulates a self-similarity with a tiny seed graph using Kronecker product denoted by $\otimes$ (Definition D.1 in Appendix D). Specifically, SKG creates a self-similar graph by recursively computing $\mathcal{A}^{(k)} = \mathcal{A}_{\text{seed}} \otimes \mathcal{A}^{(k-1)}$ where $\mathcal{A}^{(k)}$ is k-th Kronecker product result over the adjacency matrix $\mathcal{A}_{\text{seed}}$ of the seed graph. In SKG, (u, v)-th entry of $\mathcal{A}^{(k)}$ is the probability $P(u, v)$ that edge $u \rightarrow v$ exists in the graph. When a randomly generated value for each entry is within the probability, the corresponding edge is created. Several methods such as FastKronecker [24] and R-MAT [4] were proposed to reduce the generation time of SKG.

Although many research works [23, 37] have shown that SKG well captures various real-world properties (e.g., D4-6) in unsigned networks, the model is not proper for modeling signed networks since it does not consider how to form signs on edges. More essentially, it has not been revealed which self-similarity should be simulated when we generate signed networks through Kronecker products. Hence, our main challenge is to identify a desirable self-similarity for generating signed networks based on Kronecker product so that a resulting network establishes a solid foundation for the aforementioned properties.

Table 1: Table of symbols.

Symbol	Definition
V	set of nodes
E	set of singed edges
$\otimes$	Kronecker product
L	target recursion depth
$\mathbf{T}$	stochastic signed tensor $\mathbf{T} \in \mathbb{R}^{\lvert \mathrm{V}\rvert \times \lvert \mathrm{V}\rvert \times 2} = \{+\mathcal{P}, -\mathcal{M}\}$
$\mathbf{T}_{\text{seed}}$	$2 \times 2 \times 2$ seed stochastic signed adjacency tensor, i.e., $\mathbf{T}_{\text{seed}} = \{+\mathcal{P}_{\text{seed}}, -\mathcal{M}_{\text{seed}}\}$
$\mathbf{N}^{(l)}_{\text{seed}}$	$2 \times 2 \times 2$ seed tensor with noise at level l
$f_b(\cdot)$	balanced sign aggregator in Definition 3.2
$f_\alpha(\cdot)$	weight splitter with α in Definition 3.4
$\tilde{\mathbf{T}}^{(l)}$	l-th Kronecker product result with $f_b(\cdot)$ and $f_\alpha(\cdot)$
γ	parameter for noise
α	parameter for weight splitting
$\rho(\cdot)$	ratio of a given input
$\triangle_b$ and $\triangle_u$	balanced and unbalanced triangles, respectively
$\triangle_{+++}$	balanced triangles with three + signs
$\triangle_{++-}$	unbalanced triangles with two + signs and one − sign
$\triangle_{+--}$	balanced triangles with one + sign and two − signs
$\triangle_{---}$	unbalanced triangles with three − signs

3 PROPOSED METHOD

We propose BalanSiNG, a novel method for generating realistic signed networks following the desired properties in Section 2.1. The technical challenges and our approaches are as follows:

- Which self-similarity should be satisfied for generating signed networks (Section 3.1)? We suggest a novel self-similarity called *self-similar balanced structure* to be satisfied for generating balanced signed networks by investigating a real-world signed network.
- How can we generate signed networks following the self-similarity (Section 3.2)? We design Basic Stochastic Kronecker Signed Graph (SKSG-B), a basic model that produces a fully balanced signed network by simulating the self-similarity via Kronecker product.
- How can we generate realistic signed networks (Section 3.3)? We propose Stochastic Kronecker Signed Graph (SKSG), an advanced model introducing noise and weight splitting to SKSG-B so that the resulting network exhibits the aforementioned characteristics in Section 2.1.
- How can we efficiently generate large-scale signed networks (Section 3.4)? We derive Balanced Signed Network Generator (BalanSiNG) from SKSG, a fully parallelizable method that quickly generates signed edges.

We illustrate the overview of our approaches in Figure 2. Our main goal is to design a generation method for signed networks showing the distinct properties of real world signed networks. Among the various properties, we mainly focus on the balanced

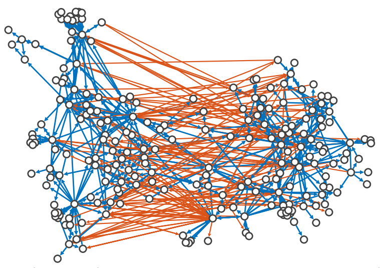


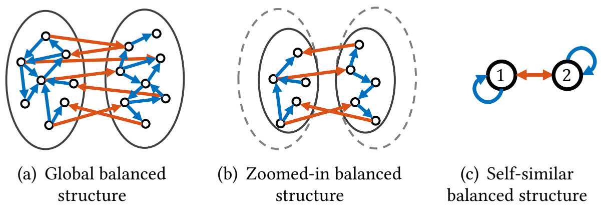
Figure 3: Balanced Structure in Congress dataset [44] where two large clusters are observed. Most nodes in each cluster are positively connected, and nodes between the clusters are negatively connected.

triangle distribution indicating *balanced signed networks* since it is one of the most distinct properties derived from signs [6, 26]. For that purpose, we first design a self-similarity to be satisfied for balanced signed networks, inspired from balanced structure in signed networks as shown in Figure 2(a).

We then propose a basic model SKSG-B and an advanced model SKSG. SKSG-B simulates the self-similarity using Kronecker product so that it produces a fully balanced signed network (i.e., there are no unbalanced triangles) as depicted in Figure 2(b). However, we observe that the fully balanced network of SKSG-B has different properties than those from real-world signed networks in terms of edge sign and balanced triangle proportions as shown in Figure 2(b). Thus, we suggest SKSG by introducing noise and weight splitting to SKSG-B so that SKSG produces realistic signed networks following the desired properties as seen in Figure 2(c). Furthermore, we develop BalanSiNG that generates balanced signed networks fully in parallel while supporting SKSG.

3.1 Self-Similarity for Signed Networks

We investigate a real-world signed network to understand its structure with signs, and then model a self-similarity behind the structure. We analyze the Congress dataset [44], a real-world signed network where nodes represent politicians, and signed edges indicate supports (i.e., positive) or oppositions (i.e., negative) between nodes. The detailed statistics of the dataset are summarized in Table 4. We visualize the signed network of the Congress dataset in Figure 3. Note that two distinct clusters appear where most nodes are mutually friends in each cluster while nodes between the clusters exhibit mutual antagonism. This structure is called *balanced signed network.* If a signed network is *fully balanced* [7], there are *two groups*; nodes in each group create only positive edges while nodes between the groups form only negative edges as in Figure 4(a). This structure is directly related to balance theory [11] since there are only balanced triangles in a fully balanced network, i.e., there are only triangles $\triangle_{+++}$ in each group and triangles $\triangle_{+--}$ between the groups.



(a) Global balanced structure (b) Zoomed-in balanced structure (c) Self-similar balanced structure

Figure 4: Self-similar pattern in balanced signed networks.

From this structure, we observe a self-similar pattern, called *self-similar balanced structure*, as illustrated in Figure 4. Figure 4(a) represents a fully balanced signed network. Then, if we zoom in the network as in Figure 4(b), a smaller but similar structure to that of Figure 4(a) appears. Note that the structure in Figure 4(b) is also balanced; hence, the balanced structure is self-similar according to the definition of self-similarity [36]. We abstract the self-similar balanced structure as shown in Figure 4(c) where each node indicates a group, and blue edges represent that positive edges are created within each group while red edges indicate that negative edges are formed between the groups.

Algorithm 1: SKSG-B

```
Input: seed tensor $\mathbf{T}_{\text{seed}} \in \mathbb{R}^{2\times2\times2}$, target recursion level $L$, and target
    number $|\text{E}|$ of edges
Output: set E of signed edges
 1: set $\tilde{\mathbf{T}}^{(1)} \leftarrow \mathbf{T}_{\text{seed}}$ and $\text{E} \leftarrow \emptyset$
 2: for $l = 2$ to $L$ do
 3:     compute $\tilde{\mathbf{T}}^{(l)} \leftarrow f_b(\tilde{\mathbf{T}}^{(1)} \otimes \tilde{\mathbf{T}}^{(l-1)})$ in Equation (4)
 4: for each $(u, v)$ such that $u, v \in \text{V}$ do
 5:     set $P(u, v, s) \leftarrow \tilde{\mathbf{T}}^{(L)}_{uvs}$ where $s \in \{+, -\}$
 6:     compute $P(u, v)$ and $P(s|u, v)$ using Equation (2)
 7:     toss a biased coin with $P(u, v)$
 8:     if head appears, i.e., $u \rightarrow v$ is formed then
 9:         $\hat{s} \leftarrow \text{argmax}_{s\in\{+,-\}} P(s|u, v)$
10:         insert a signed edge $(u \rightarrow v, \hat{s})$ into E if $|\text{E}| < m$
11: return set E of signed edges
```

3.2 SKSG-B: Basic Stochastic Kronecker Signed Graph Model

We describe our basic model SKSG-B for modeling signed networks. The main intuition of SKSG-B is to simulate the self-similarity explained in Section 3.1 using Kronecker product.

3.2.1 Formulation of SKSG-B. First of all, we define *stochastic signed tensor* used for constructing a signed network G as follows:

Definition 3.1 (Stochastic Signed Tensor). Let $|\text{V}|$ be the number of nodes. A stochastic signed tensor $\mathbf{T} \in \mathbb{R}^{|\text{V}|\times|\text{V}|\times2}$ consists of two stochastic adjacency matrices $\mathcal{P} \in \mathbb{R}^{|\text{V}|\times|\text{V}|}$ and $\mathcal{M} \in \mathbb{R}^{|\text{V}|\times|\text{V}|}$ with signs, i.e., $\mathbf{T} = \{+\mathcal{P}, -\mathcal{M}\}$ where $\mathcal{P}$ and $\mathcal{M}$ represent probabilities for positive and negative edges, respectively. ■

Then, the self-similar balanced structure in Figure 4(c) is represented as follows:

$$\mathbf{T}_{\text{seed}} = \{+\mathcal{P}_{\text{seed}}, -\mathcal{M}_{\text{seed}}\} = \left\{+\begin{bmatrix} p_{11} & 0 \\ 0 & p_{22} \end{bmatrix}, -\begin{bmatrix} 0 & m_{12} \\ m_{21} & 0 \end{bmatrix}\right\} \quad (1)$$

where + and − indicate positive and negative signs, respectively. Each entry $\mathbf{T}_{uvs}$ is a joint probability $P(u, v, s)$ where u and v are nodes, and $s \in \{+, -\}$ is a sign, e.g., $\mathbf{T}_{12-} = m_{12} = P(1, 2, -)$. The sum of all $P(u, v, s)$ is 1, i.e., $\sum_{(u,v,s)} P(u, v, s) = 1$. If we know $P(u, v, s)$, we are able to determine the creation process of edge $u \rightarrow v$ and its sign. First, we compute $P(u, v) = P(u, v, +) + P(u, v, -)$, toss a biased coin with $P(u, v)$, and determine to create the edge if the coin's head appears (line 7 in Algorithm 1). If $u \rightarrow v$ is formed, we decide its sign based on $P(s|u, v)$ as follows:

$$P(s|u, v) = \frac{P(u, v, s)}{\sum_{t\in\{+,-\}} P(u, v, t)} = \frac{P(u, v, s)}{P(u, v)} \quad (2)$$

If $P(+|u, v) > P(-|u, v)$, then its sign is determined to be positive, otherwise, it is negative (line 9 in Algorithm 1). Note that we call this approach *deterministic sign decision.*

Given a small seed signed tensor $\mathbf{T}_{\text{seed}}$, SKSG-B repeats Kronecker product multiple times over $\mathbf{T}_{\text{seed}}$. Kronecker product between two signed tensors is defined as follows:

$$\begin{aligned} \mathbf{T}^{(2)} = \mathbf{T}^{(1)} \otimes \mathbf{T}^{(1)} &= \{+\mathcal{P}, -\mathcal{M}\} \otimes \{+\mathcal{P}, -\mathcal{M}\} \\ &= \{+\mathcal{P} \otimes \mathcal{P}, -\mathcal{P} \otimes \mathcal{M}, -\mathcal{M} \otimes \mathcal{P}, +\mathcal{M} \otimes \mathcal{M}\} \end{aligned} \quad (3)$$

where $\mathbf{T}^{(k)}$ is k-th Kronecker product result on $\mathbf{T}_{\text{seed}} = \mathbf{T}^{(1)} \in \mathbb{R}^{n\times n\times2}$. Note that the dimension of $\mathbf{T}^{(2)}$ is $n^2 \times n^2 \times 2^2$ where the last dimension indicates $\{++, +-, -+, --\}$. Each entry of $\mathrm{T}^{(2)}$ indicates a joint probability $P(u, v, \{s, s\})$. However, this is not the probability that we want since we need $P(u, v, s)$ to determine the edge's sign. Hence, we aggregate the terms according to their sign using *balanced sign aggregator* $f_b(\cdot)$ defined as follows:

Definition 3.2 (Balanced Sign Aggregator). Balanced sign aggregator $f_b : \mathbb{R}^{N \times N \times 4} \rightarrow \mathbb{R}^{N \times N \times 2}$ aggregates the terms in Equation (3) according to their signs as follows:

$$\tilde{\mathrm{T}} = f_b(\mathrm{T} \otimes \mathrm{T}) = \{+(\mathcal{P} \otimes \mathcal{P} + \mathcal{M} \otimes \mathcal{M}), -(\mathcal{P} \otimes \mathcal{M} + \mathcal{M} \otimes \mathcal{P})\}$$
$$= \{+\tilde{\mathcal{P}}, -\tilde{\mathcal{M}}\}$$

where $\tilde{\mathrm{T}} \in \mathbb{R}^{N \times N \times 2}$ is a signed tensor aggregated by $f_b(\cdot)$, $\tilde{\mathcal{P}} = \mathcal{P} \otimes \mathcal{P} + \mathcal{M} \otimes \mathcal{M}$, and $\tilde{\mathcal{M}} = \mathcal{P} \otimes \mathcal{M} + \mathcal{M} \otimes \mathcal{P}$. ■

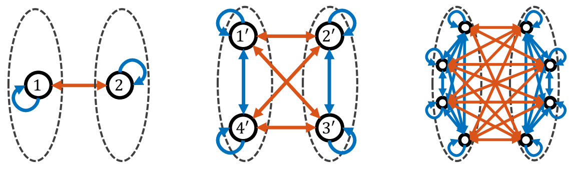
The Kronecker product result with $f_b(\cdot)$ is guaranteed to form a fully balanced signed network (see Section 3.2.2 and Lemma 3.3). Let $\tilde{\mathrm{T}}^{(l)}$ denote l-th Kronecker product result with $f_b(\cdot)$, and $\tilde{\mathrm{T}}^{(1)}$ is initially set to T_{seed} in Equation (1). Then, we generalize the Equation (3) as follows:

$$\tilde{\mathrm{T}}^{(l)} = f_b(\tilde{\mathrm{T}}^{(1)} \otimes \tilde{\mathrm{T}}^{(l-1)}) \tag{4}$$

where $\tilde{\mathrm{T}}^{(l)} \in \mathbb{R}^{n^l \times n^l \times 2}$ is used for building a signed network G given the recursion level l. Algorithm 1 summarizes SKSG-B based on Equation (4). Given T_{seed} in Equation (1), a target recursion level L, and a number $|\mathrm{E}|$ of edges, SKSG-B generates a signed network having 2^L nodes and $|\mathrm{E}|$ edges (line 3). For each pair of nodes, it decides the creation of the edge (line 7) and its sign (line 9) based on $\tilde{\mathrm{T}}^{(L)}$.

3.2.2 *Self-similar Balanced Network Simulated by SKSG-B.*

We illustrate how SKSG-B simulates the self-similarity for balanced signed networks. Given $\tilde{\mathrm{T}}^{(1)} = \mathrm{T}_{\text{seed}}$ in Equation (1), we compute $\tilde{\mathrm{T}}^{(2)}, \tilde{\mathrm{T}}^{(3)}, \cdots$ based on Equation (4). Figure 5 depicts the results of $\tilde{\mathrm{T}}^{(1)}$, $\tilde{\mathrm{T}}^{(2)}$ and $\tilde{\mathrm{T}}^{(3)}$. Note that the balanced structure is kept as level l increases, i.e., only positive edges are formed within each group (dotted ellipses), and only negative edges are allowed between the groups when we start from T_{seed} in Equation (1).



(a) $\tilde{\mathrm{T}}^{(1)}$ (b) $\tilde{\mathrm{T}}^{(2)}$ (c) $\tilde{\mathrm{T}}^{(3)}$

Figure 5: Illustrations on how SKSG-B simulates the self-similarity for balanced signed networks given $\mathrm{T}_{\text{seed}} = \tilde{\mathrm{T}}^{(1)}$.

We formalize this property of the balanced structure generated by SKSG-B in the following lemma:

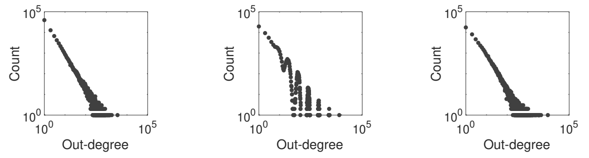
LEMMA 3.3. *Given* T_{seed} *in Equation* (1), $\tilde{\mathrm{T}}^{(l)}$ *of Equation* (4) *produces a fully balanced signed network.*

PROOF. See the detailed proof in Appendix B. □

3.3 SKSG: Exploiting Noise and Weight for Realistic Signed Networks

We propose STOCHASTIC KRONECKER SIGNED GRAPH (SKSG), an advanced model from SKSG-B for generating signed networks following the desired properties in Section 2.1. Although SKSG-B simulates a fully balanced signed network, we will observe that the network's properties deviate from those of real signed networks. We explain the issues of SKSG-B step by step, and suggest how to resolve each issue in the following subsections. The approaches of SKSG are summarized in Algorithm 2.

```
Algorithm 2: SKSG
Input: seed tensor $\mathrm{T}_{\text{seed}} \in \mathbb{R}^{2\times2\times2}$, target recursion level $L$, target
    number $|\mathrm{E}|$ of edges, noise parameter $\gamma$, and weight parameter $\alpha$
Output: set E of signed edges
 1: generate random noises $\mu^{(l)} \in [-\gamma, \gamma]$ [37], and obtain noisy seed
    tensors $\mathrm{N}^{(l)}_{\text{seed}}$ using Equation (6) with $\mathrm{T}_{\text{seed}}$ and $\mu^{(l)}$ for $1 \le l \le L$
 2: set $\tilde{\mathrm{T}}^{(1)} \leftarrow \mathrm{N}^{(1)}_{\text{seed}}$ and $\mathrm{E} \leftarrow \emptyset$
 3: for $l = 2$ to $L$ do
 4:     compute $\tilde{\mathrm{T}}^{(l)} \leftarrow f_\alpha(f_b(\mathrm{N}^{(l)}_{\text{seed}} \otimes \tilde{\mathrm{T}}^{(l-1)}))$ in Equation (7)
 5: for each $(u, v)$ such that $u, v \in \mathrm{V}$ do
 6:     set $P(u, v, s) \leftarrow \tilde{\mathrm{T}}^{(L)}_{uvs}$ for $s \in \{+, -\}$
 7:     compute $P(u, v)$ and $P(s|u, v)$ using Equation (2)
 8:     toss a biased coin with $P(u, v)$
 9:     if head appears, i.e., $u \rightarrow v$ is formed then
10:         toss a biased coin with $P(+|u, v)$
            if head appears, then $\hat{s} \leftarrow +$, otherwise, $\hat{s} \leftarrow -$
11:         insert a signed edge $(u \rightarrow v, \hat{s})$ into E if $|\mathrm{E}| < m$
12: return set E of signed edges
```



(a) Real network (b) SKSG-B without noises (c) SKSG with noises

Figure 6: Out-degree distributions of (a) the Epinions dataset, (b) a network from SKSG-B, and (c) a network from SKSG.

3.3.1 *Introducing Noise (line 1 in Algorithm 2).*

We investigate whether a degree distribution of a graph from SKSG-B follows a power-law. We focus on degree distributions regardless of edge signs (i.e., D4). For T_{seed}, we use the values of Equation (9) in Section 4.1.3. Figure 6(b) shows the out-degree distribution of SKSG-B. Note that the distribution exhibits oscillations; it is far from being monotonically decreasing unlike that of real networks as in Figure 6(a). In fact, the oscillatory behavior is a well-known issue of the standard SKG [37]. Since the edge formation of SKSG-B is equivalent to that of SKG (see the details in Appendix E), SKSG-B naturally inherits the oscillatory behavior from SKG.

Seshadhri et al. [37] analyzed the oscillatory issue of SKG, and provided a technique called Noisy SKG. For each level l, Noisy SKG defines a noise seed matrix $\mathcal{A}^{(l)}_{\text{seed}} \in \mathbb{R}^{2\times2}$ by introducing a random noise $\mu^{(l)}$ to the seed matrix $\mathcal{A}_{\text{seed}} \in \mathbb{R}^{2\times2}$. More specifically, $\mu^{(l)}$ is chosen uniformly at random in $[-\gamma, \gamma]$ for $\gamma \le \min(\frac{a_{11}+a_{22}}{2}, a_{12})$ while a_{ij} denotes (i, j)-th entry of $\mathcal{A}_{\text{seed}}$ defined as follows:

$$\mathcal{A}^{(l)}_{\text{seed}} = \begin{bmatrix} a_{11} - \frac{2\mu^{(l)} a_{11}}{a_{11}+a_{22}} & a_{12} + \mu^{(l)} \\ a_{21} + \mu^{(l)} & a_{22} - \frac{2\mu^{(l)} a_{22}}{a_{11}+a_{22}} \end{bmatrix} \tag{5}$$

Note that its entries sum to 1, and the expectation of $\mathcal{A}^{(l)}_{\text{seed}}$ is $\mathcal{A}_{\text{seed}}$. This approach introduces randomness to the degree of each node so that the fluctuation in the degree distribution is removed, which is theoretically and empirically proved in [37, 38].

In this work, we adopt this technique to our advanced model SKSG for power-law degree distributions in its signed networks. We aim to obtain a noisy seed tensor $\mathrm{N}^{(l)}_{\text{seed}}$ by adding a noise $\mu^{(l)}$ to the seed tensor $\mathrm{T}_{\text{seed}} = \{+\mathcal{P}_{\text{seed}}, -\mathcal{M}_{\text{seed}}\}$ of Equation (1) for each level l as follows (line 1 in Algorithm 2):

$$\mathbf{N}_{\text{seed}}^{(l)} = \{+\mathcal{P}_{\text{seed}}^{(l)}, -\mathcal{M}_{\text{seed}}^{(l)}\} \tag{6}$$
$$= \left\{+\begin{bmatrix} p_{11} - \frac{2\mu^{(l)}p_{11}}{p_{11}+p_{22}} & 0 \\ 0 & p_{22} - \frac{2\mu^{(l)}p_{22}}{p_{11}+p_{22}} \end{bmatrix}, -\begin{bmatrix} 0 & m_{12} + \mu^{(l)} \\ m_{21} + \mu^{(l)} & 0 \end{bmatrix}\right\}$$

where $\mu^{(l)}$ is a uniform random noise selected in $[-\gamma, \gamma]$ for $\gamma \leq \min(\frac{p_{11}+p_{22}}{2}, m_{12})$. Note that Equation (6) is derived from Equation (5) such that $\mathcal{A}_{\text{seed}}^{(l)} = \mathcal{P}_{\text{seed}}^{(l)} + \mathcal{M}_{\text{seed}}^{(l)}$, while preserving the self-similar balanced structure in Equation (1). Thus, our approach is able to model the probability of edge sign as well as the randomness of node degree while Noisy SKG with Equation (5) cannot model the probability for deciding the sign of an edge.

When generating a signed edge, we exploit $\mathbf{N}_{\text{seed}}^{(l)}$ according to level l instead of the original $\mathbf{T}_{\text{seed}}$ as in line 4 of Algorithm 2 (see Equation (7) in Section 3.3.2). Figure 6(c) depicts the out-degree distribution of SKSG using $\mathbf{N}_{\text{seed}}^{(l)}$ with $\gamma = 0.1$. The in-degree distribution of SKSG also shows the similar tendency.

3.3.2 Weight Splitting (line 4 in Algorithm 2). We analyze the properties about signs in a network of SKSG-B. As shown in Table 2, the ratio of positive edges in a network of SKSG-B is almost equal to that of negative ones, and there are only balanced triangles because SKSG-B generates fully balanced signed networks. However, real signed networks exhibit positively skewed sign and highly balanced triangle proportions (i.e., there are few unbalanced triangles) as seen in the 'BitcoinO' column of Table 2.

Table 2: Sign and balanced triangle ratios. $\rho(+)$ and $\rho(-)$ are the ratios of positive and negative edges, respectively. $\rho(\triangle_b)$ and $\rho(\triangle_u)$ are the ratios of balanced and unbalanced triangles, respectively.

	BitcoinO	SKSG-B	SKSG
$\rho(+)$	0.8999	0.5001	0.8993
$\rho(-)$	0.1001	0.4999	0.1007
$\rho(\triangle_b)$	0.8934	1.0000	0.8254
$\rho(\triangle_u)$	0.1066	0.0000	0.1746

To alleviate this issue, we suggest a weight splitting technique that increases the probabilities on generating positive signs. Note that SKSG-B produces a larger number of negative edges than expected; hence, we move a proportion of probabilities for negative signs into that for positive signs using the following function:

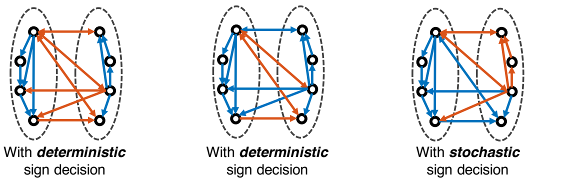
Definition 3.4 (Weight Splitter). For $0 < \alpha < 1$, weight splitter $f_\alpha : \mathbb{R}^{N\times N\times 2} \to \mathbb{R}^{N\times N\times 2}$ is defined as follows:

$$f_\alpha(\mathbf{T}) = f_\alpha(\{+\mathcal{P}, -\mathcal{M}\}) = \{+(\mathcal{P} + \alpha\mathcal{M}), -(1-\alpha)\mathcal{M}\} \quad \blacksquare$$

The function f_α increases the probabilities $\mathcal{P}$ of positive signs by $\alpha\mathcal{M}$. Equation (4) is extended with f_α and $\mathbf{N}_{\text{seed}}^{(l)}$ as follows:

$$\tilde{\mathbf{T}}^{(l)} = f_\alpha(f_b(\mathbf{N}_{\text{seed}}^{(l)} \otimes \tilde{\mathbf{T}}^{(l-1)})) \tag{7}$$

where $\tilde{\mathbf{T}}^{(l)}$ is the level-l result with f_α and f_b. The effects of f_α are that it 1) increases the proportion of positive edges, and 2) forms a few unbalanced triangles $\triangle_{++-}$ as in Figure 7(b). The reason for the latter is as follows. SKSG-B produces a fully balanced network; thus, there are two groups as in Figure 7(a). Since f_α decreases probabilities of negative sign by $\alpha\mathcal{M}$ at each level l in Equation (7), a negative edge between the groups in SKSG-B could become positive in SKSG, resulting in $\triangle_{++-}$ as in Figure 7(b). Note that after $f_\alpha(\cdot)$, probabilities for positive signs do not decrease (Definition 3.4); thus, SKSG with f_α still produces only positive edges inside a group if the edge sign is decided deterministically.


With ***deterministic*** sign decision
With ***deterministic*** sign decision
With ***stochastic*** sign decision

(a) SKSG-B without f_α (b) SKSG with f_α (c) SKSG with f_α

Figure 7: Effects of weight splitter f_α with (b) deterministic sign decision (Section 3.3.2) and (c) stochastic sign decision (Section 3.3.3).

3.3.3 Stochastic Sign Decision (line 10 in Algorithm 2). SKSG-B deterministically decides the sign of an edge based on $P(+|u,v) > P(-|u,v)$ (line 9 in Algorithm 1). However, this approach incurs a subtle issue: such decision does not produce $\triangle_{---}$ at all even though in real signed networks, there are a very few $\triangle_{---}$ as shown in Table 5. Although we introduce f_α in Section 3.3.2, SKSG with f_α does not form $\triangle_{---}$ since it generates only positive edges inside a group while for the case in Figure 7(b), the formation of $\triangle_{---}$ needs one negative edge inside a group with two negative ones between the groups. To resolve this issue, we suggest *stochastic sign decision* where SKSG stochastically decides the edge sign by tossing a biased coin with $P(+|u,v)$ (line 10 in Algorithm 2). This allows an edge to become negative with a low probability inside a group; thus, a few $\triangle_{---}$ are formed as in Figure 7(c). Based on $\tilde{\mathbf{T}}^{(L)}$ with f_α and stochastic sign decision, SKSG introduces the skewness of the sign and balanced triangle ratios similarly to those of the real network as shown in Table 2 where we use $\gamma = 0.1$, $\alpha = 0.75$, $L = 13$ and $\mathbf{T}_{\text{seed}}$ in Equation (9).

3.4 BalanSiNG: Fast and Scalable Balanced Signed Network Generator

We propose BalanSiNG, an efficient method for generating signed edges in parallel, while supporting SKSG. Algorithm 2 of SKSG is not scalable since its time and space complexities are $O(|V|^2)$, respectively, where $|V|$ is the number of nodes to be generated. The reason is that SKSG explicitly constructs signed tensor $\tilde{\mathbf{T}}^{(L)} \in \mathbb{R}^{2^L\times 2^L\times 2}$ through Kronecker product. Our main intuition to design a scalable method for the problem is to directly determine edge and track its sign probabilities without constructing $\tilde{\mathbf{T}}^{(L)}$ explicitly.

We summarize BalanSiNG in Algorithm 3. At each iteration, it exploits Generate-Edge function which determines an edge (u, v) and its sign probabilities $P(u,v,+)$ and $P(u,v,-)$ (line 4). We first explain how the function determines the edge (u, v). Intuitively, this function divides the whole region of $2^L\times 2^L$ adjacency matrix represented by $\tilde{\mathcal{P}}^{(L)} + \tilde{\mathcal{M}}^{(L)}$ of $\tilde{\mathbf{T}}^{(L)}$ into four quadrants. Then, it selects one of them with the corresponding probability, and repeats the process recursively in the chosen quadrant until the quadrant becomes a single cell where an edge is inserted.

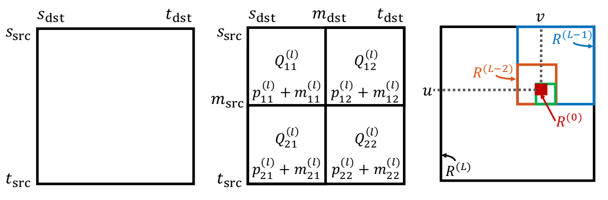
To formalize this process, we need to define *selected region* at level l of Generate-Edge as follows:

Definition 3.5 (Selected Region). $R^{(l)} = \{[s_{\text{src}}, t_{\text{src}}], [s_{\text{dst}}, t_{\text{dst}}]\}$ represents a region of an adjacency matrix, which is selected at level l, where $[s_{\text{src}}, t_{\text{src}}]$ is a range of source nodes, and $[s_{\text{dst}}, t_{\text{dst}}]$ is that of destination nodes as shown in Figure 8(a). $\blacksquare$

Suppose Generate-Edge is given $R^{(l)}$ at level l. It splits the region $R^{(l)}$ equally into four quadrants $Q_{ij}^{(l)}$ (line 9) for $1 \leq i, j \leq 2$ which are defined as follows:

Algorithm 3: BalanSiNG

```
Input: seed tensor $\mathbf{T}_{\text{seed}} \in \mathbb{R}^{2\times2\times2}$, target recursion level $L$, target
    number $|\mathrm{E}|$ of edges, noise parameter $\gamma$, and weight parameter $\alpha$
1: generate random noises $\mu^{(l)} \in [-\gamma, \gamma]$ [37], and obtain noisy seed
   tensors $\mathbf{N}^{(l)}_{\text{seed}}$ using Equation (6) with $\mathbf{T}_{\text{seed}}$ and $\mu^{(l)}$ for $1 \le l \le L$
2: parallel for $k \leftarrow 1$ to $|\mathrm{E}|$ do
3:     set $R^{(L)} \leftarrow \{[1, 2^L], [1, 2^L]\}$ as an initial region
4:     $\{+P(u,v,+), -P(u,v,-)\}$ and $(u, v) \leftarrow$ Generate-Edge($L$, $R^{(L)}$)
5:     compute $P(+|u, v)$ using Equation (2)
6:     toss a biased coin with $P(+|u, v)$
       if head appears, then $\hat{s} \leftarrow +$, otherwise, $\hat{s} \leftarrow -$
7:     write $(u \rightarrow v, \hat{s})$
8: procedure Generate-Edge(level $l$, region $R^{(l)}$)
9:     divide $R^{(l)}$ into four quadrants $Q^{(l)}_{ij}$ for $1 \le i, j \le 2$
10:    randomly select a quadrant $Q^{(l)}_{ij}$ according to probabilities
       $p^{(l)}_{ij} + m^{(l)}_{ij}$ in $\mathbf{N}^{(l)}_{\text{seed}}$ for $1 \le i, j \le 2$
11:    set $R^{(l-1)} \leftarrow Q^{(l)}_{ij}$ as a selected region for level $l-1$
12:    if $l$ is 1 then
13:        return $\{+p^{(1)}_{ij}, -m^{(1)}_{ij}\}$ and $(u, v)$ in $R^{(0)}$
14:    else
15:        $\{+\tilde{p}^{(l-1)}_{uv}, -\tilde{m}^{(l-1)}_{uv}\}$ and $(u, v) \leftarrow$ Generate-Edge($l-1$, $R^{(l-1)}$)
16:        compute $\{+\tilde{p}^{(l)}_{uv}, -\tilde{m}^{(l)}_{uv}\}$ using Equation (8)
17:        return $\{+\tilde{p}^{(l)}_{uv}, -\tilde{m}^{(l)}_{uv}\}$ and $(u, v)$
```



(a) Region $R^{(l)}$ (b) Quadrants $Q^{(l)}_{ij}$ in $R^{(l)}$ (c) Base region $R^{(0)}$

Figure 8: The concept of region and quadrants.

Definition 3.6 (Quadrants in $R^{(l)}$). Given $R^{(l)}$, let $m_{\text{src}} = s_{\text{src}} + \lfloor \frac{t_{\text{src}} - s_{\text{src}}}{2} \rfloor$ and $m_{\text{dst}} = s_{\text{dst}} + \lfloor \frac{t_{\text{dst}} - s_{\text{dst}}}{2} \rfloor$. Each quadrant $Q^{(l)}_{ij}$ is defined as in Figure 8(b) for $1 \le i, j \le 2$. ■

Then, it randomly selects a quadrant $Q^{(l)}_{ij}$ with the probability $p^{(l)}_{ij} + m^{(l)}_{ij}$ which is based on the noisy seed tensor $\mathbf{N}^{(l)}_{\text{seed}}$ (line 10). Note that $p^{(l)}_{ij} + m^{(l)}_{ij}$ indicates $P(i, j, +) + P(i, j, -) = P(i, j)$ interpreted as the probability of selecting (i, j)-th quadrant in $R^{(l)}$. For the next level $l-1$, it sets $R^{(l-1)}$ to the selected $Q^{(l)}_{ij}$ (line 11). The function recursively repeats this process for $R^{(l-1)}$ (line 15) until l becomes 1 when the selected region $R^{(0)}$ (called *base region*) is a single cell representing the edge (u, v) (line 13) as shown in Figure 8(c), after starting from the initial region $R^{(L)}$ (line 3).

The edge sign probabilities $P(u, v, +)$ and $P(u, v, -)$ are also recursively computed using the following equation (line 16):

$$\{+\tilde{p}^{(l)}_{uv}, -\tilde{m}^{(l)}_{uv}\} \leftarrow f_\alpha \left(f_b \left(\{+p^{(l)}_{ij}, -m^{(l)}_{ij}\} \otimes \{+\tilde{p}^{(l-1)}_{uv}, -\tilde{m}^{(l-1)}_{uv}\} \right) \right) \quad (8)$$

which is the entry-wise version of Equation (7) where $p^{(l)}_{ij}$ and $m^{(l)}_{ij}$ are the selected quadrant probabilities at line 10 (derivation in Lemma C.1 of Appendix C). The terms $\tilde{p}^{(l)}_{uv}$ and $\tilde{m}^{(l)}_{uv}$ denote the entries of $\tilde{\mathcal{P}}^{(l)}$ and $\tilde{\mathcal{M}}^{(l)}$ corresponding to edge (u, v), respectively, where $\tilde{\mathrm{T}}^{(l)} = \{+\tilde{\mathcal{P}}^{(l)}, -\tilde{\mathcal{M}}^{(l)}\}$. Note that each probability is scalar, i.e., $p^{(l)}_{ij}, m^{(l)}_{ij} \in \mathbb{R}^{1\times1}$; thus, $\{+p^{(l)}_{ij}, -m^{(l)}_{ij}\} \in \mathbb{R}^{1\times1\times2}$. Similarly, $\{+\tilde{p}^{(l-1)}_{uv}, -\tilde{m}^{(l-1)}_{uv}\} \in \mathbb{R}^{1\times1\times2}$. The Kronecker product result in $f_b(\cdot)$ is $\{+p^{(l)}_{ij}\tilde{p}^{(l-1)}_{uv}, -p^{(l)}_{ij}\tilde{m}^{(l-1)}_{uv}, -m^{(l)}_{ij}\tilde{p}^{(l-1)}_{uv}, +m^{(l)}_{ij}\tilde{m}^{(l-1)}_{uv}\} \in \mathbb{R}^{1\times1\times4}$ which is consistent with the input definition of $f_b(\cdot)$ when $N = 1$ (see Definition 3.2). For level $l-1$, $\{+\tilde{p}^{(l-1)}_{uv}, -\tilde{m}^{(l-1)}_{uv}\}$ is recursively computed by Generate-Edge (line 15). The final $\{+\tilde{p}^{(L)}_{uv}, -\tilde{m}^{(L)}_{uv}\}$ returned by the function is $\{+P(u,v,+), -P(u,v,-)\}$ (line 4).

Note that the generation of a signed edge of Generate-Edge is independent of the generation of other edges; thus, Algorithm 3 of BalanSiNG generates signed edges in parallel (line 2). We let Algorithm 3 call the Generate-Edge function in parallel using Apache Spark, a widely used distributed computing framework.

Table 3: BalanSiNG has the smallest time and space complexities. $|\mathrm{E}|$ and $|\mathrm{V}|$ are the number of edges and nodes, respectively, and $d_{\max}$ is the maximum node degree.

Method	Time	Space	Parallel?
IB [45]	$O(\lvert\mathrm{E}\rvert\lvert\mathrm{V}\rvert)$	$O(\lvert\mathrm{E}\rvert)$	No
Evo [29]	$O(d^3_{\max}\lvert\mathrm{E}\rvert\lvert\mathrm{V}\rvert)$	$O(\lvert\mathrm{E}\rvert)$	No
BSCL [6]	$O(d^2_{\max}\lvert\mathrm{E}\rvert + \lvert\mathrm{V}\rvert)$	$O(\lvert\mathrm{E}\rvert)$	No
BalanSiNG (proposed)	$O(\lvert\mathrm{E}\rvert \log \lvert\mathrm{V}\rvert)$	$O(\log \lvert\mathrm{V}\rvert)$	**Yes**

3.4.1 *Complexity Analysis.*

We analyze the complexities of BalanSiNG. To compare BalanSiNG with other sequential methods, we analyze the sequential complexities as follows:

Lemma 3.7 (Complexity of BalanSiNG). *The time complexity of BalanSiNG is $O(|\mathrm{E}| \log |\mathrm{V}|)$ where $|\mathrm{E}|$ and $|\mathrm{V}|$ are the number of edges and nodes, respectively. The space complexity is $O(\log |\mathrm{V}|)$.*

Proof. Let $T(L)$ be the time complexity of Generate-Edge given L; then, $T(L) = T(L-1) + O(1)$ since there is a recursive call with $L-1$ at line 15 of Algorithm 3, and other lines demand $O(1)$. Hence, it is obvious that $T(L)$ is in $O(L) = O(\log |\mathrm{V}|)$ where we set $|\mathrm{V}| = 2^L$. BalanSiNG generates $|\mathrm{E}|$ edges; thus, the total time complexity is $O(|\mathrm{E}| \log |\mathrm{V}|)$. BalanSiNG needs to have L noisy seed tensors $\mathbf{N}^{(l)}_{\text{seed}} \in \mathbb{R}^{2\times2\times2}$ where each tensor exhibits constant space complexity, i.e., $O(1)$ (line 1 in Algorithm 3). Therefore, the space complexity is $O(L) = O(\log |\mathrm{V}|)$. □

Table 3 compares signed network generation methods (see Section 4.1.2) in terms of complexities and parallelism. The time and space complexities of BalanSiNG are less than those of other sequential methods such as IB, Evo, and BSCL. Especially, these competitors require to store all generated edges in memory (i.e., they require $O(|\mathrm{E}|)$ space) since they need to retrieve the common neighbors of two nodes to determine the edge's sign between the nodes based on balance theory. On the other hand, BalanSiNG is free of such restriction; i.e., as soon as an edge is created, BalanSiNG is able to write it onto disk (line 7 of Algorithm 3).

4 EXPERIMENT

We aim to answer the following questions from experiments:

- **Q1. Properties of signed networks (Section 4.2).** Is our proposed BalanSiNG able to synthetically generate signed networks following the desired properties of real-world networks?
- **Q2. Fine-grained comparison of signed triangles (Section 4.3).** Does BalanSiNG generate graphs with realistic signed triangle distributions, compared to other methods?
- **Q3. Effects of parameters (Section 4.4).** How do weight parameter α and recursion level L of BalanSiNG affect the properties of generated networks?
- **Q4. Computational performance (Section 4.5).** How efficient is BalanSiNG for generating large-scale signed networks compared to other competitors? How does BalanSiNG scale up in terms of the number of workers and the data size on distributed machines?

4.1 Experimental Settings

We explain the detailed settings for our experiments.

4.1.1 Datasets. The datasets used for our experiments are summarized in Table 4. The BitcoinO and BitcoinA datasets [22] were extracted from online trust and directed networks served by Bitcoin Alpha and Bitcoin OTC, respectively. The Epinions dataset [10] is a directed signed network, and was scraped from Epinions, a product review site where users are able to mark their trust or distrust to others. We use the datasets to investigate their distinct properties and provide baseline statistics on signed triangle distributions in Table 5.

4.1.2 Competitors. We compare our proposed method BalanSiNG to the following competitors:

- **IB** [45]: IB (Interaction-based model) generates signed edges based on global and local interactions between nodes under ant pheromone mechanism and balance theory.
- **Evo** [29]: Evo (Evolutionary model) randomly generates signed edges, and keeps track of the number of unbalanced triangles over time. Once a node reaches a certain threshold of unbalanced triangle ratio, it randomly removes a link from the node until the threshold is not exceeded.
- **BSCL** [6]: BSCL (Balanced Signed Chung-Lu) is the-state-of-the-art model based on Transitive Chung-Lu model [35] and balance theory, which synthetically produces a signed network by imitating an input signed network.

4.1.3 Parameters. We describe the setting of the parameters for each method as follows:

- **BalanSiNG**: For the weight parameter α, we search for α on a grid between 0 and 1 by 0.05, and choose α which minimizes the absolute difference for edge signs in Equation (12) between a generated network and a real network. We set the noise parameter γ to 0.1 and the seed tensor $\mathbf{T}_{\text{seed}} = \{+\mathcal{P}_{\text{seed}}, -\mathcal{M}_{\text{seed}}\}$ to the following values:
$$\mathbf{T}_{\text{seed}} = \left\{+\begin{bmatrix}0.57 & 0\\ 0 & 0.05\end{bmatrix}, -\begin{bmatrix}0 & 0.19\\ 0.19 & 0\end{bmatrix}\right\} \quad (9)$$
which are derived from $\mathcal{A}_{\text{seed}} = \begin{bmatrix}0.57 & 0.19\\ 0.19 & 0.05\end{bmatrix}$. Many researches [30, 31, 37, 38] have empirically proved that these values produce monotonically decreasing power-law degree distributions. Note that other values of γ and $\mathbf{T}_{\text{seed}}$ can be used as well.
- **IB**: M_G and M_L are the numbers of edges added globally and locally, respectively. p_G and p_L are the probabilities of the positive sign of the globally and locally added edges, respectively. δ is the initial weight of an added edge. ϵ is the parameter for the evaporation. According to their work [45], we set $M_G = M_L$ to 1 and $p_G = p_L$ to $\rho(+)$ for each dataset in Table 4. For δ and ϵ, we perform grid searches from 0 to 1.0 by 0.05 to minimize the absolute difference for edge signs in Equation (12).
- **Evo**: In Evo, α is a friendliness index affecting the formation of the positive sign of an edge, and β is a tolerance threshold for unbalanced triangles. For α and β, we perform grid searches from -1 to 1 by 0.05 to minimize the absolute difference for edge signs in Equation (12).
- **BSCL**: ρ_{BSCL} is a parameter for closing wedge, α_{BSCL} is for creating positive edge, and β_{BSCL} is for closing balanced triangle. Given a real network, those parameters are approximately tuned by the estimation phase of BSCL.

Table 4: Dataset statistics. |V| is the number of nodes, |E| is the number of edges, and $\rho(+)$ is the ratio of positive edges.

Dataset	|V|	|E|	$\rho(+)$	Description
Epinions[1]	131,828	841,372	0.85	Online social network
BitcoinO[2]	5,881	35,592	0.89	Bitcoin social network
BitcoinA[2]	3,783	24,186	0.93	Bitcoin social network
Congress[3]	219	764	0.78	Politician network

[1] http://www.trustlet.org/wiki/Extended_Epinions_dataset
[2] https://snap.stanford.edu/data/soc-sign-bitcoin-otc.html
[3] http://www.cs.cornell.edu/home/llee/data/convote.html

4.1.4 Machines and Implementation. We describe the settings of machines and implementation used for evaluating the computational performance of each method in Section 4.5 as follows:

- **Setting on single machine (Section 4.5.1).** We use a single thread in a machine with an Intel Xeon E3-1240v5 CPU and 32GB RAM, and implement all tested methods including BalanSiNG based on g++ v5.4.0.
- **Setting on distributed machines (Section 4.5.2).** We implement BalanSiNG on Spark to test the scalability on a cluster (managed by Hadoop YARN) that consists of 17 machines: a master and 16 worker nodes. Each worker node has 4 physical cores (Intel Xeon E3-1240v5 CPU) with 32GB RAM, and can run 4 workers. Java v1.8.0, Scala v2.11.8, Hadoop v2.7.3, and Spark v2.11.9 are used.

4.2 Properties of Signed Networks (Q1)

We compare real-world signed network BitcoinO with those generated by BalanSiNG and competitors in Figure 1 to investigate if they exhibit the desired properties of real-world signed networks listed in Section 2.1. We omit the comparisons for other datasets due to the space limit, but the overall tendency is similar. We adjust the parameters of each method so that the resulting networks have almost the same positive edge sign ratios as that of BitcoinO (details in Appendix F); thus, the sign distributions in Figure 1(a) are similar for all graphs.

The signed network generated by BalanSiNG follows the desired properties w.r.t. signs (D1-3) as well as those regardless of signs (D4-6). The balanced triangle distribution is highly skewed as shown in Figure 1(b), and degree distributions follow a power-law as seen from Figure 1(c) to Figure 1(f). The hop plot of BalanSiNG in Figure 1(g) is similar to that of BitcoinO. Also, top-k singular values of graphs from BalanSiNG and BitcoinO monotonically decrease as shown in Figure 1(h).

On the other hand, the signed networks generated by IB and Evo do not follow power-law degree distributions as shown in the third and forth rows (Figure 1(c) to Figure 1(f)). The main reason is that IB and Evo naively create random edges without the consideration of power-law degree distribution. The hop plot and singular value distributions of both methods are also different from those of the real-world network as shown in Figures 1(g) and 1(h). BSCL generates signed networks obeying most of the desired properties, but its balanced triangle distribution (D2) does not; it is not skewed enough compared to the real network (at the first row) and BalanSiNG (at the second row) as shown in Figure 1(b). We further provide the fine-grained comparison about these signed triangles in Section 4.3.

4.3 Fine-grained Comparison of Signed Triangles (Q2)

We compare BalanSiNG to other competitors in terms of signed triangle distribution. As described in Section 2.1, the balanced

Table 5: Comparison of signed triangle distributions by BalanSiNG and competitors. $\rho(\triangle_b)$ and $\rho(\triangle_u)$ indicate the ratios of balanced and unbalanced triangles, respectively. $\rho(\triangle_{+++})$, $\rho(\triangle_{+--})$, $\rho(\triangle_{++-})$, and $\rho(\triangle_{---})$ denote the ratios of the triangle types $\triangle_{+++}$, $\triangle_{+--}$, $\triangle_{++-}$, and $\triangle_{---}$, respectively. Note that BalanSiNG (marked †) generates the most closest signed networks to the corresponding real-world signed networks in terms of absolute difference and Kolmogorov–Smirnov statistic (the lower the better).

Datasets	BitcoinA					BitcoinO					Epinions				
Methods	Real	BalanSiNG†	IB	Evo	BSCL	Real	BalanSiNG†	IB	Evo	BSCL	Real	BalanSiNG†	IB	Evo	BSCL
$\rho(\triangle_b)$	0.8805	0.8740	0.7604	0.8184	0.8366	0.8934	0.8254	0.7023	0.7402	0.7579	0.9240	0.8109	0.7061	0.7025	0.7104
$\rho(\triangle_u)$	0.1195	0.1260	0.2396	0.1816	0.1634	0.1066	0.1746	0.2977	0.2598	0.2422	0.0760	0.1891	0.2939	0.2975	0.2896
Abs. Diff.	-	**0.0130**	0.2402	0.1242	0.0879	-	**0.1360**	0.3822	0.3064	0.2711	-	**0.2261**	0.4358	0.4430	0.4272
K-S Stat.	-	**0.0065**	0.1201	0.0621	0.0439	-	**0.0680**	0.1911	0.1532	0.1356	-	**0.1131**	0.2179	0.2215	0.2136
$\rho(\triangle_{+++})$	0.8413	0.8632	0.7285	0.7954	0.8240	0.8260	0.8014	0.6649	0.6975	0.7281	0.8723	0.7782	0.6688	0.6297	0.6677
$\rho(\triangle_{+--})$	0.0393	0.0108	0.0319	0.0231	0.0126	0.0675	0.0240	0.0374	0.0427	0.0298	0.0517	0.0328	0.0373	0.0728	0.0427
$\rho(\triangle_{++-})$	0.1166	0.1259	0.2377	0.1816	0.1630	0.1026	0.1743	0.2945	0.2598	0.2408	0.0694	0.1886	0.2913	0.2975	0.2875
$\rho(\triangle_{---})$	0.0028	0.0001	0.0009	0.0000	0.0005	0.0040	0.0003	0.0032	0.0000	0.0014	0.0066	0.0005	0.0026	0.0000	0.0021
Abs. Diff.	-	**0.0625**	0.2432	0.1299	0.0927	-	**0.1434**	0.3839	0.3145	0.2764	-	**0.2383**	0.4438	0.4984	0.4362
K-S Stat.	-	**0.0219**	0.1202	0.0621	0.0441	-	**0.0681**	0.1912	0.1533	0.1357	-	**0.1131**	0.2179	0.2426	0.2136

triangle proportion is the most distinct property in real-world signed networks. Hence, we analyze signed triangles of generated signed networks to check if they exhibit distributions similar to that of a real-world signed network.

For the purpose, we first enumerate directed signed triangles in each real signed network as in [43] since all of the signed networks used in this paper are directed, and then measure the ratio $\rho(\cdot)$ for each triangle type. For example, $\triangle_{+++}$ indicates triangles with three positive signs; thus, $\rho(\triangle_{+++}) = |\triangle_{+++}|/|\triangle_{\text{total}}|$ where $|\triangle_{\text{total}}|$ is the total number of triangles. For large-scale signed networks, distributed algorithms [33] can be used to enumerate triangles. Then, we generate synthetic signed networks for each dataset following the procedure in Section 4.2 (see the parameters of each method in Appendix F), and compare the triangle distributions of both real and synthetic networks. To measure the distance between two distributions, we utilize *Absolute Difference* [6] and *Kolmogorov–Smirnov statistic* (K-S statistic) metrics. The absolute difference [6] is defined as the sum of absolute differences between each ratio of real and synthetic triangles (see Definition D.2). The K-S statistic is defined as the maximum gap between the two cumulative distributions; it has been traditionally used for measuring the difference between two distributions. We repeat the above procedure 10 times, and report the average for each method and each dataset. For both of the metrics, small values indicate that the synthetic network has a similar tendency to the corresponding real network in terms of signed triangles.

Table 5 shows the fine-grained comparison on the four types of signed triangles by BalanSiNG and competitors for each dataset. Note that BalanSiNG gives the best signed triangle distribution, showing the smallest absolute difference and K-S statistic. Specifically, BalanSiNG shows about 1.5 $\sim$ 2× better performance than the second best method for each dataset.

4.4 Effect of Parameters (Q3)

We investigate the effect of parameters of BalanSiNG. We focus on the effects of weight parameter α and target recursion level L while noise parameter γ is set to 0.1 and seed ten sor $\mathbf{T}_{\text{seed}}$ is set to the values of Equation (9), as described in Section 4.1. The weight parameter α is introduced to increase the probability of generating positive edges, and the level L controls the size of networks to be generated.

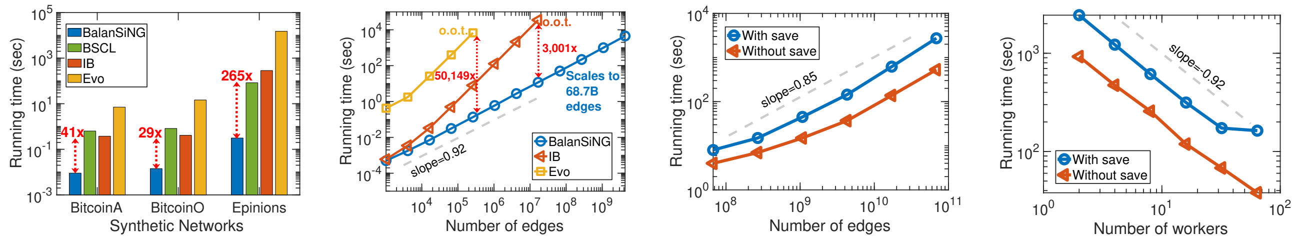
Figure 10(a) shows the effect of the weight parameter α on positive sign ratio $\rho(+)$ and balanced triangle ratio $\rho(\triangle_b)$. We set $L = 17$ and $|\mathrm{E}| = 2^{19}$, and vary α form 0.1 to 0.9. As shown in the figure, both of the ratios $\rho(+)$ and $\rho(\triangle_b)$ increase as α increases. The reason is that according to Definition 3.4, as we increase α, the probability of the positive term becomes large while that of the negative term diminishes. Also, as the number of positive edges increases, balanced triangles $\triangle_{+++}$ and $\triangle_{++-}$ are more likely to be formed. Note that α between 0.7 and 0.85 introduces the skewness of both ratios similarly to those of real signed networks. Thus, our method is able to control the skewness of those ratios according to users' preference through adjusting α.

Figures 10(b) and 10(c) demonstrate the effect of the recursion level L on the ratios $\rho(+)$ and $\rho(\triangle_b)$, and the out-degree distributions of networks generated by BalanSiNG. We set α to 0.8, and vary L from 12 to 21 to generate networks with $|\mathrm{V}| = 2^{L+1}$ and $|\mathrm{E}| = 2^{L+6}$. In Figure 10(b), $\rho(+)$ and $\rho(\triangle_b)$ do not change much as L increases. Also, as shown in Figure 10(c), the degree distributions for different L have almost the same tendency w.r.t. power-law distribution. These results indicate the effect of L on those ratios and degree distribution is marginal, i.e., our method is able to control the size of signed networks to be generated while the tendency of such properties is preserved.

4.5 Computational Performance (Q4)

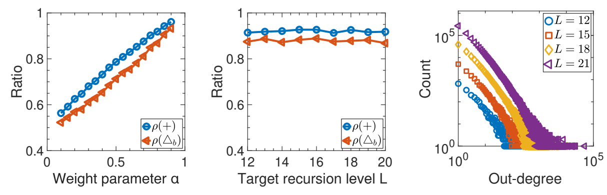
We evaluate the computational performance of BalanSiNG on single and distributed machines.

4.5.1 Performance on Single Machine. We examine the performance of BalanSiNG and competitors on a single machine. The detailed setting is in Section 4.1.4. We fix the size of each synthetic network to that of the corresponding real-world network, and compare the generation time of each method. As shown in Figure 9(a), the generation time of BalanSiNG is up to 265× faster than that of BSCL. Figure 9(b) shows the data scalability of methods. Note that BSCL is excluded since it cannot generate synthetic networks having arbitrary numbers of nodes and edges. BSCL aims to imitate an input network, and thus the size of the generated network of BSCL is fixed to that of the input network. We vary $|V| = 2^{L+1}$ and $|E| = 2^{L+6}$ for $L = 4..26$ where L is the target recursion level. We report out of time (o.o.t.) error when the generation time is more than 24 hours. As shown in Figure 9(b), only BalanSiNG generates the largest network for $L = 26$ within the limited time while IB and Evo generate o.o.t. errors. BalanSiNG is 50, 149× and 3, 001× faster than Evo and IV, respectively. Furthermore, the slope of BalanSiNG is 0.92, indicating the data scalability of BalanSiNG is near linear w.r.t. the number of edges. To sum, BalanSiNG provides the fastest running time and the best scalability.



(a) Generation time on a single machine (b) Data scalability on a single machine (c) Data scalability on distributed machines (d) Distributed machine scalability

Figure 9: Computational performance of BalanSiNG. (a) BalanSiNG generates signed networks up to 265× faster than existing methods. (b-c) The data scalability of BalanSiNG is near linear w.r.t. the number of edges on both single and distributed machines where o.o.t. stands for out of time (more than 24 hours). (d) BalanSiNG also scales up well with the increase of the number of workers on distributed machines.



(a) Effect of weight parameter α (b) Effect of level L on $\rho(+)$ and $\rho(\triangle_b)$ (c) Effect of level L on out-degree distributions

Figure 10: Effects of weight parameter α, and target recursion level L. BalanSiNG generates graphs of various sizes following the power laws, while controlling the positive sign ratio $\rho(+)$ and the balanced triangle ratio $\rho(\triangle_b)$.

4.5.2 Performance on Distributed Machines. We demonstrate the performance of BalanSiNG on distributed machines. The detailed setting is in Section 4.1.4. We report the generation times *with* and *without* writing edges onto disks (line 7 of Algorithm 3). The former is execution time with disk I/O, and the latter is only CPU execution time without disk I/O. To evaluate data scalability, we use 64 workers, and vary $|V| = 2^{L+1}$ and $|E| = 2^{L+6}$ for $L = 20..30$ where L is the target level. Figure 9(c) shows that BalanSiNG has near linear scalability w.r.t. the number of edges with the slope 0.85 in the plot. Note that BalanSiNG generates $|E| = 2^{36} \simeq 68.7$ billion signed edges within 45.5 minutes including disk I/O time on the distributed machines; the generated network is 81,675× larger than the Epinions dataset, the largest real signed network currently open to the public, with respect to the number of edges. Figure 9(d) shows BalanSiNG also scales up well with the increase of the number of workers from 2 to 64 where we set $|V| = 2^{27}$ and $|E| = 2^{32}$. The last point of the blue line at 64 is due to the bottleneck of HDFS I/O, i.e., there are too many workers trying to write edges to disks at the same time.

5 RELATED WORK

Models for generating graphs from scratch. There are various methods for generating unsigned networks following real-world properties described in Section 2.1.2. Barabási et al. [1] proposed Barabási-Albert model through a preferential attachment process for generating scale-free networks. Leskovec et al. [27] identified densification laws and shrinking diameters inherent in graphs over time, and developed Forest Fire for modeling such graphs. Also, they proposed Stochastic Kronecker Graph (SKG) [23], a general generation model that simulates a self-similarity using Kronecker product. They developed FastKronecker [24] that chooses edges in a recursive way to reduce the generation time. However, those models cannot generate signed networks, while BalanSiNG generates signed networks following real-world properties. There are a few methods for generating signed networks from scratch. Vukašinović et al. [45] proposed an interaction based model (IB) using ant pheromone mechanism and balance theory for simulating signed edge generation. Ludwig et al. [29] suggested an evolutionary model (Evo) that simulates an evolving network by inserting or removing signed edges so that the network keeps obeying balance theory. However, their resulting networks give different properties from those of the real-world signed networks, while BalanSiNG generates realistic signed networks as shown in Figure 1.

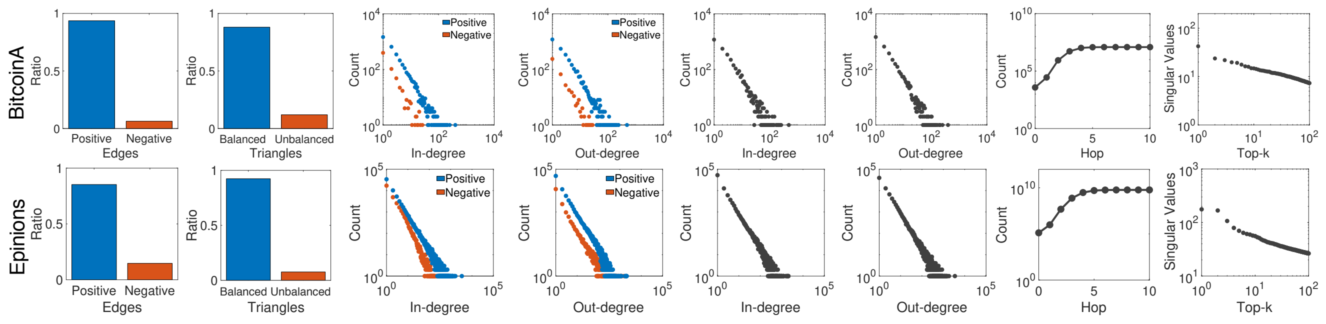
Models for generating graphs imitating an input network. Chung-Lu [35] model aims to generate a synthetic unsigned network by randomly selecting an edge with its associated degree probability. Transitive Chung-LU (TCL) model [35] stochastically performs a two-hop random walk from a node in order to explicitly form at least one triangle, thereby imitating clustering coefficients in the input graph. Derr et al. [6] proposed Balanced Signed Chung-LU (BSCL) model, the state-of-the-art model for synthetic signed networks. They combined balance theory and TCL model in order that the resulting network imitates the signed triangle distribution of the input graph. However, BSCL is not fast, does not generate networks which fully follow the properties of real-world signed networks, and cannot generate signed networks having an arbitrary number of nodes from scratch. On the other hand, BalanSiNG is fast and scalable, generates the most similar networks to real signed networks as in Table 5, and generates graphs of arbitrary sizes as in Figure 9.

6 CONCLUSION

We propose BalanSiNG, a novel, scalable, and fully parallelizable method for generating realistic signed networks from scratch. BalanSiNG exploits the self-similar balanced structure with Kronecker product, and produces realistic signed networks by introducing noises and weights. We implement BalanSiNG in parallel using Spark, a widely used distributed computing platform. Experiments show that BalanSiNG generates the most realistic signed networks. BalanSiNG is up to 265× faster than existing methods for generating signed networks, and scales up near linearly with the size of networks and the number of workers on both single and distributed machines, successfully generating graphs with 68.7 billion edges.

ACKNOWLEDGMENTS

This work was supported by Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) [2013-0-00179, Development of Core Technology for Context-aware Deep-Symbolic Hybrid Learning and Construction of Language Resources]. The Institute of Engineering Research at Seoul National University provided research facilities for this work. The ICT at Seoul National University provides research facilities for this study. U Kang is the corresponding author.



(a) Global sign proportion (b) Balanced triangle distribution (c) Signed in-degree distribution (d) Signed out-degree distribution (e) In-degree distribution (f) Out-degree distribution (g) Hop plot (h) Singular value distribution

Figure 11: Various properties of real-world signed networks. (a)-(d) illustrate properties derived from edge signs, and (e)-(h) depict traditional properties of real-world networks regardless of edge signs (see Section 2.1).

REFERENCES

[1] Réka Albert and Albert-László Barabási. 2002. Statistical mechanics of complex networks. *Rev. Mod. Phys.* 74 (Jan 2002), 47–97. Issue 1.

[2] Dorwin Cartwright and Frank Harary. 1956. Structural balance: a generalization of Heider's theory. *Psychological review* 63, 5 (1956), 277.

[3] Deepayan Chakrabarti and Christos Faloutsos. 2006. Graph mining: Laws, generators, and algorithms. *ACM computing surveys* 38, 1 (2006), 2.

[4] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *SDM*. SIAM.

[5] Lingyang Chu, Zhefeng Wang, Jian Pei, Jiannan Wang, Zijin Zhao, and Enhong Chen. 2016. Finding gangs in war from signed networks. In *KDD*. ACM.

[6] Tyler Derr, Charu Aggarwal, and Jiliang Tang. 2018. Signed Network Modeling Based on Structural Balance Theory. In *CIKM*. ACM.

[7] David Easley, Jon Kleinberg, et al. 2012. Networks, crowds, and markets: Reasoning about a highly connected world. *Significance* 9 (2012), 43–44.

[8] Victor M Eguiluz, Dante R Chialvo, Guillermo A Cecchi, Marwan Baliki, and A Vania Apkarian. 2005. Scale-free brain functional networks. *Physical review letters* 94, 1 (2005), 018102.

[9] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On power-law relationships of the internet topology. In *SIGCOMM*, Vol. 29. ACM.

[10] Ramanthan Guha, Ravi Kumar, Prabhakar Raghavan, and Andrew Tomkins. 2004. Propagation of trust and distrust. In *WWW*. ACM.

[11] Paul W Holland and Samuel Leinhardt. 1971. Transitivity in structural models of small groups. *Comparative group studies* 2, 2 (1971), 107–124.

[12] ByungSoo Jeon, Inah Jeon, and U Kang. 2015. TeGViz: Distributed Tera-Scale Graph Generation and Visualization. In *IEEE International Conference on Data Mining Workshop, ICDMW 2015, Atlantic City, NJ, USA, November 14-17, 2015*.

[13] ByungSoo Jeon, JungWoo Lee, and U Kang. 2016. TeT: Distributed Tera-Scale Tensor Generator. *Journal of KIISE* 43, 8 (2016), 910–918.

[14] Woojeong Jin, Jinhong Jung, and U Kang. 2019. Supervised and extended restart in random walks for ranking and link prediction in networks. *PLOS ONE* 14, 3 (03 2019), 1–23.

[15] Jinhong Jung, Woojeong Jin, and U Kang. 2019. Random walk-based ranking in signed social networks: model and algorithms. *Knowledge and Information Systems* (2019).

[16] Jinhong Jung, Woojeong Jin, Lee Sael, and U Kang. 2016. Personalized Ranking in Signed Networks Using Signed Random Walk with Restart.. In *ICDM*. IEEE.

[17] Jinhong Jung, Namyong Park, Lee Sael, and U Kang. 2017. BePI: Fast and Memory-Efficient Method for Billion-Scale Random Walk with Restart. In *SIGMOD*. ACM.

[18] Jinhong Jung, Kijung Shin, Lee Sael, and U Kang. 2016. Random walk with restart on large graphs using block elimination. *ACM Transactions on Database Systems (TODS)* 41, 2 (2016), 12.

[19] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. 2009. PEGASUS: A Peta-Scale Graph Mining System. In *ICDM*.

[20] Junghwan Kim, Haekyu Park, Ji-Eun Lee, and U Kang. 2018. SIDE: Representation Learning in Signed Directed Networks. In *WWW*. ACM.

[21] Srijan Kumar, Francesca Spezzano, and VS Subrahmanian. 2014. Accurately detecting trolls in slashdot zoo via decluttering. In *ASONAM*. IEEE.

[22] Srijan Kumar, Francesca Spezzano, VS Subrahmanian, and Christos Faloutsos. 2016. Edge Weight Prediction in Weighted Signed Networks.. In *ICDM*. IEEE.

[23] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. 2005. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *ECML/PKDD*. Springer.

[24] Jure Leskovec, Deepayan Chakrabarti, Jon M. Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker Graphs: An Approach to Modeling Networks. *JMLR* 11 (2010), 985–1042.

[25] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Predicting positive and negative links in online social networks. In *WWW*. ACM.

[26] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Signed networks in social media. In *CHI*. ACM.

[27] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*.

[28] Xiaoming Li, Hui Fang, and Jie Zhang. 2019. Supervised User Ranking in Signed Social Networks. In *AAAI*, Vol. 33. 184–191.

[29] Mark Ludwig and Peter Abell. 2007. An evolutionary model of social networks. *The European Physical Journal B* 58, 1 (2007), 97–105.

[30] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[31] Himchan Park and Min-Soo Kim. 2017. TrillionG: A trillion-scale synthetic graph generator using a recursive vector model. In *SIGMOD*. ACM.

[32] Ha-Myung Park, Namyong Park, Sung-Hyon Myaeng, and U Kang. 2016. Partition Aware Connected Component Computation in Distributed Systems. In *ICDM*.

[33] Ha-Myung Park, Sung-Hyon Myaeng, and U Kang. 2016. Pte: Enumerating trillion triangles on distributed systems. In *KDD*. ACM.

[34] Ha-Myung Park, Chiwan Park, and U Kang. 2018. PegasusN: A Scalable and Versatile Graph Mining System. In *AAAI*.

[35] Joseph J Pfeiffer, Timothy La Fond, Sebastian Moreno, and Jennifer Neville. 2012. Fast generation of large scale social networks while incorporating transitive closures. In *PASSAT*. IEEE, 154–165.

[36] Manfred Schroeder. 2009. *Fractals, chaos, power laws: Minutes from an infinite paradise*. Courier Corporation.

[37] Comandur Seshadhri, Ali Pinar, and Tamara G Kolda. 2011. An in-depth study of stochastic Kronecker graphs. In *ICDM*. IEEE.

[38] Comandur Seshadhri, Ali Pinar, and Tamara G Kolda. 2013. An in-depth analysis of stochastic Kronecker graphs. *J. ACM* 60, 2 (2013), 13.

[39] Kijung Shin, Jinhong Jung, Lee Sael, and U Kang. 2015. BEAR: Block Elimination Approach for Random Walk with Restart on Large Graphs. In *SIGMOD*.

[40] Dongjin Song, David A Meyer, and Dacheng Tao. 2015. Efficient latent link recommendation in signed networks. In *KDD*. ACM.

[41] Michael Szell, Renaud Lambiotte, and Stefan Thurner. 2010. Multirelational organization of large-scale social networks in an online world. *Proceedings of the National Academy of Sciences* 107, 31 (2010), 13636–13641.

[42] Jiliang Tang, Charu Aggarwal, and Huan Liu. 2016. Node classification in signed social networks. In *SDM*. SIAM.

[43] Jiliang Tang, Yi Chang, Charu Aggarwal, and Huan Liu. 2016. A survey of signed network mining in social media. *Comput. Surveys* 49, 3 (2016), 42.

[44] Matt Thomas, Bo Pang, and Lillian Lee. 2006. Get out the vote: Determining support or opposition from Congressional floor-debate transcripts. In *EMNLP*. Association for Computational Linguistics.

[45] Vida Vukašinović, Jurij Šilc, and Risth Škrekovski. 2014. Modeling acquaintance networks based on balance theory. *International Journal of Applied Mathematics and Computer Science* 24, 3 (2014), 683–696.

[46] Suhang Wang, Jiliang Tang, Charu Aggarwal, Yi Chang, and Huan Liu. 2017. Signed network embedding in social media. In *SDM*. SIAM, 327–335.

[47] Pinghua Xu, Wenbin Hu, Jia Wu, and Bo Du. 2019. Link Prediction with Signed Latent Factors in Signed Social Networks. In *SIGKDD*. ACM, 1046–1054.

[48] Bo Yang, William Cheung, and Jiming Liu. 2007. Community mining from signed social networks. *TKDE* 19, 10 (2007), 1333–1348.

APPENDIX

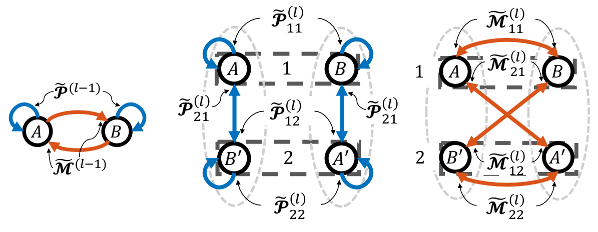
A PROPERTIES OF SIGNED NETWORKS

Figure 11 shows properties of other real-world signed networks. The properties of the BitcoinO dataset are depicted in Figure 1.

B PROOF OF LEMMA 3.3

PROOF. We use mathematical induction. For the base case, $\tilde{\mathrm{T}}^{(1)} = \mathrm{T}_{\text{seed}}$ is trivially fully balanced as shown in Figure 5(a). Assume $\tilde{\mathrm{T}}^{(l-1)}$ is fully balanced. Then, $\tilde{\mathrm{T}}^{(l)}$ of Equation (4) with $\mathrm{T}_{\text{seed}} = \tilde{\mathrm{T}}^{(1)}$ is represented as follows:

$$f_b(\tilde{\mathrm{T}}^{(1)} \otimes \tilde{\mathrm{T}}^{(l-1)})$$
$$= \{+(\mathcal{P} \otimes \tilde{\mathcal{P}}^{(l-1)} + \mathcal{M} \otimes \tilde{\mathcal{M}}^{(l-1)}), -(\mathcal{P} \otimes \tilde{\mathcal{M}}^{(l-1)} + \mathcal{M} \otimes \tilde{\mathcal{P}}^{(l-1)})\}$$



$\tilde{\mathcal{P}}^{(l-1)}$
$\tilde{\mathcal{M}}^{(l-1)}$
A
B
$\tilde{\mathcal{P}}^{(l)}_{11}$
$\tilde{\mathcal{P}}^{(l)}_{21}$
$\tilde{\mathcal{P}}^{(l)}_{12}$
$\tilde{\mathcal{P}}^{(l)}_{22}$
1
2
A
B
B′
A′
$\tilde{\mathcal{M}}^{(l)}_{11}$
$\tilde{\mathcal{M}}^{(l)}_{21}$
$\tilde{\mathcal{M}}^{(l)}_{12}$
$\tilde{\mathcal{M}}^{(l)}_{22}$

(a) $\{+\tilde{\mathcal{P}}^{(l-1)}, -\tilde{\mathcal{M}}^{(l-1)}\}$ (b) $\tilde{\mathcal{P}}^{(l)}$ (c) $\tilde{\mathcal{M}}^{(l)}$

Figure 12: Structures of (a) $\tilde{\mathcal{P}}^{(l-1)}$&$\tilde{\mathcal{M}}^{(l-1)}$, (b) $\tilde{\mathcal{P}}^{(l)}$, & (c) $\tilde{\mathcal{M}}^{(l)}$.

$$= \left\{ + \begin{bmatrix} p_{11}\tilde{\mathcal{P}}^{(l-1)} & m_{12}\tilde{\mathcal{M}}^{(l-1)} \\ m_{21}\tilde{\mathcal{M}}^{(l-1)} & p_{22}\tilde{\mathcal{P}}^{(l-1)} \end{bmatrix}, - \begin{bmatrix} p_{11}\tilde{\mathcal{M}}^{(l-1)} & m_{12}\tilde{\mathcal{P}}^{(l-1)} \\ m_{21}\tilde{\mathcal{P}}^{(l-1)} & p_{22}\tilde{\mathcal{M}}^{(l-1)} \end{bmatrix} \right\}$$

$$= \left\{ + \begin{bmatrix} \tilde{\mathcal{P}}^{(l)}_{11} & \tilde{\mathcal{P}}^{(l)}_{12} \\ \tilde{\mathcal{P}}^{(l)}_{21} & \tilde{\mathcal{P}}^{(l)}_{22} \end{bmatrix}, - \begin{bmatrix} \tilde{\mathcal{M}}^{(l)}_{11} & \tilde{\mathcal{M}}^{(l)}_{12} \\ \tilde{\mathcal{M}}^{(l)}_{21} & \tilde{\mathcal{M}}^{(l)}_{22} \end{bmatrix} \right\} = \left\{ +\tilde{\mathcal{P}}^{(l)}, -\tilde{\mathcal{M}}^{(l)} \right\}$$

Figure 12(a) shows $\tilde{\mathrm{T}}^{(l-1)}$ where $\tilde{\mathcal{P}}^{(l-1)}$ represents edges within each group (A and B), and $\tilde{\mathcal{M}}^{(l-1)}$ represents edges between the two groups. We depict the block structure of $\tilde{\mathcal{P}}^{(l)}$ in Figure 12(b) where $\tilde{\mathcal{P}}^{(l)}_{ij}$ indicates (i, j)-th block of $\tilde{\mathcal{P}}^{(l)}$. Figure 12(b) has two copies: (A, B) of 1st copy and (A', B') of 2nd copy. Then, $\tilde{\mathcal{P}}^{(l)}_{11}$ means edges within A and B of 1st copy because they are from $p_{11}\tilde{\mathcal{P}}^{(l-1)} = \tilde{\mathcal{P}}^{(l)}_{11}$. Also, $\tilde{\mathcal{P}}^{(l)}_{12}$ represents directed edges from A to B', and from B to A' by $m_{12}\tilde{\mathcal{M}}^{(l-1)}$. Other blocks in $\tilde{\mathcal{P}}^{(l)}$ are similarly interpreted; thus, there are two groups (A, B') and (A', B) having positive between-group edges in the graph of $\tilde{\mathcal{P}}^{(l)}$. Each block in $\tilde{\mathcal{M}}^{(l)}$ represents edges between the groups as shown in Figure 12(c). These indicate $\tilde{\mathrm{T}}^{(l)}$ is also fully balanced. Hence, $\tilde{\mathrm{T}}^{(l)}$ is fully balanced for any $l \geq 1$. □

C LEMMA OF ENTRY-WISE RECURSIVE REPRESENTATION OF BALANSING

Lemma C.1. *Let $R^{(l)}$ be the selected region at level l with probability $p^{(l)}_{ij} + m^{(l)}_{ij}$ in* Generate-Edge. *Let (u, v) be decided through $R^{(L)}, \cdots, R^{(0)}$. Equation* (7) *for (u, v) is equivalent to Equation* (8).

Proof. Equation (7) is represented as follows:

$$\tilde{\mathrm{T}}^{(l)} = f_\alpha(f_b(\mathrm{N}^{(l)}_{\text{seed}} \otimes \tilde{\mathrm{T}}^{(l-1)})) \Leftrightarrow \tag{10}$$

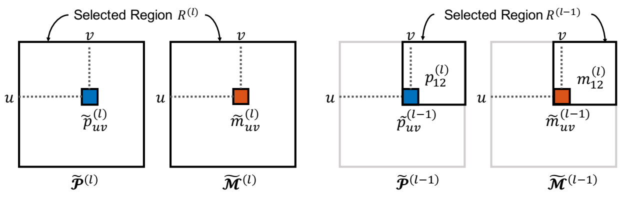
$$\{+\tilde{\mathcal{P}}^{(l)}, -\tilde{\mathcal{M}}^{(l)}\} = f_\alpha(f_b(\{+\mathcal{P}^{(l)}_{\text{seed}}, -\mathcal{M}^{(l)}_{\text{seed}}\} \otimes \{+\tilde{\mathcal{P}}^{(l-1)}, -\tilde{\mathcal{M}}^{(l-1)}\}))$$

Let $\tilde{p}^{(l)}_{uv}$ and $\tilde{m}^{(l)}_{uv}$ indicate the fixed location (u, v) in $\tilde{\mathcal{P}}^{(l)}$ and $\tilde{\mathcal{M}}^{(l)}$ under $R^{(l)}$ as shown in Figure 13(a). Let $g(\cdot)$ be a function that extracts entries participating in the computation related to (u, v) in a signed tensor of Equation (7). For $\{+\tilde{\mathcal{P}}^{(l)}, -\tilde{\mathcal{M}}^{(l)}\}$, $g(\cdot)$ extracts $\tilde{p}^{(l)}_{uv}$ and $\tilde{m}^{(l)}_{uv}$:

$$\{+\tilde{p}^{(l)}_{uv}, -\tilde{m}^{(l)}_{uv}\} \leftarrow g\left(\{+\tilde{\mathcal{P}}^{(l)}, -\tilde{\mathcal{M}}^{(l)}\}, (u, v)\right)$$

Note that $R^{(l-1)}$ is a selected region with probability $p^{(l)}_{ij} + m^{(l)}_{ij}$ where $p^{(l)}_{ij} \in \mathcal{P}^{(l)}_{\text{seed}}$ and $m^{(l)}_{ij} \in \mathcal{M}^{(l)}_{\text{seed}}$. As shown in Figure 13(b), suppose $p^{(l)}_{ij}$ and $m^{(l)}_{ij}$ correspond to $(1, 2)$-th quadrant, respectively, i.e., $p^{(l)}_{ij} = p^{(l)}_{12}$ and $m^{(l)}_{ij} = m^{(l)}_{12}$. Then, other quadrant probabilities except for $p^{(l)}_{12}$ and $m^{(l)}_{12}$ do not affect the computation of $\{+\tilde{p}^{(l)}_{uv}, -\tilde{m}^{(l)}_{uv}\}$ through Kronecker product. Also, since (u, v) is fixed, the only locations corresponding to (u, v) of $\tilde{\mathcal{P}}^{(l-1)}$ and $\tilde{\mathcal{M}}^{(l-1)}$ affect the final result as shown in Figure 13(b). In other words, only $\tilde{p}^{(l-1)}_{uv}$ and $\tilde{m}^{(l-1)}_{uv}$ participate in the computation for $\{+\tilde{p}^{(l)}_{uv}, -\tilde{m}^{(l)}_{uv}\}$, and $\{+\tilde{p}^{(l-1)}_{uv}, -\tilde{m}^{(l-1)}_{uv}\}$ are recursively obtained by $g(\cdot)$ as follows:

$$\{+\tilde{p}^{(l-1)}_{uv}, -\tilde{m}^{(l-1)}_{uv}\} \leftarrow g\left(\{+\tilde{\mathcal{P}}^{(l-1)}, -\tilde{\mathcal{M}}^{(l-1)}\}, (u, v)\right)$$



Selected Region $R^{(l)}$
v
u
$\tilde{p}^{(l)}_{uv}$
$\tilde{m}^{(l)}_{uv}$
$\tilde{\mathcal{P}}^{(l)}$
$\tilde{\mathcal{M}}^{(l)}$
Selected Region $R^{(l-1)}$
$p^{(l)}_{12}$
$m^{(l)}_{12}$
$\tilde{p}^{(l-1)}_{uv}$
$\tilde{m}^{(l-1)}_{uv}$
$\tilde{\mathcal{P}}^{(l-1)}$
$\tilde{\mathcal{M}}^{(l-1)}$

(a) Recursion level l (b) Recursion level $l-1$

Figure 13: The locations corresponding to (u, v) in (a) $\{+\tilde{\mathcal{P}}^{(l)}, -\tilde{\mathcal{M}}^{(l)}\}$ and (b) $\{+\tilde{\mathcal{P}}^{(l-1)}, -\tilde{\mathcal{M}}^{(l-1)}\}$.

Hence, Equation (10) is represented with $g(\cdot)$ as follows:

$$g\left(\{+\tilde{\mathcal{P}}^{(l)}, -\tilde{\mathcal{M}}^{(l)}\}, (u, v)\right)$$
$$= f_\alpha\left(f_b\left(\{+p^{(l)}_{ij}, -m^{(l)}_{ij}\} \otimes g\left(\{+\tilde{\mathcal{P}}^{(l-1)}, -\tilde{\mathcal{M}}^{(l-1)}\}, (u, v)\right)\right)\right)$$
$$\Leftrightarrow \{+\tilde{p}^{(l)}_{uv}, -\tilde{m}^{(l)}_{uv}\} = f_\alpha(f_b(\{+p^{(l)}_{ij}, -m^{(l)}_{ij}\} \otimes \{+\tilde{p}^{(l-1)}_{uv}, -\tilde{m}^{(l-1)}_{uv}\})). \quad \square$$

Note that Generate-Edge$(\cdot)$ represents the recursive function $g(\cdot)$, and $\tilde{p}^{(L)}_{uv} = P(u, v, +)$ and $\tilde{m}^{(L)}_{uv} = P(u, v, -)$.

D DEFINITIONS

Definition D.1 (Kronecker Product). Given $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{p \times q}$, the Kronecker product of $\mathbf{A}$ and $\mathbf{B}$ is defined as follows:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}$$

where a_{ij} is the (i, j)-th entry of $\mathbf{A}$, and $\mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{mp \times nq}$. ■

Definition D.2 (Absolute Difference for Signed Triangles and Edge Signs [6]*).* Let $\rho_{\text{real}}(\cdot)$ and $\rho_{\text{syn}}(\cdot)$ denote ratios from a real network and a synthetic network, respectively. Let T be the set of signed triangles, i.e., $\mathrm{T} = \{\triangle_{+++}, \triangle_{++-}, \triangle_{+--}, \triangle_{---}\}$. Then, *absolute difference* for signed triangles is defined as follows:

$$\text{Abs. Diff. (T)} = \sum_{\triangle \in \mathrm{T}} |\rho_{\text{real}}(\triangle) - \rho_{\text{syn}}(\triangle)| \tag{11}$$

Let S be the set of signs, i.e., $\mathrm{S} = \{+, -\}$. Then, *absolute difference* for edge signs is defined as follows:

$$\text{Abs. Diff. (S)} = \sum_{s \in \mathrm{S}} |\rho_{\text{real}}(s) - \rho_{\text{syn}}(s)| \tag{12}$$

E CONNECTION TO SKG AND NOISY SKG

In terms of edge determination process (line 7 in Algorithm 1 and line 8 in Algorithm 2) without signs, SKSG-B and SKSG are equivalent to Stochastic Kronecker Graph (SKG) [23] and Noisy SKG [37], respectively. SKG constructs a stochastic adjacency matrix $\mathcal{A}$ using Kronecker product where each entry $\mathcal{A}_{uv}$ indicates a probability $P(u, v)$ of forming edge $u \longrightarrow v$. In our models, the probability $P(u, v)$ is divided into $P(u, v, +)$ and $P(u, v, -)$, i.e., $P(u, v) = P(u, v, +) + P(u, v, -)$, implying that $\mathcal{A} = \mathcal{P} + \mathcal{M}$ where $\{+\mathcal{P}, -\mathcal{M}\}$ is a stochastic signed tensor. Thus, the formation of edges without signs in SKSG-B is equivalent to that of SKG; consequently, networks from SKSG-B naturally inherit characteristics of those of SKG. Similarly, the edge formation of SKSG with noises corresponds to that of Noisy SKG.

F PARAMETER SETTING

Table 6 describes the selected α and the target recursion level L of BalanSing for each dataset.

Table 6: Parameters used in BalanSing

Parameters	BitcoinA	BitcoinO	Epinions
α	0.84	0.75	0.65
L	12	13	17

Distributed Similarity Joins over Top-K Rankings[*]

Evica Milchevski
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
milchevski@cs.uni-kl.de

Sebastian Michel
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
michel@cs.uni-kl.de

ABSTRACT

Top-k rankings are a commonly used technique to summarize the most important entities of a specific domain. In this work, we focus on efficiently solving the problem of similarity joins for top-k rankings, for instance, for determining users with similar affinity, or for grouping related queries in search engines, based on their results. We put forward a novel algorithmic multi-step solution, realized via Apache Spark, harnessing mathematical properties of the distance function and a preceding near-duplicate detection phase, for search-space pruning. We further show how existing state-of-the-art algorithms for set similarity joins can be adapted to handle top-k rankings and how the data partitioning internals of Spark can be used to enable efficient data processing. The experimental study over standard benchmark datasets reveals that the proposed solution outperforms the state-of-the-art competitor by up to a factor of 5, despite involving additional stages of processing.

1 INTRODUCTION

Similarity joins have been a popular research topic in the database community for more than a decade now. Previous research in this topic is concerned with solving the problem of similarity joins for sets [7, 11, 25, 28], strings [13] or the more general problem of finding the similar objects in metric space [12]. Many distributed solutions, developed for the MapReduce framework, have also been proposed. Recently, Fier et al. [10] summarized and compared these distributed solutions. In this paper, we specifically focus on solving the problem of similarity joins for top-k rankings. Top-k rankings are a very popular and widely used technique to summarize the most relevant entities from a certain domain. Fast and efficient solutions for similarity joins of top-k rankings is of great value in many contexts. For instance, the case of query suggestion or expansion in search engines based on finding similar queries by comparing their result lists, in a dating portal where we can use the preferences and affinities of users, presented in a form of top-k lists, for matchmaking, or in the case of recommender systems, where the similarity between the top sold (liked, favored) items for different clients can help in recommending products. For instance, consider Table 1 containing favorite movies of members of some dating portal. By comparing the lists, we see that Alice and Chris have similar taste so the system should match them for a date.

Spearman's Footrule distance is used as a distance measure for comparing two top-k lists. Fagin et al. [9] show that there is a Spearman's Footrule adaptation for top-k rankings that is a metric. This immediately entails the use of existing metric space similarity join approaches. On the other hand, rankings

[*]This work has been supported by the German Research Foundation (DFG) under grant MI 1794/1-1/2.

	Alice	Bob	Chris
1.	Pulp Fiction	The Schindler List	Indiana Jones
2.	E. T.	Lord of the Rings	Pulp Fiction
3.	Forrest Gump	Avengers	Forrest Gump
4.	Indiana Jones	Indiana Jones	E. T.
5.	Titanic	E. T	Titanic

Table 1: Dating portal users' favorite movies

can be considered as plain sets and accordingly indexed using inverted indices, that keep for each item a list of rankings where this item appears. Thus, many of the distributed algorithms that solve the problem of similarity joins for sets can be applied for top-k rankings. The best performing one, according to a recent study [10], is the algorithm proposed by Vernica et al. [24], based on the principle of prefix filtering. This approach, as shown in the study [10], also outperforms existing metric space similarity join approaches. Furthermore, Fier et al. [10] showed that the existing distributed solutions in MapReduce do not scale well, and propose that Apache Spark is used as a platform for developing new alternative solutions. In this paper, we specifically focus on studying an efficient and scalable top-k rankings similarity joins using Apache Spark [4].

We propose a novel approach implemented in Apache Spark that is better tailored to the properties of this platform. In contrast to MapReduce, where each stage is composed from only a *map* and *reduce* function, and the data from each stage is written to disk, Apache Spark is more suitable for iterative processing of data and performs the computation in memory. Thus, we propose an iterative approach that computes the similarity join in several stages, while storing the intermediate results in memory. As Spearman's Footrule adaptation for top-k rankings is a metric, the algorithm uses the triangle inequality to reduce the number of candidate pairs generated. Very similar rankings are clustered together, and then, only the cluster representatives are joined, reducing the size od the data processed, and thus, finding more efficiently the join results. Through a detailed experimental study we show that our algorithm outperforms the competitor, especially for larger values of the similarity threshold θ.

1.1 Problem Statement and Setup

As **input** we are provided with a dataset $\mathcal{T}$ of rankings τ_i (Table 2). Each ranking has a domain D_{τ_i} of items it contains. We consider fixed-length rankings of size k, i.e., $|D_{\tau_i}| = k$, but investigate the impact of different choices of k on the join performance [1]. The considered rankings do not contain any duplicate items.

The ranked items in a ranking are represented as arrays or lists of items, where the left-most position denotes the top ranked item. In addition, each ranking has an id associated to it. Without

[1]Working with fixed-length rankings gives better insights into the difference of performance of the algorithms. For handling variable-length rankings, only the length boundaries for the Footrule distance, given a distance threshold, need to be computed .

ranking id	ranking content
τ_1	[2, 5, 4, 3, 1]
τ_2	[1, 4, 5, 9, 0]
τ_3	[0, 8, 5, 7, 3]

Table 2: Sample dataset $\mathcal{T}$ of top-5 rankings (items are represented by their ids).

loss of generality, in the remainder of the paper, we assume that items are also represented by their ids. The rank of an item i in a ranking τ is given as $\tau(i)$.

A distance function d quantifies the distance between two rankings—the larger the distance the less similar the rankings are. *Given a dataset of top-k rankings $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ and a distance threshold θ we want to find all pairs (τ_i, τ_j), $\tau_i, \tau_j \in \mathcal{T}$, $i \neq j$, where the distance d between τ_i and τ_j is smaller or equal to θ, i.e., $d(\tau_i, \tau_j) \leq \theta$.*

In this work, we focus on the computation of Spearman's Footrule distance, but the proposed algorithm can be applied on any distance measure that satisfies the triangle inequality. Spearman's Footrule distance is computed as a sum over the difference in position of the items in the two rankings, i.e., $F(\tau, \sigma) = \sum_{i \in D_\tau \cup D_\sigma} |\tau(i) - \sigma(i)|$. An artificial rank l for items not contained in a ranking, i.e., $\tau(i) = l$ if $i \notin D_\tau$, is considered. Consider the rankings τ_1 and τ_2, in Table 2. For a rank $l = 6$ for not-contained items, we obtain $F(\tau_1, \tau_2) = 4+1+1+5+2+2+1 = 16$. A more detailed introduction to rankings, specifically top-k rankings, distance functions, and how to handle items i that are not in a ranking τ is described in Section 3.

1.2 Contributions and Outline

The contributions of our work can be summarized as follows.

- We adapt existing set-based similarity join algorithm to the problem of top-k rankings. We furthermore implement and adapt this algorithm to the Apache Spark framework.
- We introduce a new iterative, highly configurable algorithm that combines metric space distance-based filtering with state-of-the-art set-based similarity join algorithms.
- We propose further optimization to the proposed algorithm by presenting a method for repartitioning large partitions.
- We implemented our methods and competitors in Apache Spark and through an extensive experimental study on two real-world datasets we show that our methods consistently outperform state-of-the-art approaches for larger values of the threshold θ.

The rest of the paper is structured as follows. Section 2 discusses related work and, in Section 3, we present background on top-k rankings, a state-of-the-art distributed set similarity join algorithm, and Apache Spark. In Section 4, we describe how existing set-based similarity join algorithm is adapted to top-k rankings. The clustering algorithm and its components is introduced in Section 5. Section 6 proposes a Spark-based repartitioning technique. We experimentally evaluate the presented approaches in Section 7. Finally, we give the conclusion in Section 8.

2 RELATED WORK

To the best of our knowledge, the problem of similarity join for top-k rankings has not been addressed so far. As top-k rankings can also be seen as sets, we mainly focus here on explaining set-based similarity joins.

In-memory similarity join approaches: There is an ample work on computing the similarity join for sets or strings. Mann et al. summarize and compare the in-memory based approaches in [16]. Previous approaches are mainly based on a filter and verification framework which uses inverted indices as the initial filter for pairs that do not have any items in common and applying additional filters that prune dissimilar pairs. In the verification phase the candidates are verified by computing their true similarity score. The prefix-filtering approach, initially proposed by Chaudhuri et al. [7] is the most well know algorithm for finding the similar pairs. It works by first sorting all records in the dataset in the same canonical order and then indexing only a prefix of the record with an inverted index. The size of the prefix depends on the threshold and distance, i.e., the similarity measure used. The records are usually sorted by the ascending frequency of the elements in the sets. There are many works [5, 6, 21, 25, 28] that propose improvements over the initial prefix-filtering algorithm, by introducing additional position or length-based filters, reducing the size of the prefix, introducing variable length prefixes, grouping based on the prefix, etc. Recently Wang et al. [26], motivated by the conclusions presented in [16], proposed an approach that improves upon existing prefix-filtering approaches by introducing index level and answer-level skipping. The index level skipping reduces the unnecessary checks done by position and length-based filters, by using length-sorted skipping blocks in the posting lists, augmented with the positions of the elements in the sets. The answer-level skipping is based on the idea that the answer sets of similar sets should be also similar, thus the already computed answer set of one set is used for computing the answer set of another, similar, set. Bouros et al. [6] propose an approach for spatio-textual similarity joins. They describe algorithms that partitions and filters the data based on the spatial (Euclidean) distance between the points in the dataset, and, in addition, they extend the prefix-filtering method by introducing grouping based on the prefix of the textual data. The grouping based on the prefix of the textual data is orthogonal to our presented approach, i.e., it can be applied in our approach in addition to (instead of) the VJ algorithm. Their method for distance based partitioning of the data space resembles our idea for clustering based on the distance threshold. However, the solution proposed in [6] works specifically for two dimensional data. Top-k rankings, on the other hand, can be interpreted as points in multidimensional space, where the dimension is determined by the size of the rankings, usually 10 or larger. In addition, our clustering approach is more general, and works for any data in metric space.

MapReduce-based similarity join approaches: To handle larger datasets, many distributed solutions for similarity join of sets have also been proposed. Recently, Fier et al. [10] summarized and compared the MapReduce-based similarity join solutions. Vernica et al. [24] present a distributed solution, referred to as VJ, based on the well known prefix filtering method. Since we use this algorithm in our implementation, we describe it in more details in Section 3. The *V-SMART* algorithm [17] adopts a different idea, by computing the ingredients of the similarity measure in a distributed manner, which are later joined to compute the final results. It works in two phases, a joining phase and a similarity phase. In the first phase, the joining phase, the partial results for each set is computed and joined to all the elements in the sets. In the similarity phase, the algorithm takes as input the output from the first phase, builds an inverted index, and then, while traversing the posting lists for each element s_i, emits pairs of sets together with information needed to compute the final similarity. Deng et al. present MassJoin [8]. This approach is based on PassJoin [15], a main memory method for string similarity joins. The idea behind their method is to generate signatures for the sets $r \in R$, and then for each signature of r they generate signatures for $s \in S$. In order for s and r to be similar, they should share at least one signature. Additional filters are applied to reduce the number of candidate pairs generated. Rong et al. present FS-Join [20]. They claim that their algorithm outperforms the competitors because it addresses some of the issues that previous approaches had, i.e., it does not generate duplicate results and achieves better load balancing. The dataset is vertically partitioned, by dividing each set into segments and then partitioning the data according to the segments. Interestingly, Fier et al. [10] came to the conclusion that the approach proposed by Vernica et al. [24] outperforms the other approaches in most scenarios. Therefore, in this work we compare our approach to the one presented in [24]. Distributed metric space approaches have also been proposed [22, 27]. Wang et al. [27] for a dataset D, partition the input dataset D into N disjoint partitions P_i, $P_i \cup P_j = \emptyset, \cup_{i=1}^{N} P_i = D$, created by randomly choosing N centroids p_i and assigning each point $p \in D$ to the partition represented by the closest centroid. Further, they define inner and outer sets of a partition and based on that they decide the data distribution. The proposed MapReduce algorithm consists of two main stages, partitioning the data, and, in the second stage, computing the similarity join. Sarma et al. [22] propose a MapReduce method that works very well for very small distance thresholds. In fact, they evaluate their approach using only threshold values up to 0.1. The novelty in their work is that they apply several filtering techniques, both distance specific and not, which lead to having tighter partitions, and thus, fewer comparisons.

In prior work [18], we solve the problem of answering similarity range queries over top-k rankings. There, in addition to an algorithm based to the prefix-filtering framework, we also presented a so-called coarse index, that combines an inverted index with a metric index structure to reduce the number of distance function computations.

3 PRELIMINARIES

Complete rankings are considered to be permutations over a fixed domain $\mathcal{D}$. We follow the notation by Fagin et al. [9] and references within. A permutation σ is a bijection from the domain $\mathcal{D} = \mathcal{D}_\sigma$ onto the set $[n] = \{1, \ldots, n\}$. For a permutation σ, the value $\sigma(i)$ is interpreted as the rank of element i. An element i is said to be ahead of an element j in σ if $\sigma(i) < \sigma(j)$. We consider incomplete rankings, called top-k lists in [9]. Formally, a top-k list τ is a bijection from D_τ onto $[k]$. The key point is that individual top-k lists, say τ_1 and τ_2 do not necessarily share the same domain, i.e., $D_{\tau_1} \neq D_{\tau_2}$.

Pairwise similar rankings can be retrieved by means of distance functions, like Kendall's Tau or Spearman's Footrule distance. In this work we use Spearman's Footrule adaptation for top-k lists proposed in [9]. Spearman's Footrule distance is computed as a sum over the difference in position of the two rankings, i.e., $F(\tau, \sigma) = \sum_{e \in D_\tau \cup D_\sigma} |\tau(i) - \sigma(i)|$. An artificial rank l for items not contained in a ranking, i.e., $\tau(i) = l$ if $i \notin D_\tau$ is considered.

In this work, we assume that $\tau(i)$ takes values from 0 to $k - 1$ (instead of 1 to k), and we fix the value of l to k as suggested in [9]. It is clear that this does not affect our algorithms. We further consider only rankings of same size k, thus the largest possible value of the Footrule distance is $k * (k + 1)$ and occurs if two disjoint rankings are compared. The smallest distance is 0, for the compared rankings are identical. In the rest of the paper, for ease of presentation, we use normalized values for the Footrule distance and the threshold values, ranging from 0 to 1.

3.1 Vernica Join (VJ) Algorithm

According to a recent experimental study [10], the VJ algorithm outperforms other distributed similarity join algorithms in most cases. The algorithm is implemented in MapReduce and is based on the well known prefix filtering method. It works in several phases, each representing one map reduce job. For each phase, the authors propose several variations, however, here we describe the version which, according to their evaluation, showed the best performance.

In the first phase, all records are read and the tokens in the universe are sorted according to the increasing frequency of appearance in the sets. Then, in the next phase, the sorted tokens are loaded into the memory of the mappers and used for sorting the sets into a canonical form. Then the mappers emit a composite key consisting of the token and the size of the set, plus the whole set as value, but only those elements that belong to the prefix. For grouping the records in the reducers, only the token is used, while the size of the set is used for sorting the records by size. The latter allows utilizing size-based filters. At the reducers, for all the rankings that share at least one element together, the PPJoin+ algorithm [28] is used, to find the similar rankings. In the final phase, duplicate pairs must be removed, since the same pair can be generated at several machines.

3.2 Apache Spark

Apache Spark [4] is a general purpose platform that enables easy and fast development and execution of distributed applications. It can be considered successor of MapReduce, as it provides similar capabilities with generally better performance. Additionally, several other functionalities are provided and many libraries are built on top of its core. The parallelization of applications is easier when using Apache Spark due to the notions of RDD, *transformations* and *actions* used in the platform. RDDs are collections of elements distributed across the nodes of a cluster [14]. Once created, they are then partitioned among the available nodes of a cluster. This way, each node handles a subset of the input. RDDs are evaluated lazily, meaning that, instead of directly computing each RDD transformation, the computation is performed only at the end, when the final RDD data needs to be materialized. This allows Spark to optimize job execution, by analyzing and grouping the transformations that are performed over the RDDs.

Another important characteristic of Apache Spark is its ability to execute iterative processes, using the main memory of the nodes, in order to reduce disk I/O, thus, reducing the overall execution time of the application, leading to superior performance over MapReduce, as shown by Shi et al. [23].

4 A VJ-STYLE ALGORITHM FOR TOP-K RANKINGS

To find all pairs of similar top-k rankings for a given set of rankings $\mathcal{T}$ and a threshold θ, we can use the Vernica Join (VJ) algorithm. However, in order to be able to apply it on top-k rankings,

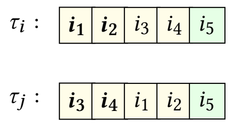

τ_i :
$\boldsymbol{i_1}$ $\boldsymbol{i_2}$ i_3 i_4 i_5
τ_j :
$\boldsymbol{i_3}$ $\boldsymbol{i_4}$ i_1 i_2 i_5

Figure 1: Example rankings with $k = 5$ and $p = 2$ with maximum Footrule distance $F(\tau_i, \tau_j) = 8$.

we need to derive the prefix size p of top-k rankings, when Spearman's Footrule distance is used to compare them[2]. There are two ways for computing the prefix size of top-k rankings, one considering the overlap of the rankings, and the other, considering their position. The latter provides slightly tighter prefix sizes than the first. However, the former allows more freedom in choosing the items in the prefix. In prior work [18], we found the minimum overlap between two rankings τ_i and τ_j, such that $F(\tau_j, \tau_j) = \theta$, as $\omega = \lfloor 0.5 * (1 + 2 * k - \sqrt{1 + 4 * \theta})$ and the prefix size as $p = k - \omega + 1$. We refer to this as **prefix based on overlap**. In addition, we define an ordered prefix, p_o, as:

Lemma 4.1. *For a given Spearman's Footrule distance threshold θ and a ranking length k, the* **ordered prefix** *p_o of the top-k rankings is given by the best ranked:*

$$p_o = \lfloor \frac{\sqrt{\theta}}{\sqrt{2}} \rfloor + 1$$

items of the rankings.

Proof. The lowest Footrule distance that two top-k rankings τ_i and τ_j can have, when none of the first p items of each ranking are overlapping, $L(p, k)$, is when the items are overlapping in the rankings, i.e., $D_{\tau_i} = D_{\tau_j}$, but they are positioned in the next p places in the other ranking. This is so, because the partial Footrule distance of an item we get either by the difference in its positions, when they are overlapping, or as $k - \tau_i(i)$ when the item is non overlapping. For the items i positioned in the first p places in ranking, where $p < \frac{k}{2}$, the partial distance of the items being overlapping and placed at the next p places is always lower than if an item is non overlapping. $L(p, k)$ can be computed as $\frac{(p*2)^2}{2}$. An example of such rankings,where $p = 2$, are the rankings τ_i and τ_j, shown in Figure 1. These rankings have the same domain, i.e., $D_{\tau_i} = D_{\tau_j} = \{i_1, i_2, i_3, i_4, i_5\}$, however, when looking only the first p items, written in bold, they have no overlap. The Footrule distance between them is $F(\tau_i, \tau_j) = 8$, the lowest that they can have when the first p items are not shared.

Solving $L(p, k) = \theta$ gives us the first $p = \lfloor \frac{\sqrt{\theta}}{\sqrt{2}} \rfloor$ items that can be non-overlapping in case of θ. Taking one more item guaranties that we will not miss any candidate pair. [3] □

Given the Footrule distance threshold θ, we now describe the VJ algorithm for top-k rankings. The first step in the VJ algorithm is counting the frequency of the elements in the sets and ordering them by frequency. This step is not needed for top-k rankings and can be skipped. However, since most real world datasets follow a skewed distribution, through experiments we concluded that reordering the rankings by the item's frequency leads to major performance gains, and thus, we keep this step for top-k rankings as well. This entails that the prefix size based on the overlap between the rankings should be used. To perform the reordering, we first count the frequency of the items in the rankings. Then, in order to make this collection available to all the nodes, in Spark, we use a broadcast variable which is cached on each machine and then used to sort the items of all rankings $\tau \in \mathcal{T}$ by increasing order of their frequency. Note that, while we reorder the items in the rankings, we still need to keep track of their original rank for the computation of the Footrule distance, thus rankings are transformed to arrays of $(i_{id}, \tau(i))$ pairs. In the next step we transform the rankings, into (i_{id}, τ) pairs, where as key we have the item id, and as value we have the ranking. This we only do for the items that belong to the prefix of the ranking. Then in the next step, in order to bring all rankings that share an item to the same partition, we aggregate the tuples (RDD), created in the previous step, by key. In the next step, for the rankings that share an item, a main memory approach for finding the similar pairs is applied. For the rankings on each item list we index their prefixes using an inverted index. In addition, based on our previous work [19] we apply a position filter in order to filter out more candidate pairs. In [19], we proved that two rankings τ_i and τ_j cannot have distance smaller than θ if at least one of the items in the rankings have a difference in their ranks larger than $\frac{k*(k+1)*\theta}{2}$. That means, if there is at least one item $i \in \tau_i, \tau_j$, such that, $|\tau_i(i) - \tau_j(i)| > \frac{k*(k+1)*\theta}{2}$, we can be sure that $d(\tau_i - \tau_j) > \theta$. For the candidate pairs that pass the filters, we compute the Footrule distance. Note that, since we work with rankings of same size k, filtering based on the length of the rankings is not applicable. Finally, before writing the final result, we remove the duplicate pairs.

4.1 Improved Memory Usage

Previous distributed approaches for similarity joins were designed and implemented in MapReduce. Spark, as a successor of MapReduce, has different characteristics than MapReduce, and thus existing approaches can be adapted to the computational properties of Spark in order to improve their performance.

Datasets in Spark are represented as RDDs, which are immutable, distributed collections of objects, stored in the memory of the executors. This means that for every transformation of an RDD, a new RDD is created. In addition to this, Spark runs in the JVM, which means that garbage collection can easily cause performance issues for Spark jobs. Thus, keeping objects in the memory of executors is not recommended, since it can lead to crashes or performance degradation when dealing with large datasets. Instead, working with iterators is more native to the Sparks computational model, since this allows the framework to spill some data to disk, when needed.

The VJ algorithm that shows the best performance, according to [24] works such that rankings that share the same item are distributed to different partitions. Next, on each partition, an in memory join algorithm is executed, to compute the rankings with distance smaller than θ. This entails, first, storing a dictionary of the items, second, storing an inverted index for the rankings for this partition, and storing the partial result sets until the final computation is done. In addition to this, since Spark works with immutable objects, sorting the objects for performing the per partition in memory join, imposes creating new objects for each ranking. This means that the VJ algorithm can lead to having

[2]To use another distance measure with a prefix-filtering based algorithm, these bounds will need to be recomputed. However, our approach is flexible and any other algorithm can be used.

[3]Note that this only holds when $\theta \leq \frac{k^2}{2}$. In the case when $\theta > \frac{k^2}{2}$ computing the formula for the ordered prefix size is more complicated and we leave it as future work, since using values of $\theta \leq \frac{k^2}{2}$ is more than enough for our problem setting, as it is common practice to use values of $\theta \leq 0.4$. $\theta = \frac{k^2}{2}$ is around 0.45 when normalized, depending on the value of k.

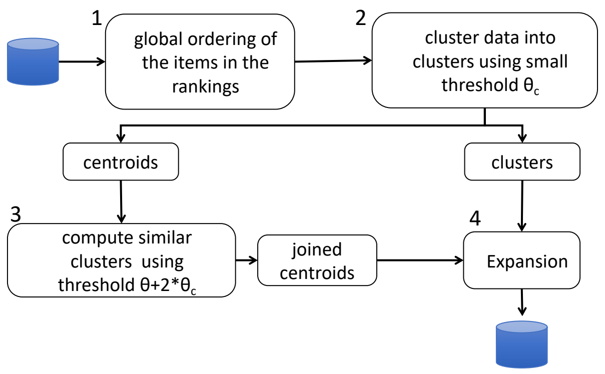

1
global ordering of the items in the rankings
2
cluster data into clusters using small threshold θc
centroids
clusters
3
compute similar clusters using threshold θ+2*θc
joined centroids
4
Expansion

Figure 2: Overall architecture. The algorithm has four main phases: ordering, clustering, joining and expansion.

both issues that we mentioned above, bad performance caused by the garbage collector overhead, and, memory overhead crashes, due to keeping data structures and objects in memory.

Instead, we claim that a nested loop modification to the VJ algorithm, is more native to Spark's processing style. Instead of indexing the rankings per partition, we propose using iterators to walk through the rankings in a nested loop fashion. For each ordered pair of rankings,(τ_i, τ_j), where $\tau_i < \tau_j$ that passes the position filter defined above, we compute the Footrule distance, and output those pairs where $d(\tau_i, \tau_j) \leq \theta$. This approach, as we will show in our experiments, performs better for large datasets, since allows Spark to spill the data to disk, when needed.

5 APPROACH

Driven by the idea that similar rankings should have similar result sets, we propose a novel approach having a pre-processing step, where very similar rankings are grouped. Then, only one representative ranking from the clusters, called centroid, is considered in the next similarity join phase. The idea is that by doing this, the number of records being joined is reduced and, thus, the execution time of the main joining phase is reduced too—which is actually the most expensive part of similarity join algorithms. Since Spark is suitable for iterative processing, adding an additional phase should be acceptable. However, this pre-processing phase should be very efficient, such that we do not end up with having a higher overhead than real benefit. Another key observation is that the Footrule distance is a metric and, thus, the triangle inequality can be used for forming, and expanding the clusters, after finding the similar centroids, to compute the final result set more efficiently.

Making use of the above observations, we propose an approach consisting of four main phases: Ordering, Clustering, Joining, and Expansion, depicted in Figure 2.

Ordering: The first phase of our approach is ordering the items in the rankings by their occurrence, i.e., items that occur less frequently in the rankings, are moved to the top positions of the rankings. In our proposed algorithm, as later described, the VJ similarity join algorithm is applied twice, once for clustering the rankings, and once for finding the similar clusters. Instead of reordering the rankings twice, we choose to do this only once, using the original dataset $\mathcal{T}$. As our approach does not depend on the similarity join algorithm used for clustering or for joining the clusters, the re-ordering of the items in the rankings can be skipped if it is of no use to the joining algorithm applied later on. The reordering is done just for determining which items will be included into the prefix of the rankings, while the rankings still preserve their original item ordering for the computation of the distance.

Clustering: The second phase of our approach is forming clusters, such that similar rankings will belong to the same cluster, C_i. First, a similarity join algorithm is executed, to find the similar rankings that need to be grouped together. Our experiments revealed that VJ is the most efficient one to be used here, which supports the findings by Fier et al. [10]. In principle, however, any similarity algorithm could be employed at this stage. Then, clusters are formed such that the pairwise distance between each member of the cluster and its representative is at most θ_c. We will refer to θ_c as the *clustering threshold*. In contrast to other similarity join algorithms in metric space, where clusters have different radius, the radius of all clusters formed by our approach is bounded by the clustering threshold, θ_c. We write $\tau_i \prec c_i$ to denote that ranking τ_i belongs to the cluster, C_i, represented by ranking (centroid) c_i. Rankings in the dataset $\mathcal{T}$ that do not have any similar rankings with distance smaller than the clustering threshold, θ_c, form *singleton* clusters, i.e., one element clusters.

Joining: In this phase a similarity join algorithm is executed over the *centroids* with a threshold $\theta_o = \theta + 2 * \theta_c$. Using threshold θ_o instead of θ is necessary in order to insure the correctness of the algorithm. Note that any similarity join algorithm can be applied here, independently from the algorithm used in the clustering phase. Due to the aforementioned reason, we implement the VJ algorithm.

Expansion: In the last step of the algorithm the final result set is computed, by joining the results from the joining phase with the formed clusters in the clustering phase. The members of the joined clusters from the joining phase are checked against each other if the distance between them is smaller than θ. Using the metric properties of the distance measure, we are able to directly filter out some candidates, and thus compute the final result list more efficiently.

Before we describe each phase more formally, how each phase is realized and how the final join results is computed, we first discuss the correctness of the proposed algorithm.

Lemma 5.1. *For given join threshold θ and clustering threshold θ_c, in the joining phase, all pairs of centroids c_i, c_j with distance $d(c_i, c_j) \leq \theta + 2 * \theta_c$ need to be retrieved in order not to miss a potential join result.*

Lemma 5.1 ensures that pairs $\{(\tau_i, \tau_j) | \tau_i \prec c_i, \tau_j \prec c_j \wedge d(\tau_i, \tau_j) \leq \theta \wedge d(c_i, c_j) > \theta\}$ will not be omitted from the result set.

In other words, Lemma 5.1 avoids missing result rankings with distance $\leq \theta$, which are represented by centroids which are with distance larger than θ from each other.

This follows from the fact that for all rankings $\{\tau_i | \tau_i \prec c_i \wedge d(\tau_i, c_i) \leq \theta_c\}$. It follows that for any pair of rankings $\{\tau_i, \tau_j | \tau_i \prec c_i, \tau_j \prec c_j\}$ the distance of the corresponding centroids $d(c_i, c_j)$ must be $\leq \theta + 2 * \theta_c$. Thus, using a threshold $\theta_o = \theta + 2 * \theta_c$ in the joining phase is enough to ensure that no true result will be missed.

5.1 Clustering

When forming the clusters the following points need to be considered: ***(i)*** To ensure correctness, the radius of all the clusters should be the same, i.e., for any ranking $\tau_i \in C_i$, represented by a ranking c_i, $d(\tau_i, c_i) \leq \theta_c$. ***(ii)*** The clustering method should be very efficient, otherwise the cost of the clustering would overweight its benefit. ***(iii)*** The performance of the expansion phase depends on the clusters formed. We address each point individually while explaining our design choices for the clustering algorithm.

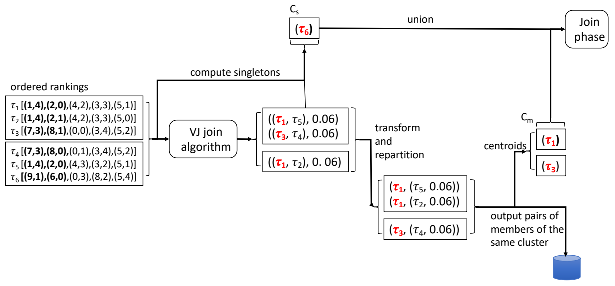
For forming the clusters, we could turn to existing methods [22, 27], where, first, the centroids of the clusters are randomly chosen, and then, by computing the distance from the centroids to the other points in the dataset, the members of the clusters are found. However, considering that we aim at forming equal range clusters, where the points are very close to each other, this approach has two main drawbacks, which make it not suitable for our use case. First, due to the very small clustering threshold, and the random choice of the clusters, it could happen that for some, or in the worst case for all, of the chosen centroids, there are no other points in the dataset such that their distance to the centroids is smaller than the clustering threshold, θ_c. This leads to having singleton clusters which do not cause any performance benefit in the joining phase. Another drawback of this approach is that the number of clusters needs to be chosen upfront.

First, to find the rankings that are very similar to each other, instead of selecting the centroids first, and comparing the distance for each point to the centroids, we execute a similarity join algorithm with the clustering threshold over the whole dataset, $\mathcal{T}$. Any similarity join algorithm can be applied, however, since prefix filtering approaches are especially efficient for very small thresholds, in our implementation we use the VJ algorithm. Note that the rankings have already been sorted, so we do not perform any additional sorting in this phase. The result of the VJ algorithm are all pairs of rankings (τ_i, τ_j) whose distance is smaller than the clustering threshold, i.e. $d(\tau_i, \tau_j) \leq \theta_c$. The clusters are formed such that, from the pairs, we take the first ranking, i.e., the one with a smaller id, as the cluster centroid, and the second one as their member. This does not only keep the clustering phase efficient, but also simplifies the expansion of the results in the last phase, since then the expansion can simply be performed by joining the result set from the joining and clustering phase. Furthermore, this way we can also efficiently apply filters based on the distance of the elements to their centroids, explained in Section 5.3. Clusters formed this way theoretically correspond to clusters formed by grouping the results by the first ranking, and taking the first ranking as the centroid. For instance, in Figure 3, the following clusters would be formed $C_1 = \{\tau_1, \tau_2, \tau_5\}$, $C_2 = \{\tau_3, \tau_4\}$ with centroids τ_1 and τ_3, respectively.

Since Spearman's Footrule distance is a metric, we know that for any two rankings $\tau_i, \tau_j \in C_i$ it holds that $d(\tau_i, \tau_j) \leq 2 * \theta_c$, and thus, members of the same clusters can directly be written to disk as partial results, as long as $\theta_c * 2 < \theta$.

By creating the clusters in this way, all of the aforementioned requirements are satisfied. The radius of all the clusters is the same and both forming the clusters and expanding the result set is kept simple, and thus, very efficient. One minor drawback of this approach is that the formed clusters would be overlapping. However, resolving this overlap would negatively impact the performance of the clustering and the expansion phase.

As input to the next, joining phase, we union the set of centroids C_m that contains all centroids representatives of clusters with at least two members, i.e., $|C| \geq 2$ with the set of centroids C_s that represent the singleton clusters, i.e., $|C| = 1$. The set C_s is derived from the original dataset, by finding those rankings $\tau_i \in \mathcal{T}$ such that there is no other ranking $\tau_j \in \mathcal{T}$, such that, $d(\tau_i, \tau_i) \leq \theta_c$. An example of a singleton cluster in Figure 3 is $C_3 = \{\tau_6\}$.



C_s
(τ6)
union
Join phase
compute singletons
ordered rankings
τ1 [(1,4),(2,0),(4,2),(3,3),(5,1)]
τ2 [(1,4),(2,1),(4,2),(3,3),(5,0)]
τ3 [(7,3),(8,1),(0,0),(3,4),(5,2)]
τ4 [(7,3),(8,0),(0,1),(3,4),(5,2)]
τ5 [(1,4),(2,0),(4,3),(3,2),(5,1)]
τ6 [(9,1),(6,0),(0,3),(8,2),(5,4)]
VJ join algorithm
((τ1, τ5), 0.06)
((τ3, τ4), 0.06)
((τ1, τ2), 0. 06)
transform and repartition
(τ1, (τ5, 0.06))
(τ1, (τ2, 0.06))
(τ3, (τ4, 0.06))
C_m
centroids
(τ1)
(τ3)
output pairs of members of the same cluster

Figure 3: Example of how clusters are formed and centroids (marked with red) are chosen, where $\theta_c = 0.1$.

Example 5.2. Figure 3 shows through an example the creation of the clusters, for $\theta_c = 0.1$. The items in the rankings $\tau_1, \ldots \tau_6$ have already been sorted by increasing order of their frequency. For instance, in τ_1 item 1 with position $\tau_1(1) = 4$ is placed on the first position, as it appears three times in the rankings (ties are arbitrarily broken). After running a similarity join algorithm with distance threshold $\theta_c = 0.1$ on these rankings, the pairs (τ_1, τ_5), (τ_1, τ_2) and (τ_3, τ_4). In the following step clusters $C_1 = \{\tau_1, \tau_2, \tau_5\}$, $C_2 = \{\tau_3, \tau_4\}$ with centroids τ_1 and τ_3, respectively are formed. Furthermore, the ranking τ_6 forms a singleton cluster since it does not belong to any of the formed clusters C_1, C_2.

5.2 Joining

In the joining phase we need to find all centroids pairs (c_i, c_j) such that $d(c_i, c_j) \leq \theta_o$. To do this, we execute the VJ algorithm over all centroids c_i, with a threshold $\theta_o = \theta + 2 * \theta_c$. However, the VJ algorithm, as almost all similarity join algorithms, is sensitive to the threshold value—for larger threshold values the algorithm performs worse. Thus, it could happen that, even though we are joining a dataset $C \subseteq \mathcal{T}$, due to the larger threshold used, the joining phase performs worse than simply executing the VJ algorithm over the whole dataset $\mathcal{T}$. Again, note that we do not perform additional reordering of the rankings here, but the VJ algorithm is executed on the initially ordered rankings.

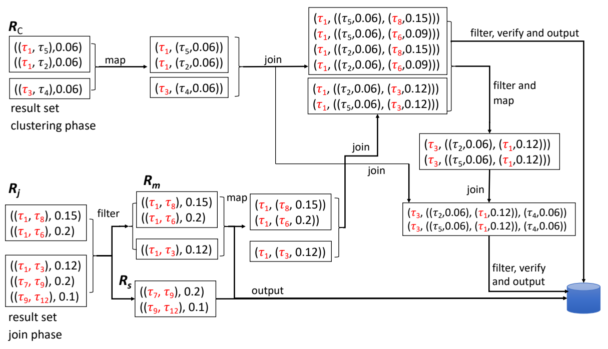
According to Lemma 5.1, using a threshold θ_o is only needed to avoid missing pairs of rankings $\{(\tau_i, \tau_j) | \tau_i \prec c_i, \tau_j \prec c_j \wedge d(\tau_i, \tau_j) \leq \theta \wedge d(c_i, c_j) > \theta)\}$. Furthermore, due to the small clustering threshold, in the dataset C we have many centroids which are representatives of singleton clusters. For these centroids, we can avoid unnecessary computation, by using a smaller threshold, without missing any true result. Lemma 5.3 defines this:

Lemma 5.3. *Given join threshold θ and clustering threshold θ_c, and a set of centroids $C = C_m \cup C_s$, where C_s is the set of centroids that represent the singleton clusters and $C_m = C \setminus C_s$ is the set of centroids representing non-singleton clusters. The following pairs of centroids need to be retrieved in order not to miss a potential join result:*

$$\{(c_i, c_j) \mid d(c_i, c_j) \leq \theta + 2 * \theta_c \quad if \ c_i, c_j \in C_m\} \tag{1}$$
$$\{(c_i, c_j) \mid d(c_i, c_j) \leq \theta + \theta_c \quad if \ c_i \in C_m \wedge c_j \in C_s \quad or \ v.v.\} \tag{2}$$
$$\{(c_i, c_j) \mid d(c_i, c_j) \leq \theta \quad if \ c_i, c_j \in C_s\} \tag{3}$$

```
method: Centroids Join
input: Dataset $C = C_m \cup C_s$, double $\theta$, $\theta_c$
output: all pairs $(c_i, c_j)$ s.t. $d(c_i, c_j) \leq \theta + 2 * \theta_c$
    $p_m$ = get_prefix($\theta + 2 * \theta_c$,$k$)
    $p_s$ = get_prefix($\theta$, $k$)
    grouped $\leftarrow$ transform_and_emit($C_m, C_s, p_m, p_s$)
    $\mathcal{R} \leftarrow$ compute_sim(grouped, $k$, $\theta$, $\theta_c$)
return $\mathcal{R}$
```

Algorithm 1: Joining of centroids based on the type of the centroid.



R_C
$((\tau_1, \tau_5),0.06)$
$((\tau_1, \tau_2),0.06)$
$((\tau_3, \tau_4),0.06)$
result set clustering phase
map
$(\tau_1, (\tau_5,0.06))$
$(\tau_1, (\tau_2,0.06))$
$(\tau_3, (\tau_4,0.06))$
join
$(\tau_1, ((\tau_5,0.06), (\tau_8,0.15)))$
$(\tau_1, ((\tau_5,0.06), (\tau_6,0.09)))$
$(\tau_1, ((\tau_2,0.06), (\tau_8,0.15)))$
$(\tau_1, ((\tau_2,0.06), (\tau_6,0.09)))$
$(\tau_1, ((\tau_2,0.06), (\tau_3,0.12)))$
$(\tau_1, ((\tau_5,0.06), (\tau_3,0.12)))$
filter, verify and output
filter and map
$(\tau_3, ((\tau_2,0.06), (\tau_1,0.12)))$
$(\tau_3, ((\tau_5,0.06), (\tau_1,0.12)))$
join
join
join
$(\tau_3, ((\tau_2,0.06), (\tau_1,0.12)), (\tau_4,0.06))$
$(\tau_3, ((\tau_5,0.06), (\tau_1,0.12)), (\tau_4,0.06))$
filter, verify and output
R_j
$((\tau_1, \tau_8), 0.15)$
$((\tau_1, \tau_6), 0.2)$
$((\tau_1, \tau_3), 0.12)$
$((\tau_7, \tau_9), 0.2)$
$((\tau_9, \tau_{12}), 0.1)$
result set join phase
filter
R_m
$((\tau_1, \tau_8), 0.15)$
$((\tau_1, \tau_6), 0.2)$
$((\tau_1, \tau_3), 0.12)$
map
$(\tau_1, (\tau_8, 0.15))$
$(\tau_1, (\tau_6, 0.2))$
$(\tau_1, (\tau_3, 0.12))$
R_s
$((\tau_7, \tau_9), 0.2)$
$((\tau_9, \tau_{12}), 0.1)$
output

Figure 4: Example of computing the final result set using the result set from the joining phase and the clusters where $\theta_c = 0.1$ and $\theta = 0.2$. Cluster's centroids are marked with red.

Lemma 5.3 allows us to more efficiently join the centroids. It follows that, only for the centroids $c_m \in C_m$ we need to use θ_o for joining and, thus, only for these centroids, we need to use a prefix based on the threshold θ_o. For the centroids $c_s \in C_s$, we can actually use the prefix based on the original threshold θ. Then, when computing the distance between the candidate pairs, we keep track of the type of the centroid, and accordingly, we output the pair if it satisfies the corresponding threshold. This is outlined in Algorithm 1.

Since we propose using small values for the clustering threshold θ_c, we expect that in practice, the cardinality of C_m will be significantly smaller than $|C|$, and thus, by applying a threshold of θ_o only for centroids $c_m \in C_m$, the savings should be notable.

5.3 Expansion

In the last phase, the final result set is generated. For this purpose, the results from the clustering phase, $\mathcal{R}_c$, and the result from the joining phase $\mathcal{R}_j$, need to be joined together, and the generated pairs need to be verified. Depending on the joined pairs from the joining phase, the expansion is done differently. The pairs where both centroids are singletons do not need to be expanded and are directly written to disc. Pairs where at least one of the rankings is not a singleton, need to be joined with the set of clusters, so that similar pairs of rankings between cluster members from different clusters, or with other singleton centroids, are generated.

Algorithm 2 outlines how the final result set is computed. First, the result set from the join phase, $\mathcal{R}_j$, is divided into two sets: $\mathcal{R}_s = \{(c_i, c_j) | c_i, c_j \in C_s \wedge d(c_i, c_j) \leq \theta\}$ and $\mathcal{R}_m = \mathcal{R}_j \setminus \mathcal{R}_s$. $\mathcal{R}_s$ is the set of candidate pairs, where both centroids are singletons. These pairs can be directly written to disc without further processing and verification. In addition, a subset of $\mathcal{R}_m$,

```
method: expand
input: Dataset $\mathcal{R}_c$, $\mathcal{R}_j$, double $\theta$, $\theta_c$
output: all pairs $(\tau_i, \tau_j)$ s.t. $d(\tau_i, \tau_j) \leq \theta$
    $\mathcal{R}_m, \mathcal{R}_s \leftarrow$ split($\mathcal{R}_m$)
    $\mathcal{R}_p \leftarrow$ get_partial_results($\mathcal{R}_m$, $\theta$, $\theta_c$)
    $\mathcal{R}_j, \mathcal{R}_m \leftarrow$ prepare_for_join($\mathcal{R}_j$, $\mathcal{R}_m$)
    $\mathcal{R}_{R_j \bowtie R_m} \leftarrow$ join($C_m, \mathcal{R}_j, \mathcal{R}_j$)
    $\mathcal{R}_{m,c} \leftarrow$ get_partial_results($\mathcal{R}_{R_j \bowtie R_m}$, $\theta$, $\theta_c$)
    $\mathcal{R}_{R_j \bowtie R_m} \leftarrow$ prepare_for_join($\mathcal{R}_{R_j \bowtie R_m}$ )
    $\mathcal{R}_{(R_j \bowtie R_m) \bowtie R_j} \leftarrow$ join($\mathcal{R}_{R_j \bowtie R_m}, C$)
    $\mathcal{R}_{m,c}, \mathcal{R}_{m,m} \leftarrow$ get_partial_results($\mathcal{R}_{(R_j \bowtie R_m) \bowtie R_j}, \theta, \theta_c$)
return distinct($\mathcal{R}_p \cup \mathcal{R}_s \cup \mathcal{R}_{m,c} \cup \mathcal{R}_{m,m}$)
```

Algorithm 2: Computation of the final result set.

i.e., pairs $(c_i, c_j) | \theta_c < d(c_i, c_j) \leq \theta$, can already be included to the final results set.

Candidate pairs, where at least one centroid is not a singleton, $\mathcal{R}_m$, need to be further joined with the set of clusters $\mathcal{R}_c$, in order to find the result pairs where at least one ranking is a cluster member. These pairs are missing from $\mathcal{R}_j$, since in the joining phase the join was performed only over the centroids. To do this, first the set of clusters and the set $\mathcal{R}_j$ are transformed, so that they are brought into a format where as key we have the centroids. Next, $\mathcal{R}_m$ and $\mathcal{R}_c$ are joined into $\mathcal{R}_{R_c \bowtie R_m}$. Then, $\mathcal{R}_{R_c \bowtie R_m}$ is used to generate the following result pairs:

$$\mathcal{R}_{m,c} = \{(\tau_i, c_j) \mid (\tau_i, c_j) \leq \theta \wedge \tau_i \prec c_i \wedge (c_i, c_j) \in \mathcal{R}_j\}$$

$$\mathcal{R}_{m,m} = \{(\tau_i, \tau_j) \mid (\tau_i, \tau_j) \leq \theta \wedge \tau_i \prec c_i \wedge \tau_j \prec c_j \wedge (c_i, c_j) \in \mathcal{R}_j\}$$

To generate the first result set $\mathcal{R}_{m,c}$, the candidate tuples in $\mathcal{R}_{R_c \bowtie R_m}$ need to be transformed into the needed pairs and further verified, if their distance is in fact smaller then θ. For pairs (τ_i, c_j), where $\tau_i \prec c_i$, we already know $d(\tau_i, c_i)$ and $d(c_i, c_j)$. Thus, using the triangle inequality, we verify only those candidate pairs (τ_i, c_j) such that $|d(c_i, c_j) - d(\tau_i, c_i)| \leq \theta$ and the remaining ones we can filter out since we can be certain that their distance is larger than θ.

For generating the set $\mathcal{R}_{m,m}$, the set $\mathcal{R}_{R_c \bowtie R_m}$ is first transformed, so that the second centroid is set as key of the tuples, and then it is joined with the set of clusters. The joined set is then used to add pairs to the set $\mathcal{R}_{m,c}$. These will be candidate pairs from the members of the newly joined centroids to the centroids we already had in $\mathcal{R}_{R_c \bowtie R_m}$. Filtering based on the triangle inequality is applied here as well. As last step, we generate all candidate pairs (τ_i, τ_j), such that $\tau_i \prec c_i$, $\tau_j \prec c_j$ and $d(c_i, c_j) \leq \theta + 2 * \theta_c$. For these, the Footrule distance is computed, and the ones where $d(\tau_i, \tau_j) \leq \theta$ are written to disk. Before writing the results to disc, the duplicates are removed.

Example 5.4. Figure 4 illustrates the expansion through an example. As results from the clustering phase, we have tuples (τ_1, τ_5), (τ_1, τ_2), and (τ_3, τ_4). The centroids of these clusters are τ_1 and τ_3—the clusters are the same as in the aforementioned example. The join results $\mathcal{R}_j$ are split into $\mathcal{R}_s = \{(\tau_7, \tau_9), (\tau_9, \tau_{12})\}$, where none of the rankings in the pairs is a centroid and $\mathcal{R}_m = \{(\tau_1, \tau_8), (\tau_1, \tau_6), (\tau_1, \tau_3)\}$, where at least one ranking in the pair is a centroid. Pairs in $\mathcal{R}_s$ are directly written to disk. Tuples in $\mathcal{R}_c$ and $\mathcal{R}_m$ are transformed such that the centroids, τ_1 and τ_3 are placed as keys of the tuples. They are joined and $\mathcal{R}_m, c$ =

$\{(\tau_5, \tau_8), (\tau_5, \tau_6), (\tau_2, \tau_8), (\tau_2, \tau_6), (\tau_2, \tau_3), (\tau_5, \tau_3)\}$ are verified. Then we take only those pairs in $\mathcal{R}_{R_c \bowtie R_m}$ where two rankings are centroids, in the example the last two elements of $\mathcal{R}_{R_c \bowtie R_m}$. For these, we switch the places of the centroids τ_1 and τ_3 so that the members of the second cluster could be joined with members of the first cluster, (τ_4, τ_5) and (τ_4, τ_2). These pairs need to be verified if their distance is smaller than θ.

6 REPARTITIONING USING JOINS

Naturally, the way data is distributed across partitions/machines greatly influences the performance of distributed algorithms. The VJ algorithm partitions the rankings based on the items that they contain—rankings that share an item end up at the same partition. This means that in the case of a skewed data distribution, which is often the case for real-world data, items that appear very frequently cause very large partitions. This problem is partially solved by the prefix filtering framework, especially for smaller values of θ, since the most frequent items would not be included. However, as we increase the value of θ, the size of the prefix increases, leading to skewed a distribution of data across the partitions, thus, having few partitions that dominate the overall execution time of the algorithm.

To tackle this issue, we propose an algorithm where large partitions are split into smaller sub-partitions. Then, the resulting pairs are computed for each small partition, and for each pair of sub-partitions. Algorithm 3 describes this procedure. First, using a user defined partitioning threshold δ we divide the inverted index into two parts, one where the partitions per item have more that δ rankings, $\mathcal{I}_{>\delta}$, and those whose partitions per item are smaller then the partitioning threshold, δ, $\mathcal{I}_{<\delta}$. In Spark, this can be easily computed, since the distributed inverted index is kept in one RDD, which allows easy access to the sizes of each partition. For those partitions that are smaller than the partitioning threshold, we compute the similarity join as before. The partitions larger than the partitioning threshold, $\mathcal{I}_{>\delta}$, are first split into smaller sub-partitions with at most δ rankings. This is done by assigning to each sub-partition a random number as a secondary key. To compute the final result set, we first compute the similarity join over each sub-partition. Then, we self join the sub-partitions by the item id, and for those join results where the secondary key of the first join pair is smaller than the secondary key of the second join pair, we execute a R-S similarity join algorithm for the joined partitions. To better handle the increased load due to data replication and to redistribute the working load equally among nodes, we partition by both the primary and secondary key, i.e., by both the item id and the randomly assigned number and increase the number of partitions.

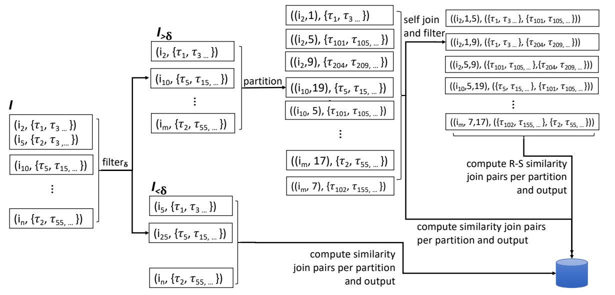
Example 6.1. Figure 5 illustrates through an example the similarity join computation in case of repartitioning. In this example, the posting list for items $i_2, i_{10}, \dots i_m$ have size larger than δ and thus are split into smaller partitions. For instance, the posting list for item i_2 is split into three smaller lists with keys $(i_1, 1)$, $(i_1, 5)$, and $(i_1, 9)$. In order to keep the correctness of the algorithms, in addition to generating the pairs for each of these lists, they are self joined, and an R-S join algorithm over the joined posting lists (with keys $(i_1, 1, 5)$, $(i_1, 1, 9)$, and $(i_1, 5, 9)$) is performed.

Choosing the Partitioning Threshold δ. In our experiments, we show that the performance of the algorithm does not significantly vary, when changing the partitioning threshold δ. However, the partitioning threshold still needs to be chosen carefully, such that it is not set to a very small value, leading to too many partitions being split into many small sub-partitions. If this happens then joining the sub-partitions in step 5 of Algorithm 3 becomes too expensive, and the benefit of the repartitioning is lost. In addition, due to the use of Spark joins, choosing a very small value of δ can also lead to memory crashes of the executors.

As a general guidance for choosing the value of the parameter δ an estimation for the size of the posting lists can be used. In our previous work on similarity search for top-k rankings [18], we devised a formula for estimating this:

$$E[\mathit{index\ list\ length}] = \sum_i n * f(i; s, v')^2 \tag{4}$$

where n is the number of rankings indexed, and $f(i; s, v')$ is the frequency of the item at rank i, when the items follow a Zipf's distribution with skewness parameter s. v' is the distinct number of items in the prefix of the rankings.



I
(i2, {τ1, τ3 … })
(i5, {τ2, τ3 ,… })
(i10, {τ5, τ15, … })
(in, {τ2, τ55, … })
filterδ
I>δ
(i2, {τ1, τ3 … })
(i10, {τ5, τ15, … })
(im, {τ2, τ55, … })
partition
((i2,1), {τ1, τ3 … })
((i2,5), {τ101, τ105, … })
((i2,9), {τ204, τ209, … })
((i10,19), {τ5, τ15, … })
((i10, 5), {τ101, τ105, … })
((im, 17), {τ2, τ55, … })
((im, 7), {τ102, τ155, … })
self join and filter
((i2,1,5), ({τ1, τ3 … }, {τ101, τ105, … }))
((i2,1,9), ({τ1, τ3 … }, {τ204, τ209, … }))
((i2,5,9), ({τ101, τ105, … },{τ204, τ209, … }))
((i10,5,19), ({τ5, τ15, … }, {τ101, τ105, … }))
((im, 7,17), ({τ102, τ155, … }, {τ2, τ55, … }))
compute R-S similarity join pairs per partition and output
compute similarity join pairs per partition and output
I<δ
(i5, {τ1, τ3 … })
(i25, {τ5, τ15, … })
(in, {τ2, τ55, … })
compute similarity join pairs per partition and output

Figure 5: Example of repartitioning of the large partitions using a partitioning threshold δ.

```
method: Repartitioning
input: inverted index over $\mathcal{D}$, $\mathcal{I}$. partitioning threshold, $\delta$
output: all pairs $(\tau_i, \tau_j)$ s.t. $d(c_i, c_j) \leq \theta$
1  $\mathcal{I}_{>\delta}, \mathcal{I}_{<\delta}$ = split($\mathcal{I}$,$\delta$)
2  $\mathcal{R}_{<\delta}$ = compute_sim($\mathcal{I}_{<\delta}$,$\theta$,$k$)
3  $\mathcal{P} \leftarrow$ repartition($\mathcal{I}_{>\delta}$, $\delta$)
4  $\mathcal{R}_{p1} \leftarrow$ compute_sim($\mathcal{P}$,$\theta$,$k$)
5  $\mathcal{R}_{p2} \leftarrow$ compute_sim(join($\mathcal{P}$,$\mathcal{P}$),$\theta$,$k$)
return $\mathcal{R}_{<\delta} \cup \mathcal{R}_{p1} \cup \mathcal{R}_{p2}$
```

Algorithm 3: Computing the all pair similarity join with repartitioning of large partitions using a partitioning threshold δ.

7 EXPERIMENTS

We deployed all algorithms on a Spark 1.6 (using YARN and HDFS) cluster running Ubuntu 14.04.5 LTS. The cluster consists of 8 nodes, each equipped with two Xeon E5-2603@ 1.6GHz/ 1.7GHz of 6 cores each, 128GB of RAM, out of which 40GB is reserved for execution of jobs by YARN, and 4TB hard disks. All nodes are connected via a 10GBit Ethernet connection.

Datasets: Due to the lack of real top-k ranking datasets, for the experiments we used datasets that are often used in previous work on similarity joins for sets and were also used for performing the experimental study for distributed similarity join algorithms [10]. Specifically, we use the DBLP [1] and ORKU [2] datasets. To transform the records of these dataset into top-k

spark.driver.memory	12G
spark.executor.memory	8GB
spark.executor.instances	24
spark.executor.cores	5

Table 3: Spark parameters used for the evaluation

rankings, we simply take the first k tokens in the sets, and consider them as items in the rankings. Since we are working with rankings of same size, we remove records with size smaller than k. In addition, the datasets are preprocessed as in [10], without the sorting of the records. Note that, while in the preprocessing step duplicates are removed from the dataset, since we cut the records to size k it can happen that we have a small amount of records with distance 0 to each other. However, this should *not* affect the performance of the algorithms, since duplicate records are *not* handled differently, i.e., the performance of the algorithms should be the same as if there are no duplicates. As we will show later on in our experimental study, what affects the performance of our algorithm is the number of records with distance smaller than θ_c.

After the preprocessing the DBLP dataset has approximately 1.2 million top-10 rankings, and ORKU has approximately 2 million top-10 rankings. Each datasets has a size of 67MB and 173MB. Since these datasets are relatively small for a distributed setting, we also increase their size using the same method as in [10, 24], where the domain of the items remains the same, and the join result increases approximately linearly with the size of the dataset. We use suffix xn to denote the number of times the dataset has been increased. For instance, "ORKUx5" represents the ORKU datasets increased 5 times.

The files in Spark are read as text files, and are directly partitioned into the number of partitions specified at input. Throughout the experiments we write the number of partitions that the data is divided into. Additionally, we show experiments that illustrate the behavior of the effect that the number of partitions has to the performance of the algorithms.

Algorithms under investigation We investigate the performance of the following algorithms:

- The adaptation of VJ to top-k rankings in Spark (VJ)
- The adaptation of VJ to top-k rankings using iterators instead of inverted index (VJ-NL)
- The clustering algorithm using iterators (CL)
- The clustering algorithm with iterators and re-partitioning of the data (CL-P)

Based on general recommendations for running Spark jobs, which suggest to not run 'tiny' or 'fat' executors, we assign 5 cores per executor. Then, based on the total number of cores and the available memory of the nodes in the cluster, we set the other execution parameters, reported in Table 3. The memory assigned to the executers also corresponds to the amount assigned to the reducers in a previous experimental study [10]. In case we use different settings, we write these changes for the specific experiments. We report on the average wall-clock time measured in seconds over 3 runs. If an algorithm runs more than 10 hours we stop its execution.

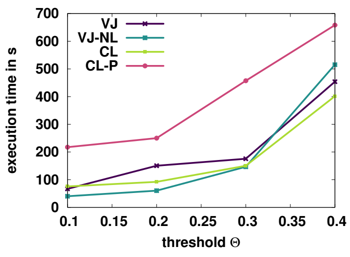
7.1 Results

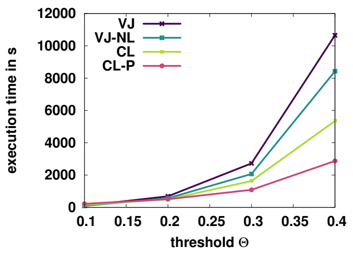
Performance Based on the Distance Threshold θ. We first evaluate and compare the performance of the above listed algorithms when we vary the distance threshold θ. Figure 6 reports on the performance of the four algorithms for both datasets DBLP and ORKU, for values of θ ranging from 0.1 to 0.4. We see that our algorithm outperforms the competitor algorithm VJ for larger values of θ. Most importantly, we see that, with the exception of the DBLP dataset, each optimization that we propose, brings additional performance improvement. For all algorithms, the execution time increases, as we increase the distance threshold θ, however, for our proposed algorithms, CL and CL-P, the increase in performance is smaller, especially for the latter. For instance, for the DBLPx5 dataset, the execution of the VJ algorithm for the largest threshold value, 0.4, is 100 times more expensive than when executing it for the smallest threshold value of 0.1. On the other hand, the increase in execution time for the CL and CL-P algorithms is 33 and 13 times, respectively. This can be attributed to the design of the CL algorithm. Since in the joining phase less rankings are being processed, the algorithm is not too much affected by the skewness of the dataset. With the partitioning of the large partitions into smaller ones, and their redistribution among the nodes in the cluster, the CL-P algorithm shows even larger performance improvement, for larger threshold values.

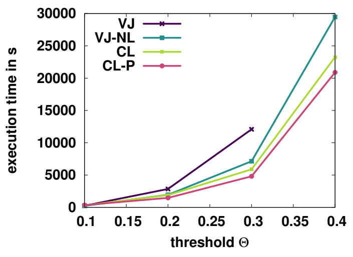
Furthermore, we see that for the datasets DBLPx5 (Figure 6(b)) and ORKU (Figure 6(d)) the gains in performance are the largest. Here, we can clearly see that using iterators over an inverted index is more efficient when it comes to a Spark implementation. Additionally, we see that the largest performance benefit from our clustering algorithm are for values of θ of 0.3 and 0.4. When θ is set to 0.4, clustering combined with partitioning based on joins (CL-P) performs 5 and 3 times better than the VJ and VJ-NL algorithms, respectively, for the ORKU dataset (Figure 6(d)). For the DBLPx5 dataset, the CL-P algorithm outperforms the VJ and VJ-NL algorithms by almost 4 and 3 times, respectively (Figure 6(b)). For lower values of the partitioning threshold, i.e., when $\theta = 0.1$ or $\theta = 0.2$, the CL and CL-P algorithms either perform slightly worse than the VJ or VJ-NL, or the gain in performance is not that large. This is especially true for $\theta = 0.1$. This is due to the fact that the VJ algorithm is very efficient for a very small thresholds, since the prefix size is then small. In these cases, the overhead from the additional clustering phase in the CL approach, or partitioning for the CL-P, is larger than the benefit that we could get from it.

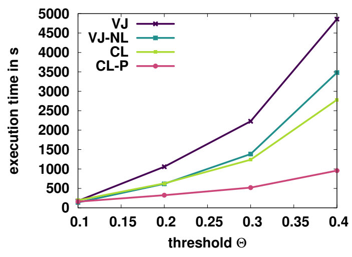
Note that in all cases, the clustering threshold for the CL and CL-P algorithms is set to 0.03. The reason for this is explained bellow, where we study the effect that this threshold has on the performance of the algorithms. The value of the partitioning threshold δ differs depending on the dataset, and the threshold value, θ. For larger thresholds θ, we choose larger partitioning threshold δ, since we expect an increase in the size of the posting lists. Later we discuss how choosing the partitioning threshold δ affects the performance. For the smallest dataset, DBLP (Figure 6(a)), where the original VJ algorithm is already very efficient, the proposed optimizations lead to worse performance. The CL-P algorithm in this case always performs worse than VJ, since it brings additional overhead of repartitioning and joining already small posting lists. The CL algorithm outperforms VJ only for large values of θ. On the other hand, for the ORKUx5 dataset (Figure 6(e)), for $\theta = 0.4$, only the CL-P algorithm finished under 10 hours. Similarly, for the DBLPx10 dataset (Figure 6(c)), the VJ algorithm did not finish under 10 hours.

Scalability. To test the scalability of the proposed algorithm, we varied the number of nodes in our cluster. We executed the CL-P algorithm on a cluster with 4 nodes and with 8 nodes. For


(a) DBLP


(b) DBLPx5


(c) DBLPx10


(d) ORKU

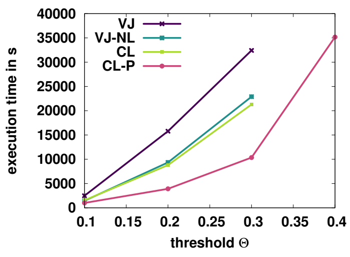
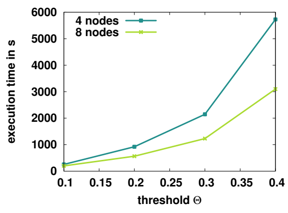

(e) ORKUx5

Figure 6: Comparison of different algorithms when varying θ


(a) DBLPx5

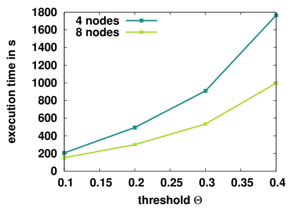

(b) ORKU

Figure 7: Performance of CL-PL algorithm when varying the number of nodes in the cluster (DBLPx5 and ORKU).

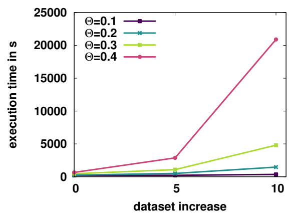


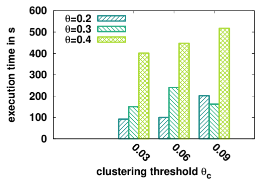
Figure 8: Performance of CL-P algorithm when varying the dataset size for the DBLP dataset.

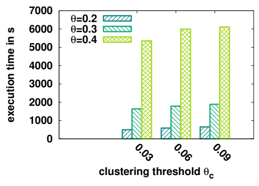
this experiment, we reduced the number of cores per executor to 3, and we did not fix the number of executors to be used, i.e., this was left to be decided by YARN, based on the cluster size. The memory restriction per executor and for the driver were kept as specified in Table 3. Figure 7 shows the performance of the CL-P algorithms for different values of the theshold θ, for the DBLPx5 and ORKU datasets. The values for the clustering threshold θ_c and the partitioning threshold δ were kept the same as for the previous experiment. We see that for both datasets, the CL-P algorithm exhibits better performance, when the number of nodes is increased. For the DBLPx5 dataset, when increasing the number of nodes from 4 to 8, the time cost decreases from 22% to 46%, and for the ORKU dataset the time savings are similar, ranging from 26% to 44%. Again, the largest performance improvement is observed for $\theta = 0.4$.

Furthermore, in Figure 8 we plotted the performance of the CL-P algorithm as we increase the size of the DBLP dataset. Note that the result size increases approximately linearly with the increase in the number of records. The rise of the execution time is the largest, i.e., for $\theta = 0.4$, when we increase the dataset size from x5 to x10. In this case the CL-P algorithm executes 7 times slower. However, the reason for this we see in the value of the partitioning threshold δ. We believe that with a more carefully chosen value for δ this increase in the execution time can be avoided. For all other cases of θ the decrease in performance is lower than 5 times.

Effect of the Clustering Threshold θ_c. Another threshold that can have impact on the performance of the proposed clustering algorithm is the clustering threshold θ_c. Depending on the value of this threshold, the size and number of the formed clusters varies, and thus the performance of the whole algorithm. Figure 9 shows the performance of the CL algorithm for different values of θ_c for both datasets. We see that, in almost all cases, setting $\theta_c = 0.03$ brings the best performance for the CL algorithm. This can be explained by two reasons. First, as we increase the clustering threshold θ_c, the running time of the clustering phase increases, since here we use the VJ algorithm to find the similar pairs. Second, the benefit by the additionally formed clusters does not seem to compensate for this increase in the running time. Thus, setting the clustering threshold θ_c to a very small value is the recommend choice, and in all further experiments we set θ_c to 0.03 for both CL and CL-P.

Effect of the Partitioning Threshold δ. The partitioning threshold δ is a parameter which decides which and how many posting lists need to be partitioned, and as such, it influences the performance of the CL-P algorithm. In Figure 10 we see the performance of the CL-P algorithm as the partitioning threshold changes, for both datasets DBLP and ORKU and for different values of the threshold θ. For the DBLP dataset we show only the DBLPx5 increased dataset, since, as we showed in Figure 6(a), the DBLP dataset is small and does not benefit from the partitioning of the posting lists. For each dataset, we chose different varying


(a) DBLP


(b) DBLPx5

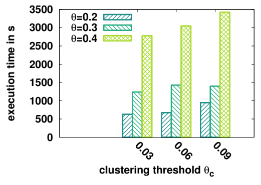
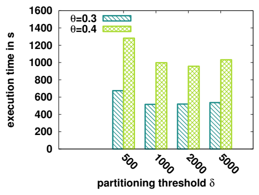
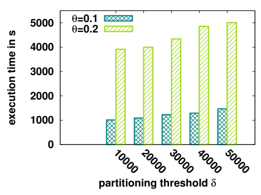

(c) ORKU

Figure 9: Performance of CL algorithm when varying the clustering threshold θ_c


(a) ORKU


(b) ORKUx5

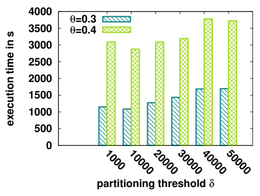

(c) DBLPx5

Figure 10: Performance of CL-P algorithm when varying the partitioning threshold δ

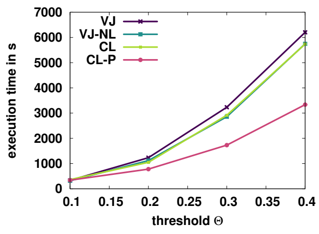
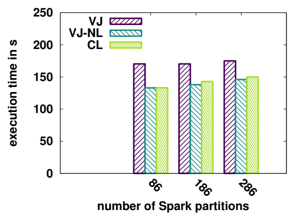


Figure 11: Performance of different algorithms for rankings of size 25 when varying the distance threshold θ (ORKU)

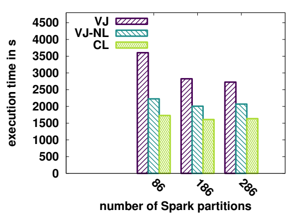
ranges for the partitioning threshold, since its value is directly dependent from the size of the dataset. For ORKU (Figure 10(a)) we vary δ from 500 to 5000, for ORKUx5 (Figure 10(b)) we vary δ from 10000 to 50000 and for DBLPx5 (Figure 10(c)) we vary δ from 1000 to 50000. Furthermore, for ORKU and DBLPx5 we plot the performance for $\theta = 0.3$ and $\theta = 0.4$ (Figures 10(c) and 10(a), respectively), while for ORKUx5, for practical reasons, due to the large execution times when having large values of θ, we plot the performance for $\theta = 0.1$ and $\theta = 0.2$ (Figure 10(b)). In Figure 6(a) we see that the performance of CL-P is not widely influenced by the partitioning threshold δ. Starting with small values of δ, the performance is slightly worse, due to the larger number of posting lists that need to be joined, and thus the overhead imposed by the Spark join is larger. Then, as we increase δ, the performance at first drops and reaches its minimum, and then starts to slightly increase. This is important to note, since it gives us more freedom of choosing the value for δ. Note, however, that choosing very small values can lead to either bad performance or crashes of the executors due to memory overhead caused by the joins. During our experiments execution, we experienced crashes due to memory overhead, whenever the δ value was set to an inappropriately small value, when considering the number of records being processed. On the other hand, setting δ to a very large value will not bring any performance benefit, since no postings lists will be partitioned.

Increasing the size of the rankings. Top-k rankings usually contain only very few items. In fact in our study [3] we showed that most of the rankings are of size 10 or 20. Therefore, in the previous experiments we focused on rankings of size 10. To see how the performance of the algorithms changes, when we have rankings of larger size, we also run experiments where $k = 25$. For this purpose we used the ORKU dataset, which contains also longer records. From the original dataset, we extracted around 1.5 million top-25 rankings, as described above. This dataset has a size of 289MB. The DBLP dataset contained only shorter records, and thus for this experiment we rely only on the ORKU dataset. Figure 11 shows the performance of the four algorithms when varying the distance threshold θ. While our algorithms still outperform the VJ algorithm, there are two important things to note here. First, the difference in the performance between VJ-NL and VJ is not so significant, and second CL performs almost the same as VJ-NL. This might be explained with the size of the dataset, since our clustering algorithms, CL and CL-P, perform better on larger datasets. The CL-P algorithm shows the best performance, except for $\theta = 0.1$, and is, as with rankings of size 10, least susceptible to the increase of the threshold θ. For $\theta = 0.1$, the VJ-NL algorithm performs slightly better than the other algorithms. The CL-P algorithm outperforms the VJ-NL algorithm for 1.5 and 1.9 times for $\theta = 0.2$, and $\theta = 0.3$ and 0.4, respectively. Note that for this experiment, for both CL and CL-P, we set $\theta_c = 0.03$ and the partitioning threshold, δ, for CL-P, we set to 5000, for all values of θ.

Varying the number of Spark partitions. The general recommendation when executing Spark jobs is to set the number of partitions to be at least four times as the number of executors running. In our setting, this means that the general recommendation is to have at least 100 partitions. Figure 12 shows the performance of different algorithms (VJ, VJ-NL and CL) for different number of partitions. For this experiments the partitioning threshold θ is fixed to 0.3. We see that for both DBLP and DBLPx5, the performance does not change much as we increase the number of partitions. In fact, we see that whether the performance increases or decreases—as we increase the number of partitions—depends on the size of the dataset. For the smaller dataset, DBLP, the best performance is observed when the number of partitions is set to 86, and then the performance slightly decreases. For DBLPx5, on the other hand, we have the best performance of both CL and VJ-NL for 186 partitions. Figure 13 shows the performance of the CL-P algorithm when changing the number of partitions. For CL-P we used a larger span of the number of partitions, from



(a) DBLP



(b) DBLPx5

Figure 12: Performance of VJ, VJ-NL and CL when varying the number of Spark partitions, $\theta = 0.3$ (DBLP and DBLPx5).

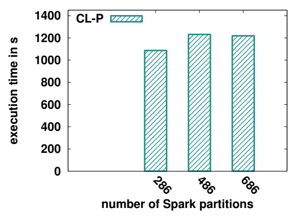


Figure 13: Performance of CL-P when varying the number of Spark partitions, $\delta = 10000$, $\theta = 0.3$ (DBLPx5).

286 to 686. Since here we additionally repartition the large partitioning into smaller ones, we believe that using a larger number of partitions is more appropriate for this approach However, as we can see from Figure 13, the performance is again not greatly influenced by the change of the number of the partitions. In fact, there is also a slight drop in the performance in the initial increase in the number of partitions, from 286 to 486. In all of the experiments presented before, the number of partitions was set to 286.

Lessons Learned. The proposed clustering algorithms, CL and CL-P, outperform the adaptation of the state-of-the-art algorithm for similarity joins over sets, VJ, for higher values of the distance threshold θ. For small values of θ the VJ algorithm is very efficient on its own, and thus, the benefits introduced by the additional stages of the CL approach, do not seem to pay off. This is also the case for small datasets. However, more importantly, for larger datasets, the CL and CL-P approaches seem to bring larger performance improvements over the VJ algorithm. Additionally, they both seem to be less susceptible to the increase of the distance threshold. This seems to be especially true for the CL-P algorithm, in particular, when the partitioning threshold is chosen right. Furthermore, our approach is more appropriate for handling datasets with skewed distribution, as first, the dataset is reduced for the joining phase, and second, large posting lists are split into smaller ones and processed in parallel. For choosing the partitioning threshold δ, statistics like the number of records in the dataset, and the size of the vocabulary, or item domain, can be used, as discussed in Section 3. For choosing the clustering threshold, as a rule of thumb, we suggest choosing a very small value, namely to set θ_c to be smaller than 0.05. A drawback of our solution is that, since we rely on Spark joins, it can run out of memory, especially where the result set is large.

8 CONCLUSION AND OUTLOOK

In this paper, we addressed distributed similarity join processing techniques for a datasets of top-k rankings. As a distance for comparing the rankings, we specifically considered Spearman's Footrule adaptation to top-k rankings. The presented approach synthesizes existing state-of-the-art, set-based, distributed similarity join algorithm with the advantages of metric-space, distance-based, filtering. It works in several stages, where each can be independently configured from each other. Furthermore, our algorithms are designed and implemented in Apache Spark, as suggested by a recent experimental study. By a comprehensive performance evaluation using two real-world datasets, we showed that the presented approach exhibits better performance than the competitor, Vernica Join. In the future, we plan to extend our approach to sets where the Jaccard distance is used as a distance measure.

REFERENCES

[1] DBLP Dataset. http://dbgroup.cs.tsinghua.edu.cn/wangjn/projects/adapt/. Accessed: 26.03.2018.
[2] ORKU Dataset. http://ssjoin.dbresearch.uni-salzburg.at/datasets.html. Accessed: 01.12.2018.
[3] F. Alvanaki, E. Ilieva, S. Michel, and A. Stupar. Interesting event detection through hall of fame rankings. In *DBSocial*, pages 7–12, 2013.
[4] Apache Spark [n.d.]. https://spark.apache.org. Accessed: 26.03.2019.
[5] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In WWW 2007, Banff, Alberta, Canada, pages 131–140..
[6] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. Spatio-textual similarity joins. *PVLDB* 6, 1 (2012), 1–12.
[7] S. Chaudhuri, V. Ganti, and R. Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. In ICDE 2006, Atlanta, GA, USA, page 5, 2006.
[8] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. MassJoin: A mapreduce-based method for scalable string similarity joins. In ICDE 2014, IL, USA, pages 340–351, 2014.
[9] R. Fagin, R. Kumar, and D. Sivakumar. Comparing Top k Lists. *SIAM J. Discrete Math.*, 17(1):134–160, 2003.
[10] F. Fier, N. Augsten, P. Bouros, U. Leser, and J. Freytag. Set Similarity Joins on MapReduce: An Experimental Survey. *PVLDB*, 11(10):1110–1122, 2018.
[11] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate String Joins in a Database (Almost) for Free. In *VLDB 2001, Roma, Italy*, pages 491–500.
[12] E. H. Jacox and H. Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2), 2008.
[13] Y. Jiang, G. Li, J. Feng, and W. Li. String Similarity Joins: An Experimental Evaluation. *PVLDB*, 7(8):625–636, 2014.
[14] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. *Learning Spark: Lightning-Fast Big Data Analytics.* 1st edition, 2015.
[15] G. Li, D. Deng, J. Wang, and J. Feng. PASS-JOIN: A Partition-based Method for Similarity Joins. *PVLDB*, 5(3):253–264, 2011.
[16] W. Mann, N. Augsten, and P. Bouros. An Empirical Evaluation of Set Similarity Join Techniques. *PVLDB*, 9(9):636–647, 2016.
[17] A. Metwally and C. Faloutsos. V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *PVLDB*, 5(8):704–715, 2012.
[18] E. Milchevski, A. Anand, and S. Michel. The Sweet Spot between Inverted Indices and Metric-Space Indexing for Top-K-List Similarity Search. In *EDBT 2015, Brussels, Belgium.*, pages 253–264.
[19] K. Panev, E. Milchevski, and S. Michel. Computing similar entity rankings via reverse engineering of top-k database queries. In ICDE Workshops 2016, Helsinki, Finland, May 16-20, 2016, pages 181–188.
[20] C. Rong, C. Lin, Y. N. Silva, J. Wang, W. Lu, and X. Du. Fast and Scalable Distributed Set Similarity Joins for Big Data Analytics. In *ICDE 2017, San Diego, CA, USA*, pages 1059–1070.
[21] C. Rong, W. Lu, X. Wang, X. Du, Y. Chen, and A. K. H. Tung. Efficient and Scalable Processing of String Similarity Join. *IEEE Trans. Knowl. Data Eng.*, 25(10):2217–2230, 2013.
[22] A. D. Sarma, Y. He, and S. Chaudhuri. ClusterJoin: A Similarity Joins Framework using Map-Reduce. *PVLDB*, 7(12):1059–1070, 2014.
[23] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. *PVLDB*, 8(13):2110–2121, 2015.
[24] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD 2010, Indianapolis, IN, USA, pages 495–506.*
[25] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In SIGMOD 2012, Scottsdale, AZ, USA, pages 85–96.
[26] X. Wang, L. Qin, X. Lin, Y. Zhang, and L. Chang. Leveraging Set Relations in Exact Set Similarity Join. *PVLDB*, 10(9):925–936, 2017.
[27] Y. Wang, A. Metwally, and S. Parthasarathy. Scalable all-pairs similarity search in metric spaces. *KDD 2013, Chicago, IL, USA*, pages 829–837.
[28] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW 2008, Beijing, China*, pages 131–140.

MV-PBT: Multi-Version Indexing for Large Datasets and HTAP Workloads

Christian Riegger
Data Management Lab,
Reutlingen University, Germany
christian.riegger@reutlingen-university.de

Tobias Vinçon
Data Management Lab,
Reutlingen University, Germany
tobias.vincon@reutlingen-university.de

Robert Gottstein
Data Management Lab,
Reutlingen University, Germany
robert.gottstein@reutlingen-university.de

Ilia Petrov
Data Management Lab,
Reutlingen University, Germany
ilia.petrov@reutlingen-university.de

ABSTRACT

Modern mixed (HTAP) workloads execute fast update-transactions and long-running analytical queries on the same dataset and system. In multi-version (MVCC) systems, such workloads result in many short-lived versions and long version-chains as well as in increased and frequent maintenance overhead.

Consequently, the *index pressure* increases significantly. Firstly, the frequent modifications cause frequent creation of new versions, yielding a surge in index maintenance overhead. Secondly and more importantly, index-scans incur extra I/O overhead to determine, which of the resulting tuple-versions are visible to the executing transaction (visibility-check) as current designs only store version/timestamp information in the base table – not in the index. Such *index-only visibility-check* is critical for HTAP workloads on large datasets.

In this paper we propose the *Multi-Version Partitioned B-Tree (MV-PBT)* as a version-aware index structure, supporting index-only visibility checks and flash-friendly I/O patterns. The experimental evaluation indicates a 2x improvement for analytical queries and 15% higher transactional throughput under HTAP workloads. MV-PBT offers 40% higher tx. throughput compared to WiredTiger's LSM-Tree implementation under YCSB.

1 INTRODUCTION

The spread of large-scale, data-intensive, real-time analytical applications is increasing. Such applications result in Hybrid Transactional and Analytical Processing workloads (*HTAP*) combining long running analytical queries (OLAP) as well as frequent and low-latency update transactions (OLTP) on the same dataset and even on the same system [19].

Multi-versioning is at the core of many approaches and system designs suitable for HTAP. Under *Multi-Version Concurrency Control (MVCC)* reading transactions, executing long-running queries, do not block the frequent low-latency modifying transactions. Under such approaches multiple versions of each data item (i.e. tuple) may physically co-exist, whereas every transaction operates against a snapshot of the database comprising all versions it is allowed to see for consistent execution. Read operations simply operate on the latest committed version, visible to them and are therefore never blocked, yielding good read performance and concurrency. An update operation produces a new version of the updated data item and invalidates the predecessor version.

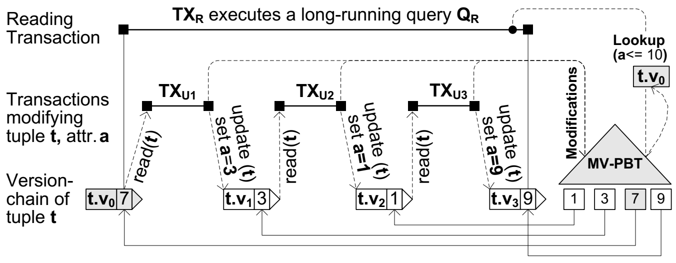


Figure 1: HTAP and Version-Chain Lengths: $TX_{U1} \ldots TX_{U3}$ **create new versions of tuple** ***t*****, which are indexed. The index scan of** TX_R **returns only the index entries (**$t.v_0$**) visible to** TX_R **filtering the invisible ones (**$t.v_1 \ldots t.v_3$**), matching the search predicate.**

All versions of a tuple form a version-chain. Timestamps placed on every physical version-record are used to determine, which of the exisiting tuple-versions is *visible* to a transaction.

Under OLTP workloads, version-chains tend to be short, due to the predominantly short-lived transactions. For instance, under TPC-C the average version-chain length is approx. 1.2 [9]. *Under HTAP the DBMS needs to handle much longer version-chains due to the mix of long-running and short-lived transactions (Figure 1).* Whenever a transaction TX_R reads a tuple *t* the DBMS returns the latest version of that tuple $t.v_0$, committed before the start of TX_R. Even though, in the meantime multiple low-latency updating transactions $TX_{U1} \ldots TX_{U3}$ might have committed, producing successor-versions ($t.v_1 \ldots t.v_3$), $t.v_0$ cannot be garbage collected as long as, it is visible to an active transaction, i.e. TX_R. Thus, the amount of such transient versions can be as high as *several hundred millions* in real systems [14].

HTAP workloads in combination with long version-chains exercise significant pressure on indices. In a single-versioned system there is one index entry per tuple. However, in a multi-versioned system, the DBMS needs to index at least all committed tuple-versions (Figure 1), even the transient ones. Thus, long version-chains put extra pressure on the index. *Although most of today's systems are multi-versioned, the majority of index approaches still handle tuple-versions of the same tuple as if they were separate tuples, ignoring the version semantics.* If naïvely integrated, these slow down index lookups and may cause significant maintenance overhead to persistent indices, as index updates are very frequent and since index entries corresponding to obsolete tuple-versions need to be frequently garbage collected. Given the read/write asymmetry of modern persistent storage technologies these operations result in prohibitively expensive in-place updates. In this context append-based index structures trading sequential writes for complex reads are a good candidate.

All in all, the following observations can be made:
1) *Version-obliviousness:* Although, all tuple-versions need to be indexed, current indexing approaches lack version information.
2) *Lack of index-only visibility-checks:* It is currently impossible to determine, which of the index-entries resulting from an index lookup/scan correspond to versions, visible to the calling transaction solely based on the index.
3) *I/O overhead:* Version-oblivious indices or naïve support for multi-versioning yield significant I/O overhead.

In the present paper we propose the *Multi-Version Partitioned B-Tree (MV-PBT)* as a version-aware index structure for MV-DBMS, in an attempt to address the above issues. MV-PBT is based on a variant of B$^+$-Trees called *Partitioned B-Trees* [13]. The contributions of this paper are:

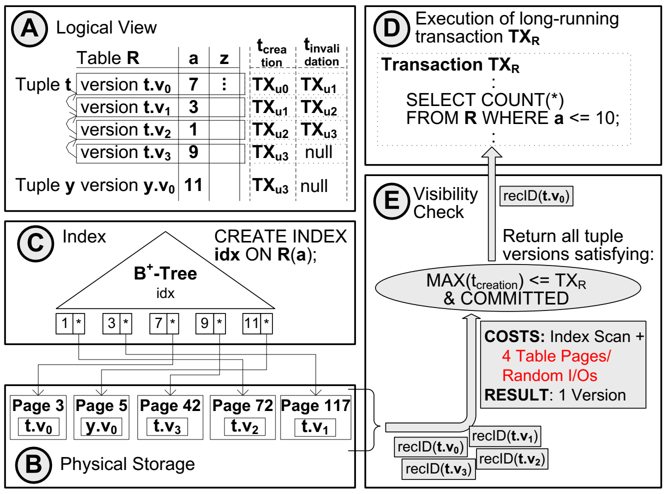
- MV-PBT is a version-aware index structure. It contains version information and supports index-only visibility-checks.
- MV-PBT supports append-based write-behavior and exhibits much lower write-amplification compared to LSM-Trees.
- MV-PBT has been implemented in PostgreSQL. The performance evaluation under HTAP workloads (CH-Benchmark [2]) indicates 2x analytical throughput improvement due to index-only visibility-checks, while improving the transactional throughput by 15% compared to PostgreSQL's highly-optimized B$^+$-Tree. Under TPC-C MV-PBT performs 15% better.
- MV-PBT has also been implemented in WiredTiger (MongoDB). The performance evaluation indicates approx. 40% higher throughput under YSCB compared to WiredTiger's highly-optimized LSM-Trees.

The rest of the paper is organized as follows. We motivate the missing *version-awareness* and the need for *index-only visibility-checks* in Section 2, while Section 3 provides some background on various multi-versioning aspects. The design and implementation of MV-PBT is described in detail in Section 4, while the experimental evaluation is presented in Section 5. We. discuss related approaches in Section 6 and conclude in Section 7.

2 MOTIVATION

In this section we give a more comprehensive perspective on the above issues of: 1) *Version-obliviousness in indices*; 2) *missing index-only visibility-check*; and 3) *I/O overhead*. Consider the example in Figure 2, which is a more detailed version of Figure 1 with a conventional B$^+$-Tree. An initial transaction TX_{U0} (not depicted) inserts tuple t prior to TX_R, creating its initial version $t.v_0$. While TX_R is running, multiple concurrent transactions $TX_{U1} \dots TX_{U3}$ update tuple t and each of them produces new versions of it ($t.v_1 \dots t.v_3$). Only TX_{U3} inserts tuple y in its initial version $y.v_0$ in addition to creating $t.v_3$. Each tuple-version is a separate physical version record (Figure 2.A). It contains *version-information*: the *recordID* of the predecessor version and two timestamps, $t_{creation}$ - the timestamp of the transaction that created that tuple-version; and $t_{invladiation}$ the timestamp of the transaction that invalidated it by creating a successor version. The invalidation-timestamp is *null* if there is no successor. If a tuple gets deleted a special *tombstone* version-record is inserted to mark the logical end of the chain. *The version-information is only available on the version-record.*

Since version-records are independent physical entities they can be stored on any DB-page with enough free space. Figure 2.B depicts an example of the physical version-storage. *For consistency, an index on a table must contain index-entries for each committed version of every tuple.* Therefore, a B$^+$-Tree index *idx* on attribute *a* of table *R* (Figure 2.C) should reflect all versions of each tuple of *R*. *Since the index is version-oblivious it contains no version-information, and treats each tuple-version as if it were a separate tuple.* Consequently, if TX_R uses the index to count all tuples satisfying *"$a \le 10$"* (Figure 2.D), the index scan will return the matching index entries (referencing versions $t.v_0 \dots t.v_3$). Now, each one of them must be checked for visibility, i.e. is it latest committed tuple-version prior to the start of TX_R. However, the necessary timestamps are available only on the version-records. Therefore, all of them are retrieved, at the cost of random I/Os.



A Logical View
Table R
a
z
$t_{creation}$
$t_{invalidation}$
Tuple t
version t.v_0 7 ⋮ TX_{u0} TX_{u1}
version t.v_1 3 TX_{u1} TX_{u2}
version t.v_2 1 TX_{u2} TX_{u3}
version t.v_3 9 TX_{u3} null
Tuple y version y.v_0 11 TX_{u3} null
D Execution of long-running transaction TX_R
Transaction TX_R
SELECT COUNT(*) FROM R WHERE a <= 10;
C Index
CREATE INDEX idx ON R(a);
B$^+$-Tree idx
1 * 3 * 7 * 9 * 11 *
E Visibility Check
recID(t.v_0)
Return all tuple versions satisfying:
MAX($t_{creation}$) <= TX_R & COMMITTED
COSTS: Index Scan + 4 Table Pages/ Random I/Os
RESULT: 1 Version
Page 3 t.v_0
Page 5 y.v_0
Page 42 t.v_3
Page 72 t.v_2
Page 117 t.v_1
B Physical Storage
recID(t.v_1)
recID(t.v_0)
recID(t.v_2)
recID(t.v_3)

Figure 2: Index-Only Visibility-Check in Multi-Version DBMS: (a) logical tuples (t and y) of a table R and their versions; (b) the physical storage of these versions into database pages; (c) an index created over table R must index all versions; (d) an index-scan retrieves all versions matching the predicate, out of which (e) the visibility-check returns only the ones visible to calling transaction TX_R.

In our example (Figure 2.D, C and E), the index-scan for the condition *"$a \le 10$"* will return versions $t.v_3$, $t.v_0$, $t.v_1$ and $t.v_2$. Subsequently, they are read to extract the *version-information* ($t_{creation}$ and $t_{invalidation}$ – Figure 2.A) yielding four random I/Os. The visibility-check then determines the latest version committed prior to the start of TX_R, returning the recordID of $t.v_0$ and ignoring the rest. *Since the index is version-oblivious and thus does not support index-only visibility-checks, the I/O costs amount to: COST(Index-Scan) + 1 random I/O for each matching tuple-version. Especially for HTAP workloads this yields significant performance degradation depending on the length of the version-chains.*

To quantify the combined effect, we designed a simple experiment with *YCSB* [7] and PostgreSQL. We run YCSB workloads A (update) and E (scan) combined, performing frequent scans and updates. In parallel, we perform a point-query on a tuple every 30 seconds (simulating an HTAP workload). Additionally, we continuously increase the version-chain, by updating the tuple, until 50 versions are reached. In realistic HTAP settings, the amount of active versions can be as high as *several hundred millions*, while analyses can take as long as 1000s [14]. The experimental results

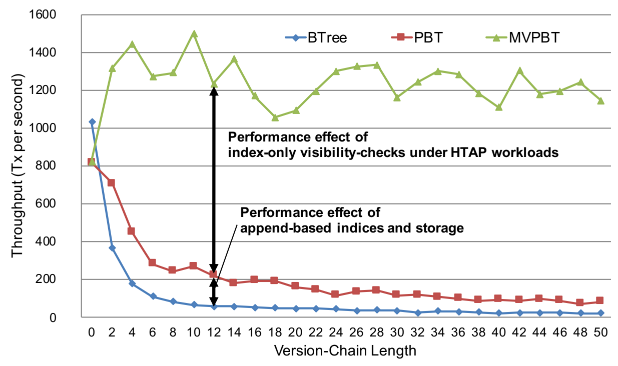


Figure 3: Performance Impact of Version Visibility Check.

are shown in Figure 3. The highly-optimized B$^+$-Tree implementation in PostgreSQL performs better than *MV-PBT* on a single tuple-version. However, as the version-chain length increases (6-8 versions) the performance drops rapidly to *approx. 50 transactions/sec*, due to version-obliviousness and random I/O. Basic Partitioned B-Trees (PBT), are likewise *version-oblivious*, but exhibit append-based write behaviour, avoiding in-place updates and perform therefore slightly better (*approx. 150 tx/sec*). Due to its *version-awareness* and support for *index-only visibility-check* MV-PBT exhibits much higher and robust performance (*approx. 1200 tx/sec*) with growing chain lengths. MV-PBT shows a performance increase with chain-lengths of two or more due to the partition buffer since: (a) the initial YCSB data load, producing the first version fills partition P_0 and evicts it; while (b) the second version is created by the benchmark workload and is in P_N.

3 BACKGROUND

Multi-Version Concurrency Control (MVCC) is one of the most popular transaction management schemes and is used in most modern DBMS: Oracle, Microsoft SQL Server, HyPer, SAP HANA, MongoDB WiredTiger, NuoDB, PostgreSQL or MySQL-InnoDB, just to name a few. These DBMS make different design decisions regarding various MVCC aspects described below.

3.1 Version Storage

Under MVCC a logical tuple corresponds to one or more tuple-versions (Figure 2.A). They form a singly linked list, which represents a version chain. There are two possible physical representations of a tuple-version (Figure 4): *physically materialized* or *delta-record based*. The former implies that each tuple-version record is stored physically materialized in its entirety and is in the focus of this paper. The latter implies that each modification of a logical tuple results in a delta-record, indicating the difference to another version (à la BW-Tree [15, 22]). The delta-records are connected and retrieved on demand by the DBMS storage manager to restore a tuple-version. *Delta-record based* system designs typically store a single version (oldest or newest) in the main store and use a separate store for the delta-records, which may be the undo log (à la InnoDB) or a temporary version store (à la MS SQL Server). Both organizations can perform modifications in-place or out-of-place. Out-of-place updates with *physically materialized* version-maintenance insert a new version-record in the base table. Based on the version ordering, additional modifications may be necessary to maintain logical timestamps or references.

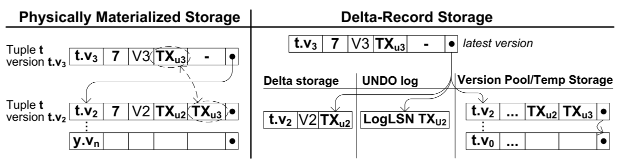


Figure 4: Version Storage Alternatives

Considering the characteristics of modern storage technologies, physically materialized version storage and out-of-place updates are preferable, due to lower write-amplification and the higher parallelism. Delta records tend to consume less space than materialized tuple-versions, but require additional processing and all predecessors or successors for tuple reconstruction.

3.2 Version Ordering

The set of tuple-versions of a database tuple is organized as a singly linked list. There are two different ordering methods (Figure 5): *old-to-new* and *new-to-old*.

Old-to-New ordering: The entry-point is the oldest tuple-version in version chain and each version contains a reference (recordID) to its successor. A visibility-check must therefore process all successors, beginning from the oldest tuple-version. This behavior is beneficial for lookups of long-running analytical (OLAP) queries under HTAP workloads, where older tuple-versions are likely to be the visible ones. Alternatively, OLTP workloads mostly require the newest version and would need to process the whole version chain. *New-to-Old ordering* implies that the entry-point is the newest tuple-version, which refers to its predecessor. Queries in the typically short OLTP transactions find the visible version very fast, but long-running OLAP queries may need to process several successors in version chain (Figure 3). *In-place* and *out-of-place* update strategies are are possible for both methods.

Considering the characteristics of modern storage technologies new-to-old ordering for physical version storage results in lower write-amplification and matches append-only storage. All other approaches require in-place updates.

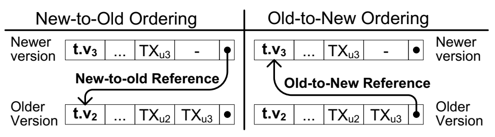


Figure 5: Version Ordering Alternatives

3.3 Version Invalidation Model

Under MVCC a version is said to be invalidated whenever a successor version exists. There are two possible invalidation models [9] (Figure 6). First, *two-point invalidation* is the state-of-the-art model, where the creation timestamp of the successor version is also placed as invalidation timestamp on the predecessor. *Two-point invalidation* works well with old-to-new ordering. However, with new-to-old ordering, the invalidation timestamp must be set on the predecessor version, yielding an in-place update and possibly a random write. Second, with *one-point invalidation* [11], the existence of a successor implicitly invalidates the predecessor and all version-records contain only the creation timestamp. *One-point invalidation* matches well new-to-old ordering, the use of indirection layer (VIDs, and entry-points) as well as append-based storage.

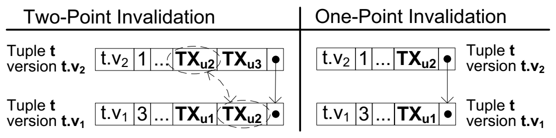


Figure 6: Version Invalidation Model

3.4 Garbage Collection

Under MVCC modifications of a tuple result in the creation of a new tuple-version. Old tuple-versions become obsolete, if they are no longer visible to any of the active transactions. Therefore, some form of *version GC* is necessary to reclaim space and can improve performance. However, GC causes performance spikes (as it interferes with foreground I/O), reduces concurrency (as some form of locking is required) and increases write-amplification on secondary storage. GC [23] can be performed on transaction [14], tuple and index levels [15, 22]. Index-level GC (Section 4.6) purges index entries, resulting from index updates, maintenance or tuple-level GC.

3.5 Version/Index-Record Referencing

There are two possibilities to map index records to tuple-versions in base tables (Figure 7). *First*, classical physical references (recordIDs) can be used. Thus, the latest tuple-version in base tables (entry-point in the version chain) can be accessed directly, but changes to the latest version or its location result in index-record modifications. Such changes comprise: creation of a successor-version; storage management and physical movement (as in append storage) or garbage collection. *Second*, an indirection layer with logical references can be employed. Each tuple-version is augmented with an unique tuple-identifier (Virtual Tuple Identifier – VID), which is also stored in the index records. An index operation resolves the VID using a mapping table (indirection layer) to locate the physical entry-point. *An indirection layer can reduce index maintenance costs for in-place and out-of-place updates, but requires additional structures and processing.*

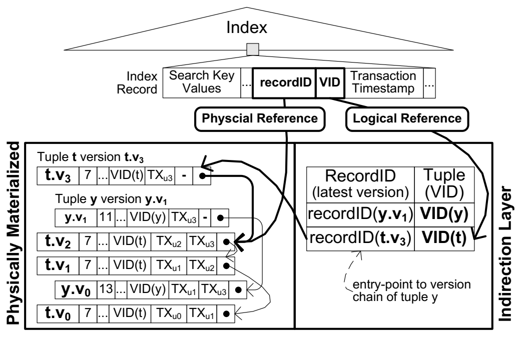


Figure 7: Version/Index-Record Referencing

Traditional index designs use physical references and contain no version-information, which tends to increase index maintenance overhead as well as the visibility check costs for lookups and scans. Alternatively, modern index-structures (BW-Tree) use an indirection layer, but contain no version-information and support no index-only visibility check. This can cause massive read amplification for mixed workloads. An optimal index structure should reduce write amplification and return only references to tuple-versions that are visible to a transaction snapshot. MV-PBT uses physical or logical references, is version-aware and produces append-only sequential write pattern.

3.6 Discussion

We have outlined some relevant design decisions for storing tuple-versions in multi-version DBMS. Modifications are preferably stored as *physically materialised tuple-versions* in base tables, rather than deltas, due to tuple reconstruction costs. Moreover, this enables direct access to each tuple-version from additional access paths. *Out-of-place updates* reduce write amplification to secondary storage. Garbage collection is required for space reclamation, but brings additional complexity to data structures.

A *new-to-old* version ordering requires index maintenance for every new tuple-version, because the entry-point of the version chain for that tuple changes. A logical indirection layer ensures fast lookups by efficiently returning the entry-point of a version chain and reduces index maintenance effort. *New-to-old* ordering is beneficial for OLTP and speeds up visibility-check as the latest version ist typically the visible one, yet older versions may require slow reconstruction. Alternatively, *old-to-new* ordering is supports long-running OLAP operations and visibility-check in HTAP settings, as the oldest version is directly accessible. Yet, modifications and maintenance may suffer low performance.

Indices for mixed workloads and large datasets should rather return visible tuple-versions. Alternatively, traditional index structures only return version candidates, which have to be subsequently verified, fetching version-records from base tables by performing random I/O. For these reasons *MV-PBT* rely on physically materialised versions, out-of-place updates, a new-to-old ordering, one-point invalidation and can do without an indirection layer.

3.7 Storage Characteristics

Modern database storage management needs to address the characteristics of semiconductor storage technologies [20]. Consider Figure 8, which deptics the I/O characteristics of the enterprise Flash storage used in the evaluation. Typical index search operations result in large amount of small (8K) random reads. Hence, optimize for read IOPS and sequential writes (≥64K). We derive the following tradeoffs for the I/O behaviour of MV-PBT: (a) transform random writes in sequential writes with higher granularity (MB); and (b) trade sequential writes for complex and possibly random reads with higher parallelism and smaller granularity (KB). Thus, append-based storage managers are beneficial for the base tables [9, 11]. Write-sequentialization is therefore necessary for indices, and MV-PBT supports it intrinsically, like LSM-Trees.

		Read		Write	
	Blocksize [KB]	8	64	8	64
Sequential	Iops	122382	24180	11104	1343
	MB/s	956	1511	87	84
Random	Iops	112479	23631	7185	56
	MB/s	879	1477	1184	74

Figure 8: I/O Characteristics of Intel DC P3600 SSD.

4 MULTI-VERSION PARTITIONED B-TREES

Multi-Version Partitioned B-Trees (Figure 9) are based on Partitioned B-Trees (PBT), introduced by Goetz Graefe [12, 13]. PBT in turn represent an enhancement on traditional B^+-Trees[4]. PBT (and MV-PBT) create index partitions based on an artificial, leading key-column – *the partition number*. All index-entires in a partition have the same partition number in the search key. PBT (and MV-PBT) utilize a portion of the database buffer (*partition buffer*) to host the latest partition P_N, where insertions and updates to existing partitions ($P_0 \ldots P_{N-1}$) are placed. Updates to existing index entries are treated as *replacement records* to avoid in-place updates. Once P_N gets full it is appended to persistent storage and becomes immutable.

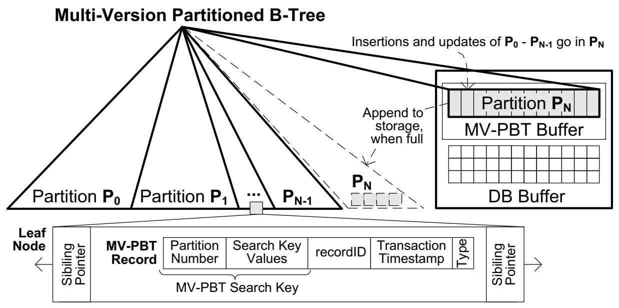


Figure 9: Structure of a Multi-Version Partitioned B-Tree.

Regular MV-PBT records comprize of a *partition number*, its *search key columns*, and a *recordID (set)*. Furthermore, MV-PBT index records contain version-information: *logical transaction timestamp* for validation or invalidation of the tuple-version and optionally an *unique virtual identifier* (indirection layer). Each partition number identifies a single partition. Partition numbers are unique, monotonically increasing, two-byte integer values. This enables the MV-PBT to maintain partitions within one single tree structure in alphanumeric sort order. The partition number is an artificial column and is therefore transparent to higher database layers. Each MV-PBT maintains partitions independent of other MV-PBTs. Partitions appear and vanish as simple as inserting or deleting records. They can be reorganized and optimized on-line in system-transaction merge steps, depending on the workload. Partitions can support additional functionalities, like bulk loads or can serve as multi-version store[13].

MV-PBTs write any modification of index records exactly once – upon eviction of a partition, except for later reorganization or garbage collection operations. This is realized by forcing sequential writes of all leaf nodes in a partition (Figure 9). Leaf nodes of modifiable main memory partitions are stored in a separate buffer cache – the *MV-PBT Buffer*. This area is shared for all MV-PBT indices in the database. Once the MV-PBT Buffer gets full, a victim MV-PBT is selected and its P_N is written to secondary storage. The MV-PBT Buffer is managed by a special replacement policy, giving active partitions the chance to grow (Section 4.5).

4.1 MV-PBT Record Types

Persistent index partitions are immutable. Direct modification-operations are forbidden. Therefore, modifications to existing index-records as well as insertions are placed in the buffered partition P_N. To handle this behavior MV-PBT introduces new index-record types. Currently the following are defined.

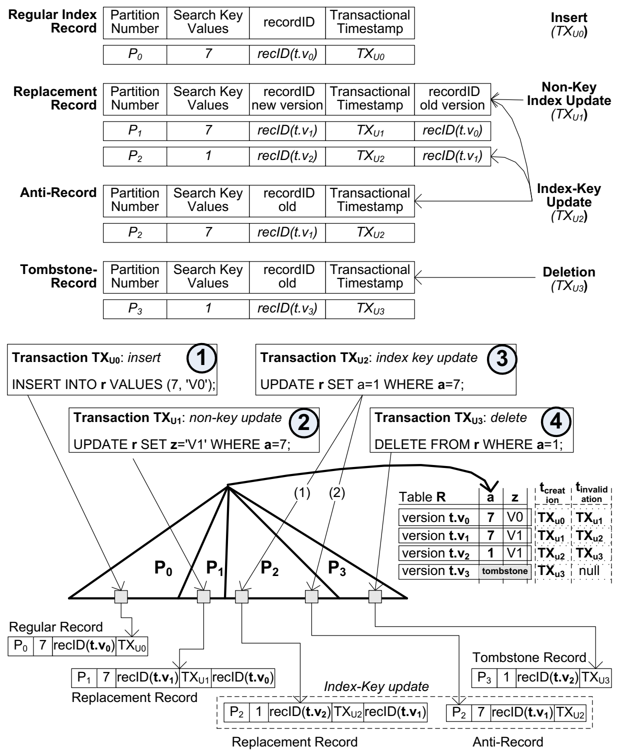

Regular Index Record
Partition Number | Search Key Values | recordID | Transactional Timestamp
P0 | 7 | recID(t.v0) | TXU0
Insert (TXU0)
Replacement Record
Partition Number | Search Key Values | recordID new version | Transactional Timestamp | recordID old version
P1 | 7 | recID(t.v1) | TXU1 | recID(t.v0)
P2 | 1 | recID(t.v2) | TXU2 | recID(t.v1)
Non-Key Index Update (TXU1)
Anti-Record
Partition Number | Search Key Values | recordID old | Transactional Timestamp
P2 | 7 | recID(t.v1) | TXU2
Index-Key Update (TXU2)
Tombstone-Record
Partition Number | Search Key Values | recordID old | Transactional Timestamp
P3 | 1 | recID(t.v3) | TXU3
Deletion (TXU3)
Transaction TXU0: insert
INSERT INTO r VALUES (7, 'V0');
1
Transaction TXU2: index key update
UPDATE r SET a=1 WHERE a=7;
3
Transaction TXU1: non-key update
UPDATE r SET z='V1' WHERE a=7;
2
Transaction TXU3: delete
DELETE FROM r WHERE a=1;
4
(1)
(2)
Table R | a | z | tcreation | tinvalidation
version t.v0 | 7 | V0 | TXu0 | TXu1
version t.v1 | 7 | V1 | TXu1 | TXu2
version t.v2 | 1 | V1 | TXu2 | TXu3
version t.v3 | tombstone | TXu3 | null
P0
P1
P2
P3
Regular Record
P0 | 7 | recID(t.v0) | TXU0
P1 | 7 | recID(t.v1) | TXU1 | recID(t.v0)
Replacement Record
Index-Key update
P2 | 1 | recID(t.v2) | TXU2 | recID(t.v1)
P2 | 7 | recID(t.v1) | TXU2
Replacement Record
Anti-Record
Tombstone Record
P3 | 1 | recID(t.v2) | TXU3

Figure 10: MV-PBT Index-Record Types and Their Use: MV-PBT record format (top), an example including a sequence of transactions and their index records (bottom).

Regular Index Records are created upon the insertion of new tuples. The *partition number* of the newest MV-PBT partition P_N is inserted together with the search key values. The *recordID* (pageID and slot) of the newly inserted tuple-version is included as well as the *transaction timestamp* of the inserting transaction (Figure 10). The latter is essential for index-only visibility-checks. For example, transaction TX_{U0} (Figure 10) inserts a new tuple (t), in its initial version ($t.v_0$), causing the creation of a *regular index record* in partition P_0.

Replacement-Records result from tuple-updates on *non-index key columns* on existing index-entries. Such updates yield a new tuple-version that becomes the new chain entry-point, which needs to be reflected in the index. Although the index-record for the previous version has not changed (non-index-key update) the version-information and recordID of the new version need to be replaced. However, this is not possible, if the index-record is already in an immutable partition ($P_0 \ldots P_{N-1}$). Therefore a *replacement record* is inserted in the newest partition P_N to logically replace the old one with the recordID and the version-information. The *Replacement Record* (Figure 10) contains: the recordID of the new version, its creation-timestamp as well as the recordID of the predecessor version. Hence the record includes some "anti-matter" [13] (recordID) invalidating the predecessors as well as some "matter", i.e. recordID and timestamp if the new version. For example, transaction TX_{U1} (Figure 10) updates the attribute z of the previously inserted tuple (t), producing a new version ($t.v_1$). Although the index-key *7* remains unchanged, the version-information of ($t.v_1$) has to be updated, causing the creation of a *replacement-record* in partition P_1.

Anti-Records are required for updates on index-key attributes and are always used in combination with *replacement records* in the same partition. If the index-key of an existing index-record (in the immutable partitions) gets updated, MV-PBT inserts a combination of an *anti-record* and a *replacement record. Anti-records* are pure "anti-matter" as they mark the extinction of the old index record (from partitions $P_0 \ldots P_{N-1}$), whereas the simultaneously inserted *replacement record* represents the new "matter" and reflects the new index-key and the new version-information. The anti-record and the replacement record are inserted in P_N and are placed according to the sort-order of the search-key value. An anti-record contains the recordID of the old version, together with its search key and the transaction timestamp of the updating transaction (Figure 10). For example, transaction TX_{U2} (Figure 10) updates the indexed attribute *a* of the previously inserted and updated tuple (*t*), producing new version ($t.v_2$), modifying the index-key from *7* to *1*. The *anti-record* (marking the extinction of the replacement-record from partition P_1) reflects the recordID of the predecessor version ($t.v_1$), contains its index-key values (7) and the transaction-timestamp of the current updating-transaction (TX_{U2}). The simultaneously inserted replacement record reflects the new and updated value of the search key (i.e. *1*), the recordIDs of the old and the new tuple-versions ($t.v_1$ *and* $t.v_2$) as well as the transaction-timestamp of TX_{U2}. Since the index records are kept in sort order of the search-key values within a partition (as in a B-Tree), the replacement record is placed first in order, followed by the anti-record.

Tombstone-records indicate the deletion of a tuple. If a tuple is logically deleted, it does not become erased immediately in MV-DBMS, because it could be visible to a concurrent transaction. Rather a tombstone tuple-version record is inserted in the DB, which needs to be reflected in the MV-PBT index. *Tombstone-records* are similar to *Anti Records* in that they represent pure "anti-matter", marking the extinction of the whole tuple-version chain. The difference is that if a tombstone-record is visible to a transaction, no further tuple-version belonging to this chain can be visible, even no *replacement record. Tombstone-records* (Figure 10) contain the recordID of the latest tuple-version and the transaction-timestamp of the deleting transaction.

For example, transaction TX_{U3} (Figure 10) deletes tuple (*t*), creating a tombstone-version ($t.v_3$) in the DB. Therefore a tombstone record is inserted in partition P_3 with the recordID of the deleted tuple-version $t.v_2$, reflecting deletion of the whole version chain $t.v_2 \longrightarrow t.v_1 \longrightarrow t.v_0$.

4.2 MV-PBT Operations

In the following we describe the index operations in an MV-PBT:

- *Insert Operations* are only performed in P_N. An *insertion* yields the creation of an *regular index-record* in P_N with the recordID of the newly created tuple-version and the timestamp of the creating transaction. The insertion traverses the buffered partition P_N and places the new index record according to the alphanumeric sort-order of the search-key (ordering issues are described in Section 4.3). The MV-PBT buffer management strategy (Section 4.5) guarantees sufficient space for the insertion and possible maintenance. In case of an *non-unique index* the insertion is performed directly. Alternatively, given a *unique index*, a lookup operation (see Search and Scan) is performed ahead of the inseartion to guarantee the non-existence of the new index-key.
- *Update Operations* are performed in different ways. If a transaction modifies a tuple-version in a way that a non-index-key attribute is changed (*non-key update*) a new tuple-version is created and its version-information needs to be reflected in the index. In case of *non-key updates* MV-PBT inserts a *replacement-record* in P_N (Figure 10), containing the version-information (recordID and timestamp) of the modifying transaction. By doing so, it logically *replaces* the index-record, which is located in an older partition, and reflects the predecessor version.
  Alternatively, if the modifying transaction updates an index-key attribute (*index-key update*) a *replacement record* as well as an *anti-record* are inserted in P_N (ordering issues are described in Section 4.3). The former reflects the new and modified index-key value in the new tuple-version, the latter indicates the extinction of the old index-record, reflecting the index-key value of the predecessor version. In case of an *unique index*, the MV-PBT first performs a lookup to ensure the non-existence of the new key-value.
- *Delete Operations* cause the insertion of a *tombstone record* in P_N. If a transaction deletes a logical-tuple a tombstone version is created indicating the deletion of the whole version-chain, to transactions to which it is visible. Analogously, MV-PBT inserts a *tombstone record* to indicate the extinction of all index-records corresponding to the version chain. Ordering issues are described in Section 4.3.
- *Search and Scan Operations* process partitions in reverse order from P_N to P_0. Filter techniques such as *Partition Range Keys*, *Minimum Transaction Timestamp* or *Bloom- and Range Filters* (Section 4.7) are needed for selecting the predeceasing partition which may contain an index record, matching the search conditions (Algorithm 1). The search conditions are extended to match the format of an MV-PBT – the partition number is prepended to the first search key column. A regular root-to-leaf traversal operation is performed and the cursor is positioned. Afterwards, the next matching index record is requested and checked for visibility (Section 4.4). This process is repeated until an index record, visible to current transaction is found, and can be returned together with the respective *recordID. Partition number* and *timestamp* are transparent for higher database layers. Index records of most recent tuple versions are found and processed first, due to index-record ordering (Section 4.3), which is very beneficial for simple search conditions, like point lookups.
  Complex scan operations (Algorithm 2) build a set of all matching index records, spreading all MV-PBT partitions. Every partition is pre-selected by filter techniques and processed from P_N to P_0. Traversal operations benefit from commonly buffered higher levels of the tree-structure. Matching index records of any record type in a partition are processed by the index-only visibility-check. Visible index records are added to the result set without *partition number* and *timestamp* in regular sort order. If no further index record matches the scan conditions, the algorithm proceeds with the preceding partition. Finally, the result set is returned. It is filled with all index records (including *recordIDs*), matching the scan and visibility conditions of the calling transaction.
  A single scan process without rechecking for concurrent modifications in P_N is sufficient, due to transaction snapshots as concurrent modifications in P_N are invisible, anyway. Expensive retrieval of version-records from the base-table (*random read I/O*) for *version-information* is avoided. In case of selection of non-index attributes, the recordID indicates the location

of version-record in the base-table, which can be directly accessed.

Algorithm 1 MV-PBT Search

```
1:  function SEARCH(Search conditions $|attr_{val,cond}|, \ldots$)
2:  Output: $IndexRecord$
3:      while HASNEXT( ) do
4:          Let $idx\_record \leftarrow$ NEXT( )  ▷ fetch next index record
5:          if VISCHECK($idx\_record$) equals $VISIBLE$ then
6:              return SET_RETURN_FORMAT($idx\_record$)
                    ▷ hide $partitionnumber$ and $timestamp$
7:      while $part \leftarrow$ PREVIOUSPARTITION($part$) do
8:          if $|attr_{val,cond}| \in part.filter$ then
9:              Let $|skeys_{part}| \leftarrow$
                    FORM_REC($part, |attr_{val,cond}|$)
10:             TRAVERSE($|skeys_{part}|$)
11:             return SEARCH( )
12:     return $\emptyset$
```

Algorithm 2 MV-PBT Scan

```
1:  function SCAN(Scan conditions $|attr_{val,cond}|, \ldots$)
2:  Output: ResultSet of $|IndexRecords|$
3:      $part \leftarrow \emptyset$    ▷ PREVIOUSPARTITION returns $P_N$ for $\emptyset$
4:      while $part \leftarrow$ PREVIOUSPARTITION($part$) do
5:          if $|attr_{val,cond}| \in part.filter$ then
6:              Let $|skeys_{part}| \leftarrow$
                    FORM_REC($part, |attr_{val,cond}|$)
7:              TRAVERSE($|skeys_{part}|$)
8:          while HASNEXT( ) do
9:              Let $idx\_record \leftarrow$ NEXT( )  ▷ neighbor in BTree
10:             if VISCHECK($idx\_record$) equals $VISIBLE$ then
11:                 $|IndexRecords|$.ADD(
                        SET_RETURN_FORMAT($idx\_record$))
12:     return $|IndexRecords|$
```

4.3 MV-PBT Index-Record(Version) Ordering

The version/partition-placement in MV-PBT is governed by modification, search and scan algorithms. *Index-records of predecessor versions are likely to be located in lower-numbered partitions, successors in higher-numbered ones (Figure 10).* This however necessitates multiple memory partitions for an MV-PBT.

To address such issues the current MV-PBT design uses a single main-memory partition P_N for each MV-PBT. *However, for index-records with the same index-key it is mandatory that records for newer/successor versions are always placed before index-records for older/predecessor versions in P_N*. In other words the *primary sort-order* of the index-records in a P_N is on the search-key (mostly descending), however all records with the same search-key are sorted in inverse *secondary sort-order* (mostly ascending) on the transactional timestamp.

Search and scan operations traverse partitions *backwards*: starting from buffered partition P_N (i.e. $P_N \rightarrow P_{N-1} \cdots \rightarrow P_0$). Yet, given the above ordering, index-records of newer tuple-versions, matching the search predicates, are processed *first* in forward direction (i.e. in descending timestamp-order). Only then the next lower-numbered partition is traversed and processed. *This is how MV-PBT ensures that in a search and scan operation, newer versions can always be found before older ones in the same partition, and across partitions.*

Consider for example Figure 11, where we have only two partitions and index-records reflecting updates to the same tuple go to P_1, and contrast to Figure 10, where all index-records with higher-timestamps are placed in higher-numbered partitions. Observe that the index-records in P_1 (Figure 11) appear in their primary-order (on the search key), i.e. records with search-key 1 precede those with 7. Observe also that the *tombstone record* with key 1 precedes the *regular record* as a result of the *secondary sort-order* since $timestamp(TX_{U3}) > timestamp(TX_{U2})$.

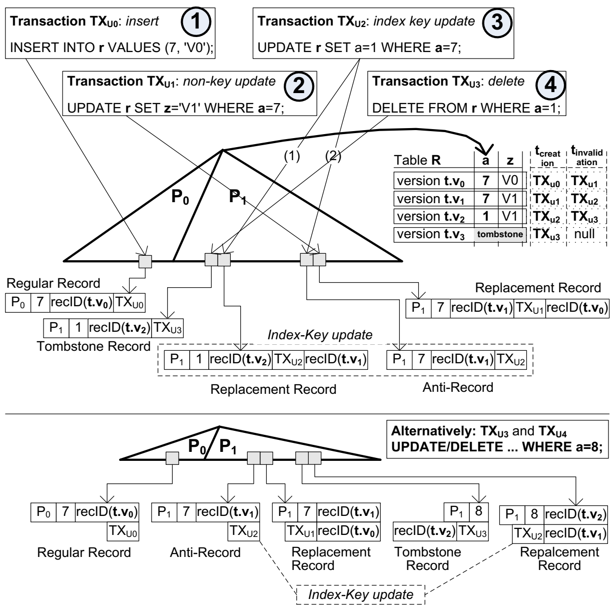

Transaction TXU0: insert 1
INSERT INTO r VALUES (7, 'V0');
Transaction TXU2: index key update 3
UPDATE r SET a=1 WHERE a=7;
Transaction TXU1: non-key update 2
UPDATE r SET z='V1' WHERE a=7;
Transaction TXU3: delete 4
DELETE FROM r WHERE a=1;
(1)
(2)
P0
P1
Table R
a
z
tcreation
tinvalidation
version t.v0 7 V0 TXu0 TXu1
version t.v1 7 V1 TXu1 TXu2
version t.v2 1 V1 TXu2 TXu3
version t.v3 tombstone TXu3 null
Regular Record
P0 7 recID(t.v0) TXU0
P1 1 recID(t.v2) TXU3
Tombstone Record
Index-Key update
P1 1 recID(t.v2) TXU2 recID(t.v1)
Replacement Record
P1 7 recID(t.v1) TXU2
Anti-Record
Replacement Record
P1 7 recID(t.v1) TXU1 recID(t.v0)
Alternatively: TXU3 and TXU4 UPDATE/DELETE ... WHERE a=8;
P0 7 recID(t.v0) TXU0
Regular Record
P1 7 recID(t.v1) TXU2
Anti-Record
P1 7 recID(t.v1) TXU1 recID(t.v0)
Replacement Record
P1 8 recID(t.v2) TXU3
Tombstone Record
P1 8 recID(t.v2) TXU2 recID(t.v1)
Repalcement Record
Index-Key update

Figure 11: MV-PBT Index-Record Ordering.

4.4 MV-PBT Index-Only Visibility-Check

MV-PBT is *version-aware* and supports *index-only visibility-check*, i.e. it returns a set of index records matching the search condition and *visible* to the calling transaction. In doing so, MV-PBT avoids the expensive retrieval of base-table version-records to extract their *version-information.*

The *index-only visibility-check* (Algorithm 3) is inherently supported by the data structure. MV-PBT index records (Section 4.1) contain version-information and define modifications and recordIDs of tuple-versions. The respective *index-record ordering* is essential to scans (Section 4.3), whereby records indicating the invalidation of a tuple-version are guaranteed to be placed before the *"validating"-records* for a given transactional timestamp.

Index records of any type, matching the search-conditions are processed by the visibility check. They are *invisible* to a transaction, if:

(a) the index record is *flagged for garbage collection*;
(b) the transaction timestamp of the index-record is *greater than* the timestamp of the calling transaction; or
the transaction corresponding to the index-record timestamp is *concurrent* to the calling transaction;
(c) visible record with *anti-matter* for the recordID (anti-matter, replacement- and tombstone-records) was already encountered (in this case also checked for GC); or

(d) the index record is either a *tombstone record* or an *anti-record.* An additional visibility-check by processing the version chain in base table is not required. Skewed updates on tuples do not lower the performance of the index-only visibility check, due to well performing garbage collection and well-cached version-chains in the main-memory partition P_N.

Algorithm 3 MV-PBT Index-Only Visibility-Check

```
1:  function VisibilityCheck( $idx\_record$)
2:  input: $idx\_record$ at current scan position
3:  output: $BOOL\_VISIBLE$
4:      Let $anti\_map \leftarrow Map$ of $(recID|TS)$            ▷ anti-matter
5:      if IS_SET($idx\_record, FLAG\_GC$) then
6:          return $INVISIBLE$
7:      if not precedes($idx\_record.ts, CurrentTxId$) OR
                isConcurrent($idx\_record.ts, CurrentTxId$) then
8:          return $INVISIBLE$
9:      if $ts_{anti} \leftarrow anti\_map$.get($idx\_record.recID_{matter}$)
          and precedes($idx\_record.ts, ts_{anti}$) then
10:         checkForGC($idx\_record$)
11:         return $INVISIBLE$
12:     if IS_SET($idx\_record, FLAG\_ANTI\_MATTER$) then
13:         $anti\_map$.put($idx\_record.recID_{anti}, idx\_record.ts$)
14:     if IS_SET($idx\_record, FLAG\_MATTER$) then
15:         return $VISIBLE$
16:     return $INVISIBLE$
```

4.5 MV-PBT Buffer Management

MV-PBT accumulate modifications to persistent partitions in the latest partition P_N, which is held in the MV-PBT partition buffer (Figure 9). All MV-PBT indices place their respective P_N in the MV-PBT buffer, which rises the question of the proper buffer management strategy. Well-known replacement policies (like LRU or ARC) are not suitable for managing the set of leaf nodes contained in the respective P_N as well as different P_N. The MV-PBT buffer should (a) only evict partitions as a whole instead of individual pages (like in LRU) to achieve sequential write patterns; and (b) give partitions of update intensive indices a fair chance to grow, and balance it across all indices. Remember that MV-PBT read operations place persistent partition nodes in the main/shared DB-Buffer. MV-PBT buffer-management strategy can be summarized as follows. Whenever the buffer-size threshold is reached the MV-PBT buffer manager selects the largest partition of all indices as a *victim* for eviction. Smaller, less update-intensive partitions are frequently evicted to avoid imbalanced number of partitions per MV-PBT and shrinking partition sizes.

The *eviction process* (Algorithm 4) can be summarized as follows. A new partition numbered P_{N+1} (initially P_{N+2}) is created for ongoing modifications. The current victim partition P_N becomes *immutable* and is scanned, as following operations are performed cooperatively and latch-free, piggybacking that *scan.*

(1) Version-chains are built for each of the Scan-ResultSet records, using their timestamps and RecordIDs, and creating a temporary VID for each chain. While doing that, obsolete index-records (parts of the version-chain) are detected and marked for garbage collection.
(2) *Garbage Collection* is performed on the marked records (no longer needed/invisible records are removed) and the result is *written out* to new leaf nodes.
(3) During this process index-records and leaf nodes are transformed to an on-disk format, whereby prefix-truncation, compression and encoding as well as dense-packing (Section 4.7) are performed. Furthermore, the partition number of each index record is decremented from P_N to P_{N-1}. Now P_{N-1} is a separate partition, which is yet unknown in the MV-PBT partition metadata. The process resembles a leaf-build in PostgreSQL: having full leaf pages the intermediary index nodes can be easily built on top. Concurrent, lookups and scans are still performed on the old P_N nodes.
(4) In parallel, well-sized *(prefix-) bloom filters* are created (Section 4.7).
(5) Dense-packing, compression and read-optimizations are performed to higher level intermediary nodes, resembling a bottom-up build. All nodes are *sequentially written out.*
(6) Finally, P_{N-1} is added to the MV-PBT partition metadata. The old P_N leaf nodes, on which concurrent non-blocking reads had been executing, are detached from the MV-PBT and are freed for reuse.

Algorithm 4 MV-PBT Partition Eviction

```
1:  function evict($|P_N|$)
2:  Input: set of $P_N$ in MV-PBT buffer
3:      Let $p_{evict} \leftarrow$SelectEvictionVictim($|P_N|$)
4:      Add $Partition\ p_{evict+1}$ to B$^+$-Tree $PartitionsList$
5:      SET($p_{evict}, FLAG\_IMMUTABLE$)
6:      Let $recordSet \leftarrow$ scanRecords($p_{evict}$)
7:      garbageCollectionP3($recordSet$)
8:      $worker1$.loadAndFlush($p_{evict}.pNo - 1, recordSet$)
9:      $worker2$.createFilters($p_{evict}, recordSet$)
10:     wait( )
11:     Let $p_{evict\_new} \leftarrow$decrementPartitionNumber($p_{evict}$)
12:     detatchAndFree($p_{evict}$)
```

4.6 MV-PBT Partition Garbage Collection

Mixed workloads with high update-rates result in massive amount of tuple-versions, which need to be garbage collected once a long-running reading/analytical query completes [14]. Same is true for the corresponding index-records. With high probability these records are located in the main-memory partition P_N of an MV-PBT due to their temporal locality. Therefore, we implemented a cooperative page-level garbage collection (GC) for P_N.

Phase (1): The GC *piggybacks* regular index-scans to identify index-records of versions, that are not visible to any active transaction (*cutoff-transaction*). As a page is already latched (shared), the following checks a performed on each record: (a) comparison with the lowest active transaction timestamp and if lower, mark predecessors as *victim-versions* for GC; (b) if higher, but a successor exists, mark all predecessors as victims for GC. In both cases, a *hasGarbage* flag is set in the page header (no exclusive latch required). This step also *piggybacks* the in-memory structures of the scan and index-only visibility check algorithms. Records with *anti-matter* (anti-matter, replacement and tombstone records) require special attention, as they are still required for invalidation of predecessors. Hence the *anti-matter* record with the highest timestamp smaller than *cutoff* transaction timestamp must not be garbage collected. Index-record ordering (Section 4.3) supports GC while scanning, since successors are mostly processed first.

Phase (2): Update operations check the *hasGarbage* flag in page header. If set they first set the *recordID* of the oldest required

record with *anti-matter* (anti-matter, replacement and tombstone records) to the recordID of the oldest victim-version of that chain on the page. Next, GC victims are removed on that page, the space is reclaimed and only then the update operation proceeds. This behavior saves memory, speeds up scans and visibility checks as well as reduces index maintenance operations (split).
Phase (3): To handle version-chains spanning several pages, and for final cleanup before partition eviction the whole partition is scanned and the version chains (based on timestamps and records) are built in memory. This scan is also piggybacked for filter creation and dense-packing (Section 4.7). Before switching to sibling page, obsolete versions are removed after updating invalidation reference and in-memory version chain is updated.

4.7 MV-PBT Filters and Optimizations

Various optimizations can be performed, based on the fact that once written to storage MV-PBT partitions are *immutable*.

Bloom Filters. Each MV-PBT partition has a bloom filter (*BF*) on the search key. Using bloom filters accelerates key lookups (point-queries) in a partition, by avoiding unnecessary scans. Whenever a key lookup is performed, a BF-query executed first, to verify whether the key does *not* exist in the partition. If it does not exist MV-PBT proceeds with the next partition. Alternatively, if the BF returns true (i.e. the key may exist), MV-PBT scans the whole partition.

Our experimental evaluation (Figure 13) indicates that the average BF size is small – in the order of few hundred KB. Therefore frequently used filters are usually cached in the MV-PBT buffer. Furthermore, their precision is *98%* on average, thus false positives and therefore superfluous scans are rare. BF is is computed efficiently on eviction, piggybacking existing maintenance scan and is persisted as part of the partition metadata.

Range Filters. Partition bloom filters accelerate point lookups, but cannot handle range predicates. Currently, we employ *prefix Bloom Filters (pBF)*, if appropriate, to speedup range scans.

Dense-packed, Read-Optimized immutable storage. Since a partition is immutable once persisted, various space and read-optimization techniques can be applied. *Dense-packing* is used to perform coalescing and free-space optimzation. When in-memory leaf nodes are on average 67% full to accumulate modifications and avoid splitting, however when persisted the the space utilization can be maximized. MV-PBT performs *dense-packing* as part of the final garbage collection and space reclamation.

Especially for *non-unique indices* MV-PBT performs *reconceliation* upon eviction to convert all regular records with the same search key to a single regular record with a set of {recordID, timestamp}, instead of holding separate record for each key instance. The same is true for replacement records, where for the same search key sets of $\{recordID_{NEW}, Timestamp_{NEW}, recordID_{OLD}\}$ are created. Last but not least, compression techniques such as *prefix-truncation* or *delta-compression* are performed on the search key. Along the same lines, various read and cache-aware optimizations can be performed.

5 EXPERIMENTAL EVALUATION

We present the analysis of Partitioned B-Trees (PBT) and MV-PBT together with traditional B^+-Trees (which serve as *baseline*) in PostgreSQL 9.04. Standard, PostgreSQL uses an *old-to-new* version ordering, *physically materialized version storage* and *two-point invalidation.* Index records have a physical reference to base tables – denoted as B-Tree (PG/HOT). PostgreSQL base table storage was also modified to Snapshot Isolation Append Storage (SIAS) [9, 11] with a beneficial append-only write pattern, one-point invalidation and new-to-old version ordering. We implemented and evaluated B^+-Trees and PBT with physical references and with logical tuple references on top of SIAS [9, 11].

Experimental Setup. We deployed PostgreSQL 9.04 and PostgreSQL with SIAS [11] on an *Ubuntu 16.04.4 LTS* server with an eight core *Intel(R) Xeon(R) E5-1620* CPU, 2GB RAM and an *Intel DC P3600 400GB SSD* drive. We used the well-known DBT-2[1] TPC-C-like OLTP benchmark and mixed workload CH-Benchmark [6] in OLTP-Bench [2, 8] for experimental evaluation. The OS page cache is cleaned every second to ensure repeatable and reliable results (even though conservative).

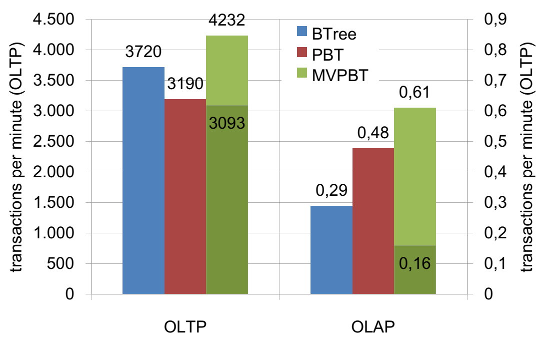
Mixed Workloads: CH-Benchmark. MV-PBT is designed for large datasets and mixed workloads. We evaluate the throughput of B^+-Trees, PBT and MV-PBT under the CH-Benchmark [6] in OLTP-Bench [2, 8]. MV-PBT *doubles* the analytical throughput compared to B^+-Trees (Figure 12a), improving it from 0.29 to 0.61 queries/transactions per minute. In the same time, MV-PBT yield *15%* higher transactional throughput than B^+-Trees (Figure 12a).

The performance improvements are mainly due to *index-only visibility-check* and *partition garbage collection.* To illustrate the combined effect we turn off both and repeat the experiment. Consider now the lower MV-PBT performance bars in Figure 12a. Without *partition garbage collection* and *index-only visibility-check* the OLAP performance drops by 75% from 0.61 to 0.16 queries per minute, whereas the OLTP throughput plummets from 4232 from to 3093 tx/min.

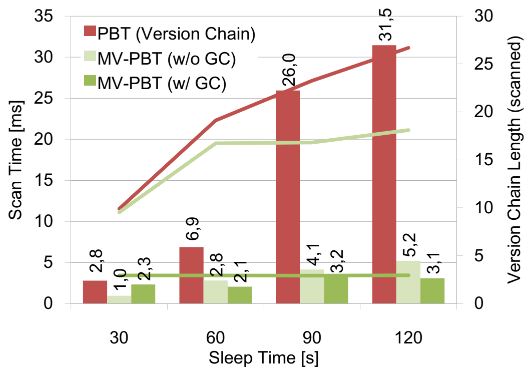
Mixed Workloads: Index-Only Visibility-Check and Garbage Collection. In a further experiment we investigate MV-PBT GC and visibility-check in more detail varying the version-chain length. We run the OLTP part of the CH-Benchmark and execute a query on the same dataset (Figure 12b), however we pause it (using *pg_sleep*) for 30/60/90/120 seconds to simulate a long-running query and vary the amount of transient versions and the chain length. Clearly, as the version-chain length increases, index-only visibility-checks gain importance, because unnecessary read I/O on base table can be reduced.

We compare PBT and standard *visibility-check in base table (VC)* to MV-PBT and *index-only visibility-check (idxVC)* (Figure 12b). As the query processing time and version-chain length increase, index scans and *VC* of slow down PBT by an order of magnitude. Even if the version-chain length has no linear growth, pages in base table get evicted and need to be fetched more frequently. MV-PBT performs *idxVC* however without garbage collection (Figure 12b *MV-PBT w/o GC*), every index record of successor tuple-versions has to be processed, likewise the scan time increases proportionally with the length of the version-chain. With garbage collection (Figure 12b *MV-PBT w/ GC*), the number of scanned index records and the scan time remain almost constant. However, GC requires additional processing and latches index nodes in P_N. Reading transactions have to wait for latches and scan time increases - consider Figure 12b at 30 seconds sleep time. As more index record get garbage collected, GC improves the index scan time - compare MV-PBT with and without GC at 30 and 120 second (Figure 12b).

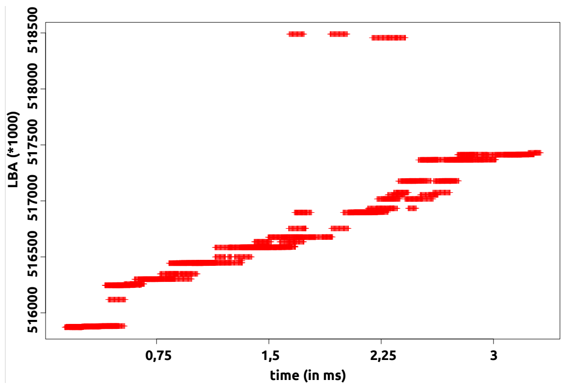
Sequential write-pattern/Append-based storage. Based on the tradeoffs derived in Section 3.7. MV-PBT needs to support write sequentialization and append based storage. In this experiment we evaluate the write pattern of MV-PBT (Figure 12c). Using *blktrace* and *blkparse* we record an I/O trace during the partition eviction from MV-PBT buffer. The X-axis represents the



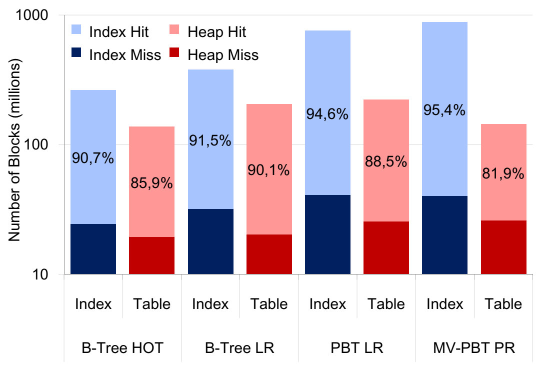
(a) Index Performance under Mixed Workloads (CH-Bnchmark)



(b) Standard vs. Index-Only Visibility-Check for Different Chain Lengths



(c) Sequential Write Pattern of Eviction of a Single MV-PBT Partition



(d) Requests / Cache Hit Rate for PostgreSQL Heap-Only Tuples (HOT), Logical(LR) and Physical(PR) references

Figure 12: Index Performance under Mixed Workloads (CH-Benchmark)

eviction time; the average write I/O time is about 1ms. The Y-axis represents the logical block addresses (LBA), i.e. the file system addresses where the blocks of the index file are written. Each red cross indicates the write of a single index node. A horizontal line, therefore indicates a *sequential write*, i.e multiple blocks are written onto neighbouring addresses over time. *Hence the sequential write pattern of MV-PBT.* The horizontal lines in Figure 12c represent *database extents* and result from the database *space allocation strategy*. Each evicted partition comprises leaf nodes allocated in new extents of the index file, allocated at (mostly) adjacent addresses by the file system. The overall *sequential pattern* confirms the sequential append behaviour of MV-PBT.

MV-PBT Buffer Efficiency. Figure 12d shows the fetch requests on index nodes (blue) and base table nodes (red) for an OLTP benchmark. Furthermore, the cache hit-rate is depicted. Requests yielding a cache-hit are displayed in brighter colours than fetches (cache-misses) from secondary storage. The scale of requests is logarithmic. The results are calculated for equal throughput over the test duration and all tables and indices.

PBT and MV-PBT require more requests on index nodes due to partitioning of index records and greater record sizes. Most requests are on buffered nodes, because many queries can be answered in the main memory partition. Index records of new tuple-versions are common to be located there. MV-PBT reduces the requests on base table by up to 40%, because the base table is not required for visibility-check. The version chains are short for this benchmark, for mixed workloads this effect is more weighty. This can be seen at the reduced cache hit rate on base table nodes in comparison to PBT. Most saved requests on base tables are on new tuple-versions, which are located in main memory.

Partition Filters. Partition-based indices like MV-PBT, PBT or LSM-Trees incur higher lookup and scan overhead than B-Trees, since matching records may exist in older partitions. Hence, the effort of lookups and especially of scans increases with number of index-partitions, since in the worst case every partition has to be traversed. Point lookups can stop partition traversal after finding the first matching record, which is visible to a transaction, since older partitions are guaranteed to contain older records.

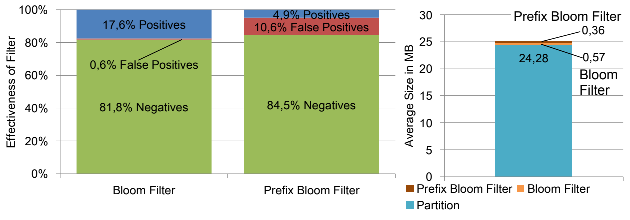
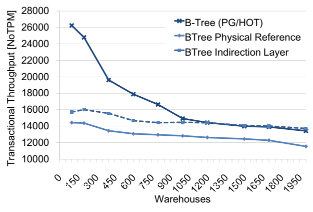
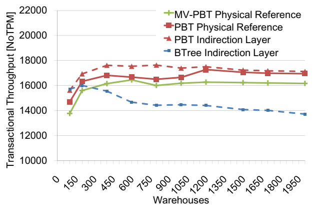


Figure 13: Effectiveness and Size of Partition Filters

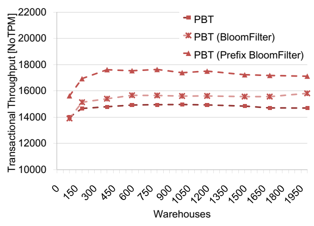
Using *Bloom filters (BF)* (Section 4.7) point lookups can skip partitions and increase throughput up to 10% under TPC-C (Figure 14c). Furthermore, *prefix Bloom filters (pBF)* may under certain conditions speedup scans by skipping partitions not matching the range predicate. *pBF* including a fixed set of scan attributes,



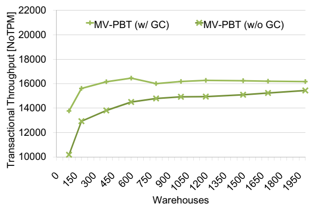
(a) **Indirection Layer vs. Physical Version-Record Reference**



(b) **Performance of Indexing Approaches under TPC-C**



(c) **Influence of filter techniques on Throughput under TPC-C**



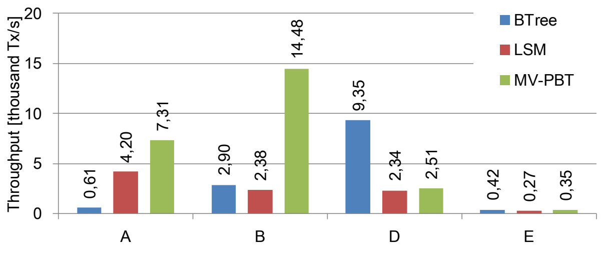
(d) **MV-PBT Garbage Collection under TPC-C**

Figure 14: OLTP Performance Evaluation under TPC-C

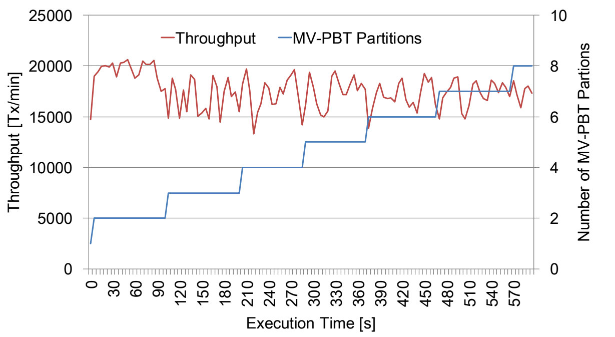
increase the throughput by another 10% (Figure 14c). The precision of both Bloom filters is relatively high (Figure 13): the false positives rate is 2% for *BF* and 10% for *pBF*, while the negatives (skipping) rate is approx. 82% for *BF* and 84.5% for *pBF*. The size *BF* and *pBF* is small relative to the partition size (Figure 14c): for a 24MB partition *BF* is 0.57MB, while *pBF* is 0.36MB.

Since index operations only have a fair share of the whole database operations under TPC-C (besides logging, CC and I/O) the above numbers yield moderate performance improvements.

Comparison to LSM-Trees. LSM-Trees [17] are used as workhorse storage structure in many Key/Value stores for large datasets. Today's highly-optimized multi-level LSM-Trees with levelling or tiering resemble MV-PBT as they exhibit an append-behaviour and employ buffered components. We implemented MV-PBT in WiredTiger [3], the high-performance KV-Store of MongoDB. In this experiment we compare MV-PBT to LSM-Tree in WiredTiger under YCSB [7] (Figure 15a). YCSB has been instrumented as follows: a dataset of 100 million keys (approx. 100GB); workloads A (30 mil. requests), B and D(10 mil. req.) and E (2 mil. req).



(a) **MV-PBT, BTree, LSM-Tree under YCSB.**



(b) **YCSB Throughput (workload A) vs. Number of MV-PBT Partitions**

Figure 15: Performance Evaluation under YCSB [7]

Workload *A* comprises 50% read and 50% update requests, which require fast lookups and updates. MV-PBT is approx. 42% faster than LSM-Trees. Each LSM level comprises multiple components, which themselves are small read-optimised BTrees. A search needs to process separate LSM components even though some can be skipped (bloom filters). MV-PBT partition search is faster than LSM component search, since the leaf nodes in each partition are under the same common index. Updates in MV-PBT hit P_N, which accommodates more KV-pairs than the main memory L_0 in LSM-Trees. Workload *B* comprises 95% read and 5% update requests, with *zipfian* distribution. BTree performs random reads, the LSM Tree caches the updates, but has an equal amount of random reads spread over more components. MV-PBT have much lower index maintenance compared to BTrees and place the updates in P_N. The reads are performed with maximum I/O parallelism. Workload *D* comprising 95% read and 5% update requests, which given the *latest* distribution stress the memory components and BTrees performs most of the operations in memory. MV-PBT is marginally better than LSM-Tree. Last but not least, we run workload *E* comprising 95% scans and 5% insert requests. Even though the *scans* are slow under MV-PBT, they outperform LSM-Trees due to the faster search and updates.

Consider, Figure 15b depicting the YCSB throughput (workload A) and the number of MV-PBT partitions over time. The throughput remains stable as the number of partitions increases.

OLTP: comparison of B-Tree alternatives. To establish the baseline we first compare standard PostgreSQL B-Trees (PG/HOT) to B$^+$-Trees with physical reference and indirection layer on top of append-only storage (SIAS [9, 11]) under TPC-C. In Figure 14a, we show the throughput for different dataset sizes. The buffer cache of the DBMS is fixed to 600MB. B-Tree(PG/HOT) performs well (Figure 14a) as long as the database buffer can accommodate most modifications. Under standard Postgres updates are performed in base tables by Heap-Only Tuples (HOT), i.e. the predecessor version is cached on the same page, on which its successor is located. Therefore the index maintenance effort is low. With growing data sizes (and therefore more modifications), the throughput falls rapidly. Append-based storage and one-point invalidation (SIAS [9, 11]) exhibit a *robust* throughput: (a) *physical references* (Section 3.5) yield lower performance, due to the higher index management overhead; (b) an *indirection layer* reduces index maintenance for insertions and index-key updates, yielding up to 30% better throughput. *With larger datasets (≥ 1200 warehouses) B-Trees with indirection outperform standard PostgreSQL PG/HOT.*

Indexing Approaches under OLTP. In a follow-up experiment, we compare B-Tree with indirection layer (Section 3.5), to

PBT and *MV-PBT* under TPC-C (Figure 14b). PBT and MV-PBT exhibit robust performance, which improves with larger datasets compared to B-Tree. PBT with *indirection layer* exhibits high and robust performance (Figure 14b). PBT with *physical reference* to close the performance gap for larger datasets as the update density decreases decreases with larger datasets. *MV-PBT are slower than PBT under OLTP workloads for several reasons.* First, less MV-PBT index records fit on the same sized P_N, since their sizes are larger because of the version-information (transaction timestamps). Consequently, the number of partitions increases, yielding more I/O. Second, the average version-chain length under TPC-C is short: 1.15/2.18 versions for *customer*/*stock* respectively [9]. Therefore, index-only visibility-checks cannot improve performance significantly. Thus, MV-PBT exhibit 6% lower performance than PBT under TPC-C (Figure 14b). We implemented MV-PBT with an *indirection layer* as well as with *physical references* (Section 3.5). Figure 14b depicts on the performance with *physical references* for brevity, both curves are almost identical. *Therefore, MV-PBT are general enough to be implemented matching the rest of the system design.*

OLTP Garbage Collection. In this experiment (Figure 14d) we quantify the performance effect of MV-PBT partition garbage collection (Section 4.6). It improves performance between 5% and 17% since old invisible versions are purged and need not be processed by scans as well as space is reclaimed letting more index records fit in P_N. The opportunity of improvement under OLTP is however limited by the short average version-chain length: 1.15 versions for *customer* and 2.18 versions for *stock* under TPC-C [9]. With HTAP workloads the amount of 'transient' (short-lived versions visible only throughout the duration of an analytical query) versions increases rapidly as does the effect of garbage collection. Garbage collecting larger amounts transient versions has a major role on the performace improvment of MV-PBT over PBT and B-Tree under mixed workloads (Figure 12a).

6 RELATED WORK

Most popular indexing approaches in database management systems are based on B^+-Trees. Their alphanumeric sorted structure can result in high write amplification for high update rates and visibility-checks require information, that is only located at tuple-versions in base table. PostgreSQL uses Heap-Only Tuples (HOT) to reduce index management operations. Index records reference items in base table, which point to tuple-versions in the heap node. Corresponding tuple-versions are held on the same node and can be located by processing the version chain. If a tuple-version become garbage collected, the item is modified to reference the next version. This indirection layer reduces index modifications, but cannot avoid write amplification of index nodes and requires the base table for visibility-checking. Furthermore the write amplification of base table nodes is increased for large datasets. MV-IDX[10] maintains a virtual identifier for each tuple and data nodes for each version as an indirection layer. With Snapshot Isolation Append Storage (SIAS)[11] write amplification on base tables is reduced in comparison to HOT, but index management operations can cause a high write amplification and base table nodes are still required for visibility-checking[21]. LSM-Trees[18] reduce write amplification due to collecting modifications in main memory components, but there is no concept for managing tuple-versions and perform an index-only visibility-check[21]. Time-Split B-Trees [16] and Multiversion B-Trees [5] are able to separate index records of old tuple-versions from current dataset and to perform an index-only visibility-check, but maintenance operations are complex and can cause a high write amplification of index nodes[21].

7 CONCLUSION

In the present paper we introduce MV-PBT as an approach to multi-version indexing. An MV-PBT is an extension of a B-Tree, where an artificial leading column is prepended to the search key of each index record and index records are placed in a buffered index partition, which if full gets evicted and appended to persistent storage. MV-PBT is version-aware, since index records contain version-information and allow for index-only visibility check. This is particularly beneficial for HTAP workloads since long chains of transient versions exist due to the mix of short-lived updating transactions and long-running queries. Furthermore, MV-PBT exhibit a sequential write pattern due to the concept of partition, which leads to less write-amplification and better utilization of modern storage technologies. Under mixed workloads (CH-Benchmark) MV-PBT doubles the analytical throughput 2x, while improving the transactional throughput by 15%.

ACKNOWLEDGMENTS

This work has been partially supported by: *HAW Promotion* and *KPK Services Computing, MWK, Baden-Würrtemberg*, Germany; *BMBF PANDAS – 01IS18081C/D*; *DFG Grant neoDBMS – 419942270.*

REFERENCES

[1] 2019. *Database Test Suite.* https://sourceforge.net/projects/osdldbt/files/dbt2/
[2] 2019. *Oltpbench.* https://github.com/oltpbenchmark/oltpbench/
[3] 2019. *WiredTiger (MongoDB).* http://www.wiredtiger.com
[4] R. Bayer and E. McCreight. 1970. Organization and Maintenance of Large Ordered Indices. In *Proc. SIGFIDET (SIGMOD) 1970.* 107–141.
[5] Bruno Becker, Stephan Gschwind, and et al. 1996. An Asymptotically Optimal Multiversion B-tree. *The VLDB Journal* 5, 4 (Dec. 1996), 264–275.
[6] Richard Cole, Florian Funke, Alfons Kemper, and et al. 2011. The Mixed Workload CH-benCHmark. In *Proc. DBTest '11.* Article 8, 6 pages.
[7] Brian F. Cooper, Adam Silberstein, and et al. 2010. Benchmarking Cloud Serving Systems with YCSB. In *In Proc. SoCC2010.*
[8] 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.
[9] Robert Gottstein. 2016. *Impact of new storage technologies on an OLTP DBMS, its architecture and algorithms.* Ph.D. Dissertation. TU, Darmstadt.
[10] Robert Gottstein, Sergej Hardock, Ilia Petrov, and Alejandro Buchmann. 2014. MV-IDX: Indexing in Multi-version Databases. In *Proc. IDEAS 2014.* 142–148.
[11] Robert Gottstein, Ilia Petrov, and et al. 2017. SIAS-Chains: Snapshot Isolation Append Storage Chains. In *ADMS@VLDB.*
[12] Goetz Graefe. 2003. Partitioned B-trees - a user's guide. In *Proc. BTW.* 668–671.
[13] Goetz Graefe. 2003. Sorting And Indexing With Partitioned B-Trees. In *CIDR.*
[14] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, and et al. 2016. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *Proc. SIGMOD 2016.* 1307–1318.
[15] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proc. ICDE 2013.* 302–313.
[16] David Lomet and Betty Salzberg. 1990. The Performance of a Multiversion Access Method. In *Proc. SIGMOD 1990.* 353–363.
[17] Chen Luo and Michael J. Carey. 2019. LSM-based storage techniques: a survey. *The VLDB Journal* (19 Jul 2019).
[18] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385.
[19] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *Proc. SIGMOD 2017.* 1771–1775.
[20] I. Petrov, R. Gottstein, and S. Hardock. 2015. DBMS on modern storage hardware. In *Proc. ICDE 2015.* 1545–1548.
[21] Christian Riegger, Tobias Vincon, and Ilia Petrov. 2017. Multi-version Indexing and Modern Hardware Technologies A Survey of Present Indexing Approaches. In *Proc. iiWAS 2017.* 266–275.
[22] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, and et al. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proc. SIGMOD 2018.* 473–488.
[23] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-memory Multi-version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (March 2017), 781–792.

Lineage-Preserving Anonymization of the Provenance of Collection-Based Workflows

Khalid Belhajjame
PSL, Université Paris-Dauphine, LAMSADE
Paris, France
khalid.belhajjame@dauphine.fr

ABSTRACT

We examine in this paper the problem of anonymizing the provenance of collection-oriented workflows, in which the constituent modules use and generate sets of data records. Despite their popularity, this kind of workflows has been overlooked in the literature w.r.t privacy. We, therefore, set out in this paper to examine the following questions: *How the provenance of a collection-based module can be anonymized? Can lineage information be preserved? Beyond a single module, how can the provenance of a whole workflow be anonymized?* As well as addressing the above questions, we report on evaluation exercises that assess the effectiveness and efficiency of our solution. In particular, we tease apart the parameters that impact the quality of the obtained anonymized provenance information.

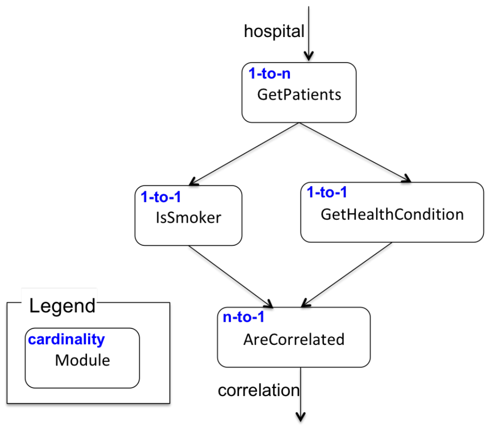


Fig. 1: Example workflow.

1 INTRODUCTION

Automated workflows have been shown to facilitate and accelerate scientific data exploration and analysis in many areas of sciences [11]. Figure 1 illustrates a simple workflow that is used to establish correlations between smoking and health conditions. Workflow provenance information, recorded during workflow executions, facilitates the interpretation of the results delivered by workflow execution. Beyond verification, workflow provenance information represents a useful dataset on its own right, that can be leveraged to answer queries that are relevant for an experiment that is (possibly related but) different from the original experiment, to learn new hypotheses, or to gain insight on the characteristics and quality of the data generated by given modules. Collected workflow provenance information can also be used to respond to the requirements of funding agencies that are increasingly requesting the publication of the data generated in the context of research investigations.

In fields such as biomedicine and social sciences, workflow executions manipulate and generate sensitive information about individuals. To promote the publication and sharing of the provenance of workflow executions, we set out in this paper to examine the problem of anonymizing workflow provenance.

1.1 Related Work

Related work has focused on the problem of securing workflow provenance and policing their access. For example, Chebotko *et al* [9] and Biton *et al* [7] proposed solutions that derive a partial view on a workflow provenance by hiding the data records of given modules Our objective is different from the above line of work in that we seek to provide the user with the provenance of all the modules of the workflow by leveraging anonymization.

Davidson *et al.* [12] investigated the problem of module privacy, whereby some of the parameters (attributes) characterizing the inputs and outputs of the modules are hidden to guarantee the privacy of modules. In our work, we seek, instead, to guarantee the privacy of the data records used and generated by the modules, instead of the behavior of the module.

We have examined in a previous work, the problem of identification of the k-anonymity degree that needs to be enforced when anonymizing the datasets used and generated by workflows [6]. In doing, we assumed that the modules that compose the workflow are 1-to-1 in that they produce a single data record, given a single data record, and we did not give much thought to the problem of lineage preservation. In this paper, we are interested in what we refer to as collection-based workflows [16–18, 27]. The modules that compose such workflows can take as input a collection of data records and deliver a collection of data records. Such workflows have been advocated as a way to better meet the needs of non-expert users to model scientific data [20], and to structure complex relationships among related pieces of information that are processed together by the workflow [22]. This class of workflows has been overlooked in the literature w.r.t. privacy.

Different techniques have been proposed in the literature for protecting the privacy of individuals, notably, k-anonymity [26] and differential privacy [15]. In particular, differential privacy [15] has recently gained momentum as the method of choice in statistical databases. It involves adding random noise to the data so that the distribution of the resulting dataset is almost invariant to the inclusion of any data record. While powerful, differential privacy is not suitable for our purposes. It assumes that the user knows up-front the queries s/he wants to issue prior to the anonymization. This is not the case in our setting, where the scientist issues exploratory queries for understanding and eventually interpreting the results of the workflows. Furthermore, for it to be useful, the scientist should be able to inspect individual data records and their relationships (lineage), both of which are not possible using differential privacy. Indeed, differential privacy is more suited for statistical (i.e., aggregation-based) queries.

For our work, we chose to use k-anonymity [26]. This method is not as powerful as differential privacy when it comes to privacy guarantees. Yet, it is better suited for our purposes since it can be instrumented, as we will show, to allow users to query and examine individual data records and their lineage within workflow provenance. k-anonymity is also still perceived by practitioners as sufficient for mitigating risk while maximizing utility, and real-world applications still utilize it for data sanitization (see e.g., [3, 10]). It is also widely popular and is used, e.g., in the healthcare world [1, 25], and is still recommended by data protection agencies (see e.g., [2]). This technique has been extensively investigated in the database and data mining communities [28]. Most of the proposals have focused on anonymizing a single relational table. In workflow provenance, however, we need to anonymize different datasets considering and preserving lineage relationships between them. One solution that can be used to anonymize workflow using k-anonymity would be to create a global relational table that is obtained by joining relations representing the input and output data records of the modules that compose the workflows. However, this solution suffers from the following issues. First, information about the same individual can be found in different records. This is because we consider collection based modules, e.g., a patient can be associated with multiple practitioners. Second, the same tuple in the global table may contain information about multiple individuals, e.g., a patient, one of its practitioners, etc. Moreover, as we will see later, different kinds of individuals may be associated with different k-anonymity degrees. For example, the k-anonymity degree associated with patients may be higher than that associated with practitioners. Traditional k-anonymity is not equipped to deal with the above issues. In this respect, the proposal by Nergiz *et al.* [24] is related to ours. They elaborated a technique that anonymizes multiple relations of a given database schema. While useful, this proposal makes a number of limiting assumptions. In particular, they consider snowflake schemas, in which there is a single relational table that represents individuals with the remaining relations containing quasi-attributes and having a single foreign key. In our work, we drop these assumptions and show that the anonymization of workflow provenance can be achieved in the presence of multiple datasets representing individuals with multiple relationships (foreign keys constraints) between them.

1.2 Contributions

Our first contribution is the formulation of the problem of k-anonymization of the provenance of collection-based workflows. This is, to our knowledge, the first paper that extends the notion of k-anonymization from a single relation to the provenance of workflows. Our second contribution is a technique for k-anonymizing the provenance of a single module, i.e., input and output records together with their lineage information. Indeed, lineage information tracing the dependencies between the output and input of a module (and more generally a workflow) is key for third-party scientists to understand and examine the validity of workflow results. We examine this problem for modules that use and generate collections of data records. Our third contribution extends the technique proposed to cater for the anonymization of the provenance of a workflow as a whole. Central to the solution we present is the notion of k-group anonymity, which we define based on the k-anonymity degree and the magnitude of the smallest input (or output) set of data records used and generated by a module. This concept allows us to gracefully reason over the different k-anonymity degrees that may be associated with the inputs and outputs of the workflow's modules. We also show how the NP-hard problem of identifying the sets of data records to be grouped together into equivalence classes that meet k-anonymity requirements can be cast as a scheduling problem that we solve using integer programming.

The paper is organized as follows. We start by laying the foundations of our work and stating the problem in Section 2. We then focus on the problem of anonymizing the provenance of a module in Section 3, and the provenance workflow in Section 4. In Section 5, we address an issue that is inherent to our anonymization technique, namely grouping sets of data records, and cast it as a scheduling problem. We report on evaluation exercises that we empirically conducted to assess the effectiveness and efficiency of our solution in Section 6, and conclude the paper in Section 7.

2 FOUNDATIONS

2.1 Collection-Based Module and Workflow

Definition 2.1 (module). A module $\mathtt{m}$ is defined by the tuple $(\mathtt{I_m}, \mathtt{O_m}, \mathtt{card})$, where $\mathtt{I_m}$ (resp. $\mathtt{O_m}$) is a set of ordered input (resp. output) ports, and $\mathtt{card}$ specifies the cardinality of $\mathtt{m}$. A port $\mathtt{p} = \langle \mathtt{a_1}, \ldots, \mathtt{a_n} \rangle$ is a list of attributes, each characterized with a basic type, e.g., String, Integer.

Assigning a data value to each attribute in a port gives rise to a data item, and assigning a data item to each input (output) port of a module gives rise to a data record.

$\mathtt{card} \in$ {1-to-1,1-to-n,n-to-1,n-to-n}: 1-to-1 specifies that the invocation of $\mathtt{m}$ takes as input a single data record and produces a single data record; n-to-1 (resp. 1-to-n) specifies that the invocation of $\mathtt{m}$ takes as input a list (ordered set) of data records (resp. single data record) and produces a single data record (resp. a list of data records); n-to-n specifies that the invocation of the module takes as input a list of data records and produces a list of data records.

Definition 2.2 (data link). A data link $\mathtt{dl}$ is defined by the pair $\mathtt{dl} = (\mathtt{m_i} : \mathtt{o_{m_i}}, \mathtt{m_j} : \mathtt{i_{m_j}})$, where $\mathtt{m_i} : \mathtt{o_{m_i}}$ designates an output port $\mathtt{o_{m_i}}$ of the module $\mathtt{m_i}$, and $\mathtt{m_j} : \mathtt{i_{m_j}}$ designates an input port $\mathtt{i_{m_j}}$ of the module $\mathtt{m_j}$.

Definition 2.3 (workflow). A workflow specification is defined by a pair $\mathtt{w} = (\mathtt{M}, \mathtt{E})$, where $\mathtt{M}$ is a set of modules and $\mathtt{E}$ is a set of data links. $\mathtt{w}$ has one initial module with no incoming data links, and one final module with no outgoing data links.

We consider acyclic workflows that have a single initial module and a single final module, and where each module in the workflow, other than the initial module, is reachable from the initial module. Workflow execution follows a pure dataflow model: a module $\mathtt{m}$ is invoked (is fireable) as soon as all of its input ports are bound to data items. During the workflow execution, data items are transferred between connected output and input ports. For example, the following data link binding $((\mathtt{m_1} : \mathtt{o_{m_1}}, \mathtt{m_2} : \mathtt{i_{m_2}}), \mathtt{di})$ specifies that the data item $\mathtt{di}$ was transferred using the data link connecting the output port $\mathtt{o_{m_1}}$ of $\mathtt{m_1}$ to the input port $\mathtt{i_{m_2}}$ of $\mathtt{m_2}$. The technical report [5] provides more information about the workflow execution model. It specifies how input data records of a module are constructed using the data records of the preceding modules in the workflow. It also specifies how mismatches in cardinalities between connected modules' outputs and inputs are resolved at execution time.

Table 1: Input and Output Provenance of admittedTo.

Input Patient DataSet			
ID	**name**	birth	lin
p_1	Garnick	1990	$\{r_1, r_2\}$
p_2	Hiyoshi	1987	$\{r_3, r_4\}$
p_3	Suessmith	1989	$\{r_5, r_6\}$
p_4	Solares	1985	$\{r_7, r_8\}$
p_5	Kading	1992	$\{r_9, r_{10}\}$
p_6	Pero	1988	$\{r_{11}, r_{12}\}$
p_7	Pehl	1986	$\{r_{13}, r_{14}\}$
p_8	Barriga	1995	$\{r_{15}, r_{16}\}$

Output Hospital DataSet		
ID	hospital	Lin
h_1	St Louis	$\{p_1, p_3\}$
h_2	St Anton	
h_3	St Anne	$\{p_2, p_4\}$
h_4	St August	
h_5	Holby	$\{p_5, p_7\}$
h_6	Larib.	
h_7	St James	$\{p_6, p_8\}$
h_8	St Mary	

2.2 Workflow Provenance as Relations

Definition 2.4. Given a workflow w, its provenance, denoted by prov(w), is the collection of modules and data link bindings that take place as a result of the executions of w.

For ease of exposition of our anonymization solution, we encode the provenance of a module m using two relational tables denoted by prov(m, w).in and prov(m, w).out. They contain the data records that were used and generated, respectively, by the invocations of m within the executions of a workflow w. we call prov(m, w).in (resp. prov(m, w).out) the input (resp. output) provenance of m. When the referred workflow w is clear from the context, we abuse the notation and simply use prov(m).in and prov(m).out to refer to such relations. The schema of such relations contains the attributes of the input ports (resp. output ports) of m. We assume that the attribute names are unique within the input (resp. output) ports of a module. From a provenance point of view, we do not keep information about the order of the data records in an input or output list, which is, therefore, viewed as a set. This is the case, for example, in the Taverna workflow system [30]. Because of this, we use in what follows the terms input/output set of data records, as opposed to list of data records. W.l.o.g, we assume that the attributes of two succeeding modules that have the same name are connected (via their ports) by data links. In other words, we can deduce data link bindings from module bindings, which allows us to write: $\text{prov(w)} = \bigcup_{\text{m} \in \text{w.M}} (\text{prov(m).in} \cup \text{prov(m).out})$

Consider a module admittedTo that given a set of patients returns a set of hospitals that those patients were admitted to[1]. Table 1 illustrates an example of two relations representing input provenance and output provenance of the admittedTo module. The names of identifying attributes are written in bold, and the names of quasi-identifying attributes are underlined. Notice that the relations contain also two additional attributes: ID and Lin. The first is an ID that is generated internally by the workflow system to identify data records, and the second is used to encode lineage information. In the case of the input provenance, Lin specifies the data records produced by the preceding modules in the workflow and that were used in the construction of the data record in question. For example, the data record p_1 was constructed using two data records r_1 and r_2 that were produced by some preceding modules. The Lin column is empty for the relational table used to store the data records used as input to the initial module in the workflow. Regarding the output provenance of admittedTo, the Lin column identifies the data records that were used as input to obtain the output data record in question. For example, it specifies that h_1 and h_2 were generated given the inputs p_1 and p_3. The lineage information we consider here is in line with the *why provenance* semantics (see [8]).

2.3 Problem Statement

Adversary Model. The data records used and generated by a workflow module are characterized by three kinds of attributes: (i) Identifying attributes allow identifying individuals, e.g., the attribute name is an identifying attribute. (ii) Sensitive attributes are attributes that carry sensitive information, e.g., health condition. (iii) Quasi-identifying attributes are non-identifying attributes, but their combination can be used to identify an individual, e.g., address, phone number, etc. Notice that the ID attribute is not considered as an identifying attribute because it is generated by the workflow system and does not carry information that allows identifying individuals such as name for example.

We assume that an adversary may know identifying and quasi-identifying attribute values about individuals, e.g., name, address, date of birth. However, we assume that s/he does not know sensitive attribute values, e.g., health-condition, income tax.

In relational databases, a relation r is k-anonymized, where k is an integer greater than 2, if any data record d in r is not distinguishable from (at least) $k - 1$ other records in r. This condition is met by masking the values of identifying attributes, and generalizing the values of quasi-identifying attributes (e.g., address, visited hospital, etc.). Sensitive attributes, such as health condition, salary, are not masked: adversaries are assumed not to be knowledgeable of the values of sensitive attributes. In what follows, we use the term identifier record to refer to a data record that has an identifying attribute value, and the term quasi-identifier record to refer to a data record that has no identifying attribute value but has a quasi-identifying attributes value. A module input (resp. output) that is bound to identifier records following module invocation is called identifier input (resp. output). It is called quasi-identifier input (resp. output) if it is bound to quasi-identifier records.

Anonymity degree of Identifier Inputs and Outputs. We assume that every identifier input (resp. identifier output) of a module m is associated with an anonymity degree, which we denote by k^i_m (resp. k^o_m) to be enforced. Note that non-identifier module inputs and output are not associated with an anonymity degree because they are not bound at execution time to records that represent individuals. We do not make the assumption that the anonymity degrees associated with the identifier inputs and outputs of the modules that compose the workflow are the same. This is because the modules that compose a workflow are likely to use different underlying data sources that are supplied by different providers who may impose different requirements when it comes to the anonymity degree to be enforced on their data. Moreover, the same data provider may impose different anonymity degrees depending on the data that is retrieved from its source. For example, an input that provides information about patients and their health condition is likely to be associated with an anonymity degree that is higher than an output that informs on the trips of practitioners. In this paper, we apply k-anonymization to the provenance prov(w) of a workflow prov(w) by creating equivalence classes for the relations prov(m).in and prov(m).out for each identifier input and output of the modules in w.M.

Definition 2.5 (Equivalence Classes). Consider the input provenance prov(m).in of a module m. We say that the set $\{E1^m_{in}, \ldots, E1^m_{in}\}$, $n \geq 1$, is a set of input equivalence classes for m and write $\text{prov}^a(\text{m}).\text{in} = \{E1^m_{in}, \ldots, E1^m_{in}\}$ iff:

1)- The set $\{E1^m_{in}, \ldots, E1^m_{in}\}$ forms a partitioning for prov(m).in. That is $\text{prov(m).in} = \bigcup_{i \in [1,n]} Ei^m_{in}$, and $Ei^m_{in} \cap Ej^m_{in} = \emptyset$ for $i, j \in [1, n]$ s.t. $i \neq j$.

2)- The identifying attribute values of the data records in every equivalence class Ei^m_{in} are masked, and their quasi-identifying attribute values are generalized such that the data records in an

[1]A hospital appears in the result only if it was visited by each of the patients in the input set.

Table 2: Input and Output Provenance of `admittedTo` **where the Input Provenance is 2-anonymized.**

2-anonymized Patient DataSet			
ID	**name**	birth	lin
p_1	⋆	{1987,1990}	$\{\texttt{r}_1, \texttt{r}_2\}$
p_2	⋆	{1987,1990}	$\{\texttt{r}_3, \texttt{r}_4\}$
p_3	⋆	{1985,1989}	$\{\texttt{r}_5, \texttt{r}_6\}$
p_4	⋆	{1985,1989}	$\{\texttt{r}_7, \texttt{r}_8\}$
p_5	⋆	{1988,1992}	$\{\texttt{r}_9, \texttt{r}_{10}\}$
p_6	⋆	{1988,1992}	$\{\texttt{r}_{11}, \texttt{r}_{12}\}$
p_7	⋆	{1986,1995}	$\{\texttt{r}_{13}, \texttt{r}_{14}\}$
p_8	⋆	{1986,1995}	$\{\texttt{r}_{15}, \texttt{r}_{16}\}$

Hospital DataSet		
ID	hospital	Lin
h_1	St Louis	$\{p_1, p_3\}$
h_2	St Anton	
h_3	St Anne	$\{p_2, p_4\}$
h_4	St August	
h_5	Holby	$\{p_5, p_7\}$
h_6	Larib.	
h_7	St James	$\{p_6, p_8\}$
h_8	St Mary	

equivalence class $\texttt{Ei}^{\texttt{m}}_{\texttt{in}}$ are indistinguishable w.r.t. their quasi-identifying attribute values.

A set of output equivalence classes are defined in a similar manner: $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{out} = \{\texttt{E1}^{\texttt{m}}_{\texttt{out}}, \ldots, \texttt{E1}^{\texttt{m}}_{\texttt{out}}\}$.

Note that the `ID` and `Lin` attribute values of the data records are not generalized. This is because the values of the `ID` attribute are generated internally by the workflow system. In other words, they are not meaningful for human users. More importantly, they are used within the `Lin` attribute to encode lineage information that we seek to preserve.

Lineage information needs to be considered when k-anonymizing the input provenance (resp. output provenance) of an identifier module input (resp. output). To illustrate this, let us consider the `admittedTo` module. It has an identifier input and a quasi-identifier output. Consider that the anonymity degree associated with its input is $\texttt{k}^{\texttt{admittedTo}}_{\texttt{i}} = 2$. Notice that its output is not associated with an anonymity degree because it is not an identifier output. Table 2 illustrates the input and output provenance of `admittedTo`, where the input provenance is 2-anonymized. The anonymization consisted in partitioning the set of input data records into input equivalence classes of size ≥ 2. Notice that this anonymization operation does not guarantee k-anonymization, however. To illustrate this, consider that an adversary knows that `Garnick` was born in `1990` and that he visited the `StLouis` hospital. By examining the output data records together with lineage information in `prov(admittedTo).out` (see Table 2), an adversary will be able to infer that the data record $\texttt{p}_1$ refers to `Garnick`. This can be more of an issue when the data record contains sensitive information such as health condition.

Problem 1 (K-anonymization of the input and output provenance of a module). *Consider a module* `m` *with an identifier input (resp. output). k-anonymizing the input provenance* `prov(m).in` *(resp.* `prov(m).out`*) of* `m` *using an anonymity degree* $\texttt{k}^{\texttt{m}}_{\texttt{i}}$ *(resp.* $\texttt{k}^{\texttt{m}}_{\texttt{o}}$*) gives rise to anonymized input provenance* $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{in}$ *(resp. anonymized output provenance* $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{out}$*) where:*
1)- $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{in} = \{\texttt{E1}^{\texttt{m}}_{\texttt{in}}, \ldots, \texttt{En}^{\texttt{m}}_{\texttt{in}}\}$ *(resp.* $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{out} = \{\texttt{E1}^{\texttt{m}}_{\texttt{out}}, \ldots, \texttt{En}^{\texttt{m}}_{\texttt{out}}\}$*), is a set of input (resp. output) equivalence classes for the input (resp. output) provenance of* `m`*, with* $\texttt{n} \geq 1$.
2)- *An input equivalence class* $\texttt{Ei}^{\texttt{m}}_{\texttt{in}}$ *(resp. output equivalence class* $\texttt{Ei}^{\texttt{m}}_{\texttt{out}}$*) contains at least* $\texttt{k}^{\texttt{m}}_{\texttt{i}}$ *(resp.* $\texttt{k}^{\texttt{m}}_{\texttt{o}}$*) data records.*
3)- *The data records in an input equivalence class* $\texttt{Ei}^{\texttt{m}}_{\texttt{in}}$ *(resp. output equivalence class* $\texttt{Ei}^{\texttt{m}}_{\texttt{out}}$*) cannot be distinguished by examining their lineage, i.e., by examining the data records that (transitively) contributed to the data records in* $\texttt{Ei}^{\texttt{m}}_{\texttt{in}}$ *(resp.* $\texttt{Ei}^{\texttt{m}}_{\texttt{out}}$*) or by examining the data records that the records in* $\texttt{Ei}^{\texttt{m}}_{\texttt{in}}$ *(resp.* $\texttt{Ei}^{\texttt{m}}_{\texttt{out}}$*) contributed to through workflow execution.*

Condition (3) is formally defined in the technical report using the notions of backward- and forward-lineage in a workflow [5].

Problem 2 (K-anonymization of the provenance of a workflow). *The provenance of a workflow* `w` *is said to be k-anonymized iff the input provenance of every identifier module input and the output provenance of every identifier module output in* `w.M` *are k-anonymized.*

The above problem is NP-Hard: Meyerson and Williams [21] demonstrated that optimal k-anonymity for a single relational table without considering lineage is an NP-hard problem. We present in this paper a heuristic that seeks to satisfy k-anonymity, to reduce the generalization (information-loss) incurred as a result, and to preserve lineage information in doing so.

3 ANONYMIZATION OF MODULE PROVENANCE

We show, in this section, how the input provenance and output provenance of a module are anonymized. The solution we present is applicable to many-to-many modules but also to modules with other cardinalities. We distinguish the case where the module input is an identifier input and its output is a quasi-identifier output, and the case where the module input and output are identifier input and identifier output. In the first case, the attribute values of the output data records are treated as quasi-identifying attribute values for their counterpart input data records. The second case is slightly more complex in the sense that the attribute values of the output data records are treated as quasi-identifying attribute values for their counterpart input data records, and vice-versa. We will not examine the case where both the module input and output carry quasi-identifier records. Indeed, it only makes sense to perform the anonymization when the input and/or the output carry identifier records, and as such associated with an anonymity degree to be enforced. That said, we will show in Section 4 how modules that carry quasi-identifier input and output records are dealt with in situations where they are used in workflows containing other modules with identifier records.

3.1 Module with Identifier Input and Quasi-Identifier Output

Consider the `admittedTo` module, presented earlier, that given a set of individuals returns a set of hospitals that those patients visited (see Table 1). And consider that the input dataset has been 2-anonymized as illustrated in Table 2. As discussed earlier, the lineage associating the output dataset to the input dataset may allow an adversary to pinpoint patients in the input dataset, even if this is anonymized. To avoid this, the hospital dataset needs to be anonymized in a way not to be able to distinguish between the hospitals visited by the patients that belong to the same equivalence class as a result of the anonymization of the patient dataset. For example, p_1 and p_2 must be associated with the same set of hospitals, and so do p_3 and p_4. Given lineage information, one way to do so consists in generalizing the hospitals in a way not to be able to distinguish between the hospitals corresponding to $\{\texttt{p}_1, \texttt{p}_3\}$ and those corresponding to $\{\texttt{p}_2, \texttt{p}_4\}$. An example of generalization of the hospital dataset that achieves this is illustrated in Table 3. Notice that similar generalization is applied to the hospitals corresponding to the groups of patients $\{\texttt{p}_5, \texttt{p}_7\}$ and $\{\texttt{p}_6, \texttt{p}_8\}$.

While acceptable, there is a more effective manner in this case to anonymize the patient and hospital datasets that yields less generalization of the quasi-attributes, thereby reducing the information loss incurred by the anonymization. Indeed, we can exploit the fact that patients are grouped into input sets to guide

Table 3: Input and Output Provenance of `admittedTo` where the Input and Output are 2-anonymized.

2-anonymized Patient DataSet			
ID	**name**	birth	Lin
p_1	★	{1987,1990}	$\{r_1, r_2\}$
p_2	★	{1987,1990}	$\{r_3, r_4\}$
p_3	★	{1985,1989}	$\{r_5, r_6\}$
p_4	★	{1985,1989}	$\{r_7, r_8\}$
p_5	★	{1988,1992}	$\{r_9, r_{10}\}$
p_6	★	{1988,1992}	$\{r_{11}, r_{12}\}$
p_7	★	{1986,1995}	$\{r_{13}, r_{14}\}$
p_8	★	{1986,1995}	$\{r_{15}, r_{16}\}$

2-anonymized Hospital DataSet		
ID	hospital	Lin
h_1	{St Louis, St Anne}	$\{p_1, p_3\}$
h_2	{St Anton, St August}	
h_3	{St Louis, St Anne}	$\{p_2, p_4\}$
h_4	{St Anton, St August}	
h_5	{Holby, St James}	$\{p_5, p_7\}$
h_6	{Larib., St Mary}	
h_7	{Holby, St James}	$\{p_6, p_8\}$
h_8	{Larib., St Mary}	

Table 4: Input and Output Provenance of `admittedTo` where the Input is 2-anonymized and the output does not need to be.

Input Patient DataSet			
ID	**name**	birth	Lin
p_1	★	{1989,1990}	$\{r_1, r_2\}$
p_2	★	{1985,1987}	$\{r_3, r_4\}$
p_3	★	{1989,1990}	$\{r_5, r_6\}$
p_4	★	{1985,1987}	$\{r_7, r_8\}$
p_5	★	{1986,1992}	$\{r_9, r_{10}\}$
p_6	★	{1988,1995}	$\{r_{11}, r_{12}\}$
p_7	★	{1986,1992}	$\{r_{13}, r_{14}\}$
p_8	★	{1988,1995}	$\{r_{15}, r_{16}\}$

Output Hospital DataSet		
ID	hospital	Lin
h_1	St Louis	$\{p_1, p_3\}$
h_2	St Anton	
h_3	St Anne	$\{p_2, p_4\}$
h_4	St August	
h_5	Holby	$\{p_5, p_7\}$
h_6	Larib.	
h_7	St James	$\{p_6, p_8\}$
h_8	St Mary	

the anonymization process. In particular, we put sets of data records that are used as input to a module invocation within the same equivalence class. For example, the patients p_1 and p_3 are put within the same equivalence class. Using this approach, we obtain the 2-anonymized patient dataset illustrated in Table 4. Notice that by doing so, we actually do not need to anonymize the hospital dataset. Indeed, starting from the hospital dataset, we cannot single out any patient: the same set of hospitals are visited by 2 patients. The approach we have just described is more effective as far as information loss is concerned. For example, one would know that $\mathtt{p}_1$ and $\mathtt{p}_3$ visited St Louis and St Antonio. Using the previous approach (described in Table 3), we would infer less specific information: that $\mathtt{p}_1$ and $\mathtt{p}_3$ visited St Louis or St Anne, and St Antonio or St Augustine.

With the above consideration in mind, we revisit the definition of equivalence classes introduced in Section 2 by requiring equivalence classes to contain sets of data records that are used as input or generated as output of module invocations. We will also introduce the notion of $\mathtt{k-group}$ anonymity degree, which allows us to gracefully reason about k-anonymity for collection-oriented modules.

Definition 3.1 (Equivalence Classes - Revisited). Given a module $\mathtt{m}$, we say that the set $\{\mathtt{E1}^{\mathtt{m}}_{\mathtt{in}}, \ldots, \mathtt{En}^{\mathtt{m}}_{\mathtt{in}}\}$ (resp $\{\mathtt{E1}^{\mathtt{m}}_{\mathtt{out}}, \ldots, \mathtt{En}^{\mathtt{m}}_{\mathtt{out}}\}$) is a set of input (resp. output) equivalence classes for $\mathtt{m}$, and write: $\mathtt{prov^a(m).in} = \{\mathtt{E1}^{\mathtt{m}}_{\mathtt{in}}, \ldots, \mathtt{E1}^{\mathtt{m}}_{\mathtt{in}}\}$ (resp. $\mathtt{prov^a(m).out} = \{\mathtt{E1}^{\mathtt{m}}_{\mathtt{out}}, \ldots, \mathtt{E1}^{\mathtt{m}}_{\mathtt{out}}\}$) iff:
1)- The conditions in Definition 2.5 are satisfied.
2)- An input (resp. output) equivalence class $\mathtt{Ei}^{\mathtt{m}}_{\mathtt{in}}$ (resp. $\mathtt{Ei}^{\mathtt{m}}_{\mathtt{out}}$) contains entire sets of input sets (resp. output sets) of data records. That is, two data records that belong to the same input set (resp. output set) that was used (resp. generated) by the invocation of $\mathtt{m}$ in $\mathtt{prov(m).in}$ (resp. $\mathtt{prov(m).out}$) cannot belong to different input (resp. output) equivalence classes.

Definition 3.2 (k-group anonymity (kg)). We say that the input provenance $\mathtt{prov^a(m).in}$ (resp. output provenance $\mathtt{prov^a(m).out}$) of a module $\mathtt{m}$ is k-group anonymized using the k-group anonymity degree $\mathtt{kg}^{\mathtt{m}}_{\mathtt{i}}$ (resp. $\mathtt{kg}^{\mathtt{m}}_{\mathtt{o}}$) iff each equivalence class in $\mathtt{prov^a(m).in}$ (resp. $\mathtt{prov^a(m).out}$) contains at least $\mathtt{kg}^{\mathtt{m}}_{\mathtt{i}}$ input sets of data records (resp. $\mathtt{kg}^{\mathtt{m}}_{\mathtt{o}}$ output sets of data records)

Property 1. *Consider a module* $\mathtt{m}$ *with an identifier input associated with an anonymity degree* $\mathtt{k}^{\mathtt{m}}_{\mathtt{i}}$ *(resp. identifier output with an anonymity degree* $\mathtt{k}^{\mathtt{m}}_{\mathtt{o}}$*). And, let* $\mathtt{l}^{\mathtt{m}}_{\mathtt{i}}$ *(resp.* $\mathtt{l}^{\mathtt{m}}_{\mathtt{o}}$*) be the magnitude of the smallest input (resp. output) set of data records in* $\mathtt{prov(m).in}$ *(resp.* $\mathtt{prov(m).out}$*). k-group anonymyzing* $\mathtt{prov(m).in}$ *(resp.* $\mathtt{prov(m).out}$*) using the k-group anonymity degree* $\mathtt{kg}^{\mathtt{m}}_{\mathtt{i}} = \left\lceil \frac{\mathtt{k}^{\mathtt{m}}_{\mathtt{i}}}{\mathtt{l}^{\mathtt{m}}_{\mathtt{i}}} \right\rceil$ *(resp.* $\mathtt{kg}^{\mathtt{m}}_{\mathtt{o}} = \left\lceil \frac{\mathtt{k}^{\mathtt{m}}_{\mathtt{o}}}{\mathtt{l}^{\mathtt{m}}_{\mathtt{o}}} \right\rceil$*) yields input provenance* $\mathtt{prov^a(m).in}$ *(resp. output provenance* $\mathtt{prov^a(m).in}$*) that is k-anonymized using the anonymity degree* $\mathtt{k}^{\mathtt{m}}_{\mathtt{i}}$ *(resp.* $\mathtt{k}^{\mathtt{m}}_{\mathtt{o}}$*).*

Proof. An input equivalence class in $\mathtt{prov^a(m).in}$ contains at least $\left\lceil \frac{\mathtt{k}^{\mathtt{m}}_{\mathtt{i}}}{\mathtt{l}^{\mathtt{m}}_{\mathtt{i}}} \right\rceil$ input sets of data records. Given that $\mathtt{l}^{\mathtt{m}}_{\mathtt{i}}$ is the magnitude of the smallest input set in $\mathtt{prov(m).in}$, we conclude that an input equivalence class in $\mathtt{prov^a(m).in}$ contains at least $\mathtt{kg}^{\mathtt{m}}_{\mathtt{i}} \cdot \mathtt{l}^{\mathtt{m}}_{\mathtt{i}}$ data records, which is equal to or greater than $\mathtt{k}^{\mathtt{m}}_{\mathtt{i}}$. In other words, $\mathtt{prov^a(m).in}$ is k-anonymized using the k-anonymity degree of $\mathtt{k}^{\mathtt{m}}_{\mathtt{i}}$. The same reasoning can be applied to show that the output provenance is k-anonymized using the degree of $\mathtt{k}^{\mathtt{m}}_{\mathtt{o}}$. □

We are now ready to discuss the general case. To anonymize the input provenance and output provenance of a module $\mathtt{m}$ with an identifier input and quasi-identifier output, we start by k-group anonymizing its input provenance using the k-group anonymity degree of $\mathtt{kg}^{\mathtt{m}}_{\mathtt{i}} = \left\lceil \frac{\mathtt{k}^{\mathtt{m}}_{\mathtt{i}}}{\mathtt{l}^{\mathtt{m}}_{\mathtt{i}}} \right\rceil$. This yields input provenance $\mathtt{prov^a(m).out}$ that is k-anonymized using the degree of $\mathtt{k}^{\mathtt{m}}_{\mathtt{i}}$ (see Property 1). Because the data records in the output provenance act as quasi-identifying records for the data records in the input provenance, we also need to anonymize the output provenance. To do so, we partition $\mathtt{prov(m).out}$ into a set of output equivalence classes $\mathtt{prov^a(m).out}$. This is done by putting the output sets of data records, that correspond to input sets pertaining to the same input equivalence class in $\mathtt{prov^a(m).in}$, within the same output equivalence class in $\mathtt{prov^a(m).out}$. This way, an adversary cannot distinguish the data records in an input equivalence class by examining their corresponding output data records, since these belong to the same output equivalence class and as such have the same quasi-identifying attribute values.

The above solution is applicable to modules with quasi-identifier input and identifier output, by inverting the roles of the input and output used above.

3.2 Modules with Identifier Input and Identifier Output

Consider a module $\mathtt{m}$ with an identifier input and an identifier output. To anonymize the input provenance and output provenance of $\mathtt{m}$, we reason using the k-group anonymity degrees associated with the input and output of $\mathtt{m}$. Specifically, we distinguish the following cases:

Case 1: $\mathtt{kg}^{\mathtt{m}}_{\mathtt{i}} \geq \mathtt{kg}^{\mathtt{m}}_{\mathtt{o}}$. We k-group the input provenance using the k-group degree $\mathtt{kg}^{\mathtt{m}}_{\mathtt{i}}$. This yields k-anonymized input provenance with an anonymity degree of $\mathtt{k}^{\mathtt{m}}_{\mathtt{i}}$ (according to Property 1). The output provenance is anonymized by partitioning it into a set of output equivalence classes $\mathtt{prov^a(m).out}$: output sets of data records, that correspond to input sets pertaining to the same input equivalence class in $\mathtt{prov^a(m).in}$, are put within the same output equivalence class in $\mathtt{prov^a(m).out}$.

An output equivalence contains the sets of data records that correspond to input sets of data records in the same input equivalence class. Given that an input equivalence class contains at least $\mathtt{kg}^{\mathtt{m}}_{\mathtt{i}}$ input sets of data records, it follows that an output equivalence class in $\mathtt{prov^a(m).out}$ contains at least $\mathtt{kg}^{\mathtt{m}}_{\mathtt{i}}$ output sets of data records. Given that $\mathtt{kg}^{\mathtt{m}}_{\mathtt{i}} \geq \mathtt{kg}^{\mathtt{m}}_{\mathtt{o}}$, it follows that $\mathtt{prov^a(m).out}$ is k-group anonymized using the k-group anonymity degree of

Table 5: Input and Output Provenance of `getPractitioners`.

Input Patient DataSet		
ID	**name**	birth
p_1	Facello	1953
p_2	Simmel	1964
p_3	Bamford	1959
p_4	Koblick	1954
p_5	Maliniak	1955
p_6	Preusig	1953
p_7	Zielinski	1957
p_8	Kalloufi	1958

Output Practitioner DataSet			
ID	**name**	birth	Lin
pr_1	Rosch	1996	
pr_2	Bellone	1987	$\{p_1, p_2\}$
pr_3	Gargeya	1993	
pr_4	Gubsky	1988	
pr_5	Heyers	1985	$\{p_3, p_4\}$
pr_6	Tokunaga	1991	
pr_7	Camarinopoulos	1995	
pr_8	Miculan	1986	$\{p_5, p_6\}$
pr_9	Birrer	1992	
pr_{10}	Keustermans	1999	
pr_{11}	Mancunian	2001	$\{p_7, p_8\}$
pr_{12}	Bond	1982	

$\mathtt{kg}_\mathtt{o}^\mathtt{m}$, which implies that $\mathtt{prov}^\mathtt{a}(\mathtt{m})\mathtt{.out}$ is k-anonymized using the anonymity degree of $\mathtt{k}_\mathtt{o}^\mathtt{m}$ (see Property 1). Note also that data records in the same input (resp. output) equivalence class cannot be distinguished by examining their corresponding output (resp. input) data records. This is because the data records in a given input equivalence class will have their corresponding output data records in the same output equivalence class, and, therefore, cannot be distinguished by examining their quasi-identifying attribute values, and vice-versa.

As an example, consider a module, `getPractitioners`, that takes a set of patients and returns the set of practitioners that have examined those patients[2]. Table 5 illustrates the input provenance and the output provenance of `getPractitioners`. We omit the lineage information (Lin column) in the input provenance because it is not useful in the example. Consider that the input of `getPractitioners` is associated with $\mathtt{k}_\mathtt{i}^\mathtt{getPractitioners} = 2$, and its output with $\mathtt{k}_\mathtt{o}^\mathtt{getPractitioners} = 2$. Given that $\mathtt{l}_\mathtt{i}^\mathtt{getPractitioners} = 2$ and $\mathtt{l}_\mathtt{o}^\mathtt{getPractitioners} = 3$ (see Table 5), we have $\mathtt{kg}_\mathtt{i}^\mathtt{getPractitioners} = \mathtt{kg}_\mathtt{o}^\mathtt{getPractitioners} = 1$. The k-group anonymity degree for both input and output in this case is 1. Tables 6 shows the anonymized input and output provenance obtained using the solution we have just described. Notice that the resulting patient dataset is 2-anonymized and that the resulting practitioner dataset is 3-anonymized. Moreover, we cannot distinguish between the practitioners of the patients in the same input equivalence class, and, similarly, we cannot distinguish between the patients of the practitioners that belong to the same output equivalence class.

Case 2: $\mathtt{kg}_\mathtt{i}^\mathtt{m} < \mathtt{kg}_\mathtt{o}^\mathtt{m}$. We perform the same processing as in (case 1) by inverting the roles of the input and output.

Table 6: 2-anonymized Input and 3-anonymized Output Provenance of `getPractitioners`.

Input Patient DataSet		
ID	**name**	birth
p_1	⋆	{53, 64}
p_2	⋆	{53, 64}
p_3	⋆	{54, 59}
p_4	⋆	{54, 59}
p_5	⋆	{53, 55}
p_6	⋆	{53, 55}
p_7	⋆	{57, 58}
p_8	⋆	{57, 58}

Output Practitioner DataSet			
ID	**name**	birth	Lin
pr_1	⋆	{87, 93, 96}	
pr_2	⋆	{87, 93, 96}	$\{p_1, p_2\}$
pr_3	⋆	{87, 93, 96}	
pr_4	⋆	{85, 88, 91}	
pr_5	⋆	{85, 88, 91}	$\{p_3, p_4\}$
pr_6	⋆	{85, 88, 91}	
pr_7	⋆	{86, 92, 95}	
pr_8	⋆	{86, 92, 95}	$\{p_5, p_6\}$
pr_9	⋆	{86, 92, 95}	
pr_{10}	⋆	{82, 99, 01}	
pr_{11}	⋆	{82, 99, 01}	$\{p_7, p_8\}$
pr_{12}	⋆	{82, 99, 01}	

4 DATA PRIVACY OF WORKFLOW PROVENANCE

Given a workflow `w`, we seek to anonymize its provenance `prov(w)` by anonymizing the input provenance and output provenance of its constituent modules. In doing so, we can use the method presented in the previous section as is to anonymize the data used and generated by each module, in an independent fashion. Unfortunately, this solution may lead to a breach of privacy. Indeed, equivalence classes will be formed without consideration to lineage between data records output by given modules and the data records used to feed the succeeding modules within the workflow, which may lead to a privacy breach. The technical report [5] contains a detailed example that illustrates this case.

We, therefore, designed an algorithm that ensures that lineage information cannot be used by an adversary to uncover private information about individuals. For the purpose of the anonymization algorithm, we will be shortly presenting, we group the workflow modules into levels as illustrated in Figure 2. A module belongs to level `0` if it does not have a previous module. A module belongs to a level `i` where `i > 0`, if it has at least an incoming data link connected to a module in level `i − 1`, and it does not have any incoming data link connected to a module in level ≥ `i`.

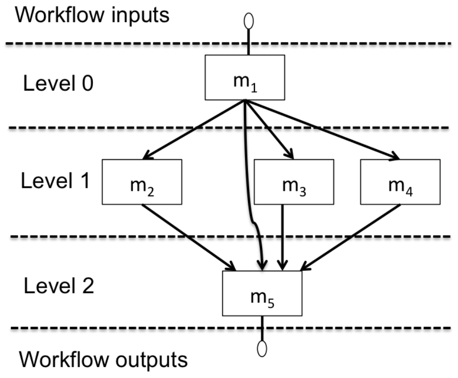


Fig. 2: Workflow levels.

Algorithm 1 Anonymize Workflow Provenance

```
Input:  w, the workflow specification.
Input:  Modules = {L_0, ..., L_k} // workflow modules grouped into levels (breadth)
     from the sink to the source.
          prov(w), the provenance of the workflow w.
          kg, group anonymity degree.
Output: Prov^a(w) // anonymized provenance
 1: for Level in Modules do
 2:     for m in Level do
 3:         if (m is the initial module) then
 4:             prov^a(m).in ← anonymizeInitialInput(m, prov(m).in, kg)
 5:             prov^a(m).out ← anonymizeOutput(m, prov(m).out, prov^a(m).in)
 6:         else
 7:             prov^a(m).in ← constructInputRecords(m, prec(m))
 8:             prov^a(m).out ← anonymizeOutput(m, prov(m).out, prov^a(m).in)
 9:         end if
10:     end for
11: end for
12: prov^a(w) ← $\bigcup_{m \in w.M}$ prof^a(m)
13: return prov^a(w)
```

To anonymize the provenance of a workflow `w`, Algorithm 1 takes as input the modules that compose the workflow organized into levels from the source to the sink, the provenance of the workflow `prov(w)`, as well as a group anonymity degree `kg` to be applied to the input provenance of the initial module of the workflow. (We will see later which k-group anonymity degree is used.) The algorithm examines the modules by level. For the first level composed of the initial module $\mathtt{m}_\mathtt{init}$, the operation `anonymizeInitialinput()` (line 4) anonymizes the input data of such module using the k-group degree `kg` and produces the set of equivalence classes $\mathtt{prov}^\mathtt{a}(\mathtt{m}_\mathtt{init})\mathtt{.in}$. The output data records of the initial module are used to feed the operation

[2]A practitioner appears in the output set only if it has examined every patient in the input set.

anonymizeOutput(), which produces $\texttt{prov}^{\texttt{a}}(\texttt{m}_{\texttt{init}}).\texttt{out}$ (line 5). For a module m that belongs to a level other than the initial one, the algorithm starts by constructing the anonymized input data records by using the anonymized data records of the preceding output in the previous level using the constructInputRecords() operation (lines 7). The output data records of m are, then, anonymized by using the same grouping applied to its inputs to produce $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{out}$ using the anonymizeOuput() operation (line 8). The algorithm terminates when the provenance of the module that belongs to the sink level is anonymized.

Having described how Algorithm 1 operates, we will describe in detail the operations used in the algorithm. We will also list the guarantees respected by each operation. Due to space limitations, the poofs of the guarantees can be found in the technical report [5]. Before proceeding to the presentation of the operations used in our algorithm, we start by defining the notion of lineage-related equivalence classes.

Definition 4.1 (Lineage-Related Equivalence Classes). Let E1 and E2 be two different equivalence classes. We say that E1 and E2 are lineage-related iff there are data records in E1 that (transitively) contributed through workflow executions to data records in E2, or vice-versa.

anonymizeInitialInput(m, prov(m).in, $\mathbf{k_g}$). This operation generates input equivalence classes for the initial module m: $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{in} = \{\texttt{E1}^{\texttt{m}}_{\texttt{in}}, \ldots, \texttt{En}^{\texttt{m}}_{\texttt{in}}\}$, $n \geq 1$. Such equivalences classes are obtained by partitioning the input sets of data records in prov(m).in into groups, each containing at least $\texttt{k}_{\texttt{g}}$ input sets. Each group gives rise to an equivalence class by masking the identifying attribute values of its data records and generalizing their quasi-identifying attribute values such that the data records in the group are indistinguishable w.r.t. their quasi-identifying attribute values. Details about the partitioning operation are presented later in Section 5.

Guarantees:

- **$\mathbf{G_1}$**: $\texttt{prov}(\texttt{m}_{\texttt{init}})^{\texttt{a}}.\texttt{in}$ is $\texttt{k}_{\texttt{g}}$ group anonymized.

anonymizeOutput(m, prov(m).out, $\mathbf{prov^a}$(m).in). Given anonymized input provenance $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{in} = \{\texttt{E1}^{\texttt{m}}_{\texttt{in}}, \ldots, \texttt{En}^{\texttt{m}}_{\texttt{in}}\}$ of a module m, this operation generates anonymized output provenance of that module: $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{out} = \{\texttt{E1}^{\texttt{m}}_{\texttt{out}}, \ldots, \texttt{En}^{\texttt{m}}_{\texttt{out}}\}$. To do so, for every input equivalence class $\texttt{Ei}^{\texttt{m}}_{\texttt{in}}$, a group $\texttt{Gi}^{\texttt{m}}_{\texttt{out}}$ containing the output sets of data records in prov(m).out that are lineage dependent on input sets of data records in $\texttt{Ei}^{\texttt{m}}_{\texttt{in}}$, is constructed. Each group $\texttt{Gi}^{\texttt{m}}_{\texttt{out}}$ gives rise to an output equivalence class $\texttt{Ei}^{\texttt{m}}_{\texttt{out}}$ by masking the identifying attribute values of the data records in $\texttt{Gi}^{\texttt{m}}_{\texttt{out}}$ and by generalizing their quasi-identifying attribute values such that the data records in the group are indistinguishable w.r.t. their quasi-identifying attribute values.

Guarantees:

- **$\mathbf{G_2}$**: for every equivalence class $\texttt{E}^{\texttt{m}}_{\texttt{in}}$ in $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{in}$, anonymizeOutput() generates one lineage-related equivalence $\texttt{E}^{\texttt{m}}_{\texttt{out}}$ in $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{out}$.
- **$\mathbf{G_3}$**: anonymizeOutput() preserves k-group anonymity degree. That is, the number of output sets of data records in an output equivalence class $\texttt{E}^{\texttt{m}}_{\texttt{out}}$, generated by anonymizeOutput(), is equal to the number of input sets of data records in the input equivalence class $\texttt{E}^{\texttt{m}}_{\texttt{in}}$ that is used as input to that operation.

ConstructInputRecords(m, prec(m)). construct input equivalence classes for m given the output equivalence classes of its preceding modules prec(m). We distinguish two cases:

Case 1: m is preceded by one module: prec(m) = {m′} There are data links connecting the output ports of m′ to the input ports of m. Given the anonymized output provenance $\texttt{prov}^{\texttt{a}}(\texttt{m}').\texttt{out} = \{\texttt{E1}^{\texttt{m}'}_{\texttt{out}}, \ldots, \texttt{En}^{\texttt{m}'}_{\texttt{out}}\}$ of m′, this operation generates the anonymized input provenance of m, $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{in} = \{\texttt{E1}^{\texttt{m}}_{\texttt{in}}, \ldots, \texttt{En}^{\texttt{m}}_{\texttt{in}}\}$, as follows: For every output equivalence class $\texttt{Ei}^{\texttt{m}'}_{\texttt{out}}$, a group $\texttt{Gi}^{\texttt{m}}_{\texttt{in}}$ is constructed containing the input sets of data records in prov(m).in that are lineage dependent on output sets of data records in $\texttt{Ei}^{\texttt{m}'}_{\texttt{out}}$. The data records in $\texttt{Gi}^{\texttt{m}}_{\texttt{in}}$ are, then, anonymized by masking their identifying attribute values, and by replacing their quasi-identifying attribute values, with the values used in their lineage-dependent data records in $\texttt{Ei}^{\texttt{m}'}_{\texttt{out}}$. Thereby, each group $\texttt{Gi}^{\texttt{m}}_{\texttt{in}}$ gives rise to an input equivalence class $\texttt{Ei}^{\texttt{m}}_{\texttt{in}}$.

Case 2: m is preceded by multiple modules. Suppose that $\texttt{prec}(\texttt{m}) = \{\texttt{m}_1, \texttt{m}_2\}$. Cases, where a module has more than two preceding modules, are handled in the same manner. Given the anonymized output provenance $\texttt{prov}^{\texttt{a}}(\texttt{m}_1).\texttt{out}$ of $\texttt{m}_1$ and the anonymized output provenance $\texttt{prov}^{\texttt{a}}(\texttt{m}_2).\texttt{out}$ of $\texttt{m}_2$, the anonymized input provenance $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{in}$ of m is obtained using the following process:

For pair $(\texttt{Ei}^{\texttt{m}_1}_{\texttt{out}}, \texttt{Ej}^{\texttt{m}_2}_{\texttt{out}})$ in $(\texttt{prov}^{\texttt{a}}(\texttt{m}_1).\texttt{out}, \texttt{prov}^{\texttt{a}}(\texttt{m}_1).\texttt{out})$, if there exists a data record d in prov(m).in that is lineage dependent on a data record in $\texttt{Ei}^{\texttt{m}_1}_{\texttt{out}}$ and a data record in $\texttt{Ej}^{\texttt{m}_2}_{\texttt{out}}$, a group $\texttt{Gij}^{\texttt{m}}_{\texttt{in}}$ containing the data records in $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{in}$ that are lineage-dependent on data records from both equivalence classes $\texttt{Ei}^{\texttt{m}_1}_{\texttt{out}}$ and $\texttt{Ej}^{\texttt{m}_2}_{\texttt{out}}$, is constructed. The data records in $\texttt{Gij}^{\texttt{m}}_{\texttt{in}}$ are anonymized, thereby giving rise to $\texttt{Eij}^{\texttt{m}}_{\texttt{in}}$, as follows. The identifying attribute values of the data records in $\texttt{Gij}^{\texttt{m}}_{\texttt{in}}$ are masked, and their quasi-identifying value attributes are replaced by the attribute values of their lineage-wise corresponding data records in $\texttt{Ei}^{\texttt{m}_1}_{\texttt{out}}$ and $\texttt{Ej}^{\texttt{m}_2}_{\texttt{out}}$.

Guarantees:

- **$\mathbf{G_4}$**: Using the operation ConstructInputRecords(), an output equivalence class of a module in prec(m) contributes to one lineage-related input equivalence class of m.
- **$\mathbf{G_5}$**: The operation constructInputRecord() preserves k-group anonymity. In other words, the number of input sets of data records in an input equivalence class $\texttt{E}^{\texttt{m}}_{\texttt{in}}$ is equal to or greater than the number of output sets of data records in the corresponding output equivalence classes of the preceding modules.

4.1 Privacy Analysis

We will show, in this section, that an adversary cannot break k-anonymized workflow provenance that is obtained using Algorithm 1. In doing so, we need to show that: i)- The data records in the anonymized input provenance $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{in}$ (resp. anonymized output provenance $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{out}$) of every identifier input (resp. identifier output) of a module m in w, belong to equivalence classes of size $\geq k_i^m$ (resp. $\geq k_o^m$). ii)- The data records in an equivalence class in $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{in}$ (resp. $\texttt{prov}^{\texttt{a}}(\texttt{m}).\texttt{out}$) cannot be distinguished by examining their lineage (i.e., by examining the data records they have been (transitively) generated from or the data records that they have (transitively) contributed to within workflow executions). To do so, we present in what follows a lemma and a theorem, each of which is accompanied by proof.

Lemma 1. An input equivalence class (resp. output equivalence class) of a given module:

(1) is lineage-related with at most one input equivalence class and one output equivalence class of a different module in the same workflow, and
(2) is lineage-related with one output equivalence class (resp. input equivalence class) of the same module, and
(3) is not lineage-related with any input equivalence class (resp. output equivalence class) of the same module.

Proof. We start by showing (1). Let $\mathtt{m}$ and $\mathtt{m}'$ be two modules in a workflow $\mathtt{w}$, and let $\mathtt{E}^{\mathtt{m}}_{\mathtt{in}}$ and $\mathtt{E}^{\mathtt{m}}_{\mathtt{out}}$ be an input and output equivalence classes of $\mathtt{m}$. There are three possible cases:
a)- There is exist a dataflow path connecting $\mathtt{m}$ to $\mathtt{m}'$ in $\mathtt{w}$. For ease of exposition, we denote $\mathtt{m}$ by $\mathtt{m}_1$, and $\mathtt{m}'$ by $\mathtt{m}_n$, and, therefore, the data flow path connecting $\mathtt{m}$ to $\mathtt{m}'$, can be represented by the sequence $(\mathtt{m}_1, \ldots, \mathtt{m}_n)$, with $n \geq 2$. The sequence $(\mathtt{m}_1, \ldots, \mathtt{m}_n)$ denotes a dataflow, i.e. there are data links connecting the output ports of $\mathtt{m}_i$ to the input ports of $\mathtt{m}_{i+1}$ for $i \in [1, n-1]$. Given the guarantees G_2 and G_4, it follows that

- Every input equivalence class in $\mathtt{m}_i$ gives rise to one lineage-related output equivalence class of $\mathtt{m}_i$ for $i \in [1, n]$.
- Every output equivalence class in $\mathtt{m}_i$ gives rise to one lineage-related input equivalence class of $\mathtt{m}_{i+1}$ for $i \in [1, n-1]$.

Given that we consider acyclic workflow, a module cannot appear twice in the dataflow path $(\mathtt{m}_1, \ldots, \mathtt{m}_n)$, which allows us to conclude that every input or output equivalence class of $\mathtt{m}_1$ gives rise to one lineage-related input equivalence class for $\mathtt{m}_n$ and one lineage-related output equivalence class for the output of $\mathtt{m}_n$. Given that we use $\mathtt{m}_1$ to denote $\mathtt{m}_1$ and $\mathtt{m}_n$ to denote $\mathtt{m}'$, we can conclude that $\mathtt{m}'$ has one input equivalence class and one output equivalence class that are lineage-related with $\mathtt{E}^{\mathtt{m}}_{\mathtt{in}}$ (resp. $\mathtt{E}^{\mathtt{m}}_{\mathtt{out}}$).
b)- There is exist a dataflow path connecting $\mathtt{m}'$ to $\mathtt{m}$ in the workflow. The same analysis in (a) allows to conclude that $\mathtt{m}'$ has one input equivalence class and one output equivalence class that are lineage-related with $\mathtt{E}^{\mathtt{m}}_{\mathtt{in}}$ (resp. $\mathtt{E}^{\mathtt{m}}_{\mathtt{out}}$).
c)- There does not exist a data flow path connecting $\mathtt{m}$ to $\mathtt{m}'$, or vice-versa. Given that we consider a data-driven workflow execution module, the data records used and generated by $\mathtt{m}$ cannot possibly contribute to the data records used and generated by $\mathtt{m}'$, and vice-versa. It follows from Definition 4.1 that the equivalence classes associated with the input or output of $\mathtt{m}$ cannot be lineage-related to the equivalence classes associated with the input or output of $\mathtt{m}$.

(a), (b) and (c) allows us to conclude (1).
We now show (2). According to G_2, given a module $\mathtt{m}$ in a workflow $\mathtt{w}$, the operation anonymizeOutput() generates one lineage-related output equivalence class $\mathtt{E}^{\mathtt{m}}_{\mathtt{out}}$ of $\mathtt{m}$, for every input equivalence class $\mathtt{E}^{\mathtt{m}}_{\mathtt{in}}$ of $\mathtt{m}$. Given that $\mathtt{w}$ is acyclic, it follows that $\mathtt{E}^{\mathtt{m}}_{\mathtt{in}}$ contains all the data records bound to the input of $\mathtt{m}$ that contributed to $\mathtt{E}^{\mathtt{m}}_{\mathtt{out}}$, and the data records in $\mathtt{E}^{\mathtt{m}}_{\mathtt{in}}$ contribute to no data records bound to the output of $\mathtt{m}$, other than those in $\mathtt{E}^{\mathtt{m}}_{\mathtt{out}}$. In other words, $\mathtt{E}^{\mathtt{m}}_{\mathtt{out}}$ is the only output equivalence class of $\mathtt{m}$ that is lineage-related with $\mathtt{E}^{\mathtt{m}}_{\mathtt{in}}$, and $\mathtt{E}^{\mathtt{m}}_{\mathtt{in}}$ is the only input equivalence class of $\mathtt{m}$ that is lineage-related with $\mathtt{E}^{\mathtt{m}}_{\mathtt{out}}$.
We now show (3). Given that we consider acyclic workflows, input data records (resp. output data records) of a given module cannot possibly contribute data records bound to the same input module (resp. module output). It follows then that an input equivalence class (resp. output equivalence class) of a given module is not lineage-related with any input equivalence class (resp. output equivalence class) of the same module. □

Theorem 4.2 (Soundness). *The workflow provenance generated for the provenance* $\mathtt{prov}(\mathtt{w})$ *of a workflow* $\mathtt{w}$ *by Algorithm 1 using as input the k-group anonymity degree:*

$$\mathtt{kg}^{\mathtt{max}} = \mathtt{max}\Big(\bigcup_{\mathtt{m_j} \in \mathtt{WF.modules}} \{\mathtt{kg}^{\mathtt{m_j}}_{\mathtt{i}}, \mathtt{kg}^{\mathtt{m_j}}_{\mathtt{o}}\}\Big) \tag{1}$$

is k-annoymized.

Proof. To prove this theorem, we need to show: i) that every equivalence class in $\mathtt{prov}^{\mathtt{a}}(\mathtt{m}).\mathtt{in}$ of an identifier input of (resp. $\mathtt{prov}^{\mathtt{a}}(\mathtt{m}).\mathtt{out}$ of an identifier output) of every module $\mathtt{m}$ in the workflow $\mathtt{w}$, contains at least $\mathtt{k}^{\mathtt{m}}_{\mathtt{i}}$ (resp. $\mathtt{k}^{\mathtt{m}}_{\mathtt{o}}$) data records, and ii) that data records in an equivalence class of the input or output of $\mathtt{m}$ cannot be distinguished by examining their lineage within the executions of $\mathtt{w}$ (see the problem statements in Section 2).

The input equivalence classes of the initial module in the workflow are generated using anonymizeInitialInput. According to G_1, the input equivalence classes generated by anonymizeInitialInput are k-grouped using the k-group anonymity degree $\mathtt{kg}^{\mathtt{max}}$. Such equivalence classes give rise to other equivalence classes by repeatedly applying the the operations anonymizeOutput() and ConstructInputRecords(). According to the guarantees G_3 and G_5 , such operations preserve k-group anonymity. It follows, then, that every equivalence class of an input (or output) of a module $\mathtt{m}$ that is generated by the algorithm contains a number of input (or output) sets that is equal to or greater than $\mathtt{kg}^{\mathtt{max}}$. In other words, the equivalence classes in $\mathtt{prov}^{\mathtt{a}}(\mathtt{m}).\mathtt{in}$ (resp. $\mathtt{prov}^{\mathtt{a}}(\mathtt{m}).\mathtt{out}$) contain at least $\mathtt{kg}^{\mathtt{max}} \cdot \mathtt{l}^{\mathtt{m}}_{\mathtt{in}}$ data records (resp. $\mathtt{kg}^{\mathtt{max}} \cdot \mathtt{l}^{\mathtt{m}}_{\mathtt{out}}$ data records). Given that, $\mathtt{kg}^{\mathtt{max}} \cdot \mathtt{l}^{\mathtt{m}}_{\mathtt{in}}$ is equal to or greater than $\mathtt{kg}^{\mathtt{m}}_{\mathtt{i}} \cdot \mathtt{l}^{\mathtt{m}}_{\mathtt{i}}$, which by definition is equal to or greater than $\mathtt{k}^{\mathtt{m}}_{\mathtt{i}}$. It follows that the equivalences classes in $\mathtt{prov}^{\mathtt{a}}(\mathtt{m}).\mathtt{in}$ contain at least $\mathtt{k}^{\mathtt{m}}_{\mathtt{i}}$ data records. Similarly, given that, $\mathtt{kg}^{\mathtt{max}} \cdot \mathtt{l}^{\mathtt{m}}_{\mathtt{o}}$ is equal to or greater than $\mathtt{kg}^{\mathtt{m}}_{\mathtt{o}} \cdot \mathtt{l}^{\mathtt{m}}_{\mathtt{o}}$, which by definition is equal to or greater than $\mathtt{k}^{\mathtt{m}}_{\mathtt{o}}$. It follows that the equivalences classes in $\mathtt{prov}^{\mathtt{a}}(\mathtt{m}).\mathtt{out}$ contain at least $\mathtt{k}^{\mathtt{m}}_{\mathtt{o}}$ data records. Thereby, we have just shown (i).

We will now show (ii). Every input equivalence class $\mathtt{E}^{\mathtt{m}}_{\mathtt{in}}$ in $\mathtt{prov}^{\mathtt{a}}(\mathtt{m}).\mathtt{in}$ is, according to lemma 1, not lineage-related to any equivalence class of the same input, and it is lineage-related with at most one input equivalence class $\mathtt{E}^{\mathtt{m}'}_{\mathtt{in}}$ of any other module input in the workflow, and is lineage-related with at most one output equivalence class $\mathtt{E}^{\mathtt{m}'}_{\mathtt{in}}$ of any module in the workflow (including $\mathtt{m}$). The data records in any lineage-related input equivalence class $\mathtt{E}^{\mathtt{m}'}_{\mathtt{in}}$ or output equivalence class $\mathtt{E}^{\mathtt{m}'}_{\mathtt{out}}$) do not carry identifying attribute values and are indistinguishable w.r.t. their quasi-identifying attribute values. Therefore, an adversary is unable to distinguish between the data records in $\mathtt{E}^{\mathtt{m}}_{\mathtt{in}}$ of an input equivalence class of a module $\mathtt{m}$ by examining the data records of its lineage-related equivalence classes. The same reasoning can be applied to the output equivalences classes in $\mathtt{prov}^{\mathtt{a}}(\mathtt{m}).\mathtt{out}$. This implies that data records in an equivalence class $\mathtt{E}$ cannot be distinguished by examining the records of any of its lineage-related equivalence class $\mathtt{E}'$. And, by recursion, the data records in $\mathtt{E}'$ cannot be distinguished by examining the data records in the equivalence classes that are lineage-related with E'., etc. That is, the data records in an equivalence class cannot be distinguished by examining their lineage, thereby showing (ii). □

5 GROUPING OF DATA RECORD SETS

Consider that the initial module of a workflow took as input the following sets of records $D = D_1 \cup \cdots \cup D_n$, with $l = \min_{i \in [1 \cdots n]} |D_i|$. Consider now that the target k-group anonymity degree is kg, i.e., the target anonymity degree $k = kg * l$. If $k > 1$, i.e. $kg > 1$, then the method for anonymization that we have described so far states that the inputs sets in $D = D_1 \cup \cdots \cup D_n$ need to be grouped (unionied) into equivalence sets of a magnitude at least equal to k. In doing so, the method we presented does not specify which inputs sets in D to group together to form equivalence classes. A naïve solution to this problem would be to union all the datasets in D into a single group, i.e. a single equivalence class, and anonymize the quasi-identifying attributes of the data records accordingly. However, the records obtained using this approach are likely to be useless since the scientists will not be able to distinguish between any of the data records used as input to the module in question. A more desirable solution would, therefore, generate groups that have a small magnitude of at least k, and yet try to keep the magnitude of such groups as close as possible to k. We can formally define the above problem as follows[3].

Given sets of data records $D = D_1 \cup \cdots \cup D_n$, and an anonymity degree k, group the sets D_i, $i = 1 \ldots n$, into groups $G = G_1 \cup \cdots \cup G_m$, $m \leq n$, such that:

(1) $|G_i| \geq k$, and
(2) $\max_{i=1 \ldots m}(|G_i|)$ is minimal.

The above problem can be viewed as a variant of the scheduling problem [29], in which the datasets D_i represent independent and non-preemptive jobs, and the cardinalities of such datasets represent jobs' lengths. There is a maximum of n machines. If a machine is used then its load must be greater or equal to k. The objective of such a scheduling problem is to minimize the makespan. To our knowledge, there does not exist any variant of the scheduling problem in the literature that meets the above criteria. We prove in the technical report that this is a strongly NP-hard problem, by reducing the *3-partition* problem to the above problem (see [5], page 14, for the proof).

Given that our problem is strongly NP-hard, we turn our attention to approximation algorithms. In particular, we devised the minimizeG integer problem (see below) to produce a good quality solution. x_{ij} is an integer that can takes the value 1 if the set D_i participates in the union that forms the group G_j, and 0, otherwise (Constraint C_4). y_j is an integer that can takes the value 1 if the group G_i contains at least one set in D, and 0, otherwise (Constraint C_5). $card_i$ represents the cardinality of the set D_i. Constraints (C_1) states that a set D_i must participate in the union of exactly one group. Constraint (C_2) specifies that a group G_j can have a cardinality of 0 (when y_j equals to 0), or a cardinality greater or equal to k (when y_j equals to 1). Constraint (C_3) specifies that the cardinalities of the groups $G_1, \ldots, G_n$ is smaller than a variable Z, which represents the makespan. The objective of the integer program is, therefore, to minimize the value of Z. Constraints (C_6) states that y_j is definitely equal to 1 if x_{ij} is equal to 1. More specifically, if the set D_i has been affected to the group G_j (i.e., $x_{ij} = 1$), then the group G_j contains at least one set (i.e.$y_j = 1$).

$$\begin{array}{llll}
\text{minimizeG} & Z & & \\
\text{subject to} & \sum_{j\in\{1,\cdots,n\}} x_{ij} \; = \; 1, & i = 1, \ldots, n & (C_1)\\
& \sum_{i\in\{1,\cdots,n\}} card_i \cdot x_{ij} \geq k.y_j, & j = 1, \ldots, n & (C_2)\\
& \sum_{i\in\{1,\cdots,n\}} card_i \cdot x_{ij} \leq Z, & j = 1, \ldots, n & (C_3)\\
& x_{ij} \in \{0,1\}, & i, j = 1, \ldots, n & (C_4)\\
& y_j \in \{0,1\}, & j = 1, \ldots, n & (C_5)\\
& y_j \geq x_{ij}, & i, j = 1, \ldots, n & (C_6)
\end{array}$$

Notice that we need to invoke the minimiseG program only once per workflow to identify the way the input sets of the initial module are to be grouped. The output of the initial module, as well as the input and output of the remaining modules in the workflow, use groupings that are derived based on lineage information (see Algorithm 1, lines 4 – 8).

6 VALIDATION

We implemented the solution that we have described in this paper using Python 2.7. We used the COIN Branch and Cut solver (CBC) provided by the LP Modeler Pulp[4] for solving the integer program MinimizeG presented in Section 5.

6.1 Experimental Setup

There is no existing solution that we can utilize as a base solution for comparison. Nonetheless, the approach that we have described raises the question as to which parameters impact the quality of the provenance anonymized using our solution. The analysis of the k-group anonymity degree computed for a workflow (see Equation 1), which dictates the degree of generalization, i.e., information loss, to be applied to the provenance of a workflow, reveals that the quality of the provenance (level of generalization) can be influenced by the anonymity degrees and magnitudes of the sets of data records used and generated by the parameters (inputs or outputs) of the workflow's modules. Note that on the other hand, the same equation allows us to rule out the topology (structure) of the workflow as a possible influencing factor. Because of this, we focus in our experiment on assessing the impact that the difference in the anonymity degrees and the magnitudes of the sets associated with two module parameters, which we take w.l.o.g to be the input and output of a module, has on the quality of anonymized provenance.

To be able to control the parameters of our experiment, we implemented a python program that given l^m_{in}, l^m_{out} and a number of module invocations, automatically generates module provenance. The provenance identifies the data records that are automatically generated by our tool. Regarding the content of data records, we use the Adult dataset [14], a de facto benchmark for anonymization solutions.

To assess the quality of anonymized data, we used the average equivalence class size [19] and the discernability metric [19]. For conciseness sake, we focus in what follows on the average equivalence class size. Readers interested in examining the results for discernability are referred to [5].

The average equivalence class measures how well equivalence classes created by the anonymization do not exceed what is required by the anonymization degree k. It can be defined as follows: $AEC(DS^*) = \frac{|DS|}{|EQ(DS^*)| \cdot k}$
where $EQ(DS^*)$ represents the set of equivalence classes created as a result of anonymizing DS, i.e., $|EQ(DS^*)|$ is the number of equivalence classes created. k represents the k-anonymity degree

[3]The problem statement formulated in Section 2.3 contains this condition.

[4] https://pypi.org/project/PuLP

required. The best value of AEC is 1. It means that none of the equivalence classes created as a result of anonymization exceeds the required anonymity degree when performing the generalization. We chose AEC as a measure because it is a good indicator for the quality of the anonymized data, with respect to a minimum requirement that is set by the anonymity degree.

As well as examining the impact of the anonymity degree and magnitude of sets of data records on the quality of anonymized provenance (in Sections 6.2, 6.3 and 6.4), we assess the utility of anonymized workflow provenance by examining the degree to which they can be used for answering workflow provenance challenge queries using real-world workflows [23] (in Section 6.5), and assess the efficiency of our solution (in Section 6.6).

6.2 Impact of the Disparity of k^m_{in}, k^m_{out} on the Quality of Anonymization

Given the provenance of the module m, one would expect that disparity between k^m_{in} and k^m_{out}, or more specifically between the ratios kg^m_{in} and kg^m_{out} have an impact on the quality of the obtained anonymized input and output datasets of m. Specifically, if kg^m_{out} is larger than kg^m_{in} then the inputs records of m will be grouped into equivalence classes that are beyond what is required by k^m_{in} to meet k^m_{output}. Thereby, the average equivalence class of the obtained anonymized input datasets is likely to suffer as a results. On the contrary, if kg^m_{in} and kg^m_{out} are close then one would expect that the average equivalence class for both the input and output anonymized datasets to be of good quality. To assess this intuition, we ran an experiment in which:
1) We generated the provenance of a module m that associates sets of input data records with sets of output data records. (Note that we ran our experiments using different numbers of module invocations, namely 50, 100, 200, 300, 400 and 500 module invocation. The results we obtained presented similar trends. We, therefore, focus on reporting on the results obtained for 100 module invocations.) We set l^m_{in} and l^m_{out} to the same value, viz. 1. Specifically, an input (resp. output) set of data records that are used or generated by m has a magnitude between 1 and 3 (resp. 1 and 4). (We did so to examine the interplay between k^m_{in} and k^m_{out}. Later on, we report on an experiment that we ran to assess the impact of the magnitudes of the sets of data records and their variability.) **2)** We then set the value of k^m_{in} to 2, and anonymized the input and output datasets using our method by varying the value of k^m_{in} between 2 and 20.

We ran this experiment three times. Figures 3 illustrates the average of the AEC obtained. Notice that the AEC of the output dataset is close (if not equal) to 1, indicating that the quality of the anonymized dataset is optimal as far as the constructed equivalence classes are concerned. On the other hand, we observe that the AEC of the input dataset increases as the disparity between k^m_{in} and k^m_{out} increases. This confirms our initial observation.

6.3 Impact of the Disparity of k^m_{in}, l^m_{in} on the Quality of Anonymization

Another aspect that can impact the quality of the anonymized datasets is the difference between the anonymity degree and the magnitude of the smallest input (resp. output) set of data records. Without loss of generality, let us consider the input of a module m. If l^m_{in} is greater than the anonymity degree k^m_{in}, then the magnitude equivalence classes obtained as a result of anonymization will be greater than what is required by k^m_{in}, thereby impacting negatively the AEC. To empirically examine this aspect, we set

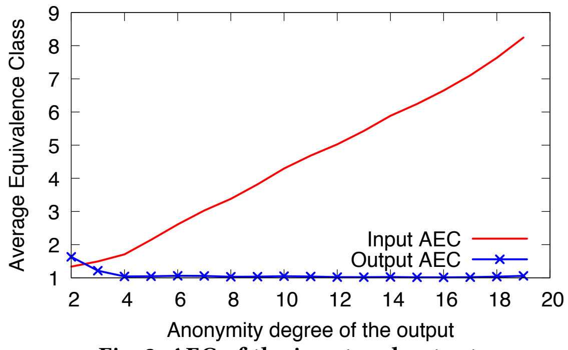


Fig. 3: AEC of the input and output.

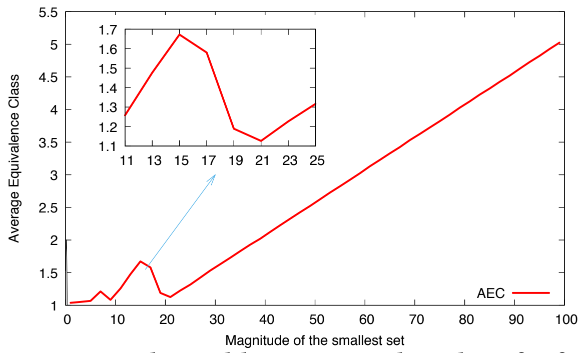


Fig. 4: AEC obtained by sweeping the value of l for $k = 20$.

the anonymity degree k^m_{in} to 20. We then varied the parameter l^m_{in} between 1 and 99, with a unit of 2, i.e., [1, 3, . . . , 97, 99]. In particular, for a given value of l^m_{in}, the input sets generated for a module have a magnitude between l^m_{in} and $l^m_{in} + 3$. In other words, the input sets have a magnitude that is close to the value of l^m_{in}. We did so to factor out the impact that the variability in the magnitude of the input sets, which we will examine later on.

For each value of l^m_{in}, we generated the provenance for the module m (100 module invocation) and anonymized the obtained input dataset. We ran this experiment three times, and averaged the results, which are depicted in Figure 4 for the average equivalence class. The chart can be partitioned into two parts. The first part where l^m_{in} ranges from 1 to 20, and in the second part where it ranges from 20 to 100. In the first part, we notice that the AEC remains relatively close to 1 until it reaches the value of 15 and 17 where we notice an increase of the AEC 1.5. The AEC then decreases to values that are close to 1 for l^m_{in} values of 19 and 21. To explain this increase in the AEC, consider the case where $l^m_{in} = 15$. The magnitude of the input sets ranges between 15 and 18 according to the above experiment setting. Consequently, the magnitude of the sets obtained using the grouping ranges between 30 and 36. Indeed, an input set on its own has a magnitude lower than the required anonymity degree of 20, and two unionied input sets will definitely have a magnitude between 30 and 36, which is larger than the required anonymity degree. This explains the fact that the AEC value is close to 1.5. In the second phase, we observe that the value of the AEC grows linearly as the magnitude of the smallest set grows. This can be explained by the following. For values of l^m_{in} greater than 20, no set grouping is actually performed: the magnitude of the input set is greater than the required anonymity degree. The larger is the magnitude of l^m_{in}, the larger the disparity between l^m_{in} and $k^m_{in} = 20$, and subsequently, the larger is the AEC.

6.4 Impact of the Disparity of the Size of Input (resp. Output) Sets

In the experiment that we ran this far, we assumed that the size of the input (resp. output) set of data records are close to 1^{m}_{in} (resp. 1^{m}_{out}). We have examined the provenance of the workflows available in ProvBench[5], namely the workflow provenance collected the workflow systems Taverna and Wings (120 workflows). For each workflow and each of its modules, we computed 1^{m}_{in} and 1^{m}_{out}. We then examined the variability of the magnitude of the input and output sets. This analysis revealed that in the majority of the cases the magnitude of the sets used and output by the modules that compose the workflow follows a uniform distribution. However, for an important proportion of the modules ($\approx$ 15%), we observed that the distribution is instead geometric in the sense that the input (resp. output) sets have a magnitude that is close to 1^{m}_{in} (resp. 1^{m}_{out}).

Given the above results, we decided to empirically examine the variability of the magnitude of the parameter sets on the quality of the anonymization considering the two distributions. For the random uniform distribution, we used three distributions where the maximum magnitude of a set is 20, 50 and 100, respectively. Regarding the geometric distribution, we used three distributions with the probabilities of 30, 50 and 80, respectively.

We then ran an experiment in which we computed the AEC by varying the anonymity degree k_{in} between 2 and 20. The results of the experiment for geometric distributions are illustrated in Figure 5, and those obtained for uniform distribution are illustrated in Figure 6. For geometric distribution, we observe that the higher the success probability, the better the AEC obtained. For example, the AEC quickly converges to the value of 1 when the success probability is equal to 0.8. On the other hand, the AEC converge to 1 only when the anonymity degree reaches 11 when the success probability is set to 0.3. That said, overall, geometric distribution delivers better results compared with uniform distribution: the AEC is much smaller. This can be explained by the fact that the variability in the magnitudes of the sets of data records is smaller in the case of the geometric distribution. And, the lower the variability of the magnitude of the data record sets, the better is the grouping of sets in the sense that it yields groups (i.e. equivalence classes) with magnitudes close to k, and therefore the better the AEC obtained (close to 1).

6.5 Assessing Utility Using Real Workflows

We assessed the degree to which anonymized provenance can be used to answer the following 3 queries that are representative of the queries defined by the workflow provenance challenge community [23].[6].

q_1 Find the workflow executions that led to a given record in the workflow results.

q_2 Find the input data records that contributed to a given data record in the workflow result.

q_3 Find the difference between two workflow execution.

For this experiment, we used 14 real-world Taverna workflows. The size of the workflows ranges from 3 modules to 24 modules, and have different structures patterns. We ran each workflow 30 times, and captured the provenance obtained using the Taverna workflow systems. We then anonymized the provenance by

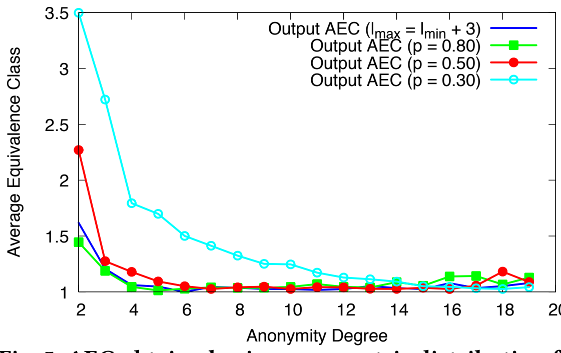


Fig. 5: AEC obtained using a geometric distribution for the magnitude of 1

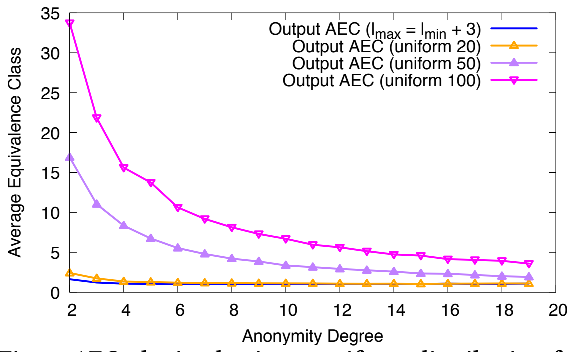


Fig. 6: AEC obtained using a uniform distribution for the magnitude of 1

varying the group anonymity degree kg^{max} from 1 to 10, and examined whether queries of the form listed above can be answered using the anonymized provenance.

Regarding q_1 and q_2, a user is presented with anonymized workflow provenance, and as such cannot pinpoint a single data record in the results that can be used as input to q_1 and q_2. Instead, s/he chooses a (set) of data records that belong to the same equivalence class. As expected, the larger is the anonymity degree, the larger is the set of data records to be considered (see Table 7). Note that on the other hand, the query results obtained had 100% precision and recall, regardless of the value of group-anonymity degree used. This was possible thanks to the fact that our anonymization method preserves lineage across data records.

Table 7: Size of the data record sets used as input to q_1 and q_2 given kg^{max}, averaged over the 14 workflows.

kg^{max}	1	2	4	6	8	10
avg size of the set of data records	3	6	11	20	25	33

Regarding q_3, the provenance challenge does not formally specify what it is meant by the difference of workflow executions [23]. That said, this question has later been thoroughly examined by Bao *et al.* [4]. They defined the difference between workflow executions of the same workflow specification using the edit distance which is the minimum number of edit operations that transform one provenance graph structure to the other. Using this definition, the edit distance between every pair of anonymized provenance graphs (of the 14 workflow specifications that we used) was the same as the edit distance computed using their counterpart original provenance graphs. This can be explained by the fact that our anonymization solution preserves the structure of the provenance graph as-is (since one of the requirements that we set is to preserve lineage information). Therefore, the

[5]https://github.com/provbench

[6]We could not use the provenance challenge queries as they are since they were specified for a single specific workflow on image processing.

original provenance graph of a given workflow specification is homomorphic to its anonymized counterpart.

This experiment has shown the utility of the workflow provenance anonymized using our solution since we were able to answer the three classes of queries. This evaluation exercise has also shown that for q_1 and q_2, the input of the query size (number of records) depends on the anonymity degree. Smaller anonymity degrees allow having smaller sets of data records that can be used for the queries, and vice-versa.

6.6 Efficiency

The only operation that is costly in the anonymization solution presented is the grouping of sets of data records, which we implemented using the integer program `minimizeG` (Section 5). Note, however, that such an operation is performed only once for the input of the initial module of the workflow. Indeed, the remaining parameters of the modules that compose the workflow apply the same grouping as the one applied to the input of the initial module. That said, we investigated the cost of such an operation to group n data sets, where n = 50, 100, 100, $\cdots$, 500. As expected, this experiment showed that time required increased as does the number of module invocations. Interestingly, the experiment also showed that the time required for performing the grouping of the sets of data records is primarily impacted by the distribution of the magnitude of the sets of data records to be grouped.

This experiment showed that using a uniform distribution, the range has little impact on the time required for grouping: it took between 16 and 18 seconds. On the other hand, it showed that sets that follow a geometric distribution with high success probability (50% and higher) require a considerable time compared to sets that follow a uniform distribution: it took up to 17 minutes. This can be explained by the fact that for a geometric distribution with high success probability, the majority of the sets have the same (or close) magnitudes. Therefore, many of the groupings that are explored by the integer program yield similar values for the objective function, hence recording little progress. The above results prompted us to develop an alternative solution when the magnitudes of the sets follow a geometric distribution with high success probability. The alternative solution we developed is simple, yet it group sets of data records in the orders of microseconds with a small impact on the AEC, which was higher by a margin of 0.03 on average compared with the situation in which we used our integer program `minimizeG`. More details can be found in the technical report [5].

7 CONCLUSIONS

We presented, in this paper, a solution for systematically anonymizing the provenance of collection-oriented workflows. Evaluation exercises allowed us to tease apart the aspects that impact the quality of the anonymization, namely the disparity between the anonymity degrees of the input and output sets of a module (or more generally the inputs and outputs of the modules that compose the workflow), the disparity between the anonymity degree and the magnitude of the sets of data records, and the distribution of the magnitudes of the record sets. We also examined the utility of the anonymized provenance using real-world workflows, and assessed the efficiency of our solution. In our ongoing work, we are investigating the applicability of our solution to anonymization techniques, other than k-anonymity, e.g., l-diversity and t-closeness [13]. We are also investigating the incorporation of vocabularies (hierarchies of concepts) to our solution. In the solution we presented, we substitute each quasi-identifier attribute value with a set containing the values that that attribute takes given a group (i.e. equivalence class) of data records. We will investigate how the use of vocabularies can be incorporated in our solution for generalizing the values of quasi-identifier attributes.

ACKNOWLEDGMENTS

I warmly thank Eun Jung Kim and Florian Sikora from the Paris-Dauphine University for the enlightening discussions on scheduling problems and integer programming. I would also like to thank the reviewers for their time and constructive comments, which helped improve the quality of the paper.

REFERENCES

[1] K. Abouelmehdi, A. B. Hssane, and H. Khaloufi. Big healthcare data: preserving security and privacy. *J. Big Data*, 5:1, 2018.
[2] AEPD. k-anonymity as a privacy measure. *Spanish Agency for Data Protection*, 2018. https://www.aepd.es/media/notas-tecnicas/nota-tecnica-kanonimidad-en.pdf.
[3] V. Ayala-Rivera, P. McDonagh, T. Cerqueus, and L. Murphy. A systematic comparison and evaluation of k-anonymization algorithms for practitioners. *Trans. Data Privacy*, 7(3):337–370, 2014.
[4] Z. Bao, S. C. Boulakia, S. B. Davidson, et al. Differencing provenance in scientific workflows. In *ICDE*, pages 808–819. IEEE, 2009.
[5] K. Belhajjame. On Anonymizing the Provenance of Collection-Based Workflows. Research report, Université Paris-Dauphine, PSL Research University, Jan. 2020. https://hal.inria.fr/hal-02430624/file/techreport.pdf.
[6] K. Belhajjame, N. Faci, Z. Maamar, V. A. Burégio, E. Soares, and M. Barhamgi. Privacy-preserving data analysis workflows for escience. In *EDBT/ICDT Workshops*, volume 2322 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.
[7] O. Biton, S. C. Boulakia, and S. B. Davidson. Zoom*userviews: Querying relevant provenance in workflow systems. In *VLDB*. ACM, 2007.
[8] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, volume 1973, pages 316–330. Springer, 2001.
[9] A. Chebotko, S. Chang, et al. Scientific workflow provenance querying with security views. In *WAIM*, pages 349–356. IEEE CS, 2008.
[10] C. Clifton and T. Tassa. On syntactic anonymity and differential privacy. *Transactions on Data Privacy*, 6(2):161–183, 2013.
[11] R. F. da Silva et al. Automating environmental computing applications with scientific workflows. In *e-Science*, pages 400–406. IEEE, 2016.
[12] S. B. Davidson, S. Khanna, T. Milo, et al. Provenance views for module privacy. In *PODS*, pages 175–186, 2011.
[13] J. Domingo-Ferrer and V. Torra. A critique of k-anonymity and some of its enhancements. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 990–993. IEEE, 2008.
[14] D. Dua and C. Graff. UCI machine learning repository, 2017.
[15] C. Dwork. Differential privacy. In *ICALP*, pages 1–12. Springer, 2006.
[16] R. Filgueira, A. Krause, M. P. Atkinson, et al. dispel4py: An agile framework for data-intensive escience. In *e-Science*, pages 454–464. IEEE, 2015.
[17] E. Griffis, P. Martin, and J. Cheney. Semantics and provenance for processing element composition in dispel workflows. In *WORKS*. ACM, 2013.
[18] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. V. den Bussche. DFL: A dataflow language based on petri nets and nested relational calculus. *Inf. Syst.*, 33(3):261–284, 2008.
[19] K. LeFevre, D. J. DeWitt, and R. Ramakrishnan. Mondrian multidimensional k-anonymity. In *ICDE*, page 25. IEEE, 2006.
[20] T. M. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher. Scientific workflow design for mere mortals. *FGCS*, 25(5):541–551, 2009.
[21] A. Meyerson and R. Williams. On the complexity of optimal k-anonymity. In *PODS*, pages 223–228. ACM, 2004.
[22] P. Missier, N. W. Paton, and K. Belhajjame. Fine-grained and efficient lineage querying of collection-based workflow provenance. In *EDBT*. ACM, 2010.
[23] L. Moreau, B. Ludäscher, I. Altintas, et al. Special issue: The first provenance challenge. *CCPE*, 20(5):409–418, 2008.
[24] M. E. Nergiz, C. Clifton, and A. E. Nergiz. Multirelational k-anonymity. *IEEE Trans. Knowl. Data Eng.*, 21(8):1104–1117, 2009.
[25] N. Park, M. Mohammadi, K. Gorde, et al. Data synthesis based on generative adversarial networks. *PVLDB*, 11(10):1071–1083, 2018.
[26] P. Samarati and L. Sweeney. Generalizing data to provide anonymity when disclosing information (abstract). In *PODS*, page 188. ACM Press, 1998.
[27] J. Sroka, J. Hidders, P. Missier, and C. A. Goble. A formal semantics for the taverna 2 workflow model. *J. Comput. Syst. Sci.*, 76(6):490–508, 2010.
[28] M. Terrovitis, N. Mamoulis, and P. Kalnis. Privacy-preserving anonymization of set-valued data. *VLDB Endowment*, 1(1):115–125, 2008.
[29] F. Werner, L. Burtseva, and Y. Sotskov. *Algorithms for Scheduling Problems*. MDPI, 2018.
[30] K. Wolstencroft et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *NAR*, 41, 2013.

Sharing Computations for User-Defined Aggregate Functions

Chao Zhang
LIMOS, CNRS, University Clermont Auvergne
zhangchaohit13sg@gmail.com

Farouk Toumani
LIMOS, CNRS, University Clermont Auvergne
ftoumani@isima.fr

ABSTRACT

UDAFs (user-defined aggregate functions) are becoming a type of fundamental operators in advanced data analytics. The UDAF mechanism provided by most of the modern systems suffers, however, from at least two severe drawbacks: defining a UDAF requires hardcoding the routine that computes an aggregation, and the semantics of a UDAF is totally or partially unknown to the query processor, which hampers the optimization possibilities. This paper presents SUDAF (Sharing User-Defined Aggregate Functions), a declarative framework that allows users to write UDAFs as mathematical expressions and use them in SQL statements. SUDAF rewrites partial aggregates of UDAFs in users' queries using built-in aggregate functions and supports efficient dynamic caching and reusing of partial aggregates. Our experiments show that rewriting UDAFs using built-in functions can significantly speed up queries with UDAFs, and the proposed sharing approach can yield up to two orders of magnitude improvement in query execution time.

1 INTRODUCTION

An aggregate function has the inherent property of taking several values as input and generating a single value based on specific criteria [17, 25]. This ability to summarize information, the intrinsic feature of aggregation, has always been a fundamental task in data analysis [18, 24]. While earlier data management and analysis systems come equipped with a set of built-in aggregate functions, e.g., max, min, sum and count, it becomes clear that a limited set of predefined functions is not sufficient to cover the needs of the new applications in the age of analytics. In addition to augmenting the set of their built-in functions, most modern systems (e.g., [1, 2, 4, 21, 28, 29]) enable users to extend the system functionalities by defining their own aggregations. The UDAF (User-Defined Aggregate Function) mechanism provides a flexible interface to allow users to define new aggregate functions that can then be used for advanced data analytics, i.e., queries with statistical functions or ML workloads.

Current UDAF mechanisms suffer, however, from at least two drawbacks. Firstly, defining a UDAF is not an easy task since it is up to users to implement the routine that computes their aggregation functions. For example, to write a custom UDAF in Spark SQL [4], a user needs to map the UDAF to four methods: initialize, update, merge and evaluate, a.k.a. the IUME pattern. The user must ensure that the merge method is commutative and associative such that the UDAF can be computed correctly in a distributed architecture. In other words, to take benefit from distributed computations in Spark SQL, it is up to the user to identify whether her function supports partial aggregates (i.e., whether it is an algebraic function [18]). Secondly, the semantics of a UDAF, i.e., computation details, are not fully captured by a query engine, which hampers optimization possibilities. For example, when computing a UDAF that is created using the IUME pattern, a query engine can only be aware of calling an update function if there is a tuple or calling a merge function if there are intermediate results. However, the specific computations that are required to compute update and merge functions are unknown to a query engine since these two functions are hardcoded. The loss of such computation details prevents a query engine from sharing partial results of different UDAFs.

In the context of aggregate queries optimization, materialized views with aggregates or cached queries are among the techniques that can be used to accelerate query processing. In this context, most existing works focus on the data dimension [8, 11, 12, 15], i.e., sharing identical aggregates computed over overlapping range predicates or different data granularities. Admittedly, considering only the data dimension restricts the sharing possibilities to queries with identical aggregation operators. To cope with such a limitation, few works propose to use predefined rules to specify how a given aggregate can be computed from the results of another one [10, 33]. However, such a static approach requires one to explicitly predefine the computation rules across prefixed aggregates, which hinders the optimization for UDAFs defined on the fly.

The objective of this work is twofold: firstly, we aim at giving full flexibility to users by providing a declarative framework that allows them to write UDAFs as mathematical expressions and use them in SQL queries[1]. Then, a UDAF is decomposed into partial aggregates, which are then rewritten using built-in functions, i.e., scalar functions and aggregations. Secondly, our goal is to develop a *dynamic* approach for caching and reusing partial aggregates of UDAFs to optimize the computations of UDAFs. More precisely, we aim at identifying when it is possible to reuse cached partial aggregates of past UDAFs to compute new UDAFs.

Contributions. Our main contributions, implemented in the SUDAF framework, are as follows:

- We present SUDAF, a declarative UDAF framework that allows users to formulate a UDAF as a mathematical expression and use them within SQL queries. When executing a given query with UDAFs, SUDAF identifies appropriate partial aggregations from the mathematical expression of a UDAF and rewrites them using built-in functions of an underlying data management and analysis system.
- We formalize the problem of identifying when a partial aggregate of a given UDAF can be used in the computation of another UDAF as the *sharing problem*, and we show that this problem is undecidable in a general setting.
- To deal with the undecidability of the *sharing problem*, we restrict the set of UDAFs supported in SUDAF by providing classes of primitive functions that can be used to describe mathematical expressions of UDAFs. This practical framework is powerful enough to be used in practical applications. From a theoretical standpoint, we characterize

[1]This approach is more intuitive than programming the procedure of an aggregation, e.g., Wolfram Mathematica provides mathematical expressions to define advanced statistical computation [34].

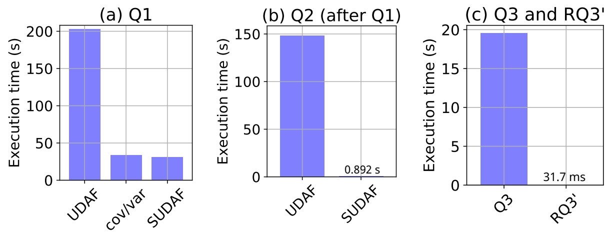


Figure 1: Experiments in PostgreSQL with the TPC-DS dataset (scale = 20). UDAFs theta1() and qm() are created in PL/pgSQL.

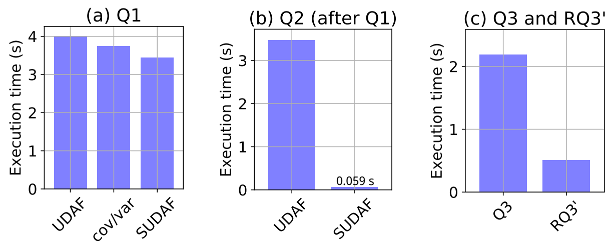


Figure 2: Experiments in Spark SQL with the TPC-DS dataset (scale = 100). UDAFs theta1() and qm() are created using UserDefinedAggregateFunction in Scala.

the sharing problem in SUDAF and provide corresponding sharing conditions (Theorem 4.1). From a practical standpoint, we design an approach based on symbolic representations of mathematical expressions to efficiently verify the proposed conditions.

- We implemented a SUDAF prototype and report on experiments using SUDAF with both PostgreSQL and Spark SQL. Our experiments show that rewriting partial aggregates of UDAFs using built-in aggregates can significantly speed up query execution. In addition, the proposed sharing technique can yield up to two orders of magnitude improvement in query execution time.

The rest of this paper is organized as follows. We present a motivating example to illustrate SUDAF's main features in Section 2. In Section 3, we introduce a canonical form of UDAFs and discuss the sharing problem in this context. In Section 4, we present the SUDAF framework and show that the sharing problem is decidable in this context. In Section 5, we introduce a practical approach, based on symbolic representations of partial aggregates, to solve the sharing problem in the SUDAF framework. In Section 6, we present an experimental evaluation of SUDAF. We discuss related works in Section 7 and conclude in Section 8. All related proofs are included in our online technical report [31].

2 MOTIVATING EXAMPLE

In this section, we present a motivating example demonstrating two SUDAF's functionalities: (i) rewriting UDAFs using built-in functions, and (ii) sharing partial aggregation results between different UDAFs. In addition, we also illustrate how the sharing mechanism can be used to extend query rewriting using aggregate views. In the following example, we consider 4 relations of the TPC-DS [27] dataset, store_sales, store, date_dim and stores.

Suppose that a user wants to analyze the price of every item sold by the stores in the state Tennessee (TN) in the past every year. Specifically, the user has a hypothesis of a *simple linear regression*: $y = \theta_1 x + \theta_0$, where y represents a value in the sales_price column and x a value in the list_price column. Using the least square error function, we have $\theta_1(X,Y) = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2}$, and $\theta_0(X,Y) = avg(Y) - \theta_1 avg(X)$.

One can hardcode θ_1 as a user-defined function and then uses it in an SQL statement, e.g., one writes a piece of Java or Scala code to create θ_1 in Spark SQL (see Scala code in [31]). Assume that a hardcoded user-defined function theta1(), that implements the function $\theta_1()$, is created and the following query Q1 is issued:

```
Q1: SELECT  ss_item_sk, d_year, avg(ss_list_price),
            avg(ss_sales_price),
            theta1(ss_list_price,ss_sales_price)
    FROM    store_sales, store, date_dim
    WHERE   ss_sold_date_sk = d_date_sk and
            ss_store_sk = s_store_sk and s_state = 'TN'
    GROUP BY ss_item_sk, d_year;
```

Alternatively, in SUDAF the function theta1() is defined declaratively by providing its mathematical expression without the needs of any programming effort.

Now, assume that a user defines the expressions of theta1() and avg() and uses them in the query Q1. We illustrate in the rest of this section two benefits of using SUDAF to execute the query Q1: (i) the partial aggregates of theta1() and avg() used in the query Q1 are rewritten into a set of partial aggregates using the built-in functions *sum* and *count*, and (ii) the partial aggregates computed during the execution of Q1 can be cached and reused to compute various other UDAFs.

Rewriting partial aggregates using built-in functions. The first step of processing Q1 in SUDAF is to factor out partial aggregates of theta1() and avg() and rewrite them using built-in functions to compute. More precisely, SUDAF identifies the following 5 partial aggregates in the expression of θ_1: $s_1 = count(), s_2 = \sum x_i, s_3 = \sum x_i^2, s_4 = \sum y_i$ and $s_5 = \sum x_i y_i$. Hence, SUDAF rewrites Q1 to the following query RQ1 where the partial aggregates are first computed and then theta1() is computed using the partial aggregates, $\theta_1 = \frac{s_1 s_5 - s_4 s_2}{s_1 s_3 - (s_2)^2}$.

```
RQ1: SELECT ss_item_sk, d_year, s2/s1 avg_list_price,
            s4/s1 avg_sales_price,
            (s1*s5-s4*s2)/NULLIF((s1*s3-power(s2,2)),0) theta1
     FROM   (SELECT ss_item_sk, d_year, count(*) s1,
                   sum(ss_list_price) s2,
                   sum(power(ss_list_price,2)) s3,
                   sum(ss_sales_price) s4,
                   sum(ss_sales_price*ss_list_price) s5
             FROM  store_sales, store, date_dim
             WHERE ss_sold_date_sk = d_date_sk and
                   ss_store_sk = s_store_sk and
                   s_state = 'TN'
             GROUP BY ss_item_sk, d_year) TEMP;
```

Compared to the original query Q1, RQ1 uses only built-in aggregate functions and hence it is expected to be much more efficient because built-in functions are better handled by existing query optimizers and execution engines than hardcoded user-defined functions. Figure 1 (a) shows that the execution of Q1 using SUDAF on top of PostgreSQL can be $10X$ faster compared to running Q1 directly over PostgreSQL. Similar results can be observed in Figure 2 (a) using SUDAF on top of Spark SQL, where Q1 is $1.25X$ faster compared to the direct execution of Q1 over Spark SQL. To be fair in our analysis, we should mention that in the context of PostgreSQL and Spark SQL systems, where the covariance (*cov*) and the variance (*var*) are built-in functions, an alternative and efficient implementation of theta1() can be obtained using the formula theta1() = cov/var. We also report the query time of using *cov/var* in Q1, respectively in Figure 1 (a) and Figure 2 (a), which is at the same order of magnitude as SUDAF execution time. However, even in this case, the benefit of using SUDAF comes from the fact that the performance of SUDAF is independent of the user's programming skill and, as shown in the next example, the partial aggregates computed by SUDAF using sum and count aggregates open wider sharing possibilities than the variance and covariance functions.

Note that SUDAF decomposes a UDAF into two parts, a set of partial aggregates and a terminating function T, then only the partial aggregates of a UDAF are rewritten using built-in functions. This is because a terminating function T is essentially a scalar function applied only on several partial aggregates, and hence it does not impact the computation time of a UDAF. Moreover, there are some UDAFs where it is not possible to write their corresponding terminating functions using built-in functions, e.g., the MomentSolver [16] used to approximate a quantile.

Sharing partial aggregates across UDAFs. Caching the result of Q1, which contains the aggregate values of theta1(), is of little interest from the sharing perspective. However, the partial aggregates $s_1, \ldots, s_5$ computed by the query RQ1 offer more possibilities to be reused in future UDAF computations. We illustrate the sharing idea by the following example. Consider a new query Q2 that computes quadratic mean qm() and standard deviation stddev() of list prices of every item sold by stores in TN for every year:

```
Q2: SELECT  ss_item_sk, d_year, qm(ss_list_price),
            stddev(ss_list_price)
    FROM    store_sales, store, date_dim
    WHERE   ss_sold_date_sk = d_date_sk and
            ss_store_sk = s_store_sk and s_state = 'TN'
    GROUP BY ss_item_sk, d_year;
```

Using SUDAF, qm() (an instance of power mean with $p = 2$ shown in Table 1) and stddev() are defined using the mathematical expressions given in Table 1. When executing Q2, SUDAF factors out their partial aggregations and generates the following query RQ2 which uses the same partial aggregates s_1, s_2 and s_3 as the query RQ1.

```
RQ2: SELECT ss_item_sk, d_year, sqrt(s3/s1) qm_list_price,
            sqrt(s3/s1-power(s2/s1,2)) std_list_price
     FROM   (SELECT ss_item_sk, d_year, count(*) s1,
                    sum(ss_list_price) s2,
                    sum(power(ss_list_price,2)) s3
             FROM   store_sales, store, date_dim
             WHERE  ss_sold_date_sk = d_date_sk and
                    ss_store_sk = s_store_sk and
                    s_state = 'TN'
             GROUP BY ss_item_sk, d_year) TEMP2;
```

SUDAF can cache the partial aggregates in the query RQ1 and identify the opportunity to reuse them for computing aggregates in the query RQ2 automatically. This makes the execution of Q2 in SUDAF significantly faster than executing the query Q2 from base data. We report the query time of Q2 when it is executed by SUDAF on top of PostgreSQL in Figure 1 (b) and on top of Spark SQL in Figure 2 (b). In both figures, the execution time of SUDAF is compared to the execution time of the query Q2 computed respectively over PostgreSQL and Spark SQL. We would like to stress the fact that the result of the UDAF theta1() computed by the query RQ1 cannot be reused to compute the UDAF qm() and stddev() of the query RQ2. However, identifying the appropriate partial aggregates of RQ1 and RQ2 enables to increase the sharing opportunities between these two queries.

Note that we only consider in our example the computation dimension, i.e., computing a UDAF from other UDAFs. Full implementation of our approach requires handling the data dimension, i.e., whether a query is semantically contained in the cached query, which is not addressed in this paper. We point out existing techniques [15, 33] based on data partitioning that can be used in our context to handle the data dimension issue. The main idea of such techniques is to partition the data into predefined chunks and then to map a given query to chunks. Extending SUDAF with such techniques enables us to share partial aggregates over predefined data chunks.

We would like to stress the following three features of the SUDAF sharing mechanism:

- Firstly, it increases performance significantly compared to SUDAF without sharing. In this example, using SUDAF without sharing over PostgreSQL to compute Q2 will take 33.61 s, which is far slower compared to 0.892 s shown in Figure 1 (b). Similarly, in the case of using SUDAF over SparkSQL, SUDAF without sharing will take 2.953 s, which is also significantly slower compared to 0.059 s shown in Figure 2 (b).
- Moreover, the sharing opportunity is dynamically identified in SUDAF by analyzing the expressions of partial aggregates in UDAFs. Note that, using a static approach, one has to predefine computation rules for specific aggregations, e.g., defining $stddev \rightarrow s_1, s_2, s_3$ to share results between RQ1 and RQ2, which is not required in SUDAF.
- Finally, the sharing mechanism of SUDAF covers also the case where partial aggregates are not identical (we present sharing conditions in Section 4.2). For example, SUDAF enables sharing computations between geometric mean and the aggregate $\sum ln(x_i)$, an element of the moment sketch [16]. This is because the partial aggregate $\prod x_i$ of geometric mean (see Table 1) can be computed from $\sum ln(x_i)$, i.e., $\prod x_i = exp(\sum ln(x_i)), \forall x_i > 0$ (see detailed experiments in Section 6).

Extending query rewriting using aggregate views. We show that factoring out partial aggregations of UDAFs can improve traditional query rewriting using aggregate views. Assuming a user is interested in computing qm() and stddev() of the list prices of all items in the category of sports sold by stores in TN for every year since 2000. This is expressed by the following query Q3.

```
Q3: SELECT  d_year, qm(ss_list_price), stddev(ss_list_price)
    FROM    store_sales, store, date_dim, item
    WHERE   ss_sold_date_sk = d_date_sk and ss_item_sk =
            i_item_sk and ss_store_sk = s_store_sk and
            i_category = 'Sports' and s_state = 'TN'
            and d_year >= 2000
    GROUP BY d_year;
```

Now, assume that a materialized view VQ1 corresponding to the query Q1 is given. One can realize that the view VQ1 is useless for rewriting Q3 since it is not possible to compute qm() and stddev() from theta1() and avg().

However, if a materialized view V1 corresponding to the subquery of RQ1 is given and if we factor out partial aggregations of qm() and stddev() in Q3 to generate the following query RQ3:

```
RQ3: SELECT d_year, sqrt(s3/s1) qm_list_price,
            sqrt(s3/s1-pow(s2/s1,2)) std_list_price
     FROM   (SELECT  d_year, count(*) s1,
                     sum(ss_list_price) s2,
                     sum(power(ss_list_price,2)) s3
             FROM    store_sales, store, date_dim, item
             WHERE   ss_sold_date_sk = d_date_sk and
                     ss_item_sk = i_item_sk and
                     ss_store_sk = s_store_sk and
                     i_category = 'Sports'
                     and s_state = 'TN'and d_year >= 2000
             GROUP BY d_year) TEMP3;
```

Then it is possible to use the rewriting algorithm proposed in [13] to rewrite the subquery of RQ3 using V1. The obtained rewriting, denoted by RQ3', is shown below.

Table 1: Examples of aggregations in canonical forms.

Aggregation	Expression	Canonical form $\left(\text{F}, \oplus, \text{T}\right)$
Power mean	$(\frac{\sum(x_i)^p}{n})^{1/p}$	$\left((1, x_i^p), (+,+), (\frac{s_2}{s_1})^{1/p}\right)$
Geometric mean	$(\prod x_i)^{1/n}$	$\left((x_i, 1), (\times, +), (s_1)^{1/s_2}\right)$
Stddev	$\sqrt{\frac{\sum x_i^2}{n} - (\frac{\sum x_i}{n})^2}$	$\left((1, x_i, x_i^2), (+,+,+), \sqrt{\frac{s_3}{s_1} - (\frac{s_2}{s_1})^2}\right)$
Central moment	$\frac{\sum(x_i - avg)^k}{n}$	$\left(((x_i - avg)^k, 1), (+,+), s_1/s_2\right)$
LogSumExp	$ln(\sum exp(x_i))$	$\left((exp(x_i)), (+), ln(s_1)\right)$

Aggregation	Expression	Canonical form $\left(\text{F}, \oplus, \text{T}\right)$
Skewness	$\frac{(\sum(x_i - avg)^3)/n}{((\sum(x_i - avg)^2)/n)^{3/2}}$	$\left(((x_i - avg)^3, (x_i - avg)^2, 1), (+,+,+), \frac{(s_1/s_3)}{(s_2/s_3)^{3/2}}\right)$
Covariance	$\frac{\sum(x_i y_i)}{n} - \frac{\sum x_i \sum y_i}{n^2}$	$\left((x_i, y_i, x_i y_i, 1), (+,+,+,+), \frac{s_3}{s_4} - \frac{s_1 s_2}{s_4}\right)$
Correlation	$\frac{n\sum(x_i y_i) - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}}$	$\left((x_i, x_i^2, y_i, y_i^2, x_i \times y_i, 1), (+,+,+,+,+,+), \frac{s_6 s_5 - s_1 s_3}{\sqrt{s_6 s_2 - (s_1)^2}\sqrt{s_6 s_4 - (s_4)^2}}\right)$

```
RQ3': SELECT d_year, sqrt(s3/s1) qm_list_price,
             sqrt(s3/s1-pow(s2/s1,2)) std_list_price
      FROM   (SELECT  d_year, sum(s1) s1, sum(s2) s2,
                      sum(s3) s3
              FROM    V1, item
              WHERE   ss_item_sk = i_item_sk and
                      d_year >= 2000 and
                      i_category = 'Sports'
              GROUP BY d_year) TEMP3;
```

The key reason that enables such a rewriting comes from the fact that the UDAFs have been rewritten using built-in aggregates: sum() and count() (we recall that the rewriting algorithm proposed in [13] supports only the *sum* and *count* aggregates). We report the execution time of Q3 and RQ3' in PostgreSQL in Figure 1 (c) and Spark SQL in Figure 2 (c).

To conclude this section, we would like to emphasis the fact that the main features of SUDAF, factoring out the partial aggregations of UDAFs, computing partial aggregations using built-in functions and sharing partial aggregates, provide abundant opportunities to speed up queries with UDAFs. In the rest of this paper, we address the following challenges:

- *how to identify appropriate partial aggregations of UDAFs to maximize sharing opportunities?*
- *how to efficiently determine when cached results of partial aggregations of UDAFs can be reused to compute other UDAFs?* (hereafter, called the sharing problem)

3 IDENTIFYING AND SHARING PARTIAL AGGREGATES

We aim at speeding up queries with UDAFs by reusing cached answers to previous queries with UDAFs during the evaluation of new ones. We deal with the following two issues in this section.

What computation results should be cached to optimize the evaluation of UDAFs? We identify a canonical form of UDAFs [10], which captures the computation pipelines of UDAFs. We analyze the caching possibilities based on the computation pipelines and identify the appropriate level of aggregation to be kept in caches.

How can we identify if a cached answer can be reused in the evaluation of a given UDAF? We formalize the problem of identifying a reusable answer as the sharing problem. Then we show that it is an undecidable problem for arbitrary cases. In Section 4, we present a restricted, yet powerful enough, framework to handle the sharing problem for practical cases.

3.1 Canonical forms of UDAFs

An aggregate function takes as inputs several values and produces as output a *single representative* value [17]. In our work, we consider aggregations operating on multisets. Let D_s and D_t be two domains i.e., countably infinite sets of values, and let $\mathcal{M}(D_s)$ denote the set of all nonempty multisets of elements from D_s. An aggregate function α is a function: $\mathcal{M}(D_s) \rightarrow D_t$.

We use the notion of well-formed aggregation to define a canonical form of aggregate functions. Well-formed aggregation was introduced in [10] to capture the manner in which a UDAF is created. An aggregation $\alpha : \mathcal{M}(D_s) \rightarrow D_t$ is a *well-formed aggregation* if α can be expressed as a triple $(F, \oplus, T)$, where F is a translating function, $\oplus$ is a commutative and associative binary operation and T is a terminating function, such that $\forall X = \{\!\{x_1, ..., x_n\}\!\} \in \mathcal{M}(D_s), \alpha(X) = T(F(x_1) \oplus \ldots \oplus F(x_n))$, or briefly $\alpha(X) = T(\sum_\oplus F(x_i))$.

In this paper, we consider the well-formed aggregation as the canonical form of UDAFs. We list some examples of aggregations with their canonical forms in Table 1 (an input of a terminating function T is denoted as s_i). It is interesting to note that practical aggregations usually have addition and multiplication as an element of $\oplus$ function in their canonical forms, e.g., the $\oplus$ function of geometric mean is $(\times, +)$.

Given an aggregation, $\alpha = (F, \oplus, T)$, the associative and commutative property of $\oplus$ ensures that $\alpha(X)$ can be computed by first applying F and $\oplus$ on arbitrary subsets of X and then the intermediate results can be merged using $\oplus$ and T to produce the final result $\alpha(X)$. Hence, we call the intermediate results $\sum_\oplus F(x_i)$ the *partial aggregations* of α.

3.2 Caching aggregate data

To obtain more sharing possibilities, we identify which results of an aggregation are worth caching based on its canonical form. Consider two aggregations $\alpha = (F_\alpha, \oplus_\alpha, T_\alpha)$ and $\beta = (F_\beta, \oplus_\beta, T_\beta)$. Suppose a scenario where an implementation of α based on its canonical form is executed first. When the UDAF β is evaluated, there are three possibilities to reuse partial or whole computation results of α: (1) the result of F_α, (2) the result of $\sum_{\oplus_\alpha} F_\alpha$, or (3) the final result of α. It is clear that caching the 1st result does not provide any added value to the computation of β since F_α is a scalar function. Storing the 3rd result is of little interest as it offers very restricted possibilities [2] to be reused in the computation of other UDAFs, e.g., β. However, the partial aggregation $\sum_{\oplus_\alpha} F_\alpha$ offers much more potentials to reuse than the others. For example, if α is a *stddev* and β is a *power mean* ($p = 2$) shown in Table 1, it is not possible to reuse the final result of α to compute β. However, using their canonical forms, one can observe that the fragments, s_1 and s_3, in the partial aggregation of α can be used to compute β. Therefore, we choose to cache the partial aggregation $\sum_{\oplus_\alpha} F_\alpha(x_i)$.

[2]Theoretically, T_α should not be expected to have an inverse function [10], such that we cannot always have the 2nd result if we cache the 3rd one. However, we can indeed have the 3rd result if we cache the 2nd result.

Table 2: **Classes of primitive functions provided in SUDAF.**

Class	**Functions**
PS	a; x; ax; x^a; $log_a x$; a^x.
PB	$+$; $-$; $\times$; $/$; ^.
PA	$\sum$; $\prod$.
PS°	$g(x) = h_l \circ ... \circ h_1(x)$, with $h_j \in PS$, for $j \in [1, ..., l]$.
$PS^\odot$	$f(x) = g_k(x) \odot_{k-1} ... \odot_1 g_1(x)$, with $g_j \in PS^\circ, \odot_z \in PB$, for $j \in (1, ..., k), z \in (1, ..., k-1), k \in \mathbb{N}_{>0}$.
PA°	$agg(X) = f' \circ \sum_\oplus \circ f(x_i)$, with $f, f' \in PS^\odot, \sum_\oplus \in PA$.
$PA^\odot$	$bagg(X) = T'(agg_k(X) \odot_{k-1} ... \odot_1 agg_1(X))$, with $agg_j \in PA^\circ$,$\odot_z \in PB$ for $j \in (1, ..., k)$,$z \in (1, ..., k-1), k \in \mathbb{N}_{>1}$ and $T' \in PS^\odot$.

Table 3: Cases analysis of the sharing problem in SUDAF.

Case	f_1 **in** s_1	f_2 **in** s_2	**Whether** $s_1 \in D(s_2)$
1	Injective	Non-injective	N (case 1 of Theorem 4.1)
2	-	Injective	Case 2 of Theorem 4.1
3	Even	Even	Case 2 of Theorem 4.1
4	Neither injective nor even	Neither injective nor even	Splitting rules (SR)

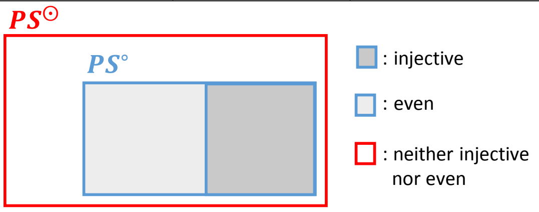


Figure 3: Injective and even functions in PS° and $PS^\odot$ (excluding constant functions).

3.3 Sharing aggregation states

Let $\alpha = (F, \oplus, T)$ be an aggregation and $\sum_\oplus F(x_i)$ be the partial aggregation of α. We decompose the partial aggregation as follows, $\sum_\oplus F(x_i) = \left(\sum_{\oplus_1} f_1(x_i), ..., \sum_{\oplus_m} f_m(x_i) \right)$, where the f_is are scalar functions and the $\oplus_i$s are commutative and associative binary operations, e.g., the partial aggregation of geometric mean is $(\prod x_i, count)$. In the sequel, we call an individual element $s_j(X) = \sum_{\oplus_j} f_j(x_i)$ as an *aggregation state*, e.g., both $\prod x_i$ and *count* are aggregation states of geometric mean.

We rely on aggregation states to define when a partial result of a UDAF α can be reused in the computation of another UDAF β. More precisely, we define below when an aggregation state s of α can be *shared* by an aggregation state s' of β.

Definition 3.1. Let $s'(X)$ and $s(X)$ be two aggregation states of two UDAFs. Then, s' shares s iff there exists a computable function r such that $s'(X) = r \circ s(X), \forall X \in \mathcal{M}(D)$.

The function r is a scalar function that enables computing the aggregation state s' without scanning the base dataset X, e.g., r is the identity function if $s'(X) = s(X)$. If an aggregation state s is cached, the sharing problem is then to decide whether s can be reused in the computation of another aggregation state s'.

We denote the problem whether s' shares s as share(s', s). As stated by the following theorem, it is not possible to solve share(s', s) in a general setting. The proof for Theorem 3.2 is included in our online technical report [31].

Theorem 3.2. *The problem share(s', s) is undecidable.*

4 THE SUDAF PRACTICAL FRAMEWORK

In this section, we present a declarative UDAF framework SUDAF, which rests on the canonical form of UDAFs to generate and share partial aggregation states of UDAFs automatically. The following main objective guided the design of SUDAF.

How to deal with the undecidability of the sharing problem? We adopt a pragmatic approach to solve this problem by restricting the class of UDAFs that can be used in SUDAF. The proposed practical framework is powerful enough to be useful in many real-world applications while making the sharing problem decidable.

We argue that it is not realistic to ask a user to provide UDAFs in their canonical forms. Therefore, SUDAF enables users to formulate UDAFs as mathematical expressions and then generates a corresponding canonical form. Consequently, in a generated canonical form, SUDAF knows the semantics of partial aggregations, i.e., computation details, which can be exploited to analyze sharing possibilities during computing UDAFs.

4.1 Declarative UDAF framework

SUDAF provides a set of predefined functions that can be used by users to write UDAFs. Three classes of primitive functions are proposed (cf. Table 2):

- *Primitive scalar functions.* This class, denoted PS (primitive scalar), contains six types of functions: constant, identity, linear, power, logarithmic and exponential functions. The elements of PS are presented in line 1 of Table 2, where a is an arbitrary constant defined by users.
- *Primitive binary functions.* This class, denoted PB (primitive binary), contains the following binary functions: addition $+$, subtraction $-$, multiplication $\times$, division $/$ and exponentiation ^.
- *Primitive aggregate functions.* This class, denoted PA (primitive aggregate functions), contains two functions: summation $\sum$ and product $\prod$.

As explained below, primitive functions can be combined using the composition operator and binary functions to create more complex scalar and aggregate functions.

Complex scalar functions. SUDAF provides a *composition operator*, denoted $\circ$, that enables creating complex scalar functions from the primitive ones. The class of such functions is denoted PS°. A function $g(x) \in PS^\circ$ can be expressed as a composition of primitive scalar functions (cf. Table 2). The length of $g(x)$, denoted $|g|$, gives the number of primitive functions used in the definition of $g(x)$. For example, if $g(x) = h_l \circ ... \circ h_1(x)$, with $h_j \in PS$, then $|g| = l$. Besides, more complex scalar functions can be expressed by using binary functions to combine scalar functions from PS°. The set of such functions, i.e., scalar functions containing binary operations, is denoted $PS^\odot$. The shape of functions in $PS^\odot$ is shown in Table 2.

Supported aggregations. SUDAF also allows using the composition operator $\circ$ between scalar functions and primitive aggregate functions to define new aggregations. More precisely, in this context, the composition can be used in two ways: *(i)* to apply a scalar function on an output of a primitive aggregate function, or *(ii)* to apply a primitive aggregation on a set of data transformed using a scalar function. The class of such functions is denoted as PA°. The expression of aggregation $agg \in PA^{\circ}$ is presented in Table 2. Moreover, more complex aggregations can be expressed using primitive binary functions to combine several aggregations in PA°. The class of such functions is denoted as $PA^{\odot}$, and a UDAF $bagg \in PA^{\odot}$ has the expression shown in Table 2.

Scope of UDAFs in SUDAF. SUDAF restricts the set of UDAFs that can be declared to the classes presented in Table 2. We shall show in the next section that this restriction enables us to cope with the undecidability of sharing problems. However, this restriction does not hamper the usability of SUDAF in real world applications since the proposed framework covers a wide range of aggregations such as the classes of power mean, arbitrary central moments [7], arbitrary standardized moments [32] and other multi-variate aggregations [3] such as covariance, correlation, and cofactor aggregates [30] used in training linear regression. Generally, algebraic aggregations can be defined in SUDAF. Although holistic aggregations, e.g., median, cannot be expressed in SUDAF, aggregates used in their approximation algorithms are supported by SUDAF, e.g., moment sketch [16],

Mapping SUDAF functions into canonical forms. SUDAF supports two scenarios to define UDAFs. We explain below how to derive canonical forms and aggregation states from UDAFs defined in each scenario.

The first scenario is that a terminating function is described using an element from $PS^{\odot}$. Such functions are expressed using a function $T' \in PS^{\odot}$ applied on compositions, using binary operations in PB, of aggregations from PA° and have the following general form:

$$\alpha(X) = T'\big(\big(f'_k \circ \sum_{\oplus_k} \circ f_k(x_i)\big) \odot_{k-1} \dots \odot_1 \big(f'_1 \circ \sum_{\oplus_1} \circ f_1(x_i)\big)\big).$$

The f_j, f'_j, for $j \in [1, ..., k]$, are scalar functions from $PS^{\odot}$ and $\sum_{\oplus_j}$ are primitive aggregations from PA. Given such a function $\alpha(X) \in PA^{\odot}$, a canonical form canonical$(\alpha) = (F, \oplus, T)$ is derived from the general expression of α as follows:

- $F = (f_1, \dots, f_k)$;
- $\oplus = (\oplus_1, \dots, \oplus_k)$ and
- $T = T'\big((f'_1 \circ \sum_{\oplus_1} \circ f_1) \odot_1 \dots \odot_{k-1} (f'_k \circ \sum_{\oplus_k} \circ f_k)\big)$.

The aggregation states of α are shown as follows: $s_j(X) = \sum_{\oplus_j} f_j(x_i)$, for $j \in [1, \dots, k]$. For instance, aggregations in Table 1 can be defined in SUDAF using their expressions in the second column. SUDAF generates their canonical forms and aggregation states from their expressions (the s_i elements in Table 1).

The second scenario is that a terminating function is created by hardcoding. Such functions have the following shapes, $\alpha(X) = T(s_1, ..., s_k)$, where $s_j, j \in (1, ..., k)$ is an aggregation state. For example, if one wants to use the MomentSolver [16] taking the MomentSketch as inputs to approximate a quantile, the MomentSketch can be defined as a set of aggregation states from PS° and the MomentSolver as a terminating function.

4.2 Dealing with the sharing problem in SUDAF

In this section, we present sharing conditions to deal with the sharing problem in SUDAF. Let $s_1(X) = \sum_{\oplus_1} f_1(x_i)$ and $s_2(X) = \sum_{\oplus_2} f_2(x_i)$ be two aggregation states of two UDAFs in the scope of SUDAF. Then both f_1 and f_2 belong to $PS^{\odot}$. We carry out a case analysis to identify the conditions that characterize situations where s_1 shares s_2. Our case analysis is based on the properties of the scalar functions f_1 and f_2 used by the aggregation states s_1 and s_2. In fact, all scalar functions in PS°, except constant functions, are either injective, or even (i.e., $f(x) = f(-x)$)), while scalar functions in $(PS^{\odot} \setminus PS^{\circ})$ are not injective because of the presence of the arithmetic binary functions $\odot$ (cf. Figure 3). Therefore, we split the *sharing problem* share(s_1, s_2) into four main cases depending on whether f_1 and f_2 are injections or even functions. The studied cases are presented in Table 3. Our main results provide a full characterization for the first three cases in Table 3. Specifically, we provide complete conditions in Theorem 4.1 for the first two cases in Table 3, and then we reduce the third case to the second case in Table 3. We also propose an incomplete solution to deal with the fourth case in Table 3.

THEOREM 4.1. *Let $X \in \mathcal{M}(\mathbb{Q})$ and let $s_1(X) = \sum_{\oplus_1} f_1(x_i)$ and $s_2(X) = \sum_{\oplus_2} f_2(x_i)$ be two aggregation states with $\sum_{\oplus_1} \in PA$ and $\sum_{\oplus_2} \in PA$, f_1 a non constant function and $s_1 \neq s_2$. Then, we have:*

(Case 1) *if f_1 is injective and f_2 is not injective, then s_1 does not share s_2.*

(Case 2) *if f_2 is injective, then: there exists a computable function r_{12} such that $s_1(X) = r_{12} \circ s_2(X)$ iff one of the following conditions holds:*

(2.1) *$\sum_{\oplus_1} = \sum_{\oplus_2} = \sum$ and $f_1 \circ f_2^{-1}(x) = ax$ with $a \in \mathbb{Q}_{\neq 0}$ a constant. Then we have $r_{12}(x) = f_1 \circ f_2^{-1}(x)$.*

(2.2) *$\sum_{\oplus_1} = \sum$, $\sum_{\oplus_2} = \prod$ and $f_1 \circ f_2^{-1}(x) = a(log_b|x|)$ with $b \in \mathbb{Q}_{>0,\neq 1}$ and $a \in \mathbb{Q}_{\neq 0}$ two constants. Then we have $r_{12}(x) = f_1 \circ f_2^{-1}(x)$.*

(2.3) *$\sum_{\oplus_1} = \prod$, $\sum_{\oplus_2} = \sum$ and $f_1 \circ f_2^{-1}(x) = b^{ax}$ with $b \in \mathbb{Q}_{>0,\neq 1}$ and $a \in \mathbb{Q}_{\neq 0}$ two constants. Then we have $r_{12}(x) = f_1 \circ f_2^{-1}(x)$.*

(2.4) *$\sum_{\oplus_1} = \sum_{\oplus_2} = \prod$ and with a constant $a \in \mathbb{Q}_{\neq 0}$:*
  (i) *when $f_1 \circ f_2^{-1}(-1) = 1$, $f_1 \circ f_2^{-1}(x) = |x|^a$;*
  (ii) *when $f_1 \circ f_2^{-1}(1) = -1$, $f_1 \circ f_2^{-1}(x) = sgn(x) \times |x|^a$;*
  Then we have $r(x) = f_1 \circ f_2^{-1}(x)$.

The proof for Theorem 4.1 is included in a technical report [31].

The case 1 of Theorem 4.1 states that, given two aggregation states $s_1(X) = \sum_{\oplus_1} f_1(x_i)$ and $s_2(X) = \sum_{\oplus_2} f_2(x_i)$ in the scope of SUDAF, when f_1 is injective and f_2 is non-injective, then except the special case of an identity function when $s_1 = s_2$, it is not possible to find a computable function r_{12} such that $s_1(X) = r_{12} \circ s_2(X)$. The case 2 of Theorem 4.1 provides necessary and sufficient conditions to characterize solutions for the problem share(s_1, s_2) when f_2 is injective. It carries out a case analysis for the four possible combinations obtained from the instantiation of $\sum_{\oplus_1}$ and $\sum_{\oplus_2}$ as operations in PA, i.e., either sum or product.

Example 4.2. We explain how Theorem 4.1 can be used as follows. Consider the problem whether $s_1(X) = \sum 4x_i$ shares $s_2(X) = \prod 2^{x_i}$. Since $\sum_{\oplus_1} = \sum$ and $\sum_{\oplus_2} = \prod$, then the case 2.2 of Theorem 4.1 is selected. Then, we have $f_1 \circ f_2^{-1}(x) = 4log_2(x)$, which satisfies the shape $a(log_b(x))$ with constants $a = 4$ and $b = 2$. Thus, we have $s_1(X) = r \circ s_2(X)$ with $r(x) = 4log_2(x)$.

The case of even scalar functions. The third case to deal with is when both $f_1(x)$ and $f_2(x)$ are not injections but even functions (case 3 of Table 3). As depicted in Figure 3, non-injective scalar functions of PS° are *even* functions. We exploit this property to reduce the study to a sharing problem over a positive domain of scalar functions and show that the case 2 of Theorem 4.1 can be applied in this setting. We denote $U_X = \{u_x = |x| | x \in X\}$.

[3]Multi-variate aggregations can be seen as a combination of several uni-variate aggregations, each of which is expressed using functions in Table 2. Moreover, the cofactor aggregate $\sum x_i y_i$ computed over columns X and Y can be seen as a uni-variate aggregate over an abstract column $Z = X \cdot Y$ with the scalar product $\cdot$.

Then, whatever x is, we have $u_x \geqslant 0$. Let $s_1(X) = \sum_{\oplus_1} f_1(x_i)$ and $s_2(X) = \sum_{\oplus_2} f_2(x_i)$ be two aggregation states in SUDAF such that $\{f_1, f_2\} \subset PS^{\circ}$. Observe that $s_1(X)$ shares $s_2(X)$ iff $s_1(U_X)$ shares $s_2(U_X)$. This is because $f_1(x) = f_1(u_x)$ (since f_1 is even), and similarly for f_2. Consequently, one can focus on solving the sharing problem only over positive domains of f_1 and f_2. In this setting (positive domain), all primitive scalar functions of SUDAF (non-constant elements in PS) are injections and hence the complex scalar functions, elements of PS°, are also injective functions. Therefore, the case 2 of Theorem 4.1 can be exploited to solve the sharing problem in this context.

The case of neither even nor injective scalar functions. The last case to deal with is when both $f_1(x)$ and $f_2(x)$ are neither injections nor even functions (case 4 of Table 3). As depicted in Figure 3, such scalar functions are from $(PS^{\odot} \setminus PS^{\circ})$. We propose splitting rules to deal with such cases. W.l.o.g, let $s(X) = \sum_{\oplus}(g_1(x_i) \odot g_2(x_i))$, $\sum_{\oplus} \in PA$, $\{g_1, g_2\} \in PS^{\circ}$. Then, we define the following two splitting rules (SR):

SR1: $\sum(g_1(x_i) \odot g_2(x_i)) = \sum(g_1(x_i)) \odot \sum(g_2(x_i)), \odot \in \{+, -\}$;
SR2: $\prod(g_1(x_i) \odot g_2(x_i)) = \prod(g_1(x_i)) \odot \prod(g_2(x_i)), \odot \in \{\times, /\}$.

By applying the above two rules, aggregation states in $(PS^{\odot} \backslash PS^{\circ})$ can be split into new ones with scalar functions in PS°, which can still be verified using Theorem 4.1. If aggregation stares are not covered by splitting rules in this case, SUDAF simply proceeds syntactic comparison between their mathematical expressions. Note that syntactic comparison is sufficient but not necessary.

5 A PRACTICAL APPROACH TO SOLVE THE SHARING PROBLEM

We present in this section a practical approach to solve the sharing problem based on the results provided by Theorem 4.1. Turning the conditions of Theorem 4.1 into an algorithm could be cumbersome because equivalent mathematical expressions may have different syntactic shapes.

Example 5.1. Consider the problem whether $s_1(X) = \sum 4x_i^2$ shares $s_2(X) = \sum(3x_i)^2$. Using Theorem 4.1, one needs to construct $f_1 \circ f_2^{-1}(x) = 4x \circ x^2 \circ \frac{1}{3}x \circ \sqrt{x}$ (over the positive domain since both f_1 and f_2 are even). Then, according to case 2.1 of Theorem 4.1, we need to check whether $f_1 \circ f_2^{-1}(x) = ax$, for some constant a. This is not an easy task, particularly for general cases, since it requires mathematical transformations of the original expression as follows: $f_1 \circ f_2^{-1}(x) = 4x \circ x^2 \circ \frac{1}{3}x \circ \sqrt{x} = 4x \circ \frac{1}{9}x \circ x^2 \circ \sqrt{x} = \frac{4}{9}x$. The first transformation is a *reordering* of $x^2 \circ \frac{1}{3}x$, which generates $\frac{1}{9}x \circ x^2$, and it is then followed by a *removal* of the composition $x^2 \circ \sqrt{x}$. Finally, $f_1 \circ f_2^{-1}(x)$ is transformed to $\frac{4}{9}x$, which satisfies the condition $f_1 \circ f_2^{-1}(x) = ax$, with $a = \frac{4}{9}$, of the case 2.1 of Theorem 4.1.

In addition, a straightforward implementation of Theorem 4.1 leads to redundant computations as illustrated below.

Example 5.2. Checking whether $s_1' = \sum 6x_i^3$ shares $s_2' = \sum(5x_i)^3$ requires redoing identical transformations as in the previous example (i.e., checking whether $s_1(X) = \sum 4x_i^2$ shares $s_2(X) = \sum(3x_i)^2$). This is because we have as a general property: $\sum a_2 x_i^{a_1}$ shares $\sum(b_1 x_i)^{b_2}$ if $a_1 = b_2$.

Hence, our general idea to deal with the two previous issues is: (i) to use symbolic representations of aggregation states to avoid redundant computations, i.e., using $\sum a_2 x_i^{a_1}$ and $\sum(b_1 x_i)^{b_2}$, where a_1, a_2, b_1 and b_2 are parameters, to represent the *concrete* states $\sum 4x_i^2$ and $\sum(3x_i)^2$, and (ii) to precompute sharing relationships between symbolic representations to avoid cumbersome transformations of mathematical expressions at execution time. For example, we precompute the relationship stating that $\sum a_2 x_i^{a_1}$ shares $\sum(b_1 x_i)^{b_2}$ if $a_1 = b_2$. Then, at execution time, this relationship can be used to efficiently identify that the *concrete* aggregation state $\sum 4x_i^2$, an instance of the abstract state $\sum a_2 x_i^{a_1}$, shares the concrete state $\sum(3x_i)^2$, an instance of the abstract state $\sum(b_1 x_i)^{b_2}$, because the condition $a_1 = b_2$ is satisfied.

5.1 Symbolic representations

In this section, we first present symbolic representations of scalar functions and then use them to introduce symbolic representations of aggregation states. In the sequel, we assume an infinite set of parameters, distinct from the set of constants. Hereafter, the parameters are denoted $p, p_1, \ldots$.

Symbolic primitive scalar functions. Intuitively, px with a parameter p is the symbolic representation of the primitive scalar function $2x$. In this case, $2x$ is an instance of px. Formally, we consider four symbolic primitive scalar functions with a parameter p: $px = \{ax | \forall a \neq 0\}$; $log_p x = \{log_a x | \forall a > 0, \neq 1\}$; $p^x = \{a^x | \forall a > 0, \neq 1\}$; $x^p = \{x^a | \forall a \neq 0\}$. We use the notation $sf_{\bar{p}}(x)$ for a symbolic primitive scalar function with a sequence $\bar{p} = (p)$ of a parameter p.

Symbolic scalar functions. Intuitively, $p_2 x^{p_1}$ with a parameter sequence (p_2, p_1) is the symbolic representation of the scalar function $3x^2$, and in this case $3x^2$ is an instance of $p_2 x^{p_1}$. Formally, let every $sf_{i\bar{p}_i}(x)$ for $i \in [1, \ldots, l]$ be a symbolic primitive scalar function. Then, $sf_{\bar{p}}(x) = sf_{l\bar{p}_l} \circ \ldots \circ sf_{1\bar{p}_1}(x)$ is a symbolic scalar function $sf_{\bar{p}}(x)$ with a sequence $\bar{p} = (p_l, \ldots, p_1)$ of parameters. Similarly, $|sf_{\bar{p}}| = l$.

Symbolic aggregation states. Intuitively, $\sum p_2 x_i^{p_1}$ is the symbolic representation of $\sum 3x_i^2$. In this case, $\sum p_2 x_i^{p_1}$ is called a symbolic (aggregation) state and we say that the concrete state $\sum 3x_i^2$ is an instance of the symbolic state $\sum p_2 x_i^{p_1}$. Formally, let $\sum_{\oplus} \in PA$ and $sf_{\bar{p}}(x)$ be a symbolic scalar function. Then, $ss(X) = \sum_{\oplus} sf_{\bar{p}}(x_i)$ is a symbolic aggregation state.

Specifically, we let $\sum x_i$ and $\prod x_i$ be also two symbolic aggregation states, which contain respectively only one instance $\sum x_i$ and $\prod x_i$, and we define $|f| = 0$ with $f(x) = x$.

5.2 Precomputed sharing relationships

Informally, we say that a symbolic state ss_1 shares a symbolic state ss_2 if and only if for any instance s_1 of ss_1, there exists an instance s_2 of ss_2, such that s_1 shares s_2. As explained previously, our aim is to precompute and store the sharing relationships between symbolic aggregation states. Specifically, we conduct an exhaustive analysis to identify the sharing relationships between symbolic states in a preprocessing step, which is performed once when SUDAF is deployed, and then the precomputed relationships are reused at runtime to handle the sharing problem between concrete aggregation states. Note that the space of symbolic states may be very huge (theoretically infinite) because symbolic scalar functions may be of arbitrary lengths. In addition, aggregation states having scalar functions with a higher length are useless from the practical point of view. For example in our experiments presented in Section 6 it was enough to use aggregation states, whose scalar functions have a length up to 2, to express aggregations in real-world applications. Therefore, SUDAF enables a user to bound the space of symbolic aggregation

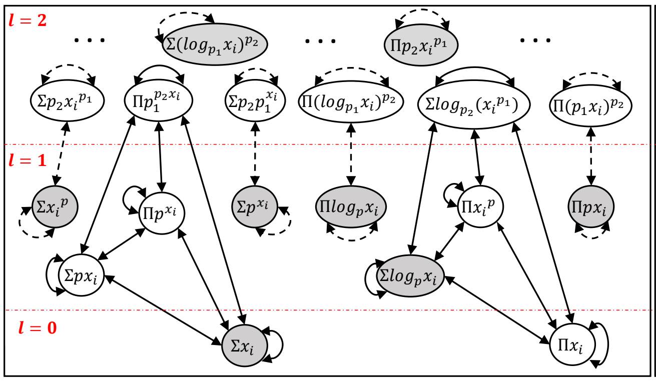


Figure 4: The digraph G of $saggs_2(X)$.

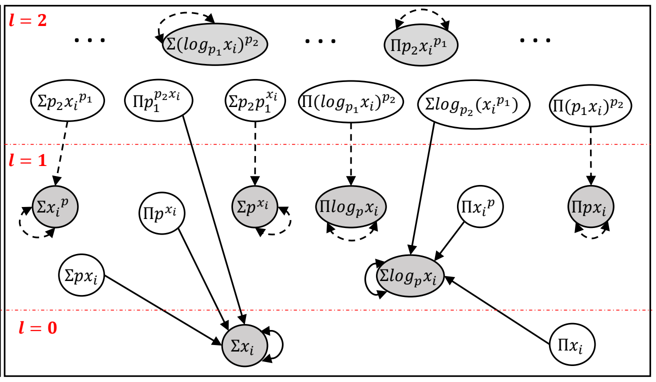


Figure 5: The simplified digraph G of $saggs_2(X)$.

states that is prebuilt in the preprocessing step using a configuration parameter, denoted by l. The obtained space, denoted by $saggs_l(X)$, is introduced below.

l-bounded symbolic space. Let $l \geqslant 0$ be an integer. We define the space $saggs_l(X)$ of symbolic aggregation states as follows: $saggs_l(X) = \{\sum_{\oplus} sf_{\bar{p}}(x_i) | sf_{\bar{p}}$ is a symbolic scalar function with $|sf_{\bar{p}}| \leqslant l\}$. We say $saggs_l(X)$ is a l-bounded symbolic space. Note that the size of the set $saggs_l(X)$ is bounded by $\frac{2(4^{l+1}-1)}{3}$.

Once the parameter l is fixed by a user, SUDAF builds space $saggs_l(X)$ and precomputes the sharing relationships between every two symbolic aggregation states in $saggs_l(X)$. An excerpt of $saggs_2(X)$ is shown in Figure 4, where each symbolic aggregation state is depicted as a node labeled with its expression (the meaning of edges in Figure 4 is explained later). As it can be observed in Figure 4, the space $saggs_2(X)$ is organized in three levels, where each level i, with $i \in \{0, 1, 2\}$, contains the symbolic states of the form $\sum_{\oplus} sf_{\bar{p}}(x_i)$ with $|sf_{\bar{p}}| = i$. Figure 4 shows all the symbolic states of level 0 and 1, and some states of level 2.

5.3 Organizing the space $saggs_l(X)$

We briefly discuss the organization of $saggs_l(X)$, w.l.o.g., focusing on the case $l = 2$. In the sequel, we first consider that the input multiset X contains only positive values, i.e., $X \in \mathcal{M}(\mathbb{Q}_+)$, then we extend the results to the case where X contains both negative and positive values. We represent the sharing relationships between symbolic states in $saggs_2(X)$ using a digraph $G = (V, E)$ where the set of vertices $V = saggs_2(X)$ is the space $saggs_2(X)$ and the set of edges $E \subseteq V \times V$ represent the sharing relationship, i.e., $(ss', ss) \in E$ if and only if ss' shares ss. Figure 4 depicts the digraph associated with the space $saggs_2(X)$. We distinguish between two kinds of sharing relationships in G (two types of edges are depicted in Figure 4). The first one is called *strong relationships* and relates two symbolic states (ss', ss) if ss' shares ss without requiring any condition on the parameters. The second one is called *weak relationships* and relates two symbolic states (ss', ss) if ss' shares ss under some conditions defined over the parameters of ss and ss'. For example, since any instance of $\sum px_i$ shares any instance of $\prod p^{x_i}$, then $\sum px_i$ and $\prod p^{x_i}$ have a strong sharing relationship denoted as $\sum px_i \rightarrow \prod p^{x_i}$. As another example, the state $\sum x_i^p$ shares $\sum p_2 x^{p_1}$ with the condition $p = p_1$, then $\sum x_i^p$ and $\sum p_2 x^{p_1}$ have a weak sharing relationship denoted as $\sum x_i^p \xrightarrow{p=p_1} \sum p_2 x^{p_1}$.

We observed that in the space $saggs_2(X)$, the sharing relationships are *equivalence relations*. For example, $\sum px_i \leftrightarrow \prod p^{x_i}$ and $\sum x_i^p \overset{p=p_1}{\longleftrightarrow} \sum p_2 x^{p_1}$. Consequently, the space $saggs_2(X)$ can be partitioned into *equivalence classes*. Intuitively, for a symbolic state ss, its associated equivalence class, denoted $[ss]$, is made of the set of symbolic aggregation states that shares (and are shared by) ss. For example, as depicted in Figure 4: $[\sum x_i] = \{\sum x_i, \sum px_i, \prod p^{x_i}, \prod p_1^{p_2 x_i}\}$ and $[\sum x_i^p] = \{\sum x_i^p, \sum p_2 x_i^{p_1}\}$.

We select a unique element in each equivalence class $[ss]$ to be a *representative* of the class, which is denoted as $rep([ss])$ and depicted as a shaded node in Figure 4. It is clear that, given an equivalence class $[ss]$, one only needs to focus on the instances of its representative $rep([ss])$ since they are able to compute an instance of any other element in $[ss]$.

We simplify G presented in Figure 5 based on the equivalence relations derived from the sharing relationships. More precisely, it is only necessary for any state $ss \in saggs_2(X)$ to store such a sharing relationship $ss \rightarrow rep([ss])$, or $ss \xrightarrow{pcon} rep([ss])$ with a parameter condition ($pcon$). Consequently, when an instance s of ss is given, we use an edge $ss \rightarrow rep([ss])$, or $ss \xrightarrow{pcon} rep([ss])$ to get a cached instance of $rep([ss])$ to compute s.

Extension to an arbitrary multiset. When a multiset X contains negative values, instances of some symbolic states in $saggs_2(X)$ do not exist, which will cause the miss of sharing opportunities. We take $\sum log_p x_i$ as an example to explain the issue. As we know that, an instance $\sum ln(x_i)$ of $\sum log_p x_i$ can only be computed over the positive domain, such that the caches for $\sum log_p x_i$ are empty in this context. To deal with this issue, we separate input values from their signs. Specifically, we translate an input multiset $X = \{x_1, \ldots, x_n\}$ to the following multiset $\hat{X} = \{(|x_1|, sgn(x_1)), \ldots, (|x_n|, sgn(x_n))\}$, where $|x_j|$ denotes the absolute value of x_j and $sgn(x_j)$ is its sign. Then, we keep in the cache such a result $(\sum ln|x_i|, \prod sgn(x_i))$ for $\sum log_p x_i$. By this way, a new aggregation state $\sum ln(x_i^2)$ can still be computed using the cache $(\sum ln|x_i|, \prod sgn(x_i))$ that is stored for $\sum log_p x_i$.

6 EXPERIMENTAL EVALUATION

We implemented a SUDAF prototype in Java and Scala, which can be used on top of PostgreSQL (through JDBC) and Spark SQL. The SUDAF prototype also comes equipped with a UDAF editor that enables users to write SUDAF-compatible UDAFs and integrate them in SQL queries.

The general scheme of our experiments is the following. We select 3 query models, and we instantiate each query model using 11 aggregations. We simulate the 11 instances of each query model coming in 2 different orders, i.e., two different sequences of

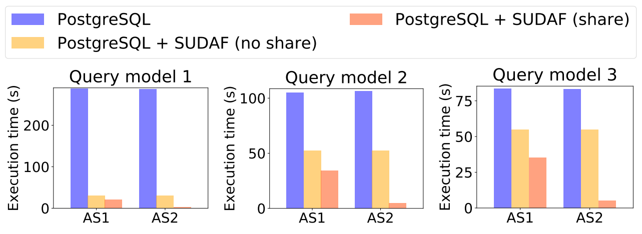


Figure 6: Total execution time of each query sequence in each query model.

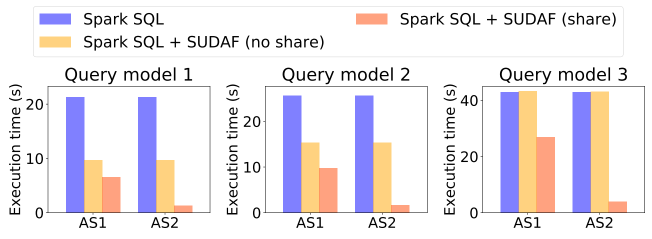


Figure 7: Total execution time of each query sequence in each query model.

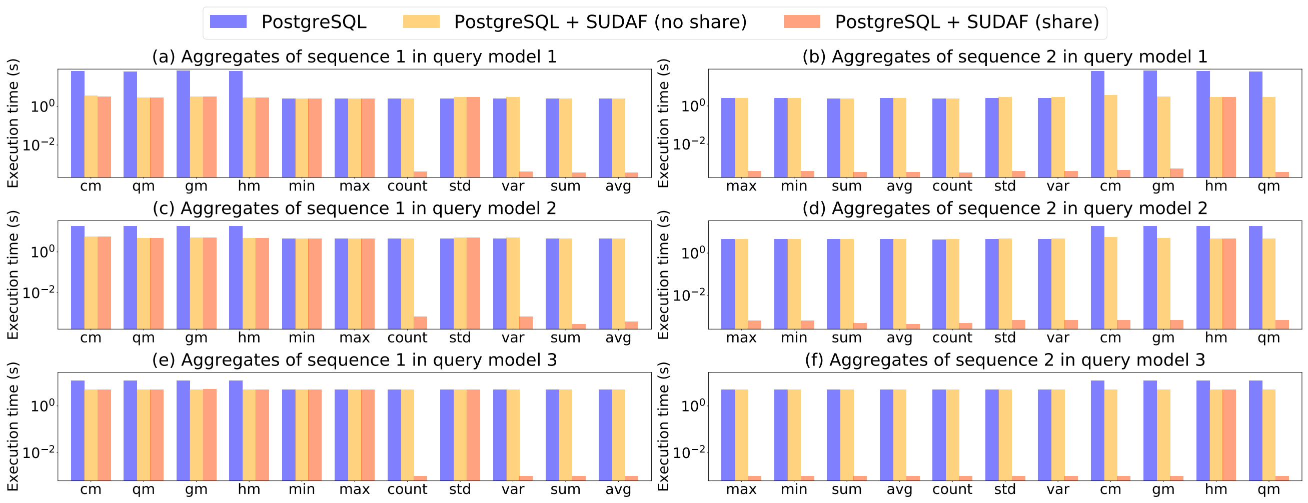


Figure 8: Execution time in PostgreSQL of each query in each query sequence.

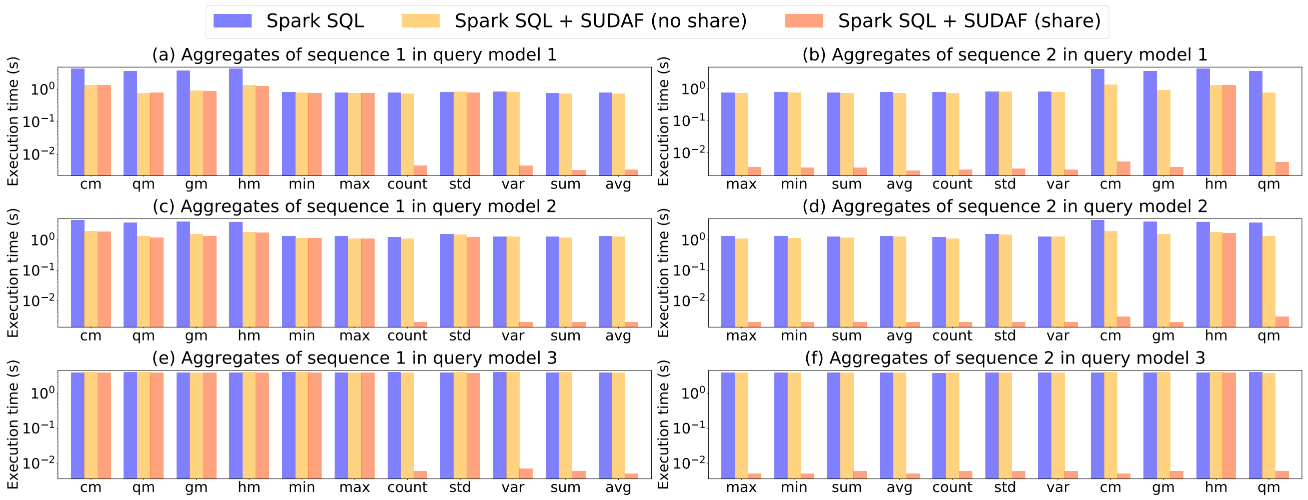


Figure 9: Execution time in Spark SQL of each query in each query sequence.

queries. Thus, the tested workload consists of 6 query sequences, where each sequence has 11 queries. We execute the query sequences in three technical contexts (i) PostgreSQL and Spark SQL, (ii) SUDAF without the sharing functionality, and (iii) SUDAF with the sharing functionality. In the PostgreSQL environment (case (i)), the aggregations are either PostgreSQL built-in or hard-coded user-defined functions, and similarly for the Spark SQL environment. PostgreSQL UDAFs are created using PL/pgSQL, and Spark SQL UDAFs are created using the UserDefinedAggregateFunction interface in Scala code. In the SUDAF environment (cases (ii) and (iii)), UDAFs are provided as mathematical expressions and used in the SQL queries. And in case (iii) of SUDAF, the precomputed sharing relationships in $saggs_2(X)$ are exploited to reuse cached aggregation states to compute new ones. In SUDAF sharing environment, we prefetch a moment sketch (MS) [16, 26] under one of the two selected query orders. At the end of this section, we also present a scenario of running a random sequence of 200 queries in the Spark SQL context.

Our main findings are twofold. First, we observed that SUDAF without sharing outperforms both PostgreSQL and Spark SQL despite the overhead in SUDAF due to the analysis and decomposition of UDAF expressions. The main reason that explains these performances comes from the fact that rewriting of UDAFs by SUDAF, which is based on canonical forms, leads to implementations that use PostgreSQL or Spark SQL built-in functions, these later ones being much faster than PostgreSQL or Spark SQL UDAFs. The second finding is SUDAF with sharing outperforms both PostgreSQL and Spark SQL. In particular, the fine-grained unit of caching used in SUDAF improves the sharing possibilities and increases the gain brought by sharing.

Experiment setup. All experiments of Spark SQL are performed on a cluster with 1 master node and 6 worker nodes, running Ubuntu server 16.04, Spark 2.2.0 and Hadoop 2.7.4. The master node has a processor of 6 cores (XEON E5-2630 2.4GHz), 16 GB of main memory and 160 GB of disk space, and every worker node has a processor of 4 cores (XEON E5-2630 2.4GHz), 8 GB of main memory and 80 GB of disk space. All experiments on PostgreSQL are only performed on the master node running PostgreSQL 11.4.

Query models. The three query models used in experiments are illustrated below, where AGG represents an aggregation.

```
-- Query model 1
SELECT AGG(internet_traffic) FROM milan_data;
-- Query model 2
SELECT square_id, AGG(internet_traffic) FROM milan_data
GROUP by square_id ORDER by square_id LIMIT 20;
-- Query model 3, the TPC-DS query 7 when AGG is avg
SELECT i_item_id, AGG(ss_quantity) agg1, AGG(ss_list_price) agg2,
       AGG(ss_coupon_amt) agg3, AGG(ss_sales_price) agg4
FROM   store_sales, customer_demographics, date_dim, item, promotion
WHERE  ss_sold_date_sk = d_date_sk and
       ss_item_sk = i_item_sk and
       ss_cdemo_sk = cd_demo_sk and
       ss_promo_sk = p_promo_sk and cd_gender = 'M'
       and cd_marital_status = 'S' and
       cd_education_status = 'College' and
       (p_channel_email = 'N' or p_channel_event = 'N')
       and d_year = 2000
GROUP BY i_item_id ORDER BY i_item_id LIMIT 100;
```

Datasets. The first two query models are evaluated on the Milan dataset [22] and the third query model is evaluated on the TPC-DS [27] dataset. For the experiments of PostgreSQL, the Milan dataset consists of 72.6 million rows in total and the TPC-DS dataset comes with scale = 20. For the experiments of Spark SQL, the Milan dataset consists of 319 million rows in total and the TPC-DS dataset comes with scale = 100. All data files in Spark SQL experiments are in Parquet format.

Aggregate functions. We use the following 11 aggregate functions to instantiate our query models: cubic_mean (cm), quadratic_mean (qm), geometric_mean (gm), harmonic_mean (hm), min, max, count, sum, average (avg), standard deviation (std), variance (var). In the used PostgreSQL and Spark SQL version, all of these functions are built-in functions except the functions cm, qm, gm and hm which are implemented using PL/pgSQL in PostgreSQL and using *UserDefinedAggregateFunction* interface in Scala code in Spark SQL.

Query sequences. We instantiate each query model using each of the 11 aggregations and define the following two sequences of query executions for each instantiated query model:
AS1 = [cm, qm, gm, hm, min, max, count, std, var, sum, avg]
AS2 = [max, min, sum, avg, count, std, var, cm, gm, hm, qm]
Thus, we obtain 6 query sequences in total, where each query sequence is made of 11 aggregate queries. In the SUDAF sharing environment (cases (ii)) with the sequence AS2, we prefetch a moment sketch (MS) [16, 26] with parameter k = 10, which consists of a set of aggregate functions ($min, max, count, \sum x_i, ..., \sum x_i^k, \sum ln(x_i), ... \sum ln^k(x_i)$) and can be used to approximate a percentile, e.g., median.

Experimental results. We executed the 6 query sequences on PostgreSQL or Spark SQL, SUDAF without sharing, and SUDAF with sharing, and we report the execution time of every query. In scenarios with sharing, we use precomputed sharing relationships of symbolic aggregation states in $saggs_2(X)$, and we also add three additional relationships for SQL standard aggregates, max, min, and count, that they share themselves. Note that in the reported results we do not take into account the overhead needed to precompute sharing relationships in $saggs_2(X)$ which is part of the initialization of SUDAF and takes 110 *ms*. However, the overhead due to the cache access is included in the global execution time reported for each query. This overhead is about 2 *ms* for query model 1 or 2, and about 5 *ms* for query model 3. Moreover, the prefetching of a moment sketch is a preprocessing step in the aggregate sequence AS2, and the corresponding time is not taken into account. In the context of PostgreSQL, the prefetching time is 13.06 *s* for query model 1, 15.16 *s* for query model 2, and 14.53 *s* for query model 3. In the context of Spark SQL, the prefetching time is 1.87 *s* for query model 1, 2.17 *s* for query model 2, and 3.82 *s* for query model 3.

The total execution time of each query sequence in each query model is presented in Figure 6 for the case of PostgreSQL and in Figure 7 for the case of Spark SQL. We observe that PostgreSQL or Spark SQL (respectively, SUDAF without sharing) always have the same execution time for the two sequences of the same model. Also, we observe that SUDAF without sharing outperforms both PostgreSQL and Spark SQL in all the considered scenarios except query model 3 in Spark SQL (the reason is explained later). SUDAF with sharing shows the best performances, whatever the considered sequence or query model. In the sequel, we discuss the execution time of every individual query depicted in Figure 8 and 9 for the cases of PostgreSQL and Spark SQL.

SUDAF without sharing. In this scenario, SUDAF only rewrites aggregations to built-in ones and it does not share computations in processing query sequences. For the case of PostgreSQL, compared to PostgreSQL UDAF queries, SUDAF speeds up UDAF queries up to 20X in query model 1 (Figure 8 (a) and (b)), 4X in query model 2 (Figure 8 (c) and (d)), and 2X in query model 3 (Figure 8 (e) and (f)). For the case of Spark SQL, compared to Spark UDAF queries, SUDAF speeds up UDAF queries up to 3X in query model 1 (Figure 9 (a) and (b)), 2X in query model 2 (Figure 9 (c) and (d)), and have identical query time in query model 3 (Figure 9 (e) and (f)). The major reason for this improvement is that SUDAF rewrites queries with UDAFs to queries with partial aggregations that can be evaluated using PostgreSQL or Spark SQL built-in functions, which are faster compared to PostgreSQL or Spark UDAFs. The performance improvements of such a rewriting depends on the number of data to be aggregated. The instances of query model 1 have the highest number of values to be aggregated while the instances of query model 3 have the smallest number of values as aggregation inputs. Therefore, for the case of query model 3, the difference between SUDAF only with the rewriting functionality and Spark SQL is less noticeable.

SUDAF with sharing. In this scenario, SUDAF rewrites aggregations to built-in ones and shares the computation results of partial aggregations in every query sequence. For the sequence AS1, we observe in Figure 8 (a), (c) and (e) and in Figure 9 (a), (c) and (e) that for all the considered query models the computation times of count, variance (var), sum and average (avg) decrease drastically w.r.t. the no sharing option. This is because SUDAF

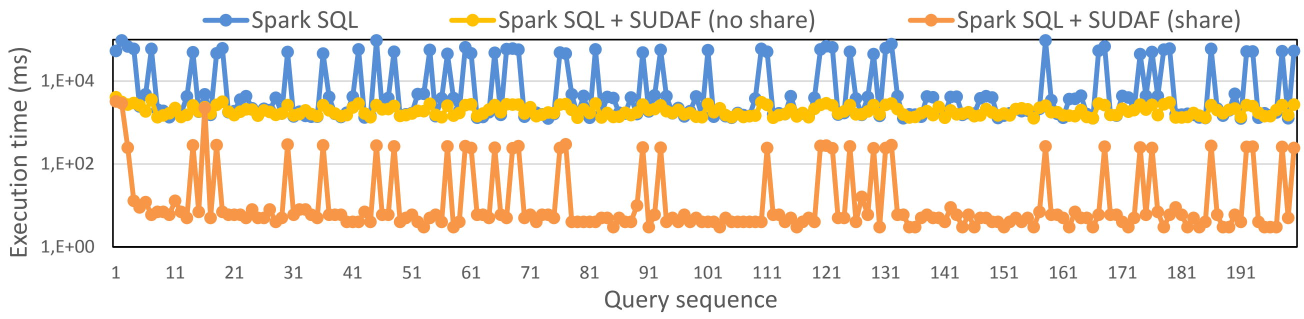


Figure 10: Execution time in Spark SQL of a random sequence of 200 queries.

is able to reuse cached results from earlier aggregates in the sequence AS1. As it can be observed in Figure 8 (b), (d) and (f) and in Figure 9 (b), (d) and (f), the sequence AS2 is more advantageous for sharing due to the prefetched moment sketch. Indeed, the moments sketch consists of 33 partial aggregates which are cached by SUDAF and reused for the computation of all the remaining aggregations in the sequence AS2 except the harmonic mean (hm). Computing queries with the harmonic mean in AS2 still requires data access since the aggregation state $\sum x_i^{-1}$ in the harmonic mean is not evaluated in previous computing.

Random query sequence. We present in Figure 10 the scenario of running a random sequence of 200 queries in Spark SQL, which are instances of the query model 2 having the following 16 aggregate functions: (min, max, sum, avg, harmonic_mean, quadratic_mean, cubic_mean, geometric_mean, stddev, variance, skewness, kurtosis, approx_median, count, approx_first_quantile, approx_thrid_quantile). The benefits of using SUDAF in this scenario are more obvious (the orange line in Figure 10).

7 RELATED WORKS

There is a wealth of research on queries with aggregations, earlier works focusing on standard aggregations (e.g., [8, 9, 12, 18, 19, 35]) and then extended to UDAFs (e.g., [6, 10, 20, 24]). Partial aggregation appeared as an essential technique used to improve the performances of aggregations: instead of computing aggregations on a complete multiset, applying aggregations on subsets and merging intermediate results is an efficient solution in numerous scenarios. In OLAP applications, partial aggregation enables computing aggregation by merging summaries of cells with different granularities across multi-dimensional data, thereby allowing aggregate queries to be executed on pre-computed results instead of base data [8]. In join-aggregate query optimization, partial aggregation enables to compute group-by aggregation before joins to decrease the size of intermediate results [35], i.e., the eager group-by technique. In distributed computing, partial aggregation allows to push the execution of aggregation before transferring data on networks [36], thereby decreasing the overhead of data shuffling, which is usually called initial reduce in MapReduce-like frameworks. An original classification of aggregations [18] distinguishes between algebraic aggregations having partial aggregation with fixed size results, and holistic functions where there is no constant bound on the storage size for partial aggregation. Several properties are proposed to have partial aggregations from algebraic aggregations, such as decomposable aggregation [35], commutative semi-group aggregation [11] and associative and commutative aggregation [36].

Most modern data management and analysis systems support UDAFs (e.g., [1, 2, 4, 21, 28, 29]). In the original MapReduce (*MR*) framework [3, 14], UDAFs are implemented according to the *MR* paradigm without requiring any specific template. This makes the semantics of UDAFs hidden in the implementations and hinders optimization possibilities (e.g., reordering with relational operators and other UDAFs [20]). However, in most of recent systems, users define UDAFs using an *IUME* pattern (initialize function, update function, merge function and evaluate function). Although such an approach enables exploiting the properties of the merging functions to allow optimization based on partial aggregations, e.g., parallel computation of the merging functions, part of the UDAF semantics still remains hidden in the implementation, which hampers the opportunity of aggregate sharing. In addition, implementing UDAFs in existing frameworks may be a tedious task since it is up to the user to map a UDAF to the implementation paradigm (*MR* or *IUME*). We build on a canonical form of UDAFs proposed in [10] to design SUDAF by allowing users to specify UDAFs as mathematical expressions and then automatically generate canonical forms of UDAFs which are compliant with the $IUME$ pattern. Consequently, with SUDAF a user does not need to handle the problem of how to obtain partial aggregations from UDAFs. Moreover, SUDAF knows the semantics of partial aggregations (primitive functions used in partial aggregation) which extends the optimization opportunities.

Different facets of the sharing problem have been studied in the literature, e.g., rewriting aggregate queries using materialized views [11, 12], reusing caches to accelerate multi-dimensional queries [8, 15], or identifying overlapping processing for multiple aggregate queries with various selection predicates [19], group-by attributes [9] and sliding-windows [5, 23]. Most of these approaches focus on the data dimension, i.e., they consider the problem of sharing the same aggregation across different ranges or granularities of data. Our work does not consider the data granularity dimension where existing techniques, e.g., [15, 33], can be used to extend SUDAF in this direction. [10, 11] proposes to predefine computation rules for sharing between different aggregations. However, SUDAF automatically identifies sharing opportunities on partial aggregates across different UDAFs.

The closest work to SUDAF is DataCanopy [33]. DataCanopy caches the basic aggregates (e.g., $\sum x_i$, $\sum x_i^2$ and $\sum x_i y_i$) of statistical measures and then is able to reuse them for queries with various range predicates. Basic aggregates are maintained at a granularity of a chunk (smallest portion of data), and DataCanopy allows sharing across queries covering overlapping chunks. In DataCanopy, basic aggregates are fixed in advance and the decomposition of an aggregate into basic ones is predefined (see Table 1 of [33]). We discuss the differences between DataCanopy and SUDAF as follows. From a theoretical standpoint, the sharing condition in SUDAF allows having a scalar function between two aggregates (see Theorem 4.1), which is more general compared to sharing identical basic aggregates in DataCanopy. From a practical standpoint, our approach is complementary to DataCanopy in the sense that DataCanopy deals with sharing w.r.t. the data dimension and proposes a static approach for sharing on the aggregation dimension, whereas SUDAF extends its static approach to a dynamic one w.r.t. the aggregation dimension. More precisely, the sharing opportunities w.r.t the aggregation dimension are automatically identified in SUDAF, which do not require any decomposition rule and are not restricted to a fixed set of aggregates. For example, if we restrict the attention to the set of predefined basic aggregates introduced in [33], the execution of a geometric mean ($gm(X) = exp(\frac{\sum ln(x_i)}{count}, \forall x_i > 0)$ cannot take any benefit from the static caching solution used in DataCanopy (i.e., cannot reuse the basic aggregates stored in the cache and do not lead to any new cached computation results). In contrast, SUDAF can reuse partial aggregates from the cache to compute *gm* and if not possible, it caches the partial aggregates $(\sum ln(x_i), count)$ after computing *gm* from base data. To obtain similar behavior, one needs to explicitly define additional basic aggregates in DataCanopy together with the appropriate decomposition rules for *gm*. In addition to being cumbersome, such a task requires to know in advance the query workloads that will be issued.

8 CONCLUSIONS AND FUTURE WORKS

In this paper, we introduce the design principles underlying SUDAF, a framework that provides a set of primitive functions together with a composition operator to enable users to define mathematical expressions of their UDAFs. SUDAF comes equipped with the ability to automatically rewrite partial aggregations, which are factored out from mathematical expressions of UDAFs, using built-in aggregates, and supports efficient dynamic caching and sharing of partial aggregates. We showed experimentally the benefits of rewriting partial aggregates of UDAFs using built-in functions and sharing partial aggregates to improve the performances of queries with UDAFs.

In this paper, we focus on the issue of *how to compute a UDAF from another UDAF*. In practice, to share computation results of different queries, we need to consider the data dimension, e.g., different range queries, or different OLAP queries. Sharing over data dimension has been extensively studied in existing works [15, 33]. The general idea is to split cached query results using chunks. For the case of range queries, a chunk is a range predicate over an attribute. For the case of OLAP queries, a chunk is a region in a multi-dimensional space. Merging our sharing approach with such approaches, we can share computation results for different queries with different UDAFs. As another future work, we envision to exploit the fact that the semantics of UDAFs is known by SUDAF to investigate query optimization and query rewriting problems for join and group-by queries with UDAFs.

9 ACKNOWLEDGMENTS

This research was supported by the French government IDEX-ISITE initiative 16-IDEX-0001 (CAP 20-25) and by the CPER of the "Région Auvergne-Rhône-Alpes", the French Government and FEDER from the European community.

REFERENCES

[1] Aache Hive. 2019. https://hive.apache.org.
[2] Apache Flink. 2019. https://flink.apache.org.
[3] Apache Hadoop. 2019. https://hadoop.apache.org.
[4] Apache Spark. 2019. https://spark.apache.org/.
[5] Arvind Arasu and Jennifer Widom. 2004. Resource Sharing in Continuous Sliding-window Aggregates *(VLDB '04)*. VLDB Endowment, 336–347.
[6] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate Sharing for User-Defined Windows. 1201–1210.
[7] Central moments. 2019. https://en.wikipedia.org/wiki/Central_moment.
[8] Surajit Chaudhuri and Umeshwar Dayal. 1997. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.* 26, 1 (March 1997), 65–74.
[9] Zhimin Chen and Vivek Narasayya. 2005. Efficient Computation of Multiple Group by Queries. In *SIGMOD '05*. ACM, New York, NY, USA, 263–274.
[10] Sara Cohen. 2006. User-defined Aggregate Functions: Bridging Theory and Practice. In *SIGMOD '06*. ACM, New York, NY, USA, 49–60.
[11] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. 2006. Rewriting Queries with Arbitrary Aggregation Functions Using Views. *ACM Trans. Database Syst.* 31, 2 (June 2006), 672–715.
[12] Sara Cohen, Werner Nutt, and Alexander Serebrenik. 1999. Rewriting Aggregate Queries Using Views. In *PODS '99*. ACM, New York, NY, USA, 155–166.
[13] Sara Cohen, Werner Nutt, and Alexander Serebrenik. 2000. Algorithms for Rewriting Aggregate Queries Using Views. In *ADBIS-DASFAA '00*. Springer-Verlag, London, UK, UK, 65–78.
[14] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04*. San Francisco, CA, 137–150.
[15] Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. 1998. Caching Multidimensional Queries Using Chunks. In *SIGMOD '98*. ACM, New York, NY, USA, 259–270.
[16] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. 2018. Moment-based Quantile Sketches for Efficient High Cardinality Aggregation Queries. *Proc. VLDB Endow.* 11, 11 (July 2018), 1647–1660.
[17] Michel Grabisch, Jean-Luc Marichal, Radko Mesiar, and Endre Pap. 2011. Aggregation function: Means. *Information Sciences* 181, 1 (January 2011), 1–22.
[18] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery* 1, 1 (01 Mar 1997), 29–53.
[19] Ryan Huebsch, Minos Garofalakis, Joseph M Hellerstein, and Ion Stoica. 2007. Sharing Aggregate Computation for Distributed Queries. In *SIGMOD '07*. ACM, New York, NY, USA, 485–496.
[20] Fabian Hueske, Mathias Peters, Aljoscha Krettek, Matthias Ringwald, Kostas Tzoumas, Volker Markl, and Johann-Christoph Freytag. 2013. Peeking into the Optimization of Data Flow Programs with MapReduce-style UDFs. *ICDE*.
[21] IBM DB2. 2019. https://www.ibm.com/analytics/db2.
[22] Telecom Italia. 2015. Telecommunications - SMS, Call, Internet - MI. https://doi.org/10.7910/DVN/EGZHFV
[23] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly Sharing for Streamed Aggregation. In *SIGMOD '06*. 623–634.
[24] Arun Kumar, Matthias Boehm, and Jun Yang. 2017. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *SIGMOD '17*. 1717–1722.
[25] Radko Mesiar Michel Grabisch, Jean-Luc Marichal and Endre Pap. 2009. *Aggregation Functions*. Cambridge University Press, Cambridge.
[26] Moment-based quantile sketches for aggregations. 2018. https://github.com/stanford-futuredata/msketch.
[27] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *VLDB '06*. 1049–1058.
[28] Oracle. 2019. https://docs.oracle.com/.
[29] PostgreSQL. 2019. https://www.postgresql.org/docs/.
[30] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *SIGMOD '16*. 3–18.
[31] Sharing computations for user-defined aggregate functions (technical report). 2019. https://github.com/CHAOHIT/SUDAF/blob/master/sudaf-technical-report.pdf.
[32] Standardized moments. 2019. https://en.wikipedia.org/wiki/Standardized_moment.
[33] Abdul Wasay, Xinding Wei, Niv Dayan, and Stratos Idreos. 2017. Data Canopy: Accelerating Exploratory Statistical Analysis. In *SIGMOD '17*. ACM, New York, NY, USA, 557–572.
[34] Wolfram Mathematica. 2019. https://reference.wolfram.com/language/guide/MathematicalFunctions.
[35] Weipeng P. Yan and Per-Ake Larson. 1995. Eager Aggregation and Lazy Aggregation. In *VLDB '95*. 345–357.
[36] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. 2009. Distributed Aggregation for Data-parallel Computing: Interfaces and Implementations. In *SOSP '09*. ACM, New York, NY, USA, 247–260.

Tracing nested data with structural provenance for big data analytics

Ralf Diestelkämper Melanie Herschel
IPVS - University of Stuttgart
Stuttgart, Germany
ralf.diestelkaemper|melanie.herschel@ipvs.uni-stuttgart.de

ABSTRACT

Big data analytics systems such as Apache Spark natively support nested data formats since they offer operators to manipulate nested lists and complex types. Compared to flat data, nested data introduces further complexity and sources of error, e.g., when developing data processing pipelines, performing auditing tasks, or performance tuning. To ease such tasks, we propose a provenance-based solution tailored to nested data processing in big data analytics systems. Unlike previous solutions, it combines (i) tracing provenance of *nested data* with (ii) *efficient* and *scalable* provenance processing, leveraging a newly proposed *structural provenance* that traces structural manipulations through data processing pipelines in addition to data. We provide a formal definition of structural provenance, as well as methods to efficiently capture and succinctly backtrace it. We implement them in our Pebble system in Apache Spark and validate its performance and usefulness on up to 500GB of real-world data.

1 MOTIVATION

Big data analytics systems such as Apache Spark or Flink are frequently the means of choice to build data processing pipelines that process large quantities of nested data. These pipelines transform nested lists and complex types stored in nested data formats like JSON, protocol buffer, or parquet. Provenance solutions that capture meta-data about the data processing [14] have proven to be useful for analyzing the internals of data processing pipelines, e.g., for debugging purposes. These solutions typically have two phases, a provenance capture phase to collect the meta-data and a provenance query phase to analyze the meta-data. For big data analytics systems, we distinguish two categories of provenance solutions: (i) Efficient and scalable solutions that track individual, flat data items (i.e., tuples) from the input to the output over each execution step [15–17, 22]. They capture so-called lineage or why-provenance [7]. (ii) System prototypes that compute provenance polynomials of nested data [2, 28]. They capture how-provenance, which provides both the input items contributing to the result and the data combination process an item undergoes.

Solutions of the first category fail to track nested items accurately. Solutions of the second category do not efficiently scale to big data processing pipelines. To capture the how-provenance, these solutions propagate the growing provenance polynomial through the entire pipeline or require annotation of each nested element, which imposes a very high and practically unacceptable overhead [16, 17]. Further, these solutions have to offload the provenance to external tools to query the captured provenance. Thereby, they miss potential performance and usability benefits compared to solutions that are fully integrated into the big data analytics or data-intensive, scalable computing (DISC) system.

We, therefore, present a *DISC system integrated* provenance solution for nested data that is as *efficient* and *scalable* as solutions of the first category while, at the same time, at least as *accurate* as solutions of the second category [9]. Our solution leverages our newly defined *structural provenance* to provide both features.

Structural provenance records identifiers for top-level data items only. For attributes and nested data items, it captures paths on a schema level. To provide accurate provenance when queried, it employs these identifiers and paths to trace back individual nested items at attribute level. Capturing paths instead of identifier annotations for nested data further allows us to distinguish between paths that are used for access (e.g., during filtering) or manipulation (e.g., during flattening). We can thereby differentiate contributing attributes, i.e., attributes needed to reproduce a result item, and influencing attributes that are accessed during data processing but not required to reproduce a result. This distinction, which is unique compared to existing data provenance models, qualifies structural provenance for use-cases beyond debugging such as auditing or determining data-usage patterns for partitioning, data compression, and workload optimization.

Auditing. Auditing aims at identifying and analyzing data breaches. These breaches commonly stem from attacks of company insiders who extract sensitive data by querying data and leaking the query result. Auditing solutions are designed to identify both these insiders and the customers whose data are leaked [19]. To address the latter challenge, the solutions typically leverage some sort of data provenance. It serves to identify those input tuples that are exposed in a leaked query result. However, after the European Union has introduced the European general data protection regulation *GDPR* [26], European companies are not only required to identify the customers (tuples) whose data are leaked, but also which of their data are leaked (i.e., attributes such as name, address, or payment details). Structural provenance precisely provides the attributes and items in nested collections that contribute to a query result. Unlike existing data provenance solutions, it further reveals which attributes are not exposed in the result but have influenced it to create awareness for reconstruction attacks.

Data-usage patterns. Data-usage patterns reveal frequently used subsets of the input data over a query workload. These patterns serve to optimize data layout and compression or to improve query performance [25]. State-of-the-art scalable provenance solutions for DISC systems can identify subsets of the input data that are frequently used. This knowledge allows for horizontal (or row-based) data partitioning and distribution. Structural provenance further provides all the information needed for vertical (or column-based) partitioning since it reveals which attributes and nested items are accessed or manipulated. It even provides insights on attribute combinations that are frequently used together for data layout optimizations.

Capturing provenance imposes runtime and space overhead during pipeline execution. The mentioned use-cases are performed infrequently. Thus, keeping the overhead low during pipeline execution is essential to ensure efficiency and scalability. During the *provenance capture* phase, a system can typically opt for computing and storing the provenance of all processed data (*eager* approach) or decide to capture it on demand when users query the provenance (*lazy* approach). Consequently, during the *provenance query* phase, retrieving the desired provenance is more or less time-consuming. We consider the provenance capture and provenance query phases holistically. To this end, we devise a meet-in-the-middle approach that eagerly collects the necessary "pebbles" (i.e., identifiers and paths on schema level) during pipeline execution to later reconstruct or backtrace attribute-level provenance of nested data at query time. Our evaluation shows that capturing structural provenance introduces comparable overhead to state-of-the-art lineage solutions in DISC systems [17], while providing attribute-level precision.

This paper also presents the first provenance solution for nested data that seamlessly integrates into a big data analytics system (Apache Spark in our implementation). Existing solutions [2, 28] require offloading captured provenance for querying to separate, non-distributed applications. This has three drawbacks: (i) It prevents adopting a holistic provenance capture and querying approach to keep capture and query overhead reasonable; (ii) it forces users to leave their familiar environment; and (iii) it prevents scalable provenance querying.

Contributions and structure. To summarize, this paper presents research on processing structural provenance in big data analytics systems to accurately trace nested data in an efficient, scalable, and integrated way. This approach enables novel use-cases that arise in the context of big data processing. After discussing a running example in Sec. 2 and related work (Sec. 3) this paper covers the following contributions:

- **Structural provenance (Sec. 4).** We present a novel provenance model for nested data that tracks structural manipulations in addition to data dependencies, and distinguishes between data access and manipulation to support use-cases beyond debugging.
- **Lightweight structural provenance capture (Sec. 5).** We discuss how to capture structural provenance in big data analytics programs composed of filter, select, map, join, union, flatten, grouping, nesting, and aggregation operations. The capture is devised to incur a minimal overhead compared to the capture of flat provenance in DISC systems.
- **Backtracing for provenance query processing (Sec. 6).** We formalize the backtracing algorithm used at provenance query time. As input, users provide a tree-pattern that, upon its scalable execution, identifies data items for which provenance is requested. The backtracing algorithm computes provenance for these items based on the previously captured information.
- **Implementation and evaluation (Sec. 7)**. We implement our contributions in Pebble [9], our system for integrated provenance capturing and querying within Apache Spark. We conduct a quantitative evaluation of runtime and space overhead incurred by our solution on two large real-world data sets, validating the scalability of our solution. In comparison to the state-of-the-art lineage solution Titian [17], Pebble has comparable runtime and space overhead. However, as our workload shows, Pebble provides sufficient insight to support the above use-cases, unlike other solutions.

2 RUNNING EXAMPLE

To distinguish our research from related work and for illustration, we use a running example based on Twitter data. Among its roughly 1000 attributes, we focus on the tweeted *text*, the *user* tweeting, the *user_mentions* in the tweet, and the *retweet_cnt*. The input data is nested as shown in Tab. 1 (ignore colors and number annotations for now). This sample data is processed in the big data processing pipeline shown in Fig. 1. It results in a list of distinct users associated with tweets that they authored or were mentioned in, as shown in Tab. 2. The upper branch of the pipeline describes how authoring users become part of the result. Their tweets require a *retweet_cnt* of 0 before the pipeline reduces them to the *text*, *id_str*, and *name*. The lower branch processes tweets mentioning users. First, it flattens the *user_mentions* attribute to select the tweeted *text*, *id_str*, and *name* of each mentioned user. Then the pipeline unifies the results of both branches and groups by the user to aggregate the tweeted *texts* into a nested list.

In the result, a duplicate *Hello World* text occurs in the nested *tweets* of user *Lisa Paul*, short *lp*. To find out how this potential data quality issue occurred, we debug the pipeline by tracing back the duplicate texts in the context of user *lp*, which are highlighted in dark-green in Tab. 2. The solution presented in this paper returns the dark- and medium-green items in Tab. 1. The dark-green items are contributing data. They suffice to reproduce the dark-green items in the result. The medium-green items reveal which attributes potentially influence the result of the pipeline.

If trivially extended to nested data, scalable lineage solutions [15–17, 22] provide all input tweets that contain the user *lp*. They are highlighted in light-grey in the input. In reality, a user typically authors more than a handful of tweets and is potentially mentioned in more than a million tweets. These tweets would all be in the provenance returned by the lineage solutions. They mask the actual two tweets causing the duplicate text.

PROVision [28] supports the unnesting of data but does not explicitly support the nesting of data. Extending it with nesting requires a list collection UDF *cl*, which yields the following provenance polynomial for the entire result item 102 in Tab. 2:

$$(p_1+p_{12}+p_{17}+(p_{29}\cdot P_{flatten}(p_{29}\cdot[0])))\cdot$$
$$P_{cl}((p_1+p_{12}+p_{17}+(p_{29}\cdot P_{flatten}(p_{29}\cdot[0]))),(\langle p_1\rangle+\langle p_{12}\rangle+\langle p_{17}\rangle+\langle(p_{29}\cdot P_{flatten}(p_{29}\cdot[0]))\rangle))$$

	text	user		user_mentions		retweet_cnt
		id_str	name	id_str	name	
1	Hello @ls @jm @ls^{2}	lp^{3}	Lisa Paul^{4}	ls^{5} jm^{7} ls^{9}	Lauren Smith^{6} John Miller^{8} Lauren Smith^{10}	0^{11}
12	Hello World^{13}	lp^{14}	Lisa Paul^{15}			0^{16}
17	Hello World^{18}	lp^{19}	Lisa Paul^{20}			0^{21}
22	This is me @jm^{23}	jm^{24}	John Miller^{25}	jm^{26}	John Miller^{27}	0^{28}
29	Hello @lp^{30}	jm^{31}	John Miller^{32}	lp^{33}	Lisa Paul^{34}	1^{35}

Table 1: Example input data

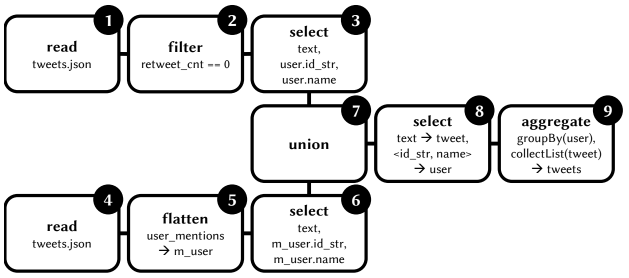


Figure 1: Example processing pipeline

	user		tweets
	id_str	name	text
101	ls	Lauren Smith	Hello @ls @jm @ls
			Hello @ls @jm @ls
102	lp	Lisa Paul	Hello @ls @jm @ls
			Hello World
			Hello World
			Hello @lp
103	jm	John Miller	Hello @ls @jm @ls
			This is me @jm
			This is me @jm

Table 2: Example result data

Essentially, the first line tells us that the result item is based on the source tuples annotated with 1, 12, and 17, denoted as p_1, as well as p_{12}, p_{17} (all these are processed by the upper branch of the pipeline in Fig. 1), and p_{29}, with some of its data flattened out during processing (corresponds to the lower part of the pipeline). The second line makes use of our extension and describes how data is combined by the remainder of the pipeline where the tuples mentioned above are grouped and aggregated. The example shows that the provenance is very verbose while not precisely tracing the dark-green data items of the user question. This is the case since it collects tuple-based provenance polynomials only.

Lipstick [2] traces provenance polynomials for each nested item. This allows pinpointing the dark-green nested values *Hello World* and *lp* correctly. However, Lipstick requires annotating all values, not just the tuples, e.g., 35 rather than 5 annotations, as indicated by the superscript italic numbers in Tab. 1. This entails a significant runtime and space overhead, rendering the solution impractical when needing to scale to large volumes of data.

We also differentiate structural provenance from where-provenance [4], which determines where a (nested) result value is copied from. In our example, the where-provenance (extended to the processing pipelines we consider) would include, for the value *lp* the "cells" with superscript annotation 14, 19, and 33 of Tab. 1. This is combined with the where-provenance of the *Hello World* result values via product. The result is not sufficiently accurate because it cannot capture that the dark-green values of the output need to be traced within their common context.

No existing solution allows recognizing (i) that the *user* attribute is unnested and nested again, (ii) that the *id_str*, *lp*, and the *text* attribute *Hello World* are subject to different, independent, structural manipulations, and (iii) that the medium-green *retweet_cnt* and *name* values in Tab. 1 are accessed for filtering and grouping, respectively. Even though these values are not needed to reproduce the queried result, they are influencing the result, which is valuable information in certain use-cases. Structural provenance captures all this information since it captures not only data dependencies but also path dependencies.

To get an understanding of querying structural provenance, consider the right tree in Fig. 2. The string labels of tree nodes denote attribute names whereas numbers refer to provenance ids (e.g., 102) or positions in nested collections (e.g., 2 and 3). The displayed tree represents the structure associated with our sample user query. It encodes the path to user *lp* and the duplicate *Hello World* items in the context of top-level data item 102. Note that *name* is absent from this tree since it is not pertinent to the user query. Backtracing this tree yields the two trees on the left of Fig. 2. These distinguish between data items that contribute to the result (dark-green) and data items that influence it (medium-green). The nodes match the green items in Tab. 1. A closer look at the medium-green *name* node reveals that this node influences the queried result since it is accessed for grouping

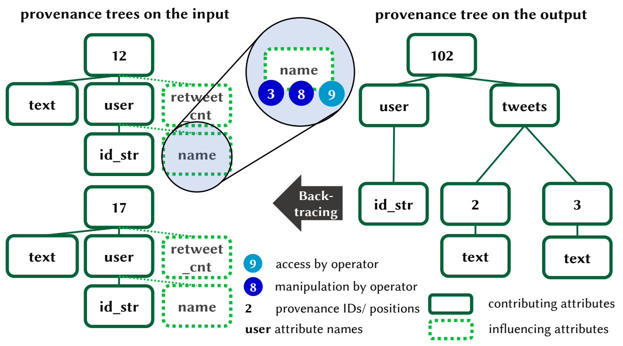


Figure 2: Example provenance trees. Tracing the tree on the right back to the input yields the trees on the left

(light-blue 9). Similarly, the *retweet_cnt* influences the result since it is accessed for filtering. Further, the *name* node undergoes structural manipulations at operator 3 and 8 (dark-blue).

3 RELATED WORK

This section generalizes the discussion of existing approaches that we provided along with the running example. We divide our discussion into research on data provenance in DISC systems and provenance models for nested data, summarized in Tab. 3.

3.1 Data provenance in DISC systems

Data provenance has been studied for various applications [14]. While the majority of approaches has focused on relational data processed by relational queries, first solutions have emerged for tracing provenance in DISC systems such as Titian [12, 16, 17] for Spark, Lipstick for PigLatin (Hadoop) [2], as well as RAMP [15], Newt [22], and PROVision [28] for multiple DISC systems.

Titian, RAMP, and Newt trace lineage of data items, i.e., they determine which top-level data items contribute to which output item. These solutions scale well but do not trivially extend to nested data. PROVision extends the provenance model for top-level data (or flat) items to also capture provenance of data items in nested collections. It lacks information on attribute level access. Lipstick is the only solution that supports provenance capture for nested data at attribute level. However, it requires annotations for each data value, not only the top-level data items. Structural provenance provides provenance on attribute level but requires annotation on top-level items only since it records access to attributes and nested data using paths. These paths are recorded on a schema level, saving space and runtime overhead.

All the above solutions except for Titian require offloading the provenance to an external tool. Titian integrates provenance querying directly into the DISC system. Thus, provenance queries can be integrated into a big data processing pipeline just like any other query. Our system extends Titian's integrated querying means with tree-patterns [13, 23] to address combinations of nested data items. Further, we present the first solution that tracks access and manipulation of attributes.

3.2 Provenance models for nested data

Focusing on nested data, at least three major directions to formalize provenance models have been researched: (i) models for why-, how-, and where-provenance, (ii) graph-based provenance models, and (iii) program slicing models.

For unions of conjunctive queries, Buneman et al. [4] define a why- and where-provenance model for nested data. This model

Feature	Titian	Ramp	Newt	Lipstick	PROVision	HowProvNested	Why/Where Prov	Kwasnikowa	Acar	Program Slicing	Structural Prov
Data provenance for nested data	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓
Provenance of acces and manipulation	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Provenance of data item structure	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Eager/lazy provenance computation	✓ / ✗	✓ / ✗	✓ / ✗	✓ / ✗	✗ / ✓	n.a.	n.a.	n.a.	✗ / ✓	✓ / ✓	✓ / ✓
Implementation-independent provenance query formalism	✗	✗	✗	✓	✗	n.a.	n.a.	na.	✗	✗	✓
DISC system compatibility/integration	✓ / ✓	✓ / ✗	✓ / ✗	✓ / ✗	✓ / ✗	✗ / ✗	✗ / ✗	✗ / ✗	✗ / ✗	✗ / ✗	✓ / ✓
Reported implementation	Spark RDDs	Hadoop	Hadoop/ Hyracks	PigLatin	Java	no	no	no	Haskell	Haskell	Spark Datasets
Evaluated for scalability	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓

✓ Supported ✗ Not Supported

Table 3: Feature overview of related work

does not extend to the programs defining data analytics pipelines in DISC systems, like the one shown in Fig. 1, since they may include map or reduce functions or any other higher-order functions in general. To model the how-provenance of nested data, a semiring-model for a subset of XQuery has been proposed [10, 18]. However, this model does not include complex operations over nested data, such as aggregations. The only how-provenance model supporting aggregations that we are aware of applies to relational data only [3].

Lipstick [2], makes use of a graph model to describe the how-provenance. This model only applies semiring annotations where possible. It provides no formal model definition for aggregations, nesting, and flattening of nested data. Kwasnikowska and Acar et al. [1, 20] also employ a graph-based provenance model to track nested data items. These solutions are essentially limited to the operations defined in the Nested Relational Calculus (NRC) [5], which do not include aggregations or joins. Also, the reported implementation and evaluation (if any) indicate that they neither integrate nor scale sufficiently to apply on DISC systems.

All provenance solutions mentioned so far do not distinguish between access and manipulation as they focus on tracing data values. In that respect, the work closest to our structural provenance model is the program slicing model [6], which tracks provenance traces for NRC operators over nested data. To provide formal guarantees, the model is limited to a small set of semantically fully specified NRC operators. In practice, it is infeasible to provide semantics for all higher-order functions such as map operations, which allow for user-defined functions. Via trace slicing it is possible to query provenance for individual nested items. However, like the other described models, this model is designed to trace data values and manipulations of them rather than structural manipulations. It is not expressive enough to faithfully capture and query structural manipulations. Its implementation is not designed or evaluated for efficiency or scalability.

The final column of Tab. 3 summarizes the capabilities of our system, which we have highlighted previously. These capabilities are based on processing structural provenance, discussed next.

4 STRUCTURAL PROVENANCE

This section formalizes structural provenance. We first present the data model and the execution model to define the corresponding structural provenance model afterwards.

4.1 Data model

DISC systems process collections of typed nested data items, which we refer to as (nested) datasets. These datasets support positional access, and, thus, the handling of ordered datasets.

Definition 4.1. (*Nested dataset*) A nested dataset D comprises constants, data items, bags, and sets, denoted and typed as shown in Tab. 4. D is a list of data items $d_1, \ldots, d_n$ with or without duplicates (ordered bag vs. set), i.e., $D = B|S$. Each data item d is a list of $a_i : v_i$ pairs. Attribute names $a_1, ..., a_n$ are unique labels within each data item. Values $v_1, ..., v_n$ may be bags, sets, data items, or constants, i.e., $v = B|S|c|d$.

Name	Notation	Type $\tau(\cdot)$
Constant	c	$Int\|Double\|String\|\ldots$
Data item	$d = \langle a_1 : v_1, ..., a_n : v_n \rangle$	$\langle a_1 : \tau(v_1), ..., a_n : \tau(v_n)) \rangle$
Bag	$B = \{\{d_1, ..., d_n\}\}$	$\{\{\tau(d)\}\}, \forall d, d' \in B, \tau(d) = \tau(d')$
Set	$S = \{d_1, ..., d_n\} \| d_1 \neq \ldots \neq d_n$	$\{\tau(d)\}, \forall d, d' \in S, \tau(d) = \tau(d')$

Table 4: Notation and types for nested collections

The type of D is defined recursively based on the type of its building blocks as described in Tab. 4, where $\tau(\cdot)$ returns the type of its parameter. Bags and sets are restricted to containing elements of the same type.

Example 4.2. All data shown in our running example conform to the above definition. The result data of Tab. 2 has type:
$\{\{\langle user{:}\langle id\_str{:}String, name{:}String\rangle, tweets{:}\{\{\langle text{:}String\rangle\}\}\rangle\}\}$

To access the different components defined by the data model, we define *access paths*, inspired by XPath expressions [24] to navigate XML data. Provided a context data item d, an access path navigates to "deeper" data in the nested model. Given that the data model ensures the order of data items in lists, we also model positional accesses in paths.

Definition 4.3. (Access path w.r.t. d) In the context of a data item d, we define an access path p by $p = d.p'$, $p' = x \mid x.p'$, $x = a \mid a[i]$. Here, p' is the path accessing x either directly or recursively. The accessed x is either an attribute a in the schema of the context data item, evaluating to its value, or the i-th component of a, denoted $a[i]$, evaluating to the item at the i-th position of a's value. For the recursive definition of p', the context data item is updated to the item referred to by x.

For simplification, we denote a path p with context data item d by p^d when the context is not clear. We refer to the enumeration of all paths that exist in a context d as path set PS^d.

Example 4.4. Considering the data item d_{102} in Fig. 2, the path $d_{102}.tweets$ evaluates to a list of four data items. Path $d_{102}.tweets[2].text$ points to the first *Hello World* in that list.

4.2 Execution model

The execution model defines the processing semantics of data analytics programs like the one in Fig. 1. These programs process data complying with our data model. We model a program as a directed acyclic graph (DAG) of individual operators, such as filter, flatten, join, etc. Each operator has its own execution semantics.

Definition 4.5. (Operator) An operator O takes a set of datasets $I = \{I_1, \ldots, I_k\}$ as input and returns a single result dataset R. Inference rules describe the execution semantics of an operator O. O has a unique identifier, a type, and its arguments.

Definition 4.6. (Program execution model) Let $G(V, E)$ be a DAG. $V = \{O_1, \ldots, O_n\}$ is the set of algebraic operators and E the set

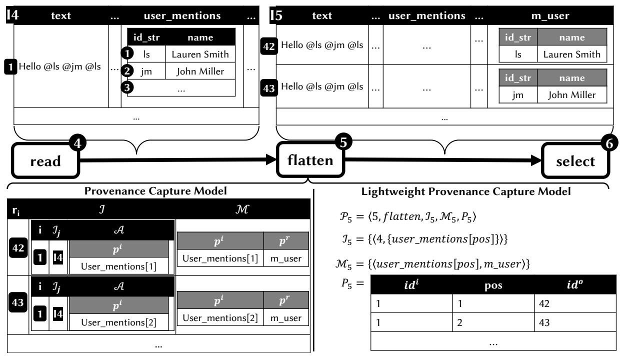


Figure 3: Provenance model and lightweight provenance capture applied on the flatten operator from the example

of directed edges that model the data flow. If and only if the result set R_i of O_i is in the set I_j of input datasets of operator O_j an edge $(O_i, O_j) \in E$ directed from O_i to O_j exists. While G may contain multiple source nodes (in-degree of 0), G has only one sink node (out-degree of 0), which outputs the final result dataset.

Example 4.7. The graph in Fig. 1 represents the execution model of our running example. The type and parameters of each operator are displayed inside the operator nodes. Further, each node is labeled with its identifier in the top right corner.

We abstract from a particular language to define the semantics of individual operators by extending the inference rules from [11] to describe the filter, select, map, join, union, flatten, grouping and aggregation operator with their semantics.

Example 4.8. We illustrate how inference rules work on our inference rule for a *join* operator:

$$\frac{\varphi(i,j) \Rightarrow \text{true}}{I_1.join[\varphi(i \in I_1, j \in I_2)](I_2) \Rightarrow \{\{\langle i,j\rangle \mid i \in I_1, j \in I_2\}\}}$$

The rule joins two input datasets I_1 and I_2 into a single result dataset. More precisely, the operator associates elements $i \in I_1$ with elements $j \in I_2$ based on a join condition ($\varphi(i,j) \rightarrow boolean$). With the precondition that φ evaluates to true, i.e., $\varphi(i,j) \Rightarrow$ true, the data item $\langle i,j\rangle$ becomes part of the result.

4.3 Provenance model

Our provenance model extends the program execution model described above by adding annotations to each node in G. More precisely, for each operator O represented by a node in G, it generates the result provenance $\mathcal{R}$ that contains the provenance of each result data item r_i in the result R of operator O.

Definition 4.9. (Result provenance w.r.t. result R of operator O) Let $\mathcal{R} = \{\rho_1, \ldots, \rho_n\}$ be the result provenance associated with $R = \{r_1, \ldots, r_n\}$. For each data item $r_i \in R$, we record the result data item provenance $\rho_i = \langle r_i, \mathcal{I}, \mathcal{M}\rangle$, where $\mathcal{I}$ is the provenance of input data items that contribute to r_i (see definition below) and $\mathcal{M}$ is a set of path pairs mapping access paths of input data items to paths of r_i to describe restructuring performed by O.

Definition 4.10. (Input provenance $\mathcal{I}$ w.r.t. result data item provenance ρ in result R of O) The input provenance $\mathcal{I}$ is a bag of triples $\langle i, I_j, \mathcal{A}\rangle$, where i is a data item from one of the input datasets of O, i,e., $i \in I_j, I_j \in I$, and $\mathcal{A}$ a set of paths recording the elements of i that are accessed by O to produce the result data item $r \in \rho$.

The above provenance model does not only contain information on the relationship between input and result data items of an operator (which is the previously mentioned lineage), it also records accesses and manipulations in $\mathcal{M}$ and $\mathcal{A}$ while transforming input items to result data items.

Example 4.11. To illustrate our model for structural provenance, we focus on the $flatten$ operator labeled 5 in Fig. 1. It unnests the nested items of attribute *user_mentions*. An excerpt from its input and output data is given at the top of Fig. 3. Black headers encode bag data types, light gray headers identify complex data items as nested data type. At the bottom left, Fig. 3 shows the provenance for the result items 42 and 43. The flatten operator derives item 42 from input item 1. It accesses path *user_mentions*[1] as recorded in $\mathcal{A}$. Further, it copies the first user of the *user_mentions* list (and implicitly, all paths in the path set $PS^{user\_mentions[1]}$) to the new item *m_user* as recorded in the mapping $\mathcal{M}$. Ignore the bottom right for now.

5 PROVENANCE CAPTURE

Based on the provenance model, we introduce inference rules describing the provenance capture of our supported operators. When the rules in Tab. 5 are annotated with a $*$, we extend an existing inference rule from [11]. Otherwise, we formalize the inference rule with and without provenance extension. Due to space constraints, we only show the complete set of inference rules with provenance. After explaining the map, flatten and aggregation rule, we show how we capture the structural provenance obtained from these rules efficiently.

5.0.1 Map. For the *map* operator, we assume that the function $\lambda(i)$ over input data item i returns a result of type data item, denoted as $\tau(\lambda(i)) \rightarrow \langle\ldots\rangle$. Given this precondition, without provenance capture, *map* returns the result of applying $\lambda(i)$, for each i in the input dataset I_1. The inference rule for the *map* operator in Tab. 5 additionally produces the provenance for each data item i. More formally, *map*, parameterized by a function λ, produces the result provenance $\mathcal{R}$, which is a bag of data items. Each data item extends the "normal" result of *map*, i.e., $\lambda(i)$ with two additional attributes: the input provenance $\mathcal{I}$ and mapping $\mathcal{M}$. For $\mathcal{I}$, the only input data item participating in producing an output data item $\lambda(i)$ is i, which originates from the single input dataset I_1. Because we generally do not know the internals of an arbitrary function λ, the set $\mathcal{A}$ is set to undefined, denoted by $\perp$. Thus, $\mathcal{I} = \{\{\langle i, I_1, \perp\rangle\}\}$. The structural mapping $M = \perp$ is also undefined because we have no knowledge of how elements from the input are restructured in the result.

Based on the rule for the *map* operator, we derive more general observations concerning our inference rules. The rules only capture structural provenance when the operator semantics clearly pinpoint paths to populate $\mathcal{A}$ and $\mathcal{M}$. Thus, the rule for the map operator captures the "undefined" semantics in $\mathcal{M} = \perp$ and $\mathcal{A} = \perp$. This semantics distinguishes from $\mathcal{M} = \emptyset$ and $\mathcal{A} = \emptyset$ semantics in the $filter$ and *union* rules. Both rules feature $\mathcal{M} = \emptyset$ since they do not restructure the data items. Each item's input structure is maintained in its entirety in the result. Further, the rule for the *union* operator holds $\mathcal{A} = \emptyset$ since it only performs an item-independent schema comparison of the input datasets.

5.0.2 Flatten. We introduced the flatten operator in Ex. 4.11 to illustrate our provenance model. Here, we explain the inference rule in Tab. 5 that captures the provenance for the $flatten$.

As preconditions, the rule requires the type of a_{col} to be either a list with duplicates (bag) or without duplicates (set). The result of the $flatten$ consists of items $r = \langle i, a_{new} : j\rangle$, where i refers

Filter*	$\dfrac{\varphi(i) \Rightarrow \text{true}}{I_1.filter[\varphi(i \in I_1)] \Rightarrow \{\{\langle i, \{\{\langle i, I_1, \cup p^i \in \varphi(i)\rangle\}\}, \emptyset\rangle \mid i \in I_1\}\}}$
Select*	$\dfrac{a_1, ..., a_n \in schema(I_1)}{I_1.select(a_1, ..., a_n) \Rightarrow \left\{\left\{\left\langle r, \left\{\left\{\left\langle i, I_1, \bigcup_{k=1}^{n} (a_k)^i \right\rangle\right\}\right\}, \bigcup_{k=1}^{n} \langle (a_k)^i, (a_k)^r \rangle \right\rangle \mid r = \langle i.a_1, ..., i.a_n\rangle, i \in I_1\right\}\right\}}$
Map*	$\dfrac{\tau(\lambda(i)) \Rightarrow \langle ... \rangle}{I_1.map[\lambda(i \in I_1)] \Rightarrow \{\{\langle \lambda(i), \{\{\langle i, I_1, \bot\rangle\}\}, \bot\rangle \mid i \in I_1\}\}}$
Join	$\dfrac{\varphi(i,j) \Rightarrow \text{true}}{I_1.join[\varphi(i \in I_1, j \in I_2)](I_2) \Rightarrow \left\{\left\{\left\langle r, \left\{\left\{\left\langle i, I_1, \bigcup_{p^i \in \varphi(i,j)} p^i \right\rangle, \left\langle j, I_2, \bigcup_{q^j \in \varphi(i,j)} q^j \right\rangle\right\}\right\}, \left\{\left\langle p^i, p^r\right\rangle \mid p^i \in schema(I_1)\right\} \cup \left\{\left\langle q^j, q^r \right\rangle \mid q^j \in schema(I_2)\right\}\right\rangle \mid r = \langle i, j\rangle, i \in I_1, j \in I_2\right\}\right\}}$
Union*	$\dfrac{\tau(I_1) = \tau(I_2)}{I_1.union(I_2) \Rightarrow \{\{\langle i, \{\{\langle i, I_1, \emptyset\rangle\}\}, \emptyset\rangle \mid i \in I_1\}\} \uplus \{\{\langle j, \{\{\langle j, I_2, \emptyset\rangle\}\}, \emptyset\rangle \mid j \in I_2\}\}}$
Flatten	$\dfrac{\tau(a_{col}) \Rightarrow \{\{\}\} \vee \tau(a_{col}) \Rightarrow \{\}}{I_1.flatten(a_{new}, explode(a_{col})) \Rightarrow \{\{\langle r, \{\{\langle i, I_1, \{\langle (a_{col}[x])^i \rangle\}\rangle\}\}, \{\langle (a_{col}[x])^i, a^r_{new}\rangle\}\rangle \mid r = \langle i, a_{new} : j\rangle, i \in I_1, j \in i.a_{col} \text{ at position } x\}\}}$
Grouping*	$\dfrac{G = \{\{g_1, ..., g_n\}\} \qquad \pi_G(i) = \langle g_1 : i.g_1, ..., g_n : i.g_n\rangle}{I_1.groupBy(g_1, ..., g_n) \Rightarrow \{\{\{\{\langle i, \{\{\langle i, I_1, \{g_1, \ldots, g_n\}\rangle\}\}, \emptyset\rangle \mid i \in I_1, \pi_G(i) == j\}\} \mid j \in set(\{\{\pi_G(i) \mid i \in I_1\}\})\}\}}$
Aggregation	$\dfrac{\begin{array}{c} \tau(I) == \{\{\tau(I_1), ..., \tau(I_n)\}\} \qquad \tau(I_1) == \ldots == \tau(I_n) == \{\{...\}\} \\ A_c == \left\{\alpha_{c_1}(a_1), ..., \alpha_{c_m}(a_m)\right\} \text{ with } \tau(\alpha_{c_k}(a_k)) \Rightarrow c \qquad A_B == \left\{\alpha_{B_1}(b_1), ..., \alpha_{B_p}(b_p)\right\} \text{ with } \tau(\alpha_{B_k}(b_k)) \Rightarrow \{\{...\}\} \\ \forall I_k \in I, \forall i, j \in I_k, \pi_G(i) == \pi_G(j) \text{ with } G == schema(I_k) \setminus \{a_1, \ldots, a_m, b_1, \ldots, b_p\} \end{array}}{I.agg(A_c, A_B) \Rightarrow \left\{\left\{\left\langle \begin{array}{l} r = \left\langle d, a_{\alpha_{c_1}} : \alpha_{c_1}\left(\pi_{a_1}(I_k)\right), \ldots, a_{\alpha_{c_m}} : \alpha_{c_m}\left(\pi_{a_m}(I_k)\right), a_{\alpha_{B_1}} : \alpha_{B_1}\left(\pi_{b_1}(I_k)\right), \ldots, a_{\alpha_{B_p}} : \alpha_{B_p}\left(\pi_{b_p}(I_k)\right)\right\rangle, d \in set(\{\{\pi_G(i) \mid i \in I_k\}\}), I_k \in I, \\ \left\{\left\{\left\langle i, I_k, \bigcup_{g \in G} g^i \cup \bigcup_{a \in A_c} a^i \cup \bigcup_{b \in A_B} b^i \right\rangle \mid i \in I_k, I_k \in I\right\}\right\}, \\ \left\{\left\langle g^i, g^r\right\rangle \mid g \in G, i \in I_k\right\} \cup \left\{\left\langle a^i_k, \left(a_{\alpha_{c_k}}\right)^r\right\rangle \mid k = 1, ..., m, i \in I_k\right\} \cup \left\{\left\langle a^i_k, \left(a_{\alpha_{B_k}}\right)^r\right\rangle \mid k = 1, ..., p, i \in I_k\right\} \end{array}\right\rangle\right\}\right\}}$

Table 5: Provenance capture semantics partially based on operator semantics from [11]. Access $\mathcal{A}$ and manipulation $\mathcal{M}$ provenance is highlighted.

to the input item and a_{new} to the newly created attribute. This attribute holds item j that is unnested from i's attribute a_{col}, i.e., $i \in I_1, j \in i.a_{col}$. In our structural provenance, we need to refer to the position of j within its bag (or set) in the context of i. Therefore, we denote the position of j in $i.a_{col}$ by pos. For each result item r, the structural provenance is $\rho = \langle r, \mathcal{I}, \mathcal{M}\rangle$ with $\mathcal{I} = \left\{\left\{\left\langle i, I_1, \left\{(a_{col}[pos])^i\right\}\right\rangle\right\}\right\}$ and $\mathcal{M} = \left\{\left\langle (a_{col}[pos])^i, a^r_{new}\right\rangle\right\}$. Here, $(a_{col}[pos])^i$ denotes the access path on the pos-th element of attribute a_{col} in the context of the input item i. a^r_{new} is the path to the new attribute in the context of the result item r.

5.0.3 Aggregation. The *aggregation* in Tab. 5 requires a bag of equally structured input collections (we only show bags for conciseness) as input, i.e., $\tau(I) = \{\{\tau(I_1), \ldots, \tau(I_n)\}\}$ such that $\tau(I_1) == \ldots == \tau(I_n) == \{\{\ldots\}\}$. These nested bags are constructed by the *grouping*. Further, the *aggregation* supports multiple aggregation functions. Among those supported by data analytics systems, we distinguish between aggregation functions that, given a bag as input, return an atomic constant value c (e.g., *count*, *sum*, *max*) and aggregation functions returning nested collections (e.g., *collect_list* and *collect_set*). We denote these by A_c and A_B, respectively. The rule also requires that all attributes G that are not aggregated by either a function in A_c or A_B, but that are present in the schema of a collection $I_k \in I$ are equal.

Given these preconditions, *aggregation* reduces each of the nested bags $I_k \in I$ to a single data item. It returns the unique value present in I_k for non-aggregated attributes in G and the results of the specified aggregate functions that are applied on the input attributes. The result of this process is the item r in the first line at the bottom of the *aggregation* rule. The second line shows $\mathcal{I}$. A result item r is based on input items that all originate from the same input collection $I_k \in I$. Thus, for each $i \in I_k$, the rule creates a data item $\langle i, I, \mathcal{A}\rangle \in \mathcal{I}$. The set of attribute accesses performed during aggregation includes the paths to all attributes in G, the paths to all attributes aggregated by functions in A_c, and the paths to all attributes aggregated by functions in A_B. That is, $\mathcal{A} = \bigcup_{g \in G} g^i \cup \bigcup_{a \in A_c} a^i \cup \bigcup_{b \in A_B} b^i$. For $\mathcal{M}$ in line 3, the rule maps aggregated attributes to the newly created attributes of r, which hold the aggregated items.

5.1 Lightweight provenance capture

The provenance capture rules have the potential for optimization, since they hold redundant information. First, recording a unique identifier suffices to identify each top-level item. Second, recording the paths accessed and manipulated on a schema level once per operator suffices since the paths are the same for all processed data items. They only differ in the identifier of the top-level item and the positions of items in nested collections. The lightweight operator provenance $\mathcal{P}$ exploits these observations to keep overhead at capture time low.

Definition 5.1. (Operator provenance $\mathcal{P}$) The operator provenance $\mathcal{P}$ is the following 5-tuple:

$$\mathcal{P} = \langle oid, type, \mathcal{I} : \{\{\langle p, \mathcal{A}\rangle\}\}, \mathcal{M}, P\rangle$$

$\mathcal{P}$ has an operator identifier oid and a $type$. The bag $\mathcal{I}$ holds one tuple for each of the operator's inputs. This tuple holds a reference to the preceding operator p and the paths accessed $\mathcal{A}$ on the input at a schema level. They are data item independent. Similarly, $\mathcal{P}$ has a bag of manipulated paths $\mathcal{M}$ on a schema level. Positions of items in nested bags are replaced with placeholders. The bag P in $\mathcal{P}$ holds the unique identifiers of the top-level input

Operator	Provenance structure
$map, select, filter$	$P = \{\{\langle id^i, id^o \rangle\}\}$
$join, union$	$P = \{\{\langle id^i_1, id^i_2, id^o \rangle\}\}$
$flatten$	$P = \{\{\langle id^i, pos, id^o \rangle\}\}$
$groupby$ and $aggregation$	$P = \{\{\langle ids^i : \{\{id^i\}\}, id^o \rangle\}\}$

Table 6: Operator-dependent provenance structure

and output items, as well as positions of accessed or manipulated items in nested collections, if needed.

The content of P depends on the operator type as summarized in Tab. 6. In this table, the attributes id^i and id^o hold the unique identifiers of top-level items in the input and the output, respectively. If the operators have multiple inputs, attributes id^i_1 and id^i_2 are indexed in the order of appearance in $\mathcal{I}$. The structure P of the flatten operator has a reference to the *pos*ition of the nested item being flattened. The aggregation holds a collection of input ids^i for each group. The position of the input id^i is equal to the position of any nested item that the aggregation produces.

Example 5.2. Fig. 3 shows the reduced operator provenance $\mathcal{P}_5$ for the flatten operator at the bottom right.

In the following section, we describe how the backtracing algorithm computes the structural provenance of nested data from the lightweight provenance structures $\mathcal{P}$.

6 BACKTRACING

Querying the provenance of items or structures in the result involves two major phases. In the first phase, the backtracing algorithm identifies those data items, for which a user queries provenance (Sec. 6.1). In the second phase, it traces these items back to the input data (Sec. 6.3).

6.1 Structural query processing

DISC systems have rudimentary means to address individual nested items, at most. They lack sophisticated means to address arbitrary combinations of them, which is essential for querying structural provenance. Thus, we devise an extension to a DISC system (i.e., Apache Spark) to support tree-pattern queries. Tree-patterns allow for addressing combinations of nested items that are related by their structure [13]. Intuitively, they express structural queries in the form of a tree, in which each node represents an attribute and edges define parent-child or ancestor-descendant relationships (depending on edge type) that should exist between two connected nodes. Further constraints may be imposed on attribute nodes, e.g., equality of an attribute's value to a constant. Therefore, we define a novel distributed tree-pattern matching algorithm to return the query result in an efficient and scalable way. Due to space constraints, we omit details on processing tree-pattern queries but show an example.

Example 6.1. Fig. 4 shows a tree-pattern for the provenance question introduced in Sec. 2. Its *root* has an ancestor-descendant edge to the *id_str* node. All other edges indicate parent-child relationships. The *id_str* and *text* nodes hold equality conditions, which require values of those attributes to be equal to *lp* and *Hello World*, respectively. Further, as indicated by the black box, the value *Hello World* has to occur twice in the nested collection.

Our algorithm matches the tree-pattern against a dataset D (in our example, the final result of the processing pipeline) to then return the matching data in the form of a backtracing structure, which we introduce next.

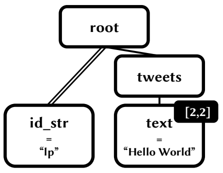


Figure 4: Example tree-pattern in a provenance question

6.2 Backtracing structure

The backtracing structure describes the items that are queried in the provenance question and traced back to the input. The backtracing algorithm updates its content while stepping backward.

Definition 6.2. (Backtracing structure) The backtracing structure $\mathcal{B} = \{\{\langle id, \mathcal{T} \rangle\}\}$ is a bag of provenance identifiers id of top-level data items associated with a backtracing tree $\mathcal{T}$ (see next definition), referencing attributes in the schema of id.

The nodes in the backtracing trees also hold information about the access and manipulation of the attribute and whether the attribute is contributing or just influencing the items queried.

Definition 6.3. (Backtracing tree) The backtracing tree $\mathcal{T} = \langle root, N \rangle$ holds complex nodes $n \in N$. Each node $n = \langle name, parent, C, A, M, c \rangle$ has a $name$ equal to the attribute name it references. Further, it references its parent node p, and its children C. A node also holds the set of operators A that access the referenced attribute and a set of operators M that manipulate the attribute. A boolean value c indicates whether the attribute contributes to the items in the provenance question ($c = true$) or whether it influences the items ($c = false$).

Example 6.4. Examples of backtracing trees are provided in Fig. 2. The right tree corresponds to the backtracing tree obtained from matching the tree-pattern in Ex. 6.1 on the data in Tab. 2. The left trees correspond to backtracing trees resulting from recursively updating the backtracing structure while stepping back through the processing pipeline to the input data.

The following two methods manipulate the trees, and the backtracing algorithm calls them repeatedly during backtracing. Their execution context is an instance of an operator provenance $\mathcal{P}$ and a backtracing structure $\mathcal{B}$.

The $manipulatePath$ method performs two tasks. First, it manipulates the nodes in $\mathcal{T}$. For each input and output path in $m \in \mathcal{P}.M$ it transforms the output path back to the specified input path in m, if the output path exists in $\mathcal{T}$. After the transformations, the nodes in the tree $\mathcal{T}$ conform to the schema of the input. Second, it adds the current operator identifier $\mathcal{P}.oid$ to each node's manipulation collection M.

The $accessPath$ method records access to attributes in the nodes of $\mathcal{T}$. During that process, one of two cases applies. In the first case, all nodes of the path $a \in \mathcal{A}, \mathcal{A} \in \mathcal{P}.\mathcal{I}$ already exist in $\mathcal{T}$. Then the method adds the $\mathcal{P}.oid$ to each node's access collection A. In the second case, nodes in path a do not exist in $\mathcal{T}$, because these attributes are neither needed to reproduce the result nor have been accessed by other operators so far. Then, the $accessPath$ method adds the according nodes to $\mathcal{T}$ but sets the contribution value to $c = false$ since these nodes are not required to reproduce the queried data items.

6.3 Backtracing algorithm

Alg. 1 shows the backtracing algorithm that traces the queried items recursively back from the result to the input. It takes the

Algorithm 1: backtrace($\mathcal{P}, \mathcal{B}$)

```
   Input: $\mathcal{P}$, $\mathcal{B}$
   Output: $\mathcal{B}$
1  switch $\mathcal{P}.type$ do
2      case "$filter$" do
3          $\langle \mathcal{P}', \mathcal{B}' \rangle \leftarrow$ backtraceFilter($\mathcal{P}, \mathcal{B}$)
4      case "$flatten$" do
5          $\langle \mathcal{P}', \mathcal{B}' \rangle \leftarrow$ backtraceFlatten($\mathcal{P}, \mathcal{B}$)
6      ...
7  if $\mathcal{P}'$ is defined then
8      backtrace($\langle \mathcal{P}', \mathcal{B}' \rangle$)
9  return $\mathcal{B}'$
```

Algorithm 2: backtraceFlatten($\mathcal{P}, \mathcal{B}$)

```
   Input: $\mathcal{P}$, $\mathcal{B}$
   Output: $\langle \mathcal{P}', \mathcal{B}' \rangle$
1  $\langle \mathcal{P}', \mathcal{B}' \rangle \leftarrow backtraceOperatorGeneric(\mathcal{P}, \mathcal{B})$
2  $\mathcal{B}' \leftarrow \alpha_{mergeTrees(\mathcal{T},pos)}(\gamma_{id}(\mathcal{B}'))$
3  return $\langle \mathcal{P}', \mathcal{B}' \rangle$
```

Algorithm 3: backtraceOperatorGeneric($\mathcal{P}, \mathcal{B}$)

```
   Input: $\mathcal{P}$, $\mathcal{B}$
   Output: $\langle \mathcal{P}', \mathcal{B}' \rangle$
1  $\mathcal{B}' \leftarrow \pi_{id^i \rightarrow id, pos, \mathcal{T}}(\mathcal{P}.P \bowtie_{id^o = id} \mathcal{B})$
2  for $t \in \mathcal{B}'.\mathcal{T}$ do
3      for $m \in \mathcal{P}.\mathcal{M}$ do
4          $manipulatePath(t, m, \mathcal{P}.oid)$
5      for $a \in \mathcal{P}.\mathcal{I}_1.\mathcal{A}$ do
6          $accessPath(t, a, \mathcal{P}.oid)$
7  return $\langle \mathcal{P}' \leftarrow \mathcal{P}.\mathcal{I}_1.p, \mathcal{B}' \rangle$
```

operator provenance of the last operator $\mathcal{P}$ in the pipeline and the backtracing structure $\mathcal{B}$ as input to call the operator-dependent backtracing method, which returns its predecessor's operator provenance $\mathcal{P}'$ and an updated backtracing structure $\mathcal{B}'$. The algorithm recursively calls the backtrace method until the input is reached. Then $\mathcal{P}'$ is not defined so that the recursion ends.

Flatten, Select, Filter, Map. As shown in Alg. 2, backtracing the flatten operator has two steps. In the first step (l. 1), the algorithm calls *backtraceOperatorGeneric* (Alg. 3) to undo the flatten on each item in $\mathcal{B}$ individually. At this step, the algorithm does not consider positions in the flattened collection. As a result, it obtains $\mathcal{P}'$ and $\mathcal{B}'$. In the second step (l. 2), the algorithm groups the trees and positions in $\mathcal{B}'$ by the top-level item *id* and merges all trees of the same *id*, considering the position *pos*.

The generic backtracing algorithm, shown in Alg. 3, also has two major steps. In the first step (l. 1), it joins B with the provenance associations P of $\mathcal{P}$ to obtain the input identifiers of the top-level items along with the trees in B (l. 1). These identifiers become the new *ids* in $\mathcal{B}'$ so that they match the id^o of the projection's predecessor $\mathcal{P}.\mathcal{I}_1.p, \mathcal{B}'$. This join is essentially the same one that existing lineage solutions [15, 17, 22] apply for backtracing. In the second step (ll. 2-6), the algorithm iterates over all the trees in $\mathcal{B}'$ to undo all recorded structural manipulations in the *manipulatePath* method and record the access to attributes in the *accessPath* method.

Example 6.5. The example input of Alg. 2 is $\mathcal{P}_5$ in Fig. 3 (bottom right) and a backtracing structure $\mathcal{B}$ with the two items of $id = 42$ and $id = 43$. They reference the items with the same identifier in Fig. 3. For simplicity, the example subtree $\mathcal{T}$ is reduced to the path *m_user.id_str*. The *backtraceFlatten* algorithm calls the *backtraceOperatorGeneric* algorithm (Alg. 2, l. 1), which joins P_5 with $\mathcal{B}$ (Alg. 3, l.1). Afterwards, both items are assigned $id = 1$. Then the algorithm modifies the trees to $user\_mentions.[pos].id_{str}$ so that they comply with the input schema of the flatten operator (Alg. 3, l.4). However, instead of

Algorithm 4: backtraceAggregation($\mathcal{P}, \mathcal{B}$)

```
    Input: $\mathcal{P}$, $\mathcal{B}$
    Output: $\langle \mathcal{P}', \mathcal{B}' \rangle$
1   $P^* \leftarrow pos\_flatten(\mathcal{P}.P.ids^i, id^i, p_P)$
2   $\mathcal{B}' \leftarrow P^* \bowtie_{id^o = id} \mathcal{B}$
3   $\mathcal{B}' \leftarrow withCol(\mathcal{B}', inProv, false)$
4   for $b \in \mathcal{B}'$ do
5       for $m \in \mathcal{P}.\mathcal{M}$ do
6           if $contains(m.out, [pos])$ then
7               $out \leftarrow replace(m.out, b.p_P)$
8           else
9               $out \leftarrow m.out$
10          if $out \in b.\mathcal{T}$ then
11              $b.inProv = true$
12              $manipulatePath(b.\mathcal{T}, \langle m.in, out \rangle, \mathcal{P}.oid)$
13          $removeNodes(b.\mathcal{T}, m.out)$
14  for $t \in \mathcal{B}'.\mathcal{T}$ do
15      for $a \in \mathcal{I}.A$ do
16          $accessPath(t, a, \mathcal{P}.oid)$
17  $\mathcal{B}' \leftarrow \pi_{id^i \rightarrow id, \mathcal{T}}(\sigma_{inProv = true}(\mathcal{B}'))$
18  return $\langle \mathcal{P}' \leftarrow \mathcal{I}.p, \mathcal{B}' \rangle$
```

holding the position of the nested items, they hold [*pos*] placeholders. The *mergeTrees* method in Alg. 2 (l. 2) replaces them with $pos = 1$ and $pos = 2$ for the items with the former $id = 42$ and $id = 43$, respectively. Further, it merges their trees because both items have $id = 1$ and, thus, are grouped together. Finally, the algorithm returns a $\mathcal{B}'$ with the item of $id = 1$ and the tree referring to positions 1 and 2 in the nested collection *user_mentions*.

The algorithms to backtrace a select, filter, or map operator are basically the same as Alg. 3, except that they do not project on the *pos* attribute (l. 1). Some optimizations are applicable to the filter. Since the filter does not manipulate any data, its backtracing algorithm does not loop over the manipulations $\mathcal{P}.\mathcal{M}$. The backtracing algorithm for the map operator has no information on the paths manipulated or accessed. Thus, it marks all nodes in the input schema as manipulated by default.

Aggregation and Nesting. As described in Sec. 5, aggregation and nesting are preceded by a grouping. Further, our model allows multiple aggregations and nestings over different attributes. Alg. 4 describes the procedure to trace aggregation and nesting back to the input of the preceding grouping.

Unlike the provenance structures of other operators, $\mathcal{P}.P$ of an aggregation holds a nested collection of input ids^i (cf. Tab. 5.1). Thus, Alg. 4 first flattens the ids^i and their positions into the columns id^i and p_P, respectively (l. 1). After joining P^* with $\mathcal{B}$ to $\mathcal{B}'$ (l.2), the algorithm adds a column *inProv* to $\mathcal{B}'$ (l. 3). This column is initialized with $false$ and used later to indicate, whether items in $\mathcal{B}'$ remain in the backtraced provenance. Then, the algorithm iterates over each item in $\mathcal{B}'$ and each manipulated path in $\mathcal{P}.\mathcal{M}$ (ll. 4-13). For each manipulated path $m.out$, it checks for a position placeholder [*pos*] (l.6), which only occurs when the operator performs bag nesting. In this case, the input item with id^i contributes exactly to the item in the nested bag that also has position p_P. Thus, the algorithm replaces the placeholder in $m.out$ and stores the result in *out* (l. 7). Otherwise, *out* is assigned $m.out$ (l. 9). If the exact path *out* is in the provenance tree, item *b* is marked as relevant and the path is adjusted accordingly (ll.10-12). In case of a bag nesting, the provenance tree may also hold information of items at other positions. The algorithm removes these nodes calling the *removeNodes* method (l. 13). It marks the accessed paths, to which the grouping attributes also belong (ll.14-16). In a final step, the algorithm removes all items and attributes from $\mathcal{B}'$ that are irrelevant for further backtracing. Their value in *inProv* was not set to true.

Example 6.6. Let us apply Alg. 4 on the nesting operator in our running example, which collects all tweeted texts in a nested bag. The backtracing structure $\mathcal{B}$ contains just the item with $id = 102$ and the right tree $\mathcal{T}$ of Fig. 2, which refers to the duplicate text *Hello World*. The operator Provenance $\mathcal{P}$ contains a provenance structure P that Alg. 4 flattens out to the following P^*:

id^i	p_P	id^o
81	1	102
82	2	102
93	3	102
95	4	102

After joining $\mathcal{B}$ with P^*, each entry with $id^o = 102$ holds a copy of the same tree $\mathcal{T}$. For a single loop iteration over $\mathcal{B}'$ and $\mathcal{M}'$ (ll. 4-13), we choose $b \in \mathcal{B}'$ with $id^i = 82$, $p_P = 2$, $id^o = 102$ and path $m.out = tweets.[pos]$. The algorithm replaces placeholder $[pos]$ with 2, so that $out = tweets.2$ (l. 7). Since out is part of $\mathcal{T}$, it sets $b.inProv = true$ (l. 11) and transforms the subtree $tweets.2.text$ to the subtree $text$ (l. 12). Then, it removes the node $tweets$ and all its children from its copy of $\mathcal{T}$ (l. 13), which includes the nodes in path $tweets.3.text$. Now, the nodes in $\mathcal{T}$ describe a subset of the schema of the aggregation's input data. The algorithm marks the accessed attributes and removes all items that are not part of the provenance. Here, it marks the $user$ and its children as accessed (ll. 14-16), since these attributes are used for grouping. Further, it removes items from $\mathcal{B}'$ whose provenance is not queried (l. 17), e.g., b' with $id^i = 95$ and $p_P = 4$.

Join and Union. Unlike the other operators, the join and the union operator have two predecessors. Thus, the backtracing algorithms require an additional parameter to specify which of the two inputs is traced back to. Based on that parameter, the algorithms pick the appropriate input tuple from the operator provenance $\mathcal{P}.\mathcal{I}$ and the appropriate input identifiers id_1^i or id_2^i from the provenance structure $\mathcal{P}.P$ (cf. Tab. 4). Then they call the generic backtracing algorithm from Alg. 3. Afterwards, the algorithm for the join operator removes all nodes in the provenance trees $\mathcal{B}'.\mathcal{T}$ that are not part of the chosen input schema, since they reference elements in the schema of the other input. The algorithm for the union operator filters out all items in $\mathcal{B}'$ whose value is undefined in the chosen field id_1^i or id_2^i . These items originate from the other input of the union operator.

7 IMPLEMENTATION & EVALUATION

We integrate the contributions described in this paper into a system prototype named Pebble. Sec. 7.1 provides some details of the system implementation, demonstrated in [9]. Sec. 7.2 then describes our evaluation setup and workload, which we use for our experimental evaluation (Sec. 7.3).

7.1 Implementation

While our contributions are generally applicable to DISC systems, we implement Pebble as a library extension for Apache Spark. This allows us to better compare it to Titian, which is the only other fully integrated provenance solution for DISC systems that has been implemented over Spark [16].

Fig. 5 shows Pebble's architecture (blue) on top of the *SparkSQL* API (grey), which is independent of the *Spark Core* module and further modules (grey) such as the MLlib. To provide a transparent user experience, Pebble has an API wrapper *PebbleAPI*. It directs user requests to the *SparkSQL* module or the *Pebble Core* module, which contains the *Capture* and *Query* submodules.

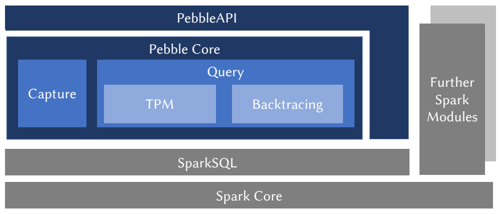


Figure 5: Pebble's architecture

S	Description (detailed descriptions available in [8])
T1	filters tweets containing the text *good*, flattens and groups by the mentioned users to collect a bag of complex tweet objects
T2	flattens the nested lists *hashtags, media, user mentions*
T3	running example
T4	associates all occurring hashtags with the authoring and mentioned users
T5	finds all users that tweet about *BTS*, and are mentioned in a *BTS* tweet
D1	associates inproceedings from 2015 with the their according proceeding(s)
D2	unites and restructures conference proceedings and articles
D3	computes nested list for aliase, co-authors, and works per author
D4	computes nested list of all associated inproceedings for each proceeding
D5	is D4 extended with a UDF in map that returns the number of authors per proceeding

Table 7: Short informal scenario descriptions

The former submodule extends Spark's dataframes and operators to capture the structural provenance as described in Sec. 5. The latter submodule implements the backtracing algorithm from Sec. 6. It utilizes maps to represent the provenance trees $\mathcal{T}$ and modifies the tree in place with user-defined functions. Each of the Algs. 2-4 iterate over all items in the backtracing structure $\mathcal{B}$ and perform changes impacting only one item at a time. Thus, the for-loops with iterator variables t or b in Algs. 2-4 are parallelized across the DISC system. However, the backtracing needs to be called for each input dataframe independently, because Spark operators always generate just one result dataframe.

7.2 Test setup & workload

For our experimental validation, we run Pebble on a cluster with three worker nodes, each having 8 cores, 256GB main memory, and SSD storage. All nodes run Scala 2.11, Hadoop 3.1.0 and Spark 2.3.1. We average five test runs framed in an additional warm-up and cool-down run. The error bars displayed in our graphs show the standard deviation. We write the result to disk to ensure that Spark computes the full result. Otherwise, Spark "optimizes" attributes away. The experiments run on 100GB input data, if not mentioned otherwise.

We base our evaluation on a nested Twitter and a DBLP dataset. We scale the datasets from 100GB to 500GB in steps of 100GB. For each of the datasets, we define five scenarios containing a Spark program to be executed with and without provenance capture and a corresponding structural query. Each supported operator occurs at least once in the scenarios. The Twitter dataset contains up to 130 million tweets (500GB). Each tweet has up to 1000 attributes and eight layers of nesting [27]. We define five test scenarios T1 - T5 (Tab. 7). The DBLP dataset contains up to 1.5 billion records (500GB) that are extracted from the dblp.xml. Records have one of ten types such as article or proceeding [21]. They are split by type and upscaled, such that important characteristics such as the average number of inproceedings per proceeding are preserved. We define five test scenarios D1 - D5 (Tab. 7).

7.3 Experimental evaluation

We conduct experiments to study the runtime and space overhead when capturing lightweight structural provenance. We also evaluate the performance of querying the structural provenance. Further, we perform a comparative evaluation with Titian [17],

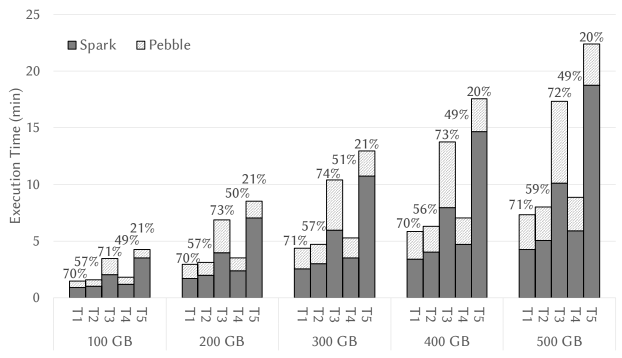


Figure 6: Runtime overhead on Twitter dataset

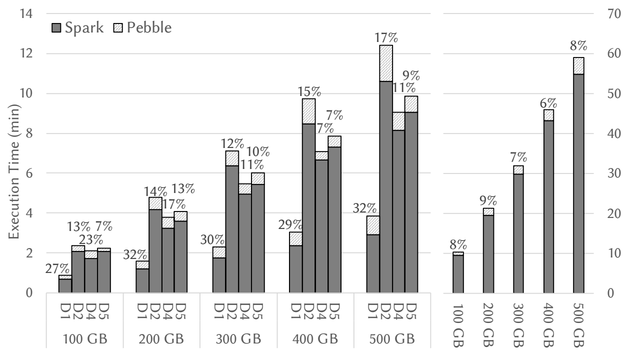


Figure 7: Runtime overhead on DBLP dataset (right: D3)

kindly provided to us by the authors, and fully lazy provenance capture as described by PROVision [28]. We conclude our evaluation with a use-case analysis for auditing and data-usage patterns.

7.3.1 Capture runtime overhead. The first series of measurements shows the runtime overhead imposed by the lightweight provenance capture for increasing data sizes. Our goal is to show (i) how Pebble scales over the input data size and (ii) how data size affects the runtime overhead.

We measure the execution time for each of the Twitter scenarios T1 to T5, once without provenance capture, i.e., with Spark's regular operator semantics, and once with Pebble's provenance capture as defined in Sec. 5.1. Fig. 6 shows the results on datasets from 100GB to 500GB. The solid dark grey part of each part shows the runtime that Spark requires without provenance collection. The textured light grey part on top of each bar shows the overhead when running provenance capture. The percentage on top of the textured bars indicate the relative overhead between the former and the latter types of experiments. Analogously, Fig. 7 reports runtimes for scenarios D1 to D5.

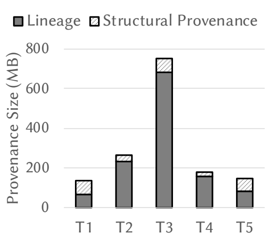
As expected, across all experiments, the runtime increases when provenance is collected since Pebble performs extra work. Runtime with and without provenance grows linearly with the data size. As the overhead percentages indicate, the relative overhead imposed by provenance capture remains constant with increasing data sizes for most scenarios. Thus, we conclude that Pebble scales with the input data size. However, the relative overhead varies significantly between the scenarios. It ranges from 75% (T3) down to 8% (D3, shown on the right of Fig. 7). A detailed analysis of D3 reveals that spilling large final and intermediate results to disk – or more generally speaking disk I/O – dominates the runtime. The time to compute the extra provenance is small. In contrast, scenario T3 reads the input tweets twice to perform a union operation. As a consequence, Pebble annotates the input data twice during provenance capture, hindering Spark to optimize reading the input.

We further investigate overhead incurred by Pebble for each individual operator (no graphs shown due to space constraints). Overall, the overhead highly depends on the size ratio between the collected provenance and the processed input data. In general, for operators with constant provenance annotation overhead (filter, select, union, join, and flatten), the relative overhead decreases with an increasing number of attributes in the input data. In the case of the DBLP dataset, which has less than 50 attributes, the overhead ranges between 5% and 25% for the mentioned operators. The overhead is particularly high for aggregations that reduce many input items to a single value. Then, Pebble stores a collection with all item identifiers contributing to the aggregated item. This collection typically is orders of magnitude larger than the result item itself. Consequently, the observed runtime overhead exceeds 100% of the actual execution time for the aggregation. However, even this overhead is negligible when disk I/O operations dominate the execution time.

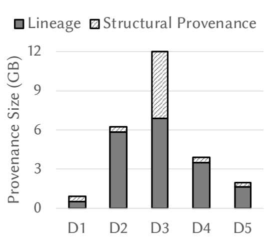
7.3.2 Capture space overhead. The second series of measurements shows the space required to store captured provenance. We show (i) that the provenance size depends on dataset and scenario characteristics, (ii) that large provenance sizes do not necessarily correlate with high runtime overhead, and (iii) that the captured structural provenance typically adds an overhead of less than 200MB compared to lineage. Thus, it typically does not significantly affect scalability. Results are reported in Fig. 8(a) and Fig. 8(b). The dark grey part of each bar shows the size of lineage for top-level items and the stacked and textured bars show the additional space required by structural provenance.

The y-axis of the Twitter graph has a Megabyte scale, whereas the y-axis of the DBLP graph has a Gigabyte scale. The reason is that the items in the Twitter dataset have about 1000 attributes, whereas the items in the DBLP dataset have less than 50 attributes. Therefore, 100GB of DBLP data contain more than 100 times as many data items as 100GB of Twitter data. Given that Pebble associates identifiers to top-level data items only, it stores more than 100 times the annotations for DBLP scenarios compared to the Twitter scenarios. Hence, the DBLP provenance is orders of magnitude larger than the Twitter provenance and lets us conclude that the size of the provenance significantly depends on the number of tracked top-level data items in the input.

Further, the sizes significantly differ among the scenarios of the same dataset. For instance, the provenance of scenario T3 amounts to 750MB, 5.5 times the size of T1's provenance. There are three reasons for the different size: (i) As mentioned above, in T3, our solution annotates the input data twice; (ii) The processing pipeline of T3 consists of 7 processing steps that trigger provenance collection, whereas the pipeline of T1 only consists of 5 steps; (iii) The filter in T1 reduces the total amount of tracked

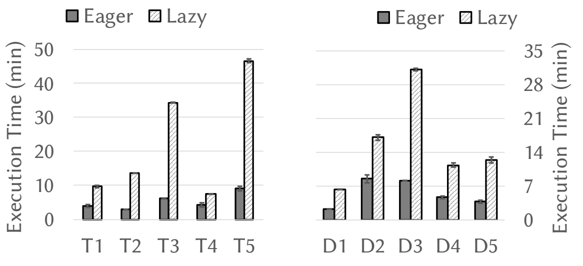


(a) Twitter dataset



(b) DBLP dataset

Figure 8: Size of collected structural provenance



(a) Twitter dataset (b) DBLP dataset

Figure 9: Runtime of Pebble's backtracing

data items early in the pipeline. Clearly, space overhead also depends on the number of operators in a program and the number of top-level items in intermediate results.

The scenarios T3 and T1 have comparable runtime overhead (see Fig. 6(a)) of around 70 - 75% and, thus, highest across all scenarios. While scenario T3 is also the scenario with the largest provenance size, T1 has a comparable runtime overhead but a much smaller provenance size. Similarly, the relationship of the runtime and space overhead is actually the inverse for scenario D3 and D1 in Fig. 8(b). D3 has the largest and D1 the smallest provenance size. However, D1 has a runtime overhead of 27%, whereas D3 only has 7%. Therefore, it is not generally true that a high runtime overhead correlates with a high space overhead.

Looking at the space overhead of lineage and structural provenance, we see that in most scenarios, structural provenance takes less than 200MB additional space, even in scenarios where lineage itself takes Gigabytes. The only exception is D3, where a flatten occurring early in the pipeline followed by a very selective join causes the comparatively high size difference.

Concerning the above experiments, which are all related to provenance capture, we make the following general observations. The provenance size highly depends on the number of top-level items in the input and intermediate results. As explained, the provenance size may not correlate with the runtime overhead. Other factors, such as processing optimizations, data width, or significant disk I/O potentially have a higher impact on the relative runtime overhead than the provenance size. While the size difference between lineage and structural provenance is small in many practical scenarios, the overhead can increase when flatten operators store positions that lineage solutions do not capture.

7.3.3 Querying structural provenance. Our third series of experiments focuses on processing structural queries over provenance-annotated result datasets. The runtimes reported in Fig. 9 include both the tree-pattern matching on the program's result items and the time needed to backtrace these result items to the input with the help of structural provenance. We report results on query processing time in our holistic approach (i.e., where structural provenance has been eagerly captured and is traced back). We also implement a fully lazy query approach that can be considered an extension of PROVision [28] to our processing pipelines. The query runtime for these two approaches, labeled eager and lazy respectively, are shown for both the Twitter scenarios (Fig. 9(a)) and the DBLP scenarios (Fig. 9(b)).

The graphs in Fig. 9 do not explicitly show the time for tree-pattern matching since the matching is integrated into Spark's processing pipeline. It becomes part of Spark's execution plan and undergoes optimizations such as filter push down. Therefore, time cannot be measured independently in a reliable way.

The dark bars in Fig. 9 show that querying structural provenance (eagerly) takes more time than the actual program execution (cf. Fig. 6 and 7). We identify two reasons for this behavior: (i) the backtracing presented in Sec. 6 performs a join operation for each operator in the actual program, even for computationally less expensive operators such as filters and selects. (ii) Backtracing has to manipulate the provenance trees for each operator.

When comparing the performance of our holistic capture and query approach with a completely lazy query approach such as PROVision [28], we see that our holistic provenance querying approach (eager) is always faster than the lazy approach. In the scenarios T3, T5, and D3 the difference amounts to a factor four to seven for two reasons. First, lazy processing needs to trace back result items for each input dataset independently and these scenarios have multiple input datasets. Hence, the extra time to query provenance lazily add up for each input. Second, the processing pipelines in these scenarios are deep, yielding high provenance query times for each input dataset.

Based on the above experiments, we draw the following conclusions for provenance querying. Lazily querying structural provenance is less attractive the more operators a program has and the more input datasets it processes. It is less time consuming to rerun a program with provenance capture and query the provenance eagerly than using lazy provenance querying approaches such as [28].

7.3.4 Comparison with Titian. We compare Pebble to Titian [17] since it is the only other provenance system integrated into a DISC system. The purpose of the evaluation is to compare the runtime overheads for capturing provenance of flat data items. A detailed comparison is not possible since Titian neither supports nested data, nor structural provenance, nor the programs in our scenarios. We run the test on a local machine with two worker nodes, using the unscaled articles and inproceedings records of the DBLP dataset. The test program reads each record as a long string value and filters lines containing *2015*. Then, the program computes the union over the filtered articles and inproceedings. Titian's program is implemented in the RDD API. Pebble's program is implemented in the SparkSQL API. Without provenance computation, the programs have an average runtime of 7.13 seconds and 7.36 seconds, respectively. The overall execution time is lower for the RDD program since the SparkSQL API imposes overhead on top of the underlying RDD API. Titian's overhead is 5.89%, Pebble's overhead is 6.98%.

The result indicates that for workloads on flat data supported by both systems, Pebble only adds marginal runtime overhead compared to Titian, even though it is capturing structural provenance. However, Pebble outperforms Titian in the sense that it additionally supports nested data and the collection and querying of structural provenance at attribute level.

7.3.5 Use-case analysis. To validate that structural provenance supports the use-cases described in our motivation, we revisit these use-cases with a prototypical implementation to analyze how they benefit from structural provenance.

Data-usage patterns. Pebble reveals data-access patterns, as well as hot items that frequently contribute to a query result and cold data items that do not influence any result. In Fig. 10, we show a heatmap of 25 randomly selected data items from the DBLP inproceedings dataset after running test scenarios D1 through D5. For that purpose, we merge the provenance of the individual scenarios. The more often a data item is used the redder (hot) it is. Items that do not influence any result are colored

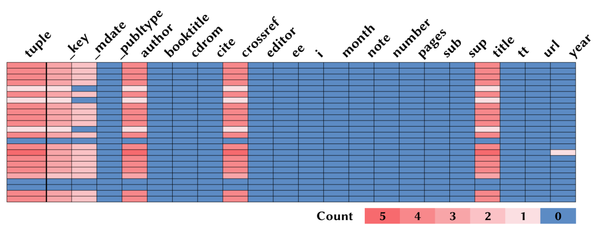


Figure 10: HeatMap for 25 data items in the DBLP inproceedings dataset after running scenarios D1-D5

blue (cold). The leftmost column indicates how often the top-level item (tuple) contributed to a result. The other columns refer to attributes of the top-level data items. The heatmap only shows top-level attributes due to space constraints. All but three top-level items have influenced at least one result. A horizontal (tuple-based) partitioning of hot and cold items, therefore, may not significantly improve system performance. However, only a fraction of all attributes contributes to the results. Thus, in this example, a vertical (column-based) partitioning of hot and cold attributes is likely to improve system performance significantly. Further, the analysis of accessed and manipulated nodes in the structural provenance reveals that the attributes *author* and *title* are frequently processed together. Thus, system performance benefits from storing these items next to each other.

In comparison, lineage solutions and PROVision [28] only provide the tuple based counter. Lipstick [2] also identifies attributes. However, Lipstick does not reveal information on access and manipulation and, thus, misses influencing attributes.

Auditing. Pebble identifies sensitive data that has been leaked directly or indirectly over the DBLP scenarios D1 through D5. All data in Fig. 10 are leaked whose count is bigger than zero. Data with count zero (blue) is not leaked. Since Pebble distinguishes access and manipulation of items, it further reveals the usage of the *year* item whose count equals one. It is marked as influencing since it does not contribute to any result item in D1 to D5. However, knowing that the *year* item is accessed is important to assess the risk of reconstruction attacks.

In comparison, lineage solutions and PROVision [28] only provide full tuples. Thus, they mark too much data as leaked. This is costly for a company, e.g., if a non-leaked (blue) attribute holds credit card numbers. Then, the company has to issue new credit cards to all marked customers, even though the information is not leaked. Lipstick [2] potentially misses leaked information, since it misses influencing attributes like the *year*. Thus, neither of the mentioned solutions allows for proper risk assessment.

8 CONCLUSION AND OUTLOOK

This paper introduced structural provenance, for which we provided a formal data model and execution semantics for operators frequently used in DISC systems. Further, we showed how to capture the structural provenance in an efficient and scalable way. Based on the captured provenance, we formalized an algorithm to backtrace structural provenance at attribute level for nested data at provenance query time. Our experimental evaluation using the Pebble system showed that our contributions result in the first DISC system integrated provenance solution for nested data that is efficient, scalable, and accurate enough to support novel provenance use-cases, such as auditing and data-usage patterns.

Future work includes extending Pebble with a user-friendly front-end to interact with structural provenance. We also intend to optimize provenance querying.

ACKNOWLEDGMENTS

Partially funded by Deutsche Forschungsgemeinschaft (DFG) under Germany's Excellence Strategy - EXC 2075 - 390740016.

REFERENCES

[1] U. Acar, P. Buneman, J. Cheney, J. Van Den Bussche, N. Kwasnikowska, and S. Vansummeren. 2010. A Graph Model of Data and Workflow Provenance. In *TaPP.*
[2] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. 2011. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *PVLDB* 5, 4 (2011).
[3] Y. Amsterdamer, D. Deutch, and V. Tannen. 2011. Provenance for aggregate queries. In *PODS.*
[4] P. Buneman, S. Khanna, and W.-C. Tan. 2001. Why and Where: A Characterization of Data Provenance. In *ICDT.*
[5] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. 1995. Principles of programming with complex objects and collection types. *Theoretical Computer Science* 149, 1 (1995).
[6] J. Cheney, A. Ahmed, and U. Acar. 2014. Database Queries That Explain Their Work. In *PDPP.*
[7] J. Cheney, L. Chiticariu, and W.-C. Tan. 2007. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2007).
[8] R. Diestelkämper. 2019. Evaluation Workload. https://www.ipvs.uni-stuttgart.de/abteilungen/de/abteilung/mitarbeiter/Ralf.Diestelkaemper_infos/resources/workload.pdf.
[9] R. Diestelkämper and M. Herschel. 2019. Capturing and Querying Structural Provenance in Spark with Pebble (Demo). In *SIGMOD.*
[10] J. N. Foster, T. J. Green, and V. Tannen. 2008. Annotated XML: Queries and Provenance. In *PODS.*
[11] M. Grabowski, J. Hidders, and J. Sroka. 2013. Representing MapReduce optimisation in the Nested Relational Calculus. In *BNCOD.*
[12] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein, and M. Kim. 2016. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. In *ICSE.*
[13] M. Hachicha and J. Darmont. 2013. A Survey of XML Tree Patterns. *TKDE* 25, 1 (2013).
[14] M. Herschel, R. Diestelkämper, and H. Ben Lahmar. 2017. A survey on provenance: What for? What form? What from? *VLDB J.* 26, 6 (2017).
[15] R. Ikeda, H. Park, and J. Widom. 2011. Provenance for Generalized Map and Reduce Workflows. In *CIDR.*
[16] M. Interlandi, A. E., K. Shah, M. A. Gulzar, S. D. Tetali, M. Kim, T. D. Millstein, and T. Condie. 2018. Adding data provenance support to Apache Spark. *VLDB J.* 27, 5 (2018).
[17] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie. 2015. Titian: data provenance support in Spark. *PVLDB* 9, 3 (2015).
[18] G. Karvounarakis and T. J. Green. 2012. Semiring-Annotated Data: Queries and Provenance. *SIGMOD Rec.* 41, 3 (2012).
[19] R. Kaushik, Y. Fu, and R. Ramamurthy. 2013. On Scaling Up Sensitive Data Auditing. *PVLDB* 6, 5 (2013).
[20] N. Kwasnikowska and J. Van den Bussche. 2008. Mapping the NRC Dataflow Model to the Open Provenance Model. In *IPAW.*
[21] M. Ley. 2009. DBLP: Some Lessons Learned. *PVLDB* 2, 2 (2009).
[22] D. Logothetis, S. De, and K. Yocum. 2013. Scalable lineage capture for debugging DISC analytics. In *SoCC.*
[23] J. Lu, T. W. Ling, Z. Bao, and C. Wang. 2011. Extended XML Tree Pattern Matching: Theories and Algorithms. *TKDE* 23, 3 (2011).
[24] J. Robie, J. Spiegel, and M. Dyck. 2017. *XML Path Language (XPath) 3.1.* W3C Recommendation. W3C. https://www.w3.org/TR/2017/REC-xpath-31-20170321/.
[25] R. Stoica, J. J. Levandoski, and P.-A. Larson. 2013. Identifying Hot and Cold Data in Main-memory Databases. In *ICDE.*
[26] The European Parliament and the Council of the European Union. 2016. Regulation (EU) no 2016/679 (General Data Protection Regulation).
[27] Z. Wang and S. Chen. 2017. Exploiting Common Patterns for Tree-Structured Data. In *SIGMOD.*
[28] N. Zheng, A. Alawini, and Z. G. Ives. 2019. Fine-Grained Provenance for Matching and ETL. In *ICDE.*

A Parallel and Distributed Approach for Diversified Top-k Best Region Search

Hamid Shahrivari
Eindhoven University of Technology
h.shahrivari.joghan@tue.nl

Matthaios Olma
EPFL
matthaios.olma@epfl.ch

Odysseas Papapetrou
Eindhoven University of Technology
o.papapetrou@tue.nl

Dimitrios Skoutas
Athena R.C.
dskoutas@athenarc.gr

Anastasia Ailamaki
EPFL
anastasia.ailamaki@epfl.ch

ABSTRACT

Given a set of points, the *Best Region Search* problem finds the optimal location of a rectangle of a specified size such that the value of a user-defined scoring function over its enclosed points is maximized. A recently proposed top-k algorithm for this problem returns results progressively, while also incorporating additional constraints, such as taking into consideration the overlap between the set of selected top-k rectangles. However, the algorithm is designed for a centralized setting and does not scale to very large datasets. In this paper, we overcome this limitation by enabling parallel and distributed computation of the results. We first propose a strategy that employs multiple rounds to progressively collect partial top-k results from each node in the cluster, while a coordinator handles the aggregation of the global top-k list, dealing with overlapping results. We then devise a single-round strategy, where the algorithm executed by each node is enhanced with additional conditions that anticipate potential overlapping solutions from neighboring nodes. Additional optimizations are proposed to further increase performance. Our experiments on real-world datasets indicate that our proposed algorithms are efficient and scale to millions of points.

1 INTRODUCTION

The amount of geospatial data generated from social networks, sensors, smart phone applications, tracking devices and so on is constantly increasing [9]. Analyzing big geospatial data at scale is of paramount importance for numerous applications in various areas such as geomarketing, mobile advertisement, urban planning, tourism and logistics. In many cases, the analysis involves identifying areas where the intensity of a studied phenomenon is maximized. This involves, for example, finding *hot spots* of user check-ins, commercial activities, crime incidents, etc. Various traditional methods exist and have been used for such purposes, such as computing global and local spatial autocorrelation [2] or finding density-based clusters [10], while several recent studies have also focused on problems related to finding areas of interest according to certain keyword-based and size-based criteria and constraints [3, 4, 7, 8, 15].

In this context, several types of *optimal location selection* problems have been studied. Range aggregate queries [18] have been proposed for scenarios where users are interested in summarized information about objects in a given region. Such queries return an aggregate score over the objects enclosed within a given region, and can be efficiently processed using aggregate spatial indices. However, these methods can not be efficiently applied when the problem is to find where the best regions are located. A typical and widely studied formulation of this problem is the *MaxRS* (Maximizing Range Sum) problem [6], which, given a set of 2-dimensional weighted points, aims at finding the optimal location of a fixed-size rectangular region that maximizes the sum of weights of the points enclosed in it.

In this paper, we focus on the *Best Region Search* (BRS) problem [11], which is a generalization of the *MaxRS* problem. Similarly, the input is a dataset $\mathcal{D}$ of spatial objects represented as weighted points in a 2-dimensional space, and two parameters $a, b \in \mathbb{R}^+$, denoting the width and height of the region to be found. The goal is to identify the optimal location of an axis-aligned $a \times b$ rectangular region R that maximizes the value of a scoring function f computed over the enclosed points. Hence, the difference lies in the fact that in *MaxRS*, f is restricted to the sum of weights of the points within R, while in *BRS* it can be any submodular monotone function. The *MaxRS* problem and its extensions and variants have their roots in problems studied in the past by the computational geometry community [14, 16], and have received much attention recently by the database community [1, 4, 6, 11, 12, 20, 21].

The *BRS* problem has numerous applications related to location planning. For instance, assume a company that is searching for the optimal location to open a new store. In that case, an $a \times b$ region R can be used to approximate the area from which the new store is expected to draw its potential customers. Assuming that a suitable scoring function f is provided, which computes a utility score for each candidate region R based on its contents, the solution to the *BRS* problem indicates the optimal location for placing this new store. Similar examples can also be found, for instance, when recommending regions to travelers. Given the size of the region that a user is willing to explore, modeled by an $a \times b$ rectangle, and a function f quantifying the utility of a region with respect to the user's preferences (e.g., presence of museums, restaurants, shops, etc.), the *BRS* problem computes the best region to be recommended.

The k-*BRS* problem has been introduced in [19], presenting an algorithm that can compute top-k best regions *progressively*. While doing so, we have also observed that in many real-world datasets, where spatial objects are typically not uniformly distributed in space, these top-k results tend to highly overlap, even for very large values of k. This is expected, since by slightly shifting a region R horizontally and/or vertically it is possible to obtain a new region R' that covers almost the same area as R and thus achieves very similar utility score to it. Yet, results of such type offer essentially no new insight over the explored data. Hence, to tackle this problem, we have introduced an additional constraint that either completely prohibits overlaps among the

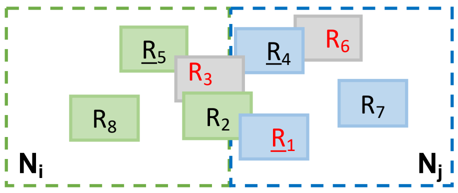


Figure 1: Illustrative example. The green and blue regions are those returned by N_i and N_j, respectively. The underlined results are those finally selected by the coordinator. The red ones are the correct top-3 results.

returned results or reduces the degree of overlap by allowing but penalizing partial overlaps among the returned top-k regions.

Being able to retrieve top-k results progressively, instead of simply computing the overall best region, as well as avoiding overlapping results that provide near-duplicate information, has many advantages in data exploration scenarios. Still, the state-of-the-art solution proposed in [19] operates in a centralized setting and cannot scale as the input dataset size grows; indicatively, it needs around 30 seconds to identify the top-10 non-overlapping regions of size 200×200 m^2 even on moderate-size datasets – around 100 thousand points.

To efficiently address the *k-BRS* problem when dealing with big geospatial data, an approach is needed that can scale out, i.e., distribute and parallelize the computation over multiple workers (computing nodes) in a cluster. In this paper, we address this problem by proposing different methods for parallel and distributed top-k best region search while taking into account the criterion of overlap among the returned results. In particular, we consider a setting where the input dataset is partitioned across several workers that perform local computations in parallel, while a coordinator node is assigned the task to coordinate the execution and merge local results to produce the global top-k result list. Most contemporary Big Data platforms, e.g., Spark, are naturally represented by the aforementioned setting.

Before we describe our approach, we explain the main challenge to this problem, which arises when non-overlapping (or only partially overlapping) results are requested. Specifically, the challenge arises from the fact that, under this constraint, the global top-k results are not contained in the union of the local top-k results computed at each node. Therefore, simply computing the local top-k results at each node and merging these top-k lists at the coordinator can lead to incorrect results in many cases. We illustrate this with the following example.

Example 1. *Consider the example illustrated in Fig. 1, assuming that the data is split in two partitions, each one assigned to a different worker, represented as N_i and N_j. Assume that the top-8 regions with highest scores in both partitions (regardless of overlaps) are those shown in the figure, with the index of each region indicating its rank (i.e., R_1 having the highest score). Suppose that each region belongs to (i.e., is computed by) the worker that contains its center.*

Assume that the user requests the top-3 non-overlapping results. Node N_i will progressively compute its results, and will return to the coordinator the regions R_2, R_5 and R_8. R_3 is skipped since it overlaps with R_2 which has a higher rank. Similarly, N_j will return to the coordinator the results R_1, R_4 and R_7. Again, R_6 is skipped as it overlaps with R_4. Based on these two sets of local top-3 results, the coordinator will compute the global top-3 list excluding overlapping regions, thus returning R_1, R_4 and R_5.

However, the above result is incorrect. The correct global top-3 results should have been R_1, R_3 and R_6. As we can see, R_3 is a false negative – it was missed because N_i skipped it, as it was overlapping with R_2; however, R_2 was eventually discarded by the coordinator because it was overlapping with a better result R_1 from the neighboring node. R_4 and R_5 are false positives – their respective nodes considered them as valid results; however they should have been discarded because eventually a better result R_3 that overlaps with both of them should have been present. R_6 is again a false negative, for reasons similar to R_3.

As revealed by the above example, the root cause of false results is the fact that the admission of a *local* top-k region in the *global* top-k list is precluded by the existence of a higher ranked region that overlaps with it. Since the latter may arise from a different worker than the former, a worker cannot independently reason regarding the validity of its own results, i.e., which of these results will eventually "survive" at the coordinator. To make matters worse, candidates that were rejected locally due to the existence of a better local overlapping result R, may actually have a place in the global top-k list if R is later dismissed at the coordinator.

Since overlapping regions between two neighboring workers occur at the borders, an intuitive – yet insufficient – approach to overcome this problem would be to replicate the contents of two neighboring nodes around the borders. In that case, in the example of Fig. 1, it could be possible for N_i to know, for instance, that R_1 overlaps with R_2 and has higher score than it. However, it may turn out that R_1 is disqualified in N_j, if it overlaps with an even better result not located in the border with N_i, in which case N_i would still have no knowledge of this. Even replicating the whole contents of N_j to N_i would not suffice, since the results of N_j may similarly be affected by its other neighbors as well.

In this paper, we explore different solutions to overcome this problem. First, we propose a *multi-round* algorithm (*MR*), where the problem of dealing with conflicts is mainly handled by the coordinator. The advantage of *MR* is that workers can readily exploit the local *k-BRS* algorithm with minor adaptations, since the decisions for forming the global top-k list are made at the coordinator level. The downside is that when overlapping results are identified, and hence discarded, at the coordinator, a new round has to be executed, where the relevant workers are informed about these results and are requested to compute new results accordingly. To overcome this drawback, we devise a *single-round* algorithm (*SR*). This requires formulating appropriate conditions to reason about the uncertainty of the validity of the local results. In *SR*, each worker proactively anticipates which results may be disqualified because of overlaps with regions from other nodes, and what the respective effects are in each case, and produces a *sufficiently extended* local set of results to guarantee a single round of communication. However, the computation of these extended results imposes additional overhead on each worker, implying potentially significantly higher execution time of the local processes. Eventually, we propose a hybrid algorithm (*HY*), which strikes a balance between the pros and cons of the multi-round and single-round strategies.

In the example described above, we assumed that the proposed regions must not overlap. Other conditions are also possible, e.g., the user might accept overlapping regions as long as their overlap is below a threshold, or even score the overlapping regions differently. Our methods support arbitrarily complex scoring functions, enabling different configurations concerning the overlap.

Summarizing, the paper makes the following contributions:

- We study the top-k best region search problem, identifying and pointing out the challenges that arise when attempting to compute non-overlapping (or partially overlapping) top-k results in a distributed and parallel setting.
- We explore the solution space of the problem, and propose two different distributed algorithms for efficiently solving it. The first is a multi-round algorithm, with the number of rounds bounded by the value of parameter k. The second is a single-round algorithm, which guarantees that the top-k best regions can be identified within a single round of interactions.
- The above two algorithms represent the two extremes in terms of number of rounds, and come with different performance tradeoffs. We thus proceed to explore the configuration space between them, leading to different optimizations, and to a hybrid, fixed-rounds algorithm that combines the benefits of the two.
- We implement all proposed algorithms in Spark and we thoroughly evaluate them using real-world datasets. Our evaluation results demonstrate the scalability and benefits of the proposed algorithms, and analyze their tradeoffs on queries and datasets of different properties.

The rest of the paper is structured as follows. Section 2 summarizes related work. Section 3 presents the problem definition and background. We introduce the multi-round and single-round algorithms in Sections 4 and 5, respectively. Section 6 presents our experimental evaluation, and Section 7 concludes the paper.

2 RELATED WORK

The *Best Region Search* problem has its roots in computational geometry problems relating to intersecting rectangular regions. An intersection graph of rectangles in the 2-dimensional space with sides parallel to the axes is defined in [14]. It is constructed by representing each rectangle with a vertex and connecting two vertices by an edge if the corresponding rectangles intersect. Then, finding the connected components and the maximum clique on this graph is investigated. Moreover, given a set of points in a 2-dimensional space, the problem of finding the placement of a rectangular region P of specified size such that it encloses the maximum or minimum number of points is investigated in [16]. The proposed algorithm is based on an interval tree.

More recently, the problem has attracted interest in spatial databases. Specifically, the *MaxRS* problem has been defined in [6] as follows: given a set of *weighted* points, and a rectangular region R of specified size, find the placement of R that maximizes the *sum of the weights* of all enclosed points. An external-memory algorithm is proposed that is optimal in terms of I/O complexity, based on an external version of the plane-sweep algorithm [13]. Furthermore, the (1-ϵ)-approximate *MaxRS* problem has been studied in [20], which returns a rectangle whose covered weight is at least(1-ϵ) m^*, where m^* is the optimal covered weight and ϵ is an arbitrarily small constant between 0 and 1.

The *MaxRS* problem has also been investigated in a streaming setting. In [1], an algorithm that exploits a graph in a grid index is proposed, using also an upper-bounding technique to avoid unnecessary update computation. A sweep-line based algorithm has also been proposed for the continuous detection of *Bursty Regions* [12]. This is a variation of the continuous *MaxRS* problem, where the burst score of a region is defined over two consecutive sliding windows, and spatial objects in different windows contribute differently to the burst score. Moreover, the continuous maintenance of *range-sum heat maps* over dynamically updating data objects has been studied in [17].

The *Best Region Search* problem generalizes the *MaxRS* problem by allowing the objective score function used to quantify a rectangle's score to be any submodular monotone function over the enclosed points, instead of the sum of their weights [11]. Each point is represented by a fixed-size rectangle centered at it. Then, the best region is identified by finding the maximal intersections of these rectangles. To this end, the input space is partitioned in vertical *slices* that run parallel to the y-axis. Each slice is processed by executing a bottom-up scan over it using a horizontal sweep line to identify so-called *maximal slabs.* The maximal slabs are then processed using a vertical sweep line to identify *maximal regions.* The best region is guaranteed to be centered inside one of those maximal regions. A progressive algorithm for top-k Best Region Search has been proposed in [19]. This algorithm is used in this paper to retrieve local top-k results at each node; hence, it is described in more detail in Section 3.2.

Variants of the *MaxRS* problem have also been considered in road networks. The *length-constrained maximum-sum region* query is introduced in [4]. An approximation algorithm is proposed, utilizing a technique that scales node weights into integers, as well as a heuristic and a greedy algorithm. A unified framework that addresses three variants of *optimal location* queries in road networks is presented in [22]. Given a set of existing facilities and a set of clients, these queries compute the location for a new facility that optimizes a certain cost metric defined based on the distances between the clients and the facilities. Finally, continuous *Best Region Search* in spatial data streams in road networks has been addressed in [5], proposing several pruning strategies and a branch-and-bound algorithm.

3 PRELIMINARIES

In this section, we first provide a formal definition of the problem addressed in this paper, and then we briefly outline the k-BRS algorithm [19], which is exploited in our algorithms to compute the local results at each node.

3.1 Problem definition

Assume a set $\mathcal{D}$ of points in a 2-dimensional space. Let f be a scoring function that assigns a score $f(R)$ to any axes-aligned rectangle R, based on its enclosed points. We assume monotone scoring functions, such that if the contents of a rectangle R' are a superset of those of R, then $f(R') \geq f(R)$. Based on these, the problem of computing the *top-k overlap-aware best regions* can be formally defined as follows:

PROBLEM 1 (TOP-k OVERLAP-AWARE BEST REGIONS). *Given a two-dimensional point dataset $\mathcal{D}$, a monotone scoring function f, width and height parameters w, h, and an integer k, the goal is to compute a ranked list of k axis-aligned rectangles R_i with dimensions $w \times h$, such that for each i, j, $1 \leq i < j \leq k$, it holds that:*

- *$f(R_i) \geq f(R_j)$, i.e., the rectangles are ranked in decreasing order of their score,*
- *$R_i \cap R_j = \emptyset$, i.e., R_j does not overlap with any higher ranked result R_i, and*
- *any other rectangle R' either has score $f(R') \leq f(R_k)$, or there exists another rectangle R_i in the top-k list such that $f(R_i) \geq f(R')$ and R' overlaps with R_i.*

Note that this definition can be relaxed to allow results that *partially* overlap, up to a user-specified threshold. In fact, this boolean condition can be generalized even further, by allowing overlaps but penalizing the score of a region by a factor depending on its degree of overlap with a higher-ranked region, as described in [19] using the notion of *marginal gain*. To simplify presentation, in this paper we focus on the boolean case, but it is straightforward to adapt the proposed algorithms to handle cases where the score of a region is penalized based on its degree of overlap with previous results.

Our goal is to compute top-k overlap-aware best regions in a *parallel and distributed* setting. A progressive algorithm for top-k overlap-aware best regions has been presented in [19]. However, it operates in a centralized setting and does not scale to big geospatial datasets containing millions of points. Transferring this solution to a distributed environment is not straightforward, because, as shown in Section 1, deriving global top-k results from the union of local top-k ones leads to incorrect answers. Hence, in this paper, we focus on overcoming this problem.

Algorithm 1: Local *k-BRS* algorithm.

```
1  Function Local($k$)
2      $\mathcal{L} \leftarrow \emptyset$ ; $\mathcal{G} \leftarrow$ ConstructGrid()
3      $Q \leftarrow$ InitializePriorityQueue($\mathcal{G}$)
4      while $|\mathcal{L}| < k$ & $|Q| > 0$ do
5          $E \leftarrow Q.nextEntry()$
6          if $E.type$ = "Cell" then
7              $\mathcal{S} \leftarrow$ GenerateSlabs($E$)
8              $Q$.addAll($\mathcal{S}$)
9          else if $E.type$ = "Slab" then
10             $\mathcal{R} \leftarrow$ GenerateRegions($E$)
11             $\mathcal{S} \leftarrow$ GetNextSlabs($E.slabTree$)
12             $Q$.addAll($\mathcal{R} \cup \mathcal{S}$)
13         else if $E.type$ = "Region" then
14             if $overlapAcceptable(\mathcal{L}, E)$ then $\mathcal{L}$.add($E$)
15             $\mathcal{R} \leftarrow$ GetNextRegions($E.regionTree$)
16             $Q$.addAll($\mathcal{R}$)
17     return $\mathcal{L}$
```

3.2 The k-BRS algorithm

Next, we briefly outline the *k-BRS* algorithm [19], which is used in this paper as the basis for retrieving local top-k results in each partition. The process is summarized in Alg. 1.

The algorithm starts (Lines 2–3) by constructing a grid with cells of size $w \times h$, i.e., with the same dimensions as the regions to be discovered. For each cell C, an upper bound $UB(C)$ is computed for the score of any region R centered inside C. This is based on the fact that R can enclose at most those points located within C or its neighboring cells. The cells are then inserted into a priority queue Q in decreasing order of their upper bound.

Whenever a cell is extracted from Q (Lines 6–8), it is scanned bottom-up using a horizontal sweep line. This generates a series of *maximal slabs*, each one associated with a respective upper bound, according to which they are inserted into Q. Moreover, the series of slabs leading up to a maximal slab are organized in a *slab tree*, so that these sub-maximal slabs can be visited later if needed. This permits the algorithm to backtrack and explore other results in case the one found is inadmissible due to overlap.

Whenever a maximal slab is extracted from Q (Lines 9–12), it is scanned from left to right using a vertical sweep line. This generates a series of *maximal regions*, each one associated with an upper bound, according to which they are added to Q. In addition, one level of the associated slab tree is traversed, generating one or two new slabs, which now become maximal, and are thus inserted in Q, along with the reduced slab tree, to be processed accordingly. Maximal regions are also associated with a corresponding *region tree*, operating similarly to the slab tree.

Finally, whenever a maximal region is extracted from Q (Lines 13–16), a result is produced, comprising the next region with the highest utility score in the local top-k list. At this point, a check is performed to determine whether this result overlaps with a previously accepted result, and thus determine whether it is admissible or not. New maximal regions are also generated from the region tree and added to Q for future consideration.

4 MULTI-ROUND ALGORITHM

Our first method is an incremental, multi-rounds algorithm (*MR*) that gradually builds the global top-k list of results by retrieving local top-k results from each worker at each round. While aggregating the local top-k results received at the end of each round, the coordinator resolves overlaps, and, if needed, contacts again the affected workers, informing them about the occurred overlaps and asking them for accordingly revised top-k results. This process may take up to k rounds to complete. The steps are outlined in Alg. 2. We start our discussion with data partitioning, and then we explain the querying algorithm.

Data partitioning. Data points are spatially partitioned by a uniform grid with partition width w_p and height h_p (Lines 2–3). Each point with coordinates (x, y) is mapped to partition $P_{i,j}$, with $i = \lceil x/w_p \rceil$ and $j = \lceil y/h_p \rceil$. Data partitioning is performed offline. In Spark, this entails a simple map and groupByKey sequence that parses the original data and generates a new pair RDD with key being the partition id, and value the list of points belonging to this partition. Partitions are held by N nodes (in our case, the Spark workers). Typically, the number of nodes is much less than the number of partitions. It is assumed that $w_p >> w$ and $h_p >> h$, where w and h denote the width and height of the query rectangle.

The node holding each partition is responsible for processing the partition to identify the top-k regions with a top-left corner[1] within the partition. The border cells of the partitions are of particular interest. Notice that a candidate region may intersect two neighboring partitions (e.g., r_8 in Fig. 2). To enable detection of these regions from exactly one partition, and to be able to compute their score, border cells belonging to the top and left borders of each partition are replicated to all partitions they share a border with (for example, the blue-colored cells of $P_{2,3}$, $P_{3,3}$ and $P_{3,2}$ of Fig. 2 are the ones replicated to the node holding $P_{2,2}$). This replication may happen either at query time, when w and h are determined, or may occur offline, assuming that a maximum value for w and h is supported.

Query execution. Upon receiving the query, each partition P is processed in parallel to compute local top-k results (Line 6). The local top-k algorithm (Alg. 1) is used for this purpose, with a

[1]An implementation detail is that, in the *k-BRS* algorithm [19], each input point p is represented by a $w \times h$ rectangle R_p centered at it. The rationale is that any region centered inside R_p encloses p. Now, identifying a result by its top-left corner instead of its center involves a minor adaptation: R_p must have its bottom-right corner (instead of its center) at p. In this way, it still holds that any result R having its top-left corner inside R_p will enclose p.

Algorithm 2: MapReduce implementation of the Multi-round algorithm.

```
1  Function AtCoordinator(k)
       /* Data partitioning */
2      data ← input.map(poi to <partitionIndex, poi>)
3      data ← data.groupByKey(partitionIndex)
4      list globalAns ← ∅
       /* Querying */
5      while |Ans| ≤ k do
6          localAns ← data.map(Local(k, globalAns))
7          roundAns ← localAns.reduce((a, b) ⟹
             AggFunc(a, b, k))
8          globalAns ← globalAns ∪ roundAns
   /* Reduce-based aggregation of results */
9  Function AggFunc(res_1, res_2, k)
10     minAcceptableScore ←
         max(min(res_1 scores), min(res_2 scores))
11     localAns ← res_1 ∪ res_2
12     localAns ← SortDesc(localAns)
13     output ← ∅
14     for pos=1 to k do
15         if overlapAcceptable(localAns[pos], roundAns) &
             sc(localAns[pos]) > minAcceptableScore then
16             output ← output ∪ localAns[pos]
17         else
18             break
19     return output
```

minor tweak such that it accepts as input the list of global results computed so far (*globalAns*). This list is taken into consideration in Alg. 1 (Line 14), so that now the overlap condition is checked with respect to $\mathcal{L} \cup globalAns$ instead of $\mathcal{L}$.

The local results per partition are then passed to an aggregation function *AggFunc*, such that a single list of top-k results ends up at the coordinator. The function takes as input two local results, constructs their union, sorts it, and retains the top-k of them, as long as these do not overlap each other. If a non-acceptable overlap is found, or if the minimum score of the two lists surpasses the score of the current region, then the aggregation interrupts and retains only the output produced so far. The logic behind this is that, after any of these cases is observed, any further results returned by the aggregation process are not guaranteed to be correct or complete. This aggregation is associative and commutative. Therefore, it is executed as a reduction in Spark (Line 7), which means that the whole processing (both maps and reductions) is fully parallelized, and the coordinator never constitutes a bottleneck for the algorithm's performance. Finally, after the reduced results are returned to the coordinator, the coordinator merges them with the results of the previous rounds (if any), and loops through the previous map/reduce steps until it collects a total of k results.

As explained previously, we focus on the case that a boolean condition is used to check whether a new result is admissible with respect to existing results based on potential overlap. In the algorithm, this condition is checked by the *overlapAcceptable* function (Line 15). A generalization of this is to penalize the score of overlapping regions using a marginal gain function, as described in [19].

The above algorithm is amenable to configurations and optimizations. First, the size of partitions is a system parameter, which involves the following tradeoff. Very large partitions reduce the area covered by border cells, thereby reducing the probability that overlapping regions of two partitions require more rounds. However, they also lead to scalability problems of the local algorithm (the workers that hold dense partitions run out of memory, swap aggressively, and eventually crash).

Also notice that the algorithm asks for k results per round, per partition. It is however unlikely that the global top-k results come from the same partition, which means that the local nodes (the Spark workers) are likely producing many more results than needed. To reduce this extra effort, the coordinator can instead ask for the top-k' results per partition at each round, with $k' < k$ set either at the beginning, or progressively at each round, in order to fine-tune the tradeoff between the number of rounds and the local effort at the nodes: a high k' value favors towards a lower number of rounds, whereas a lower k' reduces the effort spent by the workers on computing results that are not really useful, because of yet-unknown overlaps. The following lemma formalizes this tradeoff.

LEMMA 4.1. *The multi-round algorithm requires at least* $\lceil k/k' \rceil$ *rounds and at most* k *rounds.*

PROOF. For the lower bound, consider the case where answers are contained in a single partition. Since each partition generates k' answers at each round, it is not possible to retrieve all results in less than $\lceil k/k' \rceil$ rounds. For the upper bound, consider the top-1 region among all collected local results at a given round. This is guaranteed to be admissible, since it has already been checked locally against the currently existing results (*globalAns*). Hence, at each round, at least one more result is produced. Consequently, the process will terminate after k rounds. □

5 SINGLE-ROUND ALGORITHM AND EXTENSIONS

The multi-round algorithm enables processing of datasets with sizes that could not be practically processed by the centralized algorithm. Still, the iterative nature of the algorithm (and overlapping regions from different partitions) may lead to a potentially high overhead and poor performance. To overcome this drawback, we now introduce a *single-round* algorithm. The algorithm maintains auxiliary information per region that is used during the reduction phase for handing overlapping regions from different partitions. We will then explore the space between the single-round and the multi-round algorithm, proposing a *hybrid* algorithm, and discussing additional optimizations.

5.1 Single-round algorithm

The intuition for the single-round algorithm (*SR*) is the following. When processing each partition, a sufficient number of regions (typically larger than k) will be computed and sent to the coordinator, such that it is guaranteed that the coordinator will have all candidates needed to assemble the global top-k results (discarding non-admissible ones) without further communication to the workers. The challenge is to determine how many, and which, additional regions need to be sent, such that the coordinator is guaranteed to hold sufficient information for extracting the final answer. This challenge arises from the presence of overlaps between candidate regions that are detected in different partitions. For example, in Fig. 2, region r_1 of partition $P_{2,2}$ overlaps with the

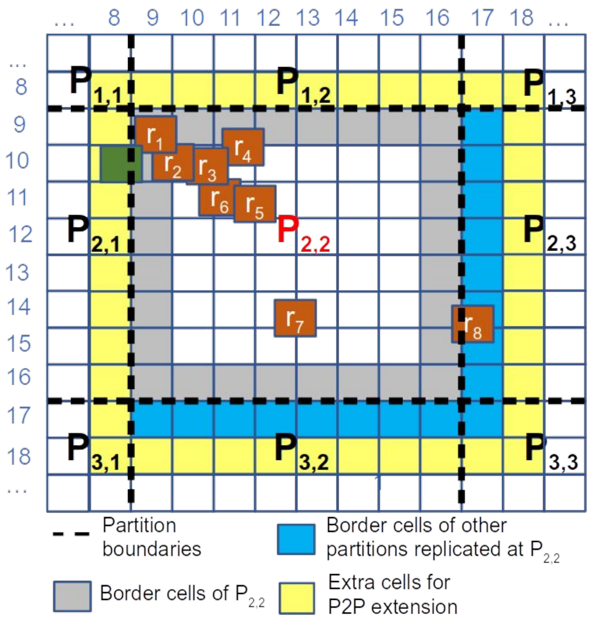


Figure 2: Identified regions within a node's partition. The regions are labeled in decreasing score, i.e., $sc(r_1) > sc(r_2)$.

green region of partition $P_{2,1}$, which has a higher score. Therefore, r_1 will not be in the final results (unless the green region of $P_{2,1}$ intersects with another region of $P_{2,1}$ with higher score), which will make r_2 a possible result. Unfortunately, existence of long chains of such overlaps prohibit local solutions that rely on data replication (e.g., replicating the border cells).

The single-round algorithm addresses this issue by forming a *dependency graph* of the identified candidate regions at each partition. These graphs allow each node to establish a lower bound on the number of regions contained in the local results that will be accepted by the coordinator, if their score is sufficiently high. We start by describing two core components of the algorithm, the *dependency graph* and the *extended dependency graph*.

Definition 5.1 (Dependency graph). Let $\mathcal{R} = r_1, r_2, \ldots$ denote the list of candidate regions detected at a partition P, ordered by their score, i.e., $sc(r_i) \geq sc(r_{i+1})$. Dependency graph $G(\mathcal{R})$ is a directed acyclic graph that contains all candidate regions from $\mathcal{R}$ as vertices, and has an edge between any two regions r_i and r_j if these overlap. The direction of the edge is from the region with the higher score towards the region having the lower score (ties between regions are broken in a consistent manner, by preferring the region with the lowest x coordinate, and then the region with the lowest y coordinate).

As an example, Fig. 3 depicts the dependency graph for the detected regions (r_1 to r_8) in Fig. 2.

Constructing the dependency graph for each partition $P_{x,y}$ is a local process, since it utilizes only the partition's data and the data in the border cells of the neighboring partitions $P_{x+1,y}$, $P_{x,y+1}$, and $P_{x+1,y+1}$ which is replicated in the node holding the partition. Recall, however, that the regions that overlap the border cells of each partition (gray-shaded cells in Fig. 2) may overlap with candidate regions detected in adjacent partitions (e.g., the green region from $P_{2,1}$ which overlaps with r_1 of $P_{2,2}$). The dependencies induced by these regions cannot be depicted in the dependency graph. Thus, to represent these dependencies,

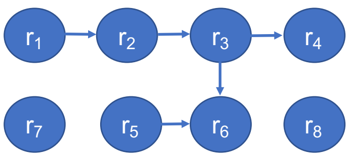


Figure 3: Dependency graph.

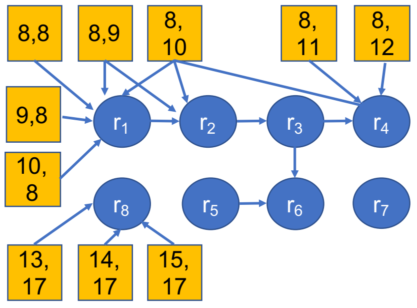


Figure 4: Extended dependency graph.

the dependency graph is extended by adding *artificial* dependencies for all (locally unknown) candidate regions that potentially overlap with regions detected at neighboring partitions.

Definition 5.2 (Extended dependency graph). The extended dependency graph $\mathcal{X}(\mathcal{R})$ of a partition is a DAG containing: (a) $G(\mathcal{R})$, and (b) for each vertex $v \in G(\mathcal{R})$ with an upper-left corner contained in a border cell (i, j), one vertex v' for each one of the cells adjacent to (i, j) that belongs to a different partition, and an edge pointing from v' to v.

Intuitively, the extended dependency graph encodes the possible dependencies from regions detected at other partitions. Since these regions are unknown at the node constructing the graph, they are represented with the coordinates of the cell that would contain their upper-left corner. For example, region r_1 of partition $P_{2,2}$ has an upper-left corner at cell $(9, 9)$. The adjacent cells of $(9, 9)$ belonging in other partitions are the cells $(8, 8)$, $(8, 9)$, $(8, 10)$, $(9, 8)$, and $(10, 8)$ of partitions $P_{1,1}$, $P_{1,2}$, and $P_{2,1}$. Any region from another partition that has a non-empty overlap with r_1 will have its upper-left corner at one of these cells. Figure 4 depicts the extended dependency graph of partition $P_{2,2}$.

The significance of the extended dependency graph is that it allows to establish an upper bound on the number of regions in the partition that may be excluded from the results due to (chains of) overlapping regions contained in other partitions. Conversely, it allows each partition to derive a lower bound on the number of *safe regions*, i.e., the regions that will not be invalidated from the contents of other partitions, even by long chains of overlaps.

The progressive construction of the extended dependency graph takes place alongside the local algorithm introduced in Alg. 1, and is summarized in Alg. 3. Precisely, we slightly modify the local algorithm in two ways: (a) for every identified region that passes the filter of acceptable overlap (Line 14), the modified algorithm invokes a function *addRegion*, to add the region and its dependencies in the extended dependency graph, (b) it terminates when *addRegion* has detected sufficient safe regions.

In more detail, before processing a partition, the node initializes an empty extended dependency graph $\mathcal{G}$, three sets *Safe*, *Unsafe*, and *detectedRegions* for keeping the regions, a set $\mathcal{M}$ for keeping the identified dependencies, and a counter for the number of identified safe regions *safeCnt*. Upon invocation, *addRegion* checks the new region to decide whether it is safe, unsafe, or inadmissible, and to maintain a lower bound for the number of safe regions (variable *safeCnt*).[2] This process is governed by a set of rules. Precisely, for a candidate region r_x, the node checks for three different cases:

Case 1: r_x does not overlap with other identified regions (either safe or unsafe), or with the border cells. r_x is a safe region, i.e., it will be included in the final answer if its score is sufficiently high. We increase *safeCnt* by one, and add it in the *Safe* set and in the extended dependency graph with no dependencies. r_7 is one such example from Fig. 2.

Case 2: r_x overlaps with a safe region r_y, which is already included in the Safe *set.* Since r_y is already identified, its score is higher than the score of r_x. Therefore, node discards r_x and continues. As an example, consider r_5 and r_6 from Fig. 2 – since r_5 is safe and has a higher score than r_6, the latter will be discarded.

Case 3: r_x overlaps with a border cell, or with an unsafe region r_y included in the Unsafe *set.* In this case, r_x is also unsafe. r_x is added in the *Unsafe* set, and in the extended dependency graph, with dependency from r_y and/or the adjacent cells of the other partitions. Let $\mathcal{D}(r_x)$ denote the set of all dependencies for r_x contained in the dependency graph after this process. We have two further sub-cases:

Case 3a: If none of the regions contained in $\mathcal{D}(r_x)$ is included in $\mathcal{M}$, none of these has been considered before. Therefore, either r_x or a region from $\mathcal{D}(r_x)$ will be part of the top-k regions, assuming that their score is sufficiently high. To capture this, r_x and the regions in $\mathcal{D}(r_x)$ are added to $\mathcal{M}$, and counter *safeCnt* is increased by one. From Fig. 2, r_1 and r_8 will belong in this category. Out of these, the coordinator will later detect that r_1 has an overlap with the green-marked region detected at partition $P_{2,1}$ (i.e., the green region will be included in the results instead), whereas r_8 will be included in the results if is score is sufficiently high (hence the increase on *safeCnt* for both of them). Another example is region r_3, which depends on r_2: since r_2 overlaps with r_1 which will be included in $\mathcal{M}$, r_2 will not be included in $\mathcal{M}$. This means that r_3 and its dependencies will be included in $\mathcal{M}$, and *safeCnt* will be increased by one.

Case 3b: If any of the regions contained in $\mathcal{D}(r_x)$ is already included in set $\mathcal{M}$, then *safeCnt* is not increased and r_x is not added in $\mathcal{M}$. As an example, consider regions r_2 and r_1 from Fig. 2. Region r_1 will be identified first, and already contained in $\mathcal{M}$ when r_2 is identified. Therefore, identification of r_2 will not lead to an increase of *safeCnt*.

After completion of *addRegion*, the control returns to the modified local algorithm, which breaks the loop when *safeCnt* reaches k. The local results for the partition are contained in *detectedRegions*, and the extended dependency graph is saved in $\mathcal{G}$.

Similar to the case of the multi-round algorithm, implementation of the single-round algorithm over Spark requires a way to hierarchically merge/reduce the local results (the dependency graphs) in order to get a final dependency graph with no further artificial dependencies in the coordinator, for extracting the final answer. Conceptually, the required merging involves the following steps: (1) we form the union of the two graphs, (2) if the two graphs share a border, we identify the artificial dependencies in the union graph that can now be eliminated, or replaced by dependencies on real regions, (3) we propagate the effect of each dependency elimination or replacement (e.g., switching a region from unsafe to safe), by performing a depth-first traversal of the graph starting from the affected region, and, (4) we reduce the graph by finding the score of the top-k safe region, and removing all regions with a lower score.

[2]The problem of counting the number of safe regions is not equivalent to the problem of actually detecting the safe regions. Addressing the latter problem typically leads to less regions marked as safe, which translates to larger dependency graphs and degradation of the algorithm's performance.

Algorithm 3: Single-round algorithm – progressive construction of extended dependency graph.

```
1  $\mathcal{G} \leftarrow \emptyset$ , $Safe \leftarrow \emptyset$ , $Unsafe \leftarrow \emptyset$ , $detectedRegions \leftarrow \emptyset$ ,
     $M \leftarrow \emptyset$ , $safeCnt \leftarrow 0$
2  Function addRegion(region)
3      if region does not overlap with any region in Safe,
         Unsafe, and border cells then
4          $\mathcal{G}$.addNode(region)
5          $Safe \leftarrow Safe \cup region$
6          $safeCnt \leftarrow safeCnt + 1$
7      else if region overlaps with a region in Safe then
8          continue
9      else if region overlaps with a region in Unsafe or
         border cells then
10         $\mathcal{G}$.addNode(region)
11         $\mathcal{G}$.addDependencies(region)
12         $Unsafe \leftarrow Unsafe \cup region$
13         if region dependencies is included in $\mathcal{M}$ then
14             continue
15         else
16             $safeCnt \leftarrow safeCnt + 1$
17             $M \leftarrow M \cup$( region dependencies)
18     $detectedRegions \leftarrow Safe \cup Unsafe$
```

Notice that the effect of the propagation step (step 3) relies on the same rules that we introduced earlier to determine whether a region is safe, unsafe, or can be safely removed. Therefore, a simple way to implement the above process is to initialize an empty graph, and keep adding the regions of the two partitions in descending order of score by invoking *addRegion* function of Alg. 3. The adding process can stop when *safeCnt* exceeds k.

THEOREM 5.3. *The final dependency graph produced by the algorithm only contains safe regions, which are the answer to the user's query.*

Proof sketch: Since the final dependency graph contains the input from all partitions, it will no longer contain artificial dependencies, which cause the unsafe regions (notice that inadmissible regions are not included in the graph – see lines 7-8 of Alg. 3). Also, regions are added in this graph in order of descending score, and the graph is completed as soon as the graph contains k nodes. The proof can be formalized with induction. □

The discussed algorithm is amenable to several optimizations and extensions for reducing network load and/or wallclock time. We present these in the following.

5.2 Spatial-aware tree-based aggregation

The aggregation function at the single-round algorithm is commutative and associative, similar to the multi-round algorithm. Therefore, it can be expressed in Spark as a reduction, enabling the Spark engine to fully distribute this part of the algorithm as well. Notice however that Spark does not take spatial proximity of partitions into account when executing the reducers. As

Algorithm 4: MapReduce implementation of Single-round algorithm. Parameters res_1, res_2, res_3, ... represent the results grouped by the same key.

```
Function AtCoordinator($k$)
    $data \leftarrow input.map$(poi to <$partitionIndex, poi$>)
    $data \leftarrow data.groupByKey(partitionIndex)$
    /* first compute the local results per
       partition                                  */
    $localAns \leftarrow data.map$(modified Alg1.$Local(k)$)
    /* recursive hierarchical aggr.               */
    for $i$=1 to $treeHeight$ do
        $localAns \leftarrow$
          $localAns.map(recomputePartitionIndexes(b))$
          $.groupByKey(partitionIndex).map(mergeRegions(k))$
    $Ans \leftarrow localAns$
Function mergeRegions(<$res_1, res_2, res_3, \ldots$>, $k$)
    $\mathcal{G} \leftarrow \emptyset$
    $pos \leftarrow 0$
    $unionRes \leftarrow sortDesc(res_1 \cup res_2 \cup res_3 \ldots)$
    while ($\mathcal{G}.safeCnt < k$) do
        $\mathcal{G}.addRegion(unionRes.get(pos))$
        $pos \leftarrow pos + 1$
    return $\mathcal{G}$
```

such, for a 16-partitions example of Fig. 5, one possible reduction order could be the following (where $\oplus$ denotes the reduction/aggregation function of the results): $(((1 \oplus 16) \oplus (5 \oplus 11)) \oplus (9 \oplus 7)) \oplus (4 \oplus 13) \ldots$. This random-order reduction precludes a core optimization, since merging distant partitions (e.g., partitions 1 and 16) does not help the reduction function to increase the per-partition number of safe regions. Ideally, we could reduce partition results in a proximity-aware hierarchical approach (i.e., order $((1 \oplus 2) \oplus (3 \oplus 4))\ldots$). Then, the merging process would, for example, exploit the partial results of partition 1 to mark the partial results of partition 2 that border with partition 1 as safe, or to exclude them, depending on their overlap and scores.

To exploit this observation, we enforce an explicit reduction order to Spark by introducing a hierarchical structure of aggregations. Fig. 5 depicts a small example with a hierarchical aggregation of 16 partitions. First, we apply a Z-order curve to assign an id to all partitions. Notice that close-by ids now mostly have spatial proximity. Then, each partition with id id assumes place in the hierarchy as the child of parent with id $\lceil id/b^2 \rceil$, where b^2 is the desired fan-out of our hierarchy. In our example, partitions with id 1 to 4 become children of a parent with id 1, i.e., their results will be merged together with a reduction in order to receive their parent node 1. The aggregation process continues recursively, until we reach to a single parent.

In concrete terms, let P denote the total number of partitions, and $p = b^i$ be the smallest power of $b \in \mathbb{N}^+$ that is greater than or equal to P, for a user-configured value of b. The space is recursively partitioned to p tiles of equal size ($b \times b$ tiles at each recursion), giving rise to a (b^2)-ary aggregation tree. This aggregation tree is represented in Spark with a chain of *map* and *groupByKey* sequences (Alg. 4, lines 5-7). The *map* functions determine the place of the tile in the hierarchy (essentially the id of the parent reducer), and the *groupByKey* functions bring together the results of the neighboring partitions, for the merging algorithm to run (function *mergeRegions*). The process continues until the root of the hierarchy, and the results are finally collected by the coordinator. Again, this whole process is executed in a decentralized fashion, and the coordinator only receives the final results. The value b that determines the number of levels in the tree hierarchy is important. A very small value of b, e.g., 2, leads to many levels, introducing significant synchronization overhead in Spark. At the other extreme, very large b values lead to low parallelism and large memory requirements at the nodes.

5.3 Peer-to-peer communication

Up to now, construction of the extended dependency graph did not exploit information regarding the neighboring partitions. As such, all artificial dependencies from neighboring partitions were set in a pessimistic way to dominate the local regions, leading to potentially long dependency chains (e.g., Fig. 6 contains a dependency chain that includes r_1 to r_4, because r_1 is unsafe). To alleviate this issue, we introduce moderate communication between the nodes, for exchanging key statistics regarding their neighboring partitions (e.g., the exact highest score of any region within all border cells, the maximum score of all regions in the border cells, or even the individual regions identified in the border cells and their scores). These statistics or results can then be used during the local algorithm to replace or tighten the artificial dependencies, i.e., to upper-bound the score of the artificial vertices in the extended dependency graph. The amount and type of data to exchange between nodes can lead to a spectrum of configurations that enable a tradeoff between the number of safe regions and the computational overhead and network volume.

This P2P-style communication, though, is not natural in Spark's MapReduce paradigm. One way to simulate it within Spark is by introducing a preliminary round, where each node runs a fast preparation step on each of its partitions and computes the desired statistics. However, this extra round introduces a high synchronization overhead for Spark. Instead, we choose to extend the idea of replication of border cells we described earlier, and assign to each node the additional task of first extracting coarse-grained statistics for the border cells of its eight neighboring partitions, prior to analyzing its own partition. In this way, the code for statistics extraction can be fully integrated in the map process of the single-round algorithm, without requiring an additional round of synchronization.

Particularly, we extend the data held for each partition by one row/column in each direction. In the example of Fig. 2, the yellow-colored cells of the neighboring partitions will also be copied to the node holding $P_{2,2}$, in addition to the blue-colored cells.[3]. When the processing for $P_{2,2}$ is initiated, the first task of the node is to process these replicated border cells and extract these statistics that will be subsequently used for constructing the extended dependency graph. We examined two levels of granularity for these statistics (in decreasing granularity):

- Executing the local algorithm on these cells to extract all contained regions – possibly overlapping each other. The scores of these regions will be an upper bound of the true scores. Therefore, these scores can be used to remove some of the dependency relations of the artificial dependencies in the extended dependency graph, i.e., if the region of the local partition has a higher score than the region in the replicated area (Fig. 6 (left)).

[3]Notice the asymmetry between the replicated cells from the left and the replicated cells from the right of $P_{2,2}$ (similarly for the cells above and below) This happens because the node holding $P_{2,2}$ is also responsible for finding the regions that overlap $P_{2,3}$, but not the ones overlapping $P_{2,1}$

Algorithm 5: Hybrid algorithm – region merging. Parameters $res_1, res_2, res_3, \ldots$ represent the results grouped by the same key.

```
1 Function mergeRegions(<$res_1$, $res_2$, $res_3$, ...>, $k$)
2     $minAcceptableScore \leftarrow$
        $\max(\min(sc(res_1)), \min(sc(res_2)), \min(sc(res_3)), \ldots)$
3     $\mathcal{G} \leftarrow \emptyset$ , $pos \leftarrow 0$
4     $unionRes \leftarrow sortDesc(res_1 \cup res_2 \cup res_3 \ldots)$
5     while $\mathcal{G}.safeCnt < k$ & $sc(unionRes.get(pos)) >$
        $minAcceptableScore$ do
6         $\mathcal{G}.addRegion(unionRes.get(pos))$
7         $pos \leftarrow pos + 1$
8     return $\mathcal{G}$
```

- For each border cell of a neighboring partition with coordinates (i, j), compute the score of the $2\epsilon \times 2\epsilon$ region consisting of cells $(i, j), (i + 1, j), (i, j + 1), (i + 1, j + 1)$. This does not require executing the local algorithm, since the exact region boundaries are known. We only need to compute the score function for the region, and use it as an upper bound for the cell. (Fig. 6 (right)).

The first approach produces the tightest upper bounds, but preludes execution of the full local algorithm on the border cells, thereby adding non-negligible time overhead. The second approach produces weaker upper bounds for the scores, but it is much more efficient. In our experiments, the second approach provided the best tradeoff in terms of overall execution time (and, thus, is the only one reported).

5.4 Hybrid algorithm

The single-round algorithm is very conservative, requiring k safe regions to be obtained from every partition. In practice, this leads to additional load for extending the local graphs, that could otherwise be avoided. We next describe a hybrid algorithm (*HY*), which covers the space between the single-round and multi-round algorithms, balancing the number of rounds and the number of results expected from each round. The intuition is that we can execute the single-round algorithm, but now requesting a smaller number of safe regions $k' << k$ per partition, aggregate the partial results, and then progressively ask for more results only from the partitions from which we already consumed at least one safe region. The partial results collected per round are a sorted subset of the final results (at least the next k' answers, but typically much more), and can be presented progressively to the user.

Similar to the case of the multi-round algorithm, the aggregation function (see Alg. 5) needs to stop accepting more data when it can no longer guarantee correctness of results. Correctness of results is guaranteed by establishing a bound for the region scores, above which all regions are guaranteed to be complete. When aggregating partial results of two partitions, say, res_1 and res_2, this score bound is simply the maximum of the two minimum scores in each of the partial results (Line 2). The intuition behind this bound is that the reducer does not have sufficient information about regions with lower score for one of the two, but all possible solutions (safe and unsafe regions) for higher scores are already included in the individual dependency graphs. Notice that this bound is naturally moved up in the hierarchy, as the aggregate results are pushed to parent reducers, and regions with lower scores are filtered out.

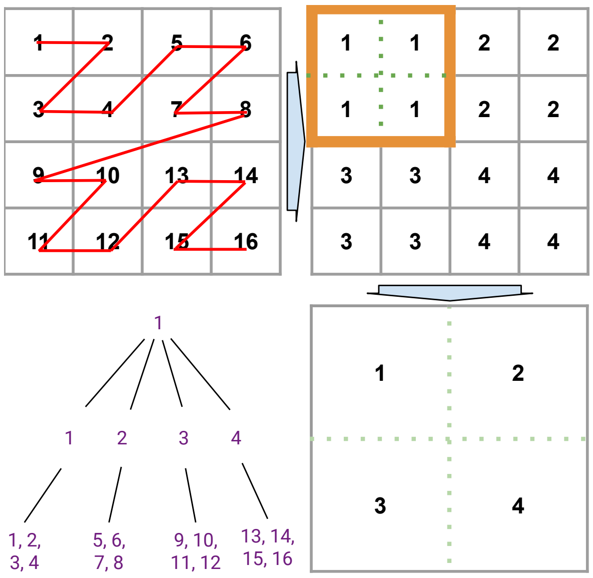


Figure 5: Hierarchical aggregation in 4×4 partitions with $b = 2$ by applying Z-order. The tree depicts the aggregation hierarchy, from 16 partitions to a single result.

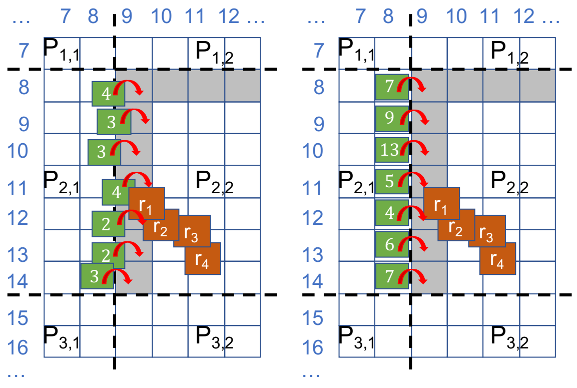


Figure 6: Exchanging region scores (left) or upper bounds for border cells (right).

6 EXPERIMENTAL EVALUATION

We have conducted an experimental evaluation to compare the performance of our proposed algorithms, investigate their scalability, and explore the impact of the various parameters.

6.1 Experimental setting

Datasets. We conducted our experiments using a real-world dataset comprising 64 million records representing Points of Interest from OpenStreetMap[4] and geolocated photos from Flickr[5] worldwide. We have mapped these to 26 million distinct locations (points). We have assigned a weight (score) to each point denoting the number of records (POIs or photos) mapped to it.

Experimental setup. The experiments are executed on a cluster of 11 nodes, each with 30 GB of RAM. Ten of the nodes are configured as workers, and the eleventh is indicated as the master/coordinator. The cluster runs Spark 2.4.3, and Hadoop HDFS

[4] https://www.openstreetmap.org
[5] https://www.flickr.com

Parameters	Values
number of points $\|\mathcal{D}\|$	5, 10, 15, 20, **26** million
number of nodes $\|\mathcal{N}\|$	1, 2, 4, 6, 8, **10**
top-k regions	50, 100, 200, **300**, 400, 500
region width and height ϵ	0.00025, 0.0005, 0.00075, **0.001**, 0.00125, 0.0015 (from approximately $27x27m^2$ to $162x162m^2$)
k' – Multi-round	3, 5, 10, 20, 30, 40, 50, 60, **300**
k' – Hybrid	3, 4, 5, 6, **10**, 15

Table 1: Experimental parameters (default value is bold).

2.9.1. The input file is distributed across the 10 workers, with replication factor set to one.

Implementation and configuration. All algorithms are implemented in Scala. Our implementations include by default the hierarchical aggregation for the hybrid algorithm and the corresponding commutative and associative reduction function for the single-round algorithm, since the coordinator becomes a bottleneck (and crashes for some parameters) otherwise.

We partitioned the input points into a uniform grid with dimensions 20,000 × 20,000, leading to partitions of size approximately $2km \times 2km$. The most densely populated partition contains approximately 18,000 points. The value of b in the tree-based aggregation was set to 8. We found these values to provide consistently good performance without creating bottlenecks. Using larger partitions creates bottlenecks during the local execution of the algorithm to the workers, whereas a significantly larger base leads to bottlenecks during the aggregation step (the reducer needs to aggregate too many partial results, which may lead to crashes around dense areas). With respect to all other parameters, unless otherwise mentioned, we use the default values shown in Table 1. The region width and height ϵ is measured in degrees.

6.2 Multi-round vs Single-round

We start by comparing the performance of the multi-round (*MR*) and the single-round (*SR*) algorithms.

We first vary the number of requested top-k regions from 50 to 500. Fig. 7 shows the results – wallclock time for both algorithms on the left Y axis and number of rounds for *MR* on the right Y axis. We see that the value of k has only a minor influence on the performance of *SR*. Conversely, the execution time of *MR* increases as k increases, eventually becoming around 40 times higher than that of *SR*. This stark difference is attributed to the number of rounds required by *MR*. Indicatively, for $k = 500$, *MR* requires 58 rounds, while *SR* only requires one. Of course, one round of execution for *SR* is more time consuming than for *MR* due to the extra time spent on creating and maintaining the dependency graph and continuing the computation of results until k safe ones are found. Indeed, we can observe that the one round of *SR* takes around 700 seconds to complete, whereas each round of *MR* takes on average 400 seconds. Yet, this difference is very small to compensate the extra overhead incurred by the high number of rounds required by *MR*.

In our second experiment (Fig. 8), we fix k to 300 and vary the width and height ϵ of the regions to be discovered. We see that the performance of both algorithms is affected by ϵ. These results are consistent to [19], which shows that the centralized algorithm (being used to retrieve the local top-k results in each partition) becomes slower as ϵ increases. Thus, they also exemplify the importance of having a parallel and distributed solution to achieve scalability. Still, since *SR* only needs to run the local algorithm once, it scales much better than *MR*, which is again affected by

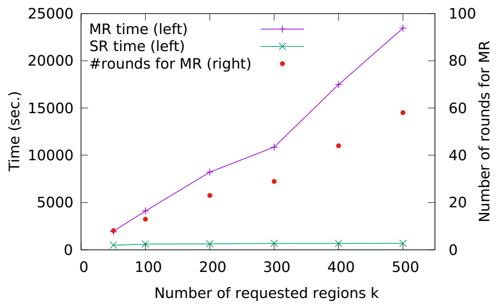


Figure 7: Performance of *MR* and *SR* for varying k.

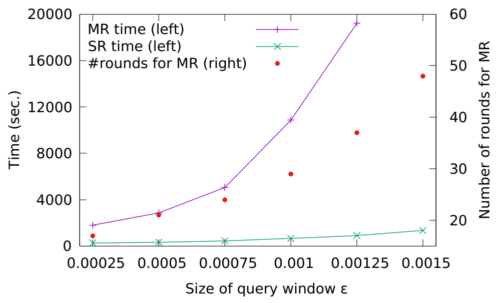


Figure 8: Performance of *MR* and *SR* for varying ϵ (region width and height).

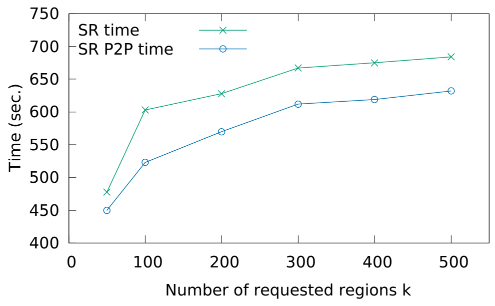


Figure 9: Effect of P2P extension on *SR* for varying k.

the need to execute multiple rounds, eventually requiring one order of magnitude more time compared to *SR*. Indicatively, for $\epsilon = 0.0015°$, *SR* takes 1352 seconds, while *MR* requires 44383 (for illustration reasons, the latter point is omitted in the plot).

We also evaluate the impact of the P2P extension on the performance of *SR*. Figure 9 plots the execution time of the algorithm with and without this extension. As expected, the extension always saves time, since it enables each node to create smaller dependency graphs per partition, and it has very low overhead. Indicatively, the average size of the dependency graph per partition with the P2P extension was reduced by approximate 40% for $k = 50$ and by 55% for $k = 500$. Since P2P consistently improves the performance of *SR*, it is applied in all remaining results.

6.3 Limiting the number of local results

We now examine how limiting the number of local results affects the performance of the algorithms. Recall that both *MR* and the Hybrid algorithm (*HY*) support requesting $k' < k$ results or safe regions from each partition at each round.

Figure 10 plots the execution time (left Y axis) and number of rounds (right axis) for *MR* for various k' values. Clearly, the

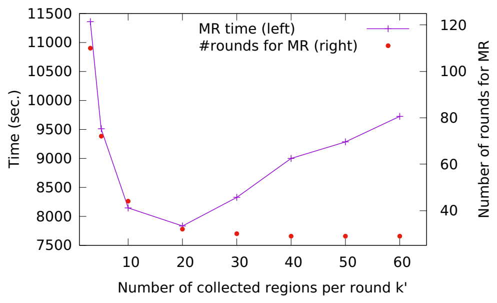


Figure 10: Execution time of *MR* for varying k'.

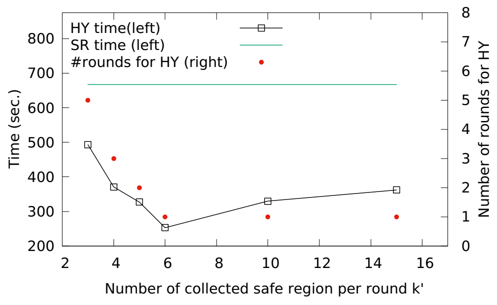


Figure 11: Execution time of *HY* for varying k' (with *SR* included for reference).

value of k' has significant influence on performance. As expected, setting a very low k' leads to a high number of rounds, which translates to excessive synchronization overhead and to longer execution times. Increasing k' until 20 reduces total time by limiting the time spent by the workers on computing the local results (which, in most cases, are anyway not needed by the coordinator). Since this time reduction is per round, and many rounds are required, the overall performance increase is significant. On the other hand, increasing k' beyond a certain point no longer reduces the number of rounds, but increases the average execution time per round, and consequently, the total execution time of the algorithm. Interestingly, already for $k' > 20$, execution time increases with the value of k', i.e., the performance gain because of reduction of the number of rounds is overshadowed by the additional time required for computing more local results, and merging them in the reduction phase.

Figure 11 depicts the influence of the respective parameter k' for *HY*. In this case, k' denotes the number of safe regions requested per partition. We see that a very low value of k' raises the need for additional rounds, since the appearance of non-admissible results prevents the coordinator from obtaining a valid top-k at the first place. However, with k' equal to 6, *HY* already completes in a single round and achieves the optimal performance, which is around one third of the baseline performance of *SR* (this would correspond to the worst performance of *HY*). It is also worth noticing that, in absolute time difference, *HY* is not as sensitive as *MR* with respect to a higher-than-optimal number of local results (cf., a high k' value in Fig 10). For example, even when collecting 15 safe regions per round (2.5 times more than the optimal), the difference in time is only around 100 seconds, compared to around 2000 seconds for *MR*. This is attributed to the fact that a sub-optimal value of k' affects much fewer rounds in *HY*, compared to a sub-optimal value of k' for *MR*. Thus, when tuning *HY*, it is relatively easier to find a good value for k'.

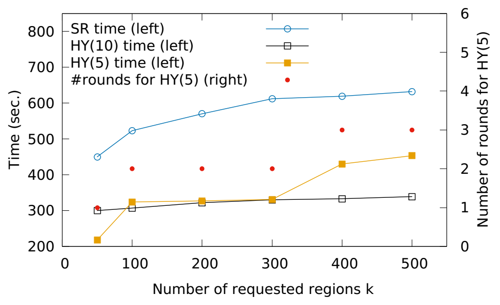


Figure 12: Comparison of *SR* and *HY* for varying k. *HY*(5) and *HY*(10) correspond to *HY* with $k' = 5$ and $k' = 10$, respectively. *HY*(10) always completes in a single round.

6.4 Comparing *SR* and *HY*

We now evaluate the performance of *SR* and *HY* for varying k and ϵ. We omit *MR* in these experiments since, as already indicated in previous results, it is always outperformed significantly by *SR*.

Figure12 presents the execution time of both algorithms, for different values of k, and for two hybrid executions with $k' = 5$ and $k' = 10$, noted as *HY*(5) and *HY*(10), respectively. The plot also includes the number of rounds for *HY*(5) (right Y axis). The number of rounds for *HY*(10) (and for *SR*) is always 1, and therefore it is omitted. As expected, the value of k brings a noticeable increase on the cost of *SR*, since it leads to larger dependency graphs. For *HY*(5), a higher k leads to more rounds, which also causes an increase in execution time. Nevertheless, *HY*(5) still outperforms *SR*, because due to the low value of k' the additional rounds are much faster compared to one round of *SR*. Also, *HY*(10) exhibits a very mild increase of the execution time (300 seconds for $k = 50$, compared to 339 seconds for $k = 500$). Since Hybrid(10) always takes 1 round, even for k=500, this increase is solely attributed to the extra cost during the hierarchical reduction: reducers need to maintain longer lists, and pass these lists to their parent nodes in the tree hierarchy. Still, this increase is negligible. Therefore, it is better to opt for a slightly higher value of k', in order to avoid the risk of running multiple rounds.

Figure 13 shows the execution time of *SR* and *HY*(10) for varying ϵ. Both algorithms exhibit a similar trend, but with *HY* consistently outperforming *SR*, and with the absolute difference between the execution time growing as ϵ increases. Detailed profiling on this result reveals that the additional time is exclusively spent on the local algorithm. As ϵ increases, the local algorithm spends more time for generating *each* result. Since *SR* has to compute 300 results in each partition, whereas *HY*(10) only 10, the total difference between the execution time of the two algorithms increases when the time spent per result is higher.

6.5 Scalability

Our last set of experiments focuses on investigating the scalability of the three algorithms, when varying the size of the dataset or number of executor nodes.

Given the original dataset, containing 26 million distinct points, we derive datasets of size from 5 to 20 million points by applying uniform sampling. Fig. 14 plots the execution time of the three algorithms. We observe that *MR* is the slowest algorithm in all cases, while *HY* performs better than *SR*. Moreover, *MR* demonstrates poor scalability compared to the other two, which exhibit similar performance, with *HY* scaling slightly better.

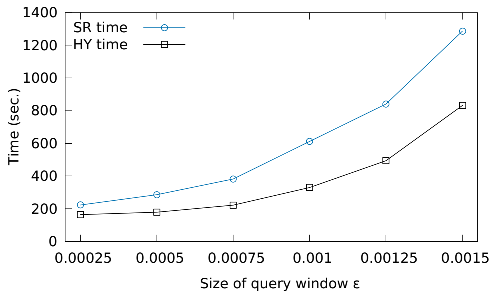


Figure 13: Comparison of *SR* and *HY* for varying ϵ. The number of rounds for *HY* is always one in this set of experiments (for k' set to 10).

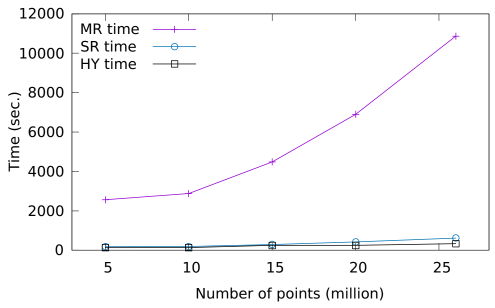


Figure 14: Execution time of *MR*, *SR*, *HY* for varying the number of points.

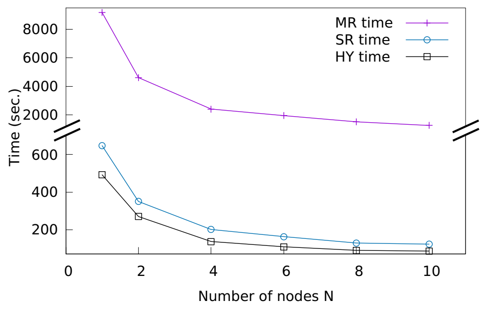


Figure 15: Execution time of *MR*, *SR*, *HY* for varying the number of nodes.

In our second experiment (Fig. 15), we vary the number of nodes from 1 to 10. Since it is not possible to process the whole dataset (26 million points) in a single node due to memory limitations, we sample 5 million points as input (i.e., the largest dataset size executable in a single node). As shown, addition of extra nodes decreases overall execution time for all algorithms, with *SR* and *HY* exhibiting linear speedup.

6.6 Summary

The experiments show that *SR* substantially outperforms *MR* in all cases, indicating that the extra cost incurred by multiple rounds dominates that for retrieving a sufficiently larger number of local results to ensure the construction of the global top-k in a single round. Overall, *HY* is the most efficient algorithm, indicating that in practice it suffices to compute just a few more than k local results to ensure that the coordinator can assemble the correct top-k list, despite any inadmissible local results.

Moreover, both algorithms *MR* and *HY* can benefit from setting the parameter k' to a much lower value than k (e.g., around 0.1 $\times$ k). This significantly increases the efficiency of *HY* due to the drastic reduction of the cost of local processing, while still obtaining the global top-k results in one or two rounds.

7 CONCLUSIONS

We have presented the first scalable algorithms for addressing the *top-k Best Region Search* problem. Our approach relies on distributing the dataset and the expensive computational part over cluster resources, thereby allowing the processing of large datasets in parallel. Starting from a multi-round algorithm, we proceed to devise one that requires a single round and is more efficient by one order of magnitude. Then, we also propose a hybrid algorithm, which further reduces computational time by a factor of two. Our future work focuses on continuous algorithms, for maintaining the answer in dynamic datasets, as well as approximation techniques, for further reducing the computational cost when small, bounded errors can be tolerated.

ACKNOWLEDGMENTS

This work was partially funded by the EU H2020 project SmartDataLake (825041).

REFERENCES

[1] D. Amagata and T. Hara. A general framework for MaxRS and MaxCRS monitoring in spatial data streams. *ACM TSAS*, 3(1):1:1–1:34, 2017.
[2] L. Anselin. Local indicators of spatial association—lisa. *Geographical analysis*, 27(2):93–115, 1995.
[3] X. Cao, G. Cong, T. Guo, C. S. Jensen, and B. C. Ooi. Efficient processing of spatial group keyword queries. *ACM Trans. Database Syst.*, 40(2):13:1–13:48, 2015.
[4] X. Cao, G. Cong, C. S. Jensen, and M. L. Yiu. Retrieving regions of interest for user exploration. *PVLDB*, 7(9):733–744, 2014.
[5] Z. Chen, Q. Yuan, and W. Liu. Monitoring best region in spatial data streams in road networks. *Data Knowl. Eng.*, 120:100–118, 2019.
[6] D. Choi, C. Chung, and Y. Tao. A scalable algorithm for maximizing range sum in spatial databases. *PVLDB*, 5(11):1088–1099, 2012.
[7] D. Choi, J. Pei, and X. Lin. Finding the minimum spatial keyword cover. In *ICDE*, pages 685–696, 2016.
[8] K. Deng, X. Li, J. Lu, and X. Zhou. Best keyword cover search. *IEEE Trans. Knowl. Data Eng.*, 27(1):61–73, 2015.
[9] A. Eldawy and M. F. Mokbel. The era of big spatial data: A survey. *Foundations and Trends in Databases*, 6(3-4):163–273, 2016.
[10] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.
[11] K. Feng, G. Cong, S. S. Bhowmick, W. Peng, and C. Miao. Towards best region search for data exploration. In *SIGMOD*, pages 1055–1070, 2016.
[12] K. Feng, T. Guo, G. Cong, S. S. Bhowmick, and S. Ma. SURGE: continuous detection of bursty regions over a stream of spatial objects. In *ICDE*, pages 1292–1295, 2018.
[13] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *FOCS*, pages 714–723, 1993.
[14] H. Imai and T. Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *J. Algorithms*, 4(4):310–323, 1983.
[15] S. Luo, Y. Luo, S. Zhou, G. Cong, and J. Guan. Distributed spatial keyword querying on road networks. In *EDBT*, pages 235–246, 2014.
[16] S. C. Nandy and B. B. Bhattacharya. A unified algorithm for finding maximum and minimum object enclosing rectangles and cuboids. *Computers & Mathematics with Applications*, 29(8):45–61, 1995.
[17] J. Qi, V. Kumar, R. Zhang, E. Tanin, G. Trajcevski, and P. Scheuermann. Continuous maintenance of range sum heat maps. In *ICDE*, pages 1625–1628, 2018.
[18] C. Sheng and Y. Tao. New results on two-dimensional orthogonal range aggregation in external memory. In *PODS*, pages 129–139, 2011.
[19] D. Skoutas, D. Sacharidis, and K. Patroumpas. Efficient progressive and diversified top-k best region search. In *SIGSPATIAL*, pages 299–308, 2018.
[20] Y. Tao, X. Hu, D. Choi, and C. Chung. Approximate MaxRS in spatial databases. *PVLDB*, 6(13):1546–1557, 2013.
[21] R. C. Wong, M. T. Özsu, P. S. Yu, A. W. Fu, and L. Liu. Efficient method for maximizing bichromatic reverse nearest neighbor. *PVLDB*, 2(1):1126–1137, 2009.
[22] X. Xiao, B. Yao, and F. Li. Optimal location queries in road network databases. In *ICDE*, pages 804–815, 2011.

Data Curation with Deep Learning

Saravanan Thirumuruganathan, Nan Tang, Mourad Ouzzani, AnHai Doan[‡]

Qatar Computing Research Institute, HBKU; [‡]University of Wisconsin-Madison
{sthirumuruganathan, ntang, mouzzani}@hbku.edu.qa, anhai@cs.wisc.edu

ABSTRACT

Data curation – the process of discovering, integrating, and cleaning data – is one of the oldest, hardest, yet inevitable data management problems. Despite decades of efforts from both researchers and practitioners, it is still one of the most time consuming and least enjoyable work of data scientists. In most organizations, data curation plays an important role so as to fully unlock the value of big data. Unfortunately, the current solutions are not keeping up with the ever-changing data ecosystem, because they often require substantially high human cost. Meanwhile, deep learning is making strides in achieving remarkable successes in multiple areas, such as image recognition, natural language processing, and speech recognition. In this vision paper, we explore how some of the fundamental innovations in deep learning could be leveraged to improve existing data curation solutions and to help build new ones. We identify interesting research opportunities and dispel common myths. We hope that the synthesis of these important domains will unleash a series of research activities that will lead to significantly improved solutions for many data curation tasks.

1 INTRODUCTION

Data Curation (DC) [32, 44] – the process of discovering, integrating [46] and cleaning data (Figure 1) for downstream analytic tasks – is critical for any organization to extract real business value from their data. The following are the most important problems that have been extensively studied by the database community under the umbrella of data curation: *data discovery*, the process of identifying relevant data for a specific task; *schema matching and schema mapping*, the process of identifying similar columns and learning the transformation between matched columns, respectively; *entity resolution*, the problem of identifying pairs of *tuples* that denote the same entity; and *data cleaning*, the process of identifying errors in the data and possibly repairing them. These are in addition to problems related to outlier detection, data imputation, and data dependencies.

Due to the importance of data curation, there has been many commercial solutions (for example, Tamr [45] and Trifacta [23]) and academic efforts for all aspects of DC, including data discovery [8, 19, 33, 36], data integration [12, 27], and data cleaning [7, 16, 42]. An oft-cited statistic is that data scientists spend 80% of their time curating their data [8]. Most DC solutions cannot be fully automated, as they are often *ad-hoc* and require substantial effort to generate things, such as features and labeled data, which are used to synthesize rules and train machine learning models. Practitioners need practical and usable solutions that can significantly reduce the human cost.

Deep Learning (DL) is a successful paradigm within the area of machine learning (ML) that has achieved significant successes in diverse areas, such as computer vision, natural language processing, speech recognition, genomics, and many more. The trifecta of big data, better algorithms, and faster processing power has resulted in DL achieving outstanding performance in many areas. Due to these successes, there has been extensive new research seeking to apply DL to other areas both inside and outside of computer science.

Data Curation Meets Deep Learning. In this article, we investigate intriguing research opportunities for answering the following questions:

- What does it take to significantly advance a challenging area such as DC?
- How can we leverage techniques from DL for DC?
- Given the many DL research efforts, how can we identify the most promising leads that are most relevant to DC?

We believe that DL brings unique opportunities for DC, which our community must seize upon. We identified two fundamental ideas with great potential to solve challenging DC problems.

Feature (or Representation) Learning. For ML- or non-ML based DC solutions, domain experts are heavily involved in feature engineering and data understanding so as to define data quality rules. *Representation learning* is a fundamental idea in DL where appropriate features for a given task are automatically learned instead of being manually crafted. Developing DC-specific representation learning algorithms could dramatically alleviate many of the frustrations domain experts face when solving DC problems. The learned representation must be generic such that it could be used for multiple DC tasks.

DL Architectures for DC. So far, no DL architecture exists that is cognizant of the characteristics of DC tasks, such as representations for tuples or columns, integrity constraints, and so on. Naively applying existing DL architectures may work well for some, but definitely not all, DC tasks. Instead, designing new DL architectures that are tailored for DC and are cognizant of the characteristics of DC tasks would allow efficient learning both in terms of training time and the amount of required training data. Given the success of domain specific DL architectures in computer vision and natural language processing, it behooves us to design an equivalent of such a DL architecture customized for DC task(s).

Contributions and Roadmap. The major elements of our proposed approach to exploiting the huge potential offered by DL to tackle key DC challenges include:

- **Representation Learning for Data Curation.** We describe several research directions for building representations that are explicitly designed for DC. Developing algorithms for representation learning that can work on multiple modalities of data (structured, unstructured, graphical),

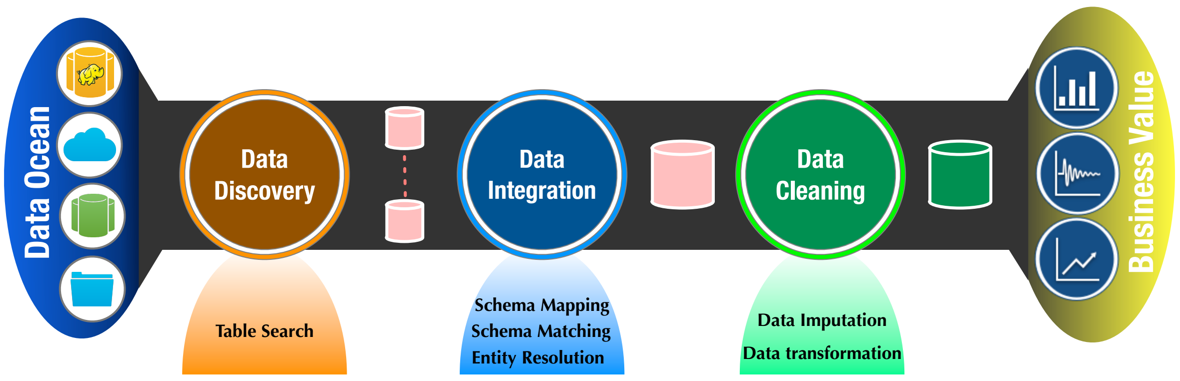


Figure 1: A Data Curation Pipeline

and can work on diverse DC tasks (such as deduplication, error detection, data repair) is challenging.

- **Deep Learning Architectures for Data Curation Tasks.** We identify some of the most common tasks in DC and propose preliminary solutions using DL. We also highlight the importance of designing DC specific DL architectures.
- **Early Successes of DL for DC.** In the last couple of years, there has been some promising developments in applying DL to some DC problems such as data discovery [19] and entity resolution [14, 35]. We provide a brief overview of these work and the lessons learned.
- **Taming DL's Hunger for Data.** DL often requires a large amount of training data. However, there are multiple promising techniques, which when adapted for DC, could dramatically reduce the required amount of labeled data.
- **Myths and Concerns about DL.** DL is still a relatively new concept, especially for database applications and is not a silver bullet. We thus identify and address a number of myths and concerns about DL.

2 DEEP LEARNING FUNDAMENTALS

We provide a quick overview of relevant DL concepts needed for the latter sections. Please refer to [5, 20, 29] for additional details.

Deep learning is a subfield of ML that seeks to learn meaningful representations from the data using computational models that are composed of multiple processing layers. The representations could be then used to solve the task at hand effectively. The most commonly used DL models are neural networks with many hidden layers. The key insight is that successive layers in this "deep" neural network can be used to learn increasingly useful representations of the data. Intuitively, the input layer often takes in the raw features. As the data is forwarded and transformed by each layer, an increasingly sophisticated and more meaningful information (representation) is extracted. This incremental manner in which increasingly sophisticated representations are learned layer-by-layer is one of the defining characteristic of DL.

2.1 Deep Learning Architectures

Fully-connected Neural Networks. The simplest model is a neural network with many hidden layers – a series of layers where each node in a given layer is connected to every other node in the next layer. They are also called feed-forward neural network. It can learn relationships between any two input features or intermediate representations. Its generality however comes with a cost; one has to learn weights and parameters which requires a lot of training data.

Domain Specific Architectures. There are a number of neural network architectures that are designed to leverage the domain specific characteristics. For example, Convolutional Neural Networks (CNNs) are widely used by the computer vision community. The input is fed through convolutions layers, where neurons in convolutional layers only connect to close neighbors (instead of all neurons connecting to all neurons). This method excels in identifying spatially local patterns and use it to learn spatial hierarchies such as nose → face → human. Recurrent Neural Networks (RNNs) are widely used in Natural Language Processing (NLP) and Speech recognition. RNN processes inputs in a sequence one step at a time. For example, given two words "data curation", it first handles "data" and then "curation". Neurons in an RNN are designed by being fed with information not just from the previous layer but also from themselves from the previous pass that are relevant for NLP.

2.2 Distributed Representations

Deep learning is based on learning data representations, and the concept of distributed representations (*a.k.a.* embeddings) is often central to DL. *Distributed representations* [24] represent one object by many representational elements (or many neurons). Each neuron is associated with more than one represented object. That is, the neurons represent features of objects. Distributed representations are very expressive and can handle semantic similarity. These advantages become especially relevant in domains such as DC. There has been extensive work on developing algorithms for learning distributed representations for different types of entities such as words and nodes.

Distributed Representations of Words (*a.k.a.* word embeddings) seek to map individual words to a vector space, which helps DL algorithms to achieve better performance in NLP tasks by grouping similar words. The dimensionality is often fixed (such as 300) and the representation is often *dense*. Word embeddings are often learned from the data in such a way that semantically related words are often close to each other. The geometric relationship between words often also encodes a semantic relationship between them. An oft-quoted example shows that by adding the vector corresponding to the concept of *female* to the distributed representation of *king*, we (approximately) obtain the distributed representation of *queen*.

Distributed Representations of Graphs (*a.k.a.* network embeddings) are a popular mechanism to effectively learn appropriate features for graphs (see [6] for a survey). Each node is represented as a dense fixed length high dimensional vector. The optimization objective is neighborhood-preserving whereby two nodes that are in the same neighborhood will have similar representations. Different definitions of neighborhood results in different representations for nodes.

Distributed Representations for Sets. Sets are a fundamental data structure where the order of items does not matter. Sets are extensively used in the Codd model where each tuple is a set of atomic units and a relation is a set of tuples. Algorithms for learning set embeddings must ensure that they have permutation invariance so that any permutation of the set should obtain the same embedding. There has been a number of popular algorithms for processing sets to produce such embeddings such as [41, 52].

Compositions of Distributed Representations. One can then use these to design distributional representations of atomic units – words in NLP, or nodes in a graph – for more complex units. For NLP, these could be sentences (*i.e.,* sentence2vec), paragraphs or even documents (*i.e.,* doc2vec) [28]. In the case of graphs, it can be subgraphs or entire graph.

3 REPRESENTATION LEARNING FOR DATA CURATION

In this section, we discuss potential research opportunities for designing DL architectures and algorithms for learning DC specific representations.

3.1 The Need for DC Specific Distributed Representations

Data curation is the process of discovering, integrating and cleaning data for downstream tasks. There are a number of challenging sub-problems in data curation. However, a number of problems could be unified through the prism of "matching".

Data Discovery is the process of identifying relevant data for a specific task. In most enterprises, data is often scattered across a large number of tables that could range in the tens of thousands. As an example, the user might be interested in identifying all tables describing user studies involving insulin. The typical approach involves asking an expert or performing manual data exploration [8]. Data discovery could be considered as identifying tables that *match* a user specification (*e.g.,* a keyword or an SQL query).

Schema Matching is the problem of identifying a pair of *columns* from different tables that are (semantically) related and provide comparable information about an underlying entity. For example, two columns *wage* and *salary* could store similar information. The problem of **schema mapping** seeks to learn the transformation between matched columns such as *hourly wage* and *monthly salary.* One can see that this problem seeks to identify *matching* columns.

Data Cleaning is the general problem of detecting errors in a dataset and repairing them in order to improve the quality of the data. This includes qualitative data cleaning which uses mainly integrity constraints, rules, or patterns to detect errors and quantitative approaches which are mainly based on statistical methods. Many sub-problems in data cleaning fall under the ambit of matching. These include:

- **Entity Resolution** is the problem of identifying pairs of *tuples* that denote the same entity. For example, the tuple ⟨Bill Gates, MS⟩ and ⟨William Gates, Microsoft⟩ refer to the same person. This has a number of applications such as identifying duplicate tuples between two tables. Once again, we can see that this corresponds to discovering *matching* entities (or tuples), *i.e.,* those that refer to the same real-world objects.
- **Data Dependencies** specify how one attribute value depends on other attribute values. This is widely used as integrity constraints for capturing data violations. For example, the social security number determines a person's name while the country name typically determines its capital. We can see that this could be treated as an instance of *relevance matching*.
- **Outlier Detection** identifies anomalous data that *does not match* a group of values, either syntactically, semantically, or statistically. For example, the phone number $123-456-7890$ is anomalous when the other values are of the form *nnnnnnnnnn*.
- **Data Imputation.** is the problem of guessing the missing value that should *match* its surrounding evidence.

The main drawback of traditional DC solutions is that they typically use syntactic similarity based features. The syntactic approach is often ad-hoc, heavily reliant on domain experts and thus not extensible. While they are effective, they cannot leverage semantic similarity. A different approach is needed.

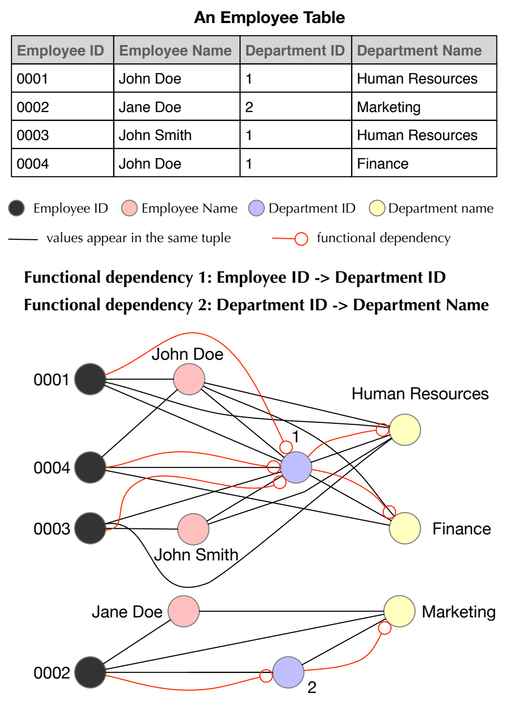
Intuitively, distributed representations, which can interpret one object in many different ways, may provide great help for various DC problems. However, there are a number of challenges before they could be used in DC. In domains such as NLP, simple co-occurrence is a sufficient approximation for distributional similarity. In contrast, domains such as DC require much more complex syntactic and semantic relationships between (multiple) attribute values, tuples, columns, and tables. Furthermore, the data itself could be noisy and the learned distributions should be resilient to errors.

3.2 Distributed Representation of Cells

A *cell*, which is an attribute value of a tuple, is the atomic data element in a relational database. Learning distributed representation for cells is already quite challenging and requires synthesis of representation learning of words and graphs.

Word Embeddings based Approaches. An initial approach inspired by word2vec [31] treats this as equivalent to the problem of learning word embeddings. Each tuple corresponds to a document where the value of each attribute corresponds to words. Hence, if two cell values occur together often in a similar context, then their distributed representation will be similar. For example, if a relation has attributes Country and Capital with many tuples containing *(USA, Washington DC)* for these two attributes, then their distributed representations would be similar.

Limitations of this Approach. The embeddings produced by this approach suffer from a number of issues. First, databases are typically well normalized to reduce redundancy, which also minimizes the frequency that two semantically related attribute values co-occur in the same tuples. Databases have many data dependencies (or integrity constraints), within tables (*e.g.,* functional dependencies [2], and conditional functional dependencies [17])



An Employee Table

Employee ID	Employee Name	Department ID	Department Name
0001	John Doe	1	Human Resources
0002	Jane Doe	2	Marketing
0003	John Smith	1	Human Resources
0004	John Doe	1	Finance

Functional dependency 1: Employee ID -> Department ID
Functional dependency 2: Department ID -> Department Name

Figure 2: A Heterogeneous Graph of A Table

or across tables (*e.g.,* foreign keys, and matching dependencies). These data dependencies are important hints about semantically related cells that should be captured by learning distributed representations of cells.

Combining Word and Graph Embeddings. To capture the relationships (*e.g.,* integrity constraints) between cells, a more natural way is to treat each relation as a *heterogeneous network.* Each relation D is modeled as a graph $G(V, E)$, where each node $u \in V$ is a unique attribute value, and each edge $(u, v) \in E$ represents multiple relationships, such as (u, v) co-occur in one tuple, there is functional dependency from the attribute of u to the attribute of v, and so on. The edges could be either directed or undirected, have labels and weights. This enriched model might provide a more meaningful distributed representation that is cognizant of both content and constraints.

A sample table and our proposed graph representation of the table is shown in Figure 2. There are four distinct Employee ID values (nodes), three distinct Employee Name values, two distinct Department ID values, and three Department Name values. There are two types of edges: undirected edges indicating values appearing in the same tuple, *e.g.,* 0001 and John Doe, and directed edges for functional dependencies, *e.g.,* Employee ID 0001 implies DepartmentID 1.

Research Opportunities.

- *Algorithms.* How can we design an algorithm for learning cell embeddings that take values, integrity constraints, and other metadata (*e.g.,* a query workload) into consideration?
- *Global Distributed Representations.* We need to learn distributed representations for the cells over the entire data lake, not only on one relation. How can we "transfer" knowledge gained from one relation to another to improve the representations?
- *Handling Rare Values.* Word embeddings often provide inadequate representations for rare values. How can we ensure that primary keys and other rare values have a meaningful representation?

3.3 Hierarchical Distributed Representations

Many DC tasks are often performed at a higher level of granularity than cells. The next fundamental question to solve is to design an algorithm to *compose* the distributed representations of more abstract units from these atomic units. As an example, how can one design an algorithm for tuple embeddings assuming one already has a distributed representation for each of its attribute values? A common approach is to simply *average* the distributed representation of all its component values.

Research Opportunities.

- *Tuple, Column and Table Embeddings (Tuple2Vec, Column2Vec, Table2Vec):* Are there any other elegant approaches to compose representations for tuples than averaging? Can it be composed from representations for cells? Or should it be learned directly? How can one extend this idea to learn representations for columns that are often useful for tasks such as schema matching? Finally, how can one learn a representation for the entire relation that can benefit a number of tasks such as copy detection or data discovery (finding similar relations)?
- *Compositional or Direct Learning*: Different domains require different ways to create hierarchical representations. For the computer vision domain, hierarchical representations such as shapes are learned by composing simpler individual representations such as edges. On the other hand, for NLP, hierarchical representations are often learned directly [28]. What is an appropriate mechanism for DC?
- *Contextual Embeddings for DC.* There has been increasing interest in the NLP community in contextual embeddings [10, 38] that can provide different embeddings for a word (such as apple) based on the surrounding context. This is often helpful for tasks requiring word disambiguation. Often, DC tasks require a number of contextual information and hence any learned representation must take that into account. What is an appropriate formalization of context and algorithms for contextual embeddings in DC?
- *Multi Modal Embeddings for DC.* Enterprises often possess data across various modalities, such as structured data (*e.g.,* relations), unstructured data (*e.g.,* documents), graphical data (*e.g.,* enterprise networks), and even videos, audios, and images. An intriguing line of research is *cross modal representation learning* [25], wherein two entities that are similar in one or more modalities, *e.g.,* occurring in the same relation, document, image, and so on, will have similar distributed representations.

4 DEEP LEARNING ARCHITECTURES FOR DATA CURATION TASKS

Representation learning and domain specific architectures are the two pillars that led to the major successes achieved by DL in various domains. While representation learning ensures that the input to the DL model contains all relevant information, the domain specific architecture often processes the input in a meaningful way requiring less training data than generic architectures.

In this section, we motivate the need for designing DC specific architecture and provide a promising design space to explore.

4.1 Need for DC Specific DL Architectures

Recall from Section 2 that a fully connected architecture is the most generic one. It does not make any domain specific assumptions and hence can be used for arbitrary domains. However, this generality comes at a cost: a lot of training data. One can argue that a major part of DL's success in computer vision and NLP is due to the design of specialized DL architectures – CNN and RNN respectively. CNN leverages spatial hierarchies of patterns where complex/global patterns are often composed of simpler/local patterns (*e.g., curves → mouth → face*). Similarly, RNN processes an input sequence one step at a time while maintaining an internal state. These assumptions allows one to design effective neural architectures for processing images and text that also require less training data. This is especially important in data curation where there is a persistent scarcity of labeled data. There is a pressing need for new DL architectures that are tailored for DC and are cognizant of the characteristics of DC tasks. This would allow them to do the learning efficiently both in terms of training time and the amount of required training data.

Desiderata for DC-DL Architectures.

- *Handling Heterogeneity.* In both CNN and RNN, the input is homogeneous – images and text. However, a relation can contain a wide variety of data, such as categorical, ordinal, numerical, textual, and image.
- *Set/Bag Semantics.* While an image can be considered as a sequence of pixels and a document as a sequence of words, such an abstraction does not work for relational data. Furthermore, the relational algebra is based on set and bag semantics with the major query languages specified as set operators. DL architectures that operate on sets are an under explored area in DL.
- *Long Range Dependencies.* The DC architecture must be able to determine long range dependencies across attributes and sometimes across relations. Many DC tasks rely on the knowledge of such dependencies. They should also be able to leverage additional domain knowledge and integrity constraints.

4.2 Design Space for DC-DL Architectures

Broadly speaking, there are two areas in which DL architectures could be used in DC. First, novel DL architectures are needed for learning effective representations for downstream DC tasks. Second, we need DL architectures for common DC tasks such as matching, data repair, imputation, and so on. Both are challenging on their own right and require significant innovations.

Architectures for DC Representation Learning. In a number of fields such as computer vision, deep neural networks learn general features in early layers and transition to task specific features in the latter layers. Initial layers learn generic features such as edges (which can be used in many tasks) while latter layers learn high level concepts such as a cat or dog (which could be used for task for recognizing cats from dogs). Models trained on large datasets such as ImageNet could be used as feature extractors followed by specific task specific models. Similarly, in NLP there has been a number of pretrained language models such as word2vec [31], ELMo [38], and BERT [10] that could then be customized for various tasks such as classification, question answering, semantic matching, and coreference resolution. In other words, the pre-trained DL models act as a set of Lego bricks that could be put together to perform the required functionality. It is thus important that the DL architecture for DC follows this property by learning features that are generic and could readily generalize to many DC tasks.

While there has been extensive work in linguistics about the hierarchical representations of language, a corresponding investigation in data curation is lacking. Nevertheless, we believe that a generic DC representation learning architecture must be inspired based on the pretrained language models. Once such an architecture is identified, the learned representations could be used for multiple DC tasks.

DL Architectures for DC Tasks. While the generic approach is often powerful, it is important that we must also work on DL architectures for specific DC tasks. This approach is often much simpler and provides incremental gains while also increasing our understanding of DL-DC nexus. In fact, this has been the approach taken by the NLP community - they worked on DL architectures for specific tasks (such as text classification or question answering) even while they searched for more expressive generative language models.

A promising approach is to take specific DC tasks and break them into simpler components and evaluate if there are existing DL architectures that could be reused. Often, it is possible to "compose" individual DL models to create a more complex model for solving a specific DC task. Many of the DC tasks could be considered as a combination of tasks such as matching, error detection, data repair, imputation, ranking, discovery, and syntactic/semantic transformations. An intriguing research question is to identify DL models for each of these and investigate how to instantiate and compose them together.

4.3 Pre-Trained DL Models for DC

In domains such as computer vision and NLP, there is a common tradition of training a DL model on a large dataset and then reusing it (after some tuning) for tasks on other smaller datasets. One example include ImageNet [9] which contains almost 14M images over 20k categories. The spatial hierarchy learned from this dataset is often a proxy for modeling the visual world [5] and can be applied on a different dataset and even different categories [5]. Similarly, in NLP, word embeddings are an effective approximation for language modeling. The embeddings are often learned from large diverse corpora such as Wikipedia or PubMed and could be used for downstream NLP tasks. These pre-trained models can be used in two ways: (a) *feature extraction* where these are used to extract generic features that are fed to a separate classifier for the task at hand; (b) *fine-tuning* where one adjusts the abstract representations from the last few hidden layers of a pre-trained model and make it more relevant to a targeted task. A promising research avenue is the design of pre-trained DL models for DC that could be used by others with comparatively less resources.

5 EARLY SUCCESSES OF DL FOR DC

In this section, we describe the early successes achieved by the DC community on using DL for major DC tasks such as data discovery and entity matching.

5.1 Data Discovery

Large enterprises typically possess hundreds or thousands of databases and relations. Data required for analytic tasks is often scattered across multiple databases depending on who collected the data. This makes the process of finding relevant data for a particular analytic task very challenging. Usually, there is no domain expert who has complete knowledge about the entire data repository. This results in a non-optimal scenario where the data analyst patches together data from her prior knowledge or limited searches – thereby leaving out potentially useful and relevant data.

As an example of applying word embeddings for data discovery, [19] shows how to discover semantic links between different data units, which are materialized in the *enterprise knowledge graph* (EKG)[1]. These links assist in data discovery by linking tables to each other, to facilitate navigating the schemas, and by relating data to external data sources such as ontologies and dictionaries, to help explain the schema meaning. A key component is a semantic matcher based on word embeddings. The key idea is that a group of words is similar to another group of words if the average similarity in the embeddings between all pairs of words is high. This approach was able to surface links that were previously unknown to the analysts, *e.g., isoform*, a type of protein, with *Protein* and *Pcr* – polymerase chain reaction, a technique to amplify a segment of DNA – with *assay*. We can see that any approach using syntactic similarity measures such as edit distance would not have identified these connections.

5.2 Entity Matching

Entity matching is a key problem in data integration, which determines if two tuples refer to the same underlying real-world entity [15].

DeepER. A recent work [14], DeepER, applies DL techniques for ER, whose overall architecture is shown in Figure 3. DeepER outperforms existing ER solutions in terms of accuracy, efficiency, and ease-of-use. For accuracy, DeepER uses sophisticated compositional methods, namely uni- and bi-directional recurrent neural networks (RNNs) with long short term memory (LSTM) hidden units to convert each tuple to a distributed representation, which are used to capture similarities between tuples. DeepER uses a locality sensitive hashing (LSH) based approach for blocking; it takes all attributes of a tuple into consideration and produces much smaller blocks, compared with traditional methods that consider only few attributes. For ease-of-use, DeepER requires much less human labeled data, and does not need feature engineering, compared with traditional machine learning based approaches which require handcrafted features, and similarity functions along with their associated thresholds.

DeepMatcher [35] proposes a template based architecture for entity matching. Figure 4 shows an illustration. It identifies four major components: attribute embedding, attribute summarization, attribute similarity and matching. It proposes various choices for each of these components leading to a rich design space. The four most promising models differ primarily in how the attribute summarization is performed and are dubbed as SIF, RNN, Attention, and Hybrid. SIF model is the simplest and computes a weighted average aggregation over word embeddings to get the attribute embedding. RNN uses a bi-directional

[1]An EKG is a graph structure whose nodes are data elements such as tables, attributes, and reference data such as ontologies and mapping tables, and whose edges represent different relationships between nodes.

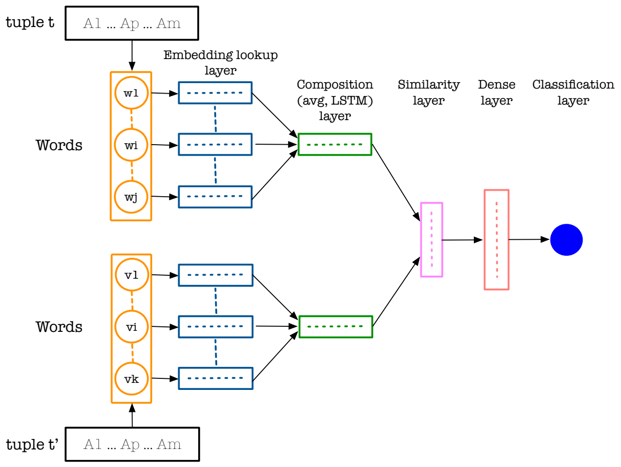


Figure 3: DeepER Framework (Figure from [14])

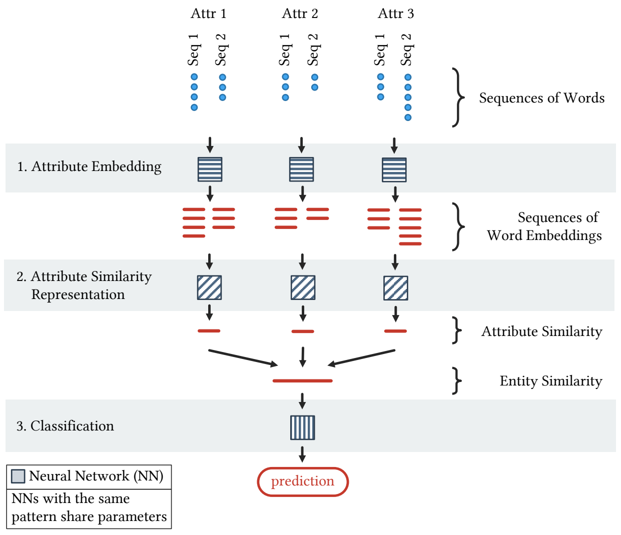


Figure 4: DeepMatcher Framework (Figure from [35])

GRU for summarizing an attribute for efficiency. Attention uses a decomposable attention model that can leverage information from aligned attributes of both tuples to summarize the attribute. Finally, the hybrid approach combines RNN and attention. DeepMatcher was evaluated on a diverse set of datasets such as structured, unstructured and noisy and provides useful rule of thumb on when to use DL for ER.

5.3 Representation Learning

A number of recent works such as DeepER [14], DeepMatcher [35] have applied a compositional approach using averaging or RNN/LSTMs for obtaining tuple embeddings from pre-trained word embeddings. For a number of benchmark datasets in entity resolution, this provides good results. There has been a number of recent efforts for learning distributed representations for data curation directly. Freddy [22] incorporated support for semantic similarity based queries by using pre-trained word embeddings inside Postgres. Termite [18] proposed an effective technique to learn a *common* (distributed) representation for both structured and unstructured data in an organization. In this approach, various entities such as rows, columns, and paragraphs are all represented as a vector. This unified representation allows Termite to identify related concepts even if they are spread across structured and unstructured data. Finally, EmbDC [4] proposed an interesting approach to learn embeddings for cells that combine ideas from word and node embeddings. It constructs a tripartite graph and performs random walks over them. The walks correspond to sentences that are passed to an algorithm that computes word embeddings. This allows for different permutations of the same data being outputted thereby partially incorporating the set semantics. The authors show that this two step approach provides promising results for unsupervised DC tasks such as schema matching and entity resolution.

5.4 Understanding the Success and What is Missing?

Both DeepER and DeepMatcher provide a mechanism to obtain tuple embeddings from pre-trained word embeddings. The similarity between the embeddings of two tuples is then used to train a DL classifier. In both cases, the classifier was generic and hence the state-of-the-art performance of these approaches could be attributed to learning effective representations for the tuples. A follow-up work [50] showed that such learned representations could improve the performance of even non-DL classifiers. [35] performed extensive empirical evaluation and found that deep learning based methods were especially effective for entity resolution involving textual and dirty data. For example, the use of word embeddings allowed the DL based approach to identify that 'Bill' and 'William' are semantically similar while no string similarity metric could do that. Learning such representations could result in some unexpected applications such as performing entity resolution between relations that are in different languages [3].

Something analogous happens in the data discovery scenario as well. The word embedding based approach learned effective representations that could show that certain concepts were relevant (*isoform* and *Protein* or *Pcr* and *assay* as mentioned above) that could not have been identified by any syntactic similarity measures such as edit distance. This ability to learn representations that could identify similar pair of words even if they are syntactically dissimilar explains the success of DL for schema matching in [34, 43].

From the above discussion, it is clear that learning effective representations was the cornerstone of improved performance of DL based methods. Such representations learn relationships that are hard to pull off using non-DL methods. However, these early works are only scratching the surface as they do not yet leverage powerful techniques such as task specific architectures or transfer learning. As an example, DeepMatcher proposed a template based architecture that has two key layers – composition/aggregation and similarity computation. This could be considered as a rudimentary task specific architecture that resulted in reduced training data. Recent work such as [50] and [26] seek to use transfer learning to improve ER. However, the days of using pre-trained ER models for multiple datasets is still far off. Even within representation learning, works have not yet explored the use of contextual embeddings such as ELMO or BERT.

6 TAMING DL'S HUNGER FOR DATA

Deep learning often requires a large amount of training data. Unfortunately, in domains such as DC, it is often prohibitively expensive to get high quality training data. In order for DL to be widely used to solve DC problems, we must overcome the problem of obtaining training data. Fortunately, there are a number of exciting innovations from DL that can be adapted to solve this issue. We highlight some of these promising ideas and discuss some potential research questions.

6.1 Unsupervised Representation Learning

While the amount of *supervised* training data is limited, most enterprises have substantial amount of *unsupervised* data in various relations and data lakes. These unlabeled data could be used to learn some generic patterns and representations that are then used train a DL model with relatively less labeled data. This technique is widely used in NLP where word embeddings are learned on large unlabeled corpus data such as Wikipedia and provide good performance for many downstream NLP tasks.

Research Opportunities.

- How can we perform a *holistic representation learning* over the enterprise data? How can the learned representations be used as features for downstream DC tasks such as entity matching, schema matching, etc? While there are some promising start in recent work such as [4, 51], more needs to be done.

6.2 Data Augmentation

Data augmentation increases the size of labeled training data without increasing the load on domain experts. The key idea is to apply a series of *label preserving transformations* over the existing training data. For an image recognition task, one can apply a transformations such as translation (moving the location of the dog/cat within the image), rotation (changing the orientation), shearing, scaling, changing brightness/color, and so on. Each of these operations does not change the label of the image (cat or dog) – yet generate additional synthetic training data.

Research Opportunities.

- *Label Preserving Transformations for DC.* What does label preserving transformations mean for DC? Is it possible to design an algebra of such transformations?
- *Domain Knowledge Aware Augmentation.* To avoid creating erroneous data, we could integrate domain knowledge and integrity constraints in the data augmentation process. This would ensure that we do not create tuples that say New York City is the capital of USA.
- *Domain Specific Transformations.* There are some recent approaches such as Tanda [40] that seek to learn domain specific transformations. For example, if one knows that *Country* $\rightarrow$ *Capital*, we can just swap the (Country, Capital) values of two tuples to generate two new tuples. In this case, even if the new tuple is not necessarily real, its label will be the same as the original tuple. Is it possible to develop an algebra for such transformations?

6.3 Synthetic Data Generation for DC

A seminal event in computer vision was the construction of ImageNet dataset [9] with many million images over thousands of categories. This was a key factor in the development of powerful DL algorithms. We believe that the DC community is in need of such an ImageNet moment.

Research Opportunities.

- *Benchmark for DC.* It is paramount to create a similar benchmark to drive research in DC (both DL and non-DL). While there has been some work for data cleaning such as BART [1], it is often limited to specific scenarios. For example, BART can be used to benchmark data repair algorithms where the integrity constraints are specified as denial constraints.
- *Synthetic Datasets.* If it is not possible to create an open-source dataset that has realistic data quality issues, a useful fall back is to create synthetic datasets that exhibit representative and realistic data quality issues. The family of TPC benchmarks involves a series of synthetic datasets that is somehow realistic and widely used for benchmarking database systems. A recent work [48] proposed a variational autoencoder based model for generating synthetic data that has similar statistical properties as the original data. Is it possible to extend that approach to encode data quality issues as well?

6.4 Weak Supervision

A key bottleneck in creating training data is that there is often an implicit assumption that it must be accurate. However, it is often infeasible to produce sufficient hand-labeled and accurate training data for most DC tasks. This is especially challenging for DL models that require a huge amount of training data. However, if one can relax the need for the veracity of training data, its generation will become much easier for the expert. The domain expert can specify a high level mechanism to generate training data without endeavoring to make it perfect.

Research Opportunities.

- *Weakly Supervised DL Models.* There has been a series of research (such as Snorkel [39]) that seek to *weakly supervise* ML models and provide a convenient programming mechanism to specify "mostly correct" training data. What are the DC specific mechanisms and abstractions for weak supervision? Can we automatically create such a weak supervision through data profiling?

6.5 Transfer Learning

Another trick to handle limited training data is to "transfer" representations learned in one task to a different yet related task. For example, DL models for computer vision tasks are often trained on ImageNet [9], a dataset that is commonly used for image recognition purposes. However, these models could be used for tasks that are not necessarily image recognition. This approach of training on a large diverse dataset followed by tuning for a local dataset and tasks has been very successful.

Research Opportunities.

- *Transfer learning.* What is the equivalent of this approach for DC? Is it possible to train on a single large dataset such that it could be used for many downstream DC tasks? Alternatively, is it possible to train on a single dataset and for a single task (such as entity matching) such that the learned representations could be used for entity matching in similar domains? How can we train a DL model for one task and tune the model for new tasks by using the limited labeled data instead of starting from scratch?
- *Pre-trained Models.* Is it possible to provide pre-trained models that have been trained on large, generic datasets? These models could then be tuned by individual practitioners in different enterprises.

7 DEEP LEARNING: MYTHS AND CONCERNS

In the past few years, DL has achieved substantial successes in many areas. However, DC has a number of characteristics that are quite different from prior domains where DL succeeded. We anticipate that applying DL to challenging real-world DC tasks can be messy. We now describe some of the concerns that could be raised by a pragmatic DC practitioner or a skeptical researcher.

7.1 Deep Learning is Computing Heavy

A common concern is that training DL models requires exorbitant computing resources where model training could take days even on a large GPU cluster. In practice, training time often depends on the model complexity, such as the number of parameters to be learnt, and the size of training data. There are many tricks that can reduce the amount of training time. For example, a task-aware DL architecture often requires substantially less parameters to be learned. Alternatively, one can "transfer" knowledge from a pre-trained model from a related task or domain and the training time will now be proportional to the amount of fine-tuning required to customize the model to the new task. For example, DeepER [14] leveraged word embeddings from GloVe (whose training can be time consuming) and built a light-weight DL model that can be trained in a matter of minutes even on a CPU. Finally, while training could be expensive, this can often be done as a pre-processing task. Prediction using DL is often very fast and comparable to that of other ML models.

7.2 Data Curation Tasks are Too Messy or Too Unique

DC tasks often require substantial domain knowledge and a large dose of "common sense". Current DL models are very narrow in the sense that they primarily learn from the correlations present in the training data. However, it is quite likely that this might not be sufficient. Unsupervised representation learning over the entire enterprise data can only partially address this issue. Current DL models are often not amenable to encoding domain knowledge in general as well as those that are specific to DC such as data integrity constraints. As mentioned before, substantial amount of new research on DC-aware DL architectures is needed. However, it is likely that DL, even in its current form, can reduce the work of domain experts.

DC tasks often exhibit a *skewed label distribution.* For the task of entity resolution (ER), the number of non-duplicate tuple pairs are orders of magnitude larger than the number of duplicate tuple pairs. If one is not careful, DL models can provide inaccurate results. Similarly, other DC tasks often exhibit *unbalanced cost model* where the cost of misclassification is not symmetric. Prior DL work utilizes a number of techniques to address these issues such as (a) cost sensitive models where the asymmetric misclassification costs are encoded into the objective function, and (b) sophisticated sampling approach where we under or over sample certain classes of data. For example, DeepER [14] samples non-duplicate tuple pairs that are abundant at a higher level than duplicate tuple pairs.

7.3 Deep Learning Predictions are Inscrutable

Domain experts could be concerned that the predictions of DL models are often uninterpretable. Deep learning models are often very complex and the black-box predictions might not be explainable by even DL experts. However, explaining why a DL model made a particular data repair is very important for a domain expert. Recently, there has been intense interest in developing algorithms for explaining predictions of DL models or designing interpretable DL models in general. Please see [21] for an extensive survey. Designing algorithms that can explain the prediction of DL models for DC tasks is an intriguing research problem. While there are some promising approaches exist for specific tasks such as Entity resolution [11, 13, 49], more research remains.

7.4 Deep Learning can be Easily Fooled

There exist a series of recent works which show that DL models (especially for image recognition) can be easily fooled by perturbing the images in an adversarial manner. The sub-field of adversarial DL [37, 47] studies the problem of constructing synthetic examples by slightly modifying real examples from training data such that the trained DL model (or any ML model) makes an incorrect prediction with high confidence. While this is indeed a long term concern, most DC tasks are often collaborative and limited to an enterprise. Furthermore, there are a series of research efforts that propose DL models that are more resistant to adversarial training such as [30].

7.5 Building Deep Learning Models for Data Curation is "Just Engineering"

Many DC researchers might look at the process of building DL models for DC and simply dismiss it as a pure engineering effort. And they are indeed correct! Despite its stunning success, DL is still at its infancy and the theory of DL is still being developed. To a DC researcher used to purveying a well organized garden of conferences such as VLDB/SIGMOD/ICDE, the DL research landscape might look like the wild west.

In the early stages, researchers might *just apply* an existing DL model or algorithm for a DC task. Or they might slightly tweak a previous model to achieve better results. We argue that database conferences must provide a safe zone in which these DL explorations are conducted in a principled manner. One could take inspiration from the computer vision community. They created a large dataset (ImageNet [9]) that provided a benchmark by which different DL architectures and algorithms can be compared. They also created one or more workshops focused on applying DL for specific tasks in computer vision (such as image recognition and scene understanding). The database community has its own TPC series of synthetic datasets that have been influential in benchmarking database systems. Efforts similar to TPC are essential for the success of DL-driven DC.

7.6 Deep Learning is Data Hungry

Indeed, this is one of the major issues in adopting DL for DC. Most classical DL architectures often have many hidden layers with millions of parameters to learn, which requires a large amount of training data. Unfortunately, the amount of training data for DC is often small. The good news is, there exist a wide variety of techniques and tricks in the DL's arsenal that can help address this issue. We could leverage the techniques described in Section 6 for addressing these issues.

8 A CALL TO ARMS

In this paper, we make two key observations. Data Curation is a long standing problem and needs novel solutions in order to handle the emerging big data ecosystem. Deep Learning is gaining traction across many disciplines, both inside and outside computer science. The meeting of these two disciplines will unleash a series of research activities that will lead to usable solutions for many DC tasks. We identified research opportunities in learning distributed representations for database aware objects such as tuples or columns and designing DC-aware DL architectures. We described a number of promising approaches to tame DL's hunger for data. We discuss the early successes in using DL for important DC tasks such as data discovery and entity matching that required novel adaptations of DL techniques. We make a call to arms for the database community in general, and the DC community in particular, to seize this opportunity to significantly advance the area, while keeping in mind the risks and mitigation that were also highlighted in this paper.

REFERENCES

[1] P. C. Arocena, B. Glavic, G. Mecca, R. J. Miller, P. Papotti, and D. Santoro. Messing up with BART: error generation for evaluating data-cleaning algorithms. *PVLDB*, 2015.

[2] L. Berti-Equille, H. Harmouch, F. Naumann, N. Novelli, and S. Thirumuruganathan. Discovery of genuine functional dependencies from relational data with missing values. *Proceedings of the VLDB Endowment*, 11(8):880–892, 2018.

[3] Ö. Ö. Çakal, M. Mahdavi, and Z. Abedjan. Clrl: Feature engineering for cross-language record linkage. In *EDBT*, pages 678–681, 2019.

[4] R. Cappuzzo, P. Papotti, and S. Thirumuruganathan. Local embeddings for relational data integration. *arXiv preprint arXiv:1909.01120*, 2019.

[5] F. Chollet. *Deep learning with Python.* Manning Publications, 2018.

[6] P. Cui, X. Wang, J. Pei, and W. Zhu. A survey on network embedding. *CoRR*, abs/1711.08752, 2017.

[7] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: a commodity data cleaning system. In *SIGMOD*, 2013.

[8] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang. The data civilizer system. In *CIDR*, 2017.

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[11] V. Di Cicco, D. Firmani, N. Koudas, P. Merialdo, and D. Srivastava. Interpreting deep learning models for entity resolution: An experience report using lime. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '19, pages 8:1–8:4, New York, NY, USA, 2019. ACM.

[12] A. Doan, A. Y. Halevy, and Z. G. Ives. *Principles of Data Integration.* Morgan Kaufmann, 2012.

[13] A. Ebaid, S. Thirumuruganathan, W. G. Aref, A. Elmagarmid, and M. Ouzzani. Explainer: Entity resolution explanations. In *ICDE*, pages 2000–2003. IEEE, 2019.

[14] M. Ebraheem, S. Thirumuruganathan, S. R. Joty, M. Ouzzani, and N. Tang. Deeper - deep entity resolution. *VLDB*, 12, 2018.

[15] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.

[16] W. Fan and F. Geerts. *Foundations of Data Quality Management.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[17] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(2), 2008.

[18] R. C. Fernandez and S. Madden. Termite: a system for tunneling through heterogeneous data. *arXiv preprint arXiv:1903.05008*, 2019.

[19] R. C. Fernandez, E. Mansour, A. A. Qahtan, A. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang. Seeping semantics: Linking datasets using word embeddings for data discovery. In *ICDE*, 2018.

[20] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* MIT Press, 2016. http://www.deeplearningbook.org.

[21] R. Guidotti, A. Monreale, F. Turini, D. Pedreschi, and F. Giannotti. A survey of methods for explaining black box models. *arXiv:1802.01933*, 2018.

[22] M. Günther. Freddy: Fast word embeddings in database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1817–1819. ACM, 2018.

[23] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive interaction for data transformation. In *CIDR*, 2015.

[24] G. E. Hinton. Learning distributed representations of concepts. In *CogSci*, volume 1, page 12. Amherst, MA, 1986.
[25] X. Huang, Y. Peng, and M. Yuan. Cross-modal common representation learning by hybrid transfer network. In *IJCAI*, 2017.
[26] J. Kasai, K. Qian, S. Gurajada, Y. Li, and L. Popa. Low-resource deep entity resolution with transfer and active learning. *arXiv preprint arXiv:1906.08042*, 2019.
[27] P. Konda, S. Das, P. S. G. C., A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. F. Naughton, S. Prasad, G. Krishnan, R. Deep, and V. Raghavendra. Magellan: Toward building entity matching management systems. *PVLDB*, 9(12):1197–1208, 2016.
[28] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. In *ICML*, 2014.
[29] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.
[30] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv:1706.06083*, 2017.
[31] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.
[32] R. J. Miller. Big data curation. In *COMAD*, page 4, 2014.
[33] R. J. Miller, F. Nargesian, E. Zhu, C. Christodoulakis, K. Q. Pu, and P. Andritsos. Making open data transparent: Data discovery on open data. *IEEE Data Eng. Bull.*, 41(2):59–70.
[34] M. J. Mior, I. Ororbia, and G. Alexander. Column2vec: Structural understanding via distributed representations of database schemas. *arXiv preprint arXiv:1903.08621*, 2019.
[35] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep learning for entity matching: A design space exploration. In *SIGMOD*, 2018.
[36] F. Nargesian, E. Zhu, K. Q. Pu, and R. J. Miller. Table union search on open data. *PVLDB*, 11(7):813–825, 2018.
[37] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *EuroS&P*, 2016.
[38] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.
[39] A. Ratner, S. H. Bach, H. Ehrenberg, J. Fries, S. Wu, and C. Ré. Snorkel: Rapid training data creation with weak supervision. *arXiv:1711.10160*, 2017.
[40] A. J. Ratner, H. R. Ehrenberg, Z. Hussain, J. Dunnmon, and C. Ré. Learning to compose domain-specific transformations for data augmentation. In *NIPS*, 2017.
[41] S. Ravanbakhsh, J. Schneider, and B. Poczos. Deep learning with sets and point clouds. *arXiv preprint arXiv:1611.04500*, 2016.
[42] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11), 2017.
[43] R. Shraga, A. Gal, and H. Roitman. Deep smane-deep similarity matrix adjustment and evaluation to improve schema matching. Technical report, Technical Report IE/IS-2019-02. Technion–Israel Institute of Technology âĂę, 2019.
[44] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. Data curation at scale: The data tamer system. In *CIDR*, 2013.
[45] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. Data curation at scale: The data tamer system. In *CIDR*, 2013.
[46] M. Stonebraker and I. F. Ilyas. Data integration: The current status and the way forward. *IEEE Data Eng. Bull.*, 41(2):3–9, 2018.
[47] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv:1312.6199*, 2013.
[48] S. Thirumuruganathan, S. Hasan, N. Koudas, and G. Das. Approximate query processing using deep generative models. *arXiv preprint arXiv:1903.10000*, 2019.
[49] S. Thirumuruganathan, M. Ouzzani, and N. Tang. Explaining entity resolution predictions: Where are we and what needs to be done? 2019.
[50] S. Thirumuruganathan, S. A. P. Parambath, M. Ouzzani, N. Tang, and S. Joty. Reuse and adaptation for entity resolution through transfer learning. *arXiv preprint arXiv:1809.11084*, 2018.
[51] R. Wu, S. Chaba, S. Sawlani, X. Chu, and S. Thirumuruganathan. Autoer: Automated entity resolution using generative modelling. *arXiv preprint arXiv:1908.06049*, 2019.
[52] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola. Deep sets. In *Advances in neural information processing systems*, pages 3391–3401, 2017.

Fairness in Clustering with Multiple Sensitive Attributes

Savitha Sam Abraham[•] Deepak P[†,•] Sowmya S Sundaram[•]
[•]Indian Institute of Technology Madras, India
[†]Queen's University Belfast, UK
savithas@cse.iitm.ac.in deepaksp@acm.org sowmya@cse.iitm.ac.in

ABSTRACT

A clustering may be considered as fair on pre-specified sensitive attributes if the proportions of sensitive attribute groups in each cluster reflect that in the dataset. In this paper, we consider the task of fair clustering for scenarios involving multiple multi-valued or numeric sensitive attributes. We propose a fair clustering method, *FairKM* (Fair K-Means), that is inspired by the popular K-Means clustering formulation. We outline a computational notion of fairness which is used along with a cluster coherence objective, to yield the FairKM clustering method. We empirically evaluate our approach, wherein we quantify both the quality and fairness of clusters, over real-world datasets. Our experimental evaluation illustrates that the clusters generated by FairKM fare significantly better on both clustering quality and fair representation of sensitive attribute groups compared to the clusters from a state-of-the-art baseline fair clustering method.

1 INTRODUCTION

Clustering is the task of grouping a dataset of objects in such a way that objects that are assigned to the same group, called a *cluster*, are more similar to each other than those in other groups/clusters. Clustering [10] is a well-studied and fundamental task, arguably the most popular task in unsupervised or exploratory data analytics. A pragmatic way of using clustering within analytics pipelines is to consider objects within the same cluster as being indistinguishable. Customers in the same cluster are often sent the same promotional material in a targeted marketing scenario in retail, whereas candidates clustered using their resumes may be assigned the same shortlisting decision in a hiring scenario. Clustering provides a natural way to tackle the infeasibility of doing manual per-object assessment or appreciation, especially in cases where the dataset in question encompasses more than a few hundreds of objects. Central to clustering is the notion of similarity which may need to be defined in a task-oriented manner. As an example, clustering to aid a task on identifying tax defaulters may use a similarity measure that focuses more on the job and income related attributes, whereas that for a health monitoring task may more appropriately focus on a very different set of attributes.

Usage of clustering algorithms in analytics pipelines for making important decisions open up possibilities of unfairness. Among two high-level streams of fairness constructs, viz., individual fairness [8] and group fairness [12], we focus on the latter. Group fairness considers fairness from the perspective of *sensitive attributes* such as gender and ethnicity and groups defined on the basis of such sensitive attributes. Consider a clustering algorithm that targets to group records of individuals to clusters; it is possible and likely that certain clusters have highly skewed distributions when profiled against particular sensitive attributes. As an example, clustering a dataset with broad representation across gender groups based on exam scores could lead to clusters that are highly gender-homogeneous due to statistical correlations[1]; this would happen even if the gender attribute were not explicitly considered within the clustering, since such information could be held implicitly across one or more other attributes. Choosing a cluster with a high gender or ethnic skew for positive (e.g., interview shortlisting) or negative (e.g., higher scrutiny or checks) action entails differentiated impact across gender and ethnic groups. This could also lead to reinforcement of societal stereotypes. For example, consistently choosing individuals from particular ethnic groups for pro-active surveillance could lead to higher reporting of violations from such ethnic groups since enhanced surveillance translates to higher crime visibility, resulting in higher reporting rates. This reporting skew results thus manifests as a data skew which provides opportunities for future invocations of the same analytics to exhibit higher ethnic asymmetry. In modern decision making scenarios within a liberal and democratic society, we need to account for a plurality of sensitive attributes within analytics pipelines to avoid causing (unintended) discrimination; these could include gender, race, religion, relationship status and country of origin in generic decision-making scenarios, and could potentially include attributes such as age and income in more specific ones. There has been much recent interest in the topic of fair clustering [1, 6].

Our Contribution. In this paper, we consider the task of fair clustering in the presence of multiple sensitive attributes. As will be outlined in a later section, this is a direction that has received less attention amidst existing work on fair clustering that has been designed for scenarios involving a single binary sensitive attribute [3, 6, 14, 17], single multi-valued/categorical sensitive attribute [1, 20–22], or multiple overlapping groups [4]. We propose a clustering formulation and algorithm to embed group fairness in clustering that incorporates multiple sensitive attributes that may include numeric, binary or multi-valued (i.e., categorical) sensitive ones. Through an empirical evaluation over multiple datasets, we illustrate the empirical effectiveness of our approach in generating clusters with fairer representations of sensitive groups, while preserving cluster meaningfulness.

2 RELATED WORK

Fairness in machine learning has received significant attention over the last few years. Our work contributes to the area of fair methods for unsupervised learning tasks. We now briefly survey the area of fairness in unsupervised learning, with a focus

[1]https://www.compassprep.com/new-sat-has-disadvantaged-female-testers/

on clustering, our task of interest. We interchangeably refer to groups defined on sensitive attributes (e.g., *ethnic groups*, *gender groups* etc.) as *protected classes* for consistency with terminology in some referenced papers. Towards developing a systematic summary, we categorize prior work into three types depending on whether the fairness modelling appears as a (i) pre-processing step, (ii) during the task of clustering, or (iii) as a post-processing step after clustering. These yield the three technique families.

2.1 Space Transformation Approaches

The family of fairness pre-processing techniques work by first representing the data points in a *'fair'* space followed by applying any existing clustering algorithm upon them. This is the largest family of techniques, and most approaches in this family seek to achieve theoretical guarantees on representational fairness. This family includes one of the first works in this area of fair clustering [6]. The work proposes a fair variant of classical clustering for scenarios involving a single sensitive attribute that can take binary values. Let each object be colored either x or y depending on its value for the binary sensitive attribute. [6] defines fairness in terms of *balance* of a cluster, which is $min\{\#x/\#y, \#y/\#x\}$. They go on and outline the concept of (b, r)*-fairlet decomposition*, where the points in the dataset are grouped into small clusters called fairlets, such that each fairlet would have a balance of b/r, where $b < r$. The clustering is then performed on these fairlets. The fairness guarantees that are provided by the clustering is based on the balance in the underlying fairlets. Fairlet decomposition turns out to be NP-hard, for which an approximation algorithm is provided. Later, [3] proposed a faster fairlet decomposition algorithm offering significant speedups. The work in [20] extends the fairlet idea to K-means for scenarios with a single multi-valued sensitive attribute. They define fair-coresets which are smaller subsets of the original dataset, such that solving fair clustering over this subset also results in giving an approximate solution for the entire dataset.

Another set of fair space transformation approaches build upon methods for dimensionality reduction and representation learning. A recent such work [2] considers the bias in the dataset (that is, representational skew) as a form of noise and uses spectral de-noising techniques to project the original points in the dataset to a new fair projected space. [17] describes a fair version of PCA (Principal Component Analysis) for data with a single binary sensitive attribute. A representation learning method may be defined as fair if the information about the sensitive attributes cannot be inferred from the learnt representations. The method uses convex optimization formulations to embed fairness within PCA. The fairness is specified in terms of failure of the classifiers in predicting the sensitive class of the dimensionality-reduced data points obtained from fair PCA. The method is fair*er* if the data points are less distinguishable with respect to their values of the sensitive attribute in this lower dimension space. Another work that projects the original data points into a fair space is the one described in [21]. This method, which is for cases involving a single multi-valued sensitive attribute, defines a clustering to be fair when there is an equal number of data points of each protected class in each cluster. They define the concept of *fairoids* (short for *fair centroids*) which are formed by grouping together all points belonging to the same protected class. The task is then to learn a latent representation of the data points such that the cluster centroids obtained after clustering on this latent representation are equi-distant from every fairoid.

2.2 Fairness Modelling within Optimization

Methods in this family incorporate the fairness constraints into the clustering step, most usually within the objective function that the clustering method seeks to optimize for. It may be noted that the method we propose in this paper, *FairKM*, also belongs to this family. Approaches within this family define a clustering to be fair if the proportional representation of the protected class in a cluster reflects that in the dataset. One of the techniques [14] describes a fair variant of spectral clustering where a linear fairness constraint is incorporated into the original ratio-cut minimization objective of spectral clustering. Another recent technique [22], the method that comes closest to ours in construction, modifies K-means clustering to add a fairness loss term. The fairness loss is computed as the KL-divergence between the probability distribution across the different values for the sensitive attribute in a cluster, and the corresponding distribution for the whole dataset. This method is designed for a single multi-valued sensitive attribute and does not generalize to multiple such sensitive attributes. *Being closest to our proposed method in spirit, we use this method as our primary baseline, in the experiments.*

In contrast to the above, another recent work [5] outlines a different notion of fairness, one that is independent of (and agnostic to) sensitive attributes. They define fairness as proportionality wrt spatial distributions, to mean that any (n/k) points can form their own cluster if there exists another center that is closer to each of these (n/k) points. This proportionality constraint is incorporated into the objective function of k-median clustering and is optimized to find a clustering that satisfies this constraint.

2.3 Cluster Perturbation Techniques

In this third family of techniques, vanilla clustering is first applied on the original set of data points, after which the generated clusters are perturbed to improve fairness of the solution. In [4], fairness is defined in terms of a lower and upper bound on the representation of a protected class in a cluster. This method is for cases with multiple binary sensitive attributes, referred to as *overlapping groups* in the paper. The k-centers generated from vanilla clustering on the data points are used to perform a fair partial assignment between points and the centers. The fair partial assignment is formulated as a linear program with constraints that ensures that the sum of the weights associated with a point's partial assignments is one, and, the representation of a protected class in a cluster is within the specified upper and lower bounds. The partial assignments are then converted to integral assignments by framing it as another linear programming problem. [1] also uses a similar idea, but it just enforces an upper bound, consequently preventing over-representation of specific groups in a cluster. The work described in [13] proposes a simple approximation algorithm for k-center clustering under a fairness constraint, for scenarios with a single multi-valued sensitive attribute. The method targets to generate a fair summary of a large set of data points, such that the summary is a representative subset of the original dataset. For example, if the original dataset has a 70:30 male:female distribution, then a fair summary should also have the same distribution.

2.4 Summary

Table 1 summarizes the different approaches in literature and our proposed approach *FairKM*, in terms of the number and type of sensitive attributes they handle and their definition for fairness. As it may be seen from the table, there has been very limited

Paper	Number	Type	Fairness Definition
[6],[3],[2]	Single	Binary	Preserve proportional representation of protected classes within clusters.
[20]	Single	Multi-valued	Preserve proportional representation of protected classes within clusters.
[17]	Single	Binary	The accuracy of the classifier predicting the protected class of a data point should be within a specified bound.
[21]	Single	Multi-valued	Each cluster should have an equal number of data points from each protected class.
[4]	Multiple	Binary	The proportional representation of a protected class in a cluster should be within the specified lower and upper bounds.
[1]	Single	Multi-valued	The proportional representation of a protected class in a cluster should not go beyond a specified upper bound.
[13]	Single	Multi-valued	The clustering should produce pre-specified number of cluster centers belonging to each specific protected class.
[14], [22]	Single	Multi-valued	Preserve proportional representation of protected classes within clusters.
[5]	-	-	There are no set of (n/k) points such that there exists another center that is closer to each of these (n/k) points.
[18]	Single	Multi-valued	Each cluster should have atleast a pre-specified number of points of a protected class.
FairKM	Multiple	Multi-valued/ Numeric	Preserve proportional representation of protected classes within clusters. as its representation in the whole dataset.

Table 1: Fair Unsupervised ML Methods indicating the Number and Type of Sensitive Attributes they are designed for.

exploration into methods that admit multiple multi-valued (*aka* categorical or multi-state) sensitive attributes, the space that *FairKM* falls in. While multiple multi-valued attributes can be treated as together forming a giant multi-valued attribute taking values that are combinations of the component attributes, this results in a large number of very fine-grained groupings. These make it infeasible to both (i) impose fairness constraints over, and (ii) ensure parity in treatment of different sensitive attributes independent of the differences in the number of values they take. Considering the fact that real-life scenarios routinely present with multiple sensitive attributes, *FairKM*, we believe addresses an important line of inquiry in the backdrop of the literature. We will empirically evaluate *FairKM* against [22], the latter coming from the same technique family and having similar construction.

3 PROBLEM DEFINITION

Let $\mathcal{X} = \{\ldots, X, \ldots\}$ be a dataset of records defined over two sets of attributes $\mathcal{N}$ and $\mathcal{S}$. $\mathcal{N}$ stands for the set of attributes that are relevant to the task of interest, and thus may be regarded non-sensitive. As examples, this may comprise experience and skills in the case of screening applicants for a job application, and attributes from medical wearables' data to inform decision making for pro-active screening. $\mathcal{S}$ stands for the set of *sensitive attributes*, which would typically include attributes such as those identifying *gender*, *race*, *religion*, *relationship status* in a citizen database and any other sensitive attribute over which fairness is to be ensured. In other scenarios such as NLP for education, representational fairness may be sought over attributes such as types of problems in a word problem database; this forms one of the scenarios in our empirical evaluation.

The (vanilla) clustering objective would be to group $\mathcal{X}$ into clusters such that it maximizes intra-cluster similarity and minimizes inter-cluster similarity, similarity gauged using the task-relevant attributes in $\mathcal{N}$. Within a fair clustering, we would additionally like the output clusters to be *fair* on attributes in $\mathcal{S}$. A natural way to operationalize fairness within a clustering that covers all objects in $\mathcal{X}$ would be to ensure that the distribution of groups defined on sensitive attributes within each cluster approximates the distribution across the dataset; this correlates with the well-explored notion of *statistical parity* [8] in fair supervised learning. For example, suppose the sex ratio in $\mathcal{X}$ is 1:1; we would ideally like each cluster in the clustering output to report a sex ratio of 1:1, or very close to it. In short, we would like a fair clustering to produce clusters, each of which are both:

- coherent when measured on the attributes in $\mathcal{N}$, *and*
- approximate the dataset distribution when measured on the attributes in $\mathcal{S}$.

It may be noted that simply *hiding* the $\mathcal{S}$ attributes from the clustering algorithm does not suffice. A *gender blind* clustering algorithm may still produce clusters that are highly gender-homogeneous, since some attributes in $\mathcal{N}$ could implicitly encode gender information. Indeed, we would like a fair clustering to surpass $\mathcal{S}$-blind clustering by significant margins on fairness.

4 FAIRKM: OUR METHOD

We now describe our proposed technique for fair clustering, code-named *FairKM*, short for Fair K-Means, indicating that it draws inspiration from the classical K-Means clustering algorithm [9, 16]. FairKM incorporates a novel fairness loss term that nudges the clustering towards fairness on attributes in $\mathcal{S}$. The FairKM objective function is as follows:

$$O = \underbrace{\sum_{C \in \mathcal{C}} \sum_{X \in C} dist_{\mathcal{N}}(X, C)}_{\text{K-Means Term over attributes in } \mathcal{N}} \quad + \lambda \quad \underbrace{deviation_{\mathcal{S}}(\mathcal{C}, \mathcal{X})}_{\text{Fairness Term over attributes in } \mathcal{S}} \tag{1}$$

As indicated, the objective function comprises two components; the first is the usual K-Means loss for the clustering $\mathcal{C}$, $dist_{\mathcal{N}}(X, C)$ computing the distance between X and prototype of cluster C, distance computed only over attributes in $\mathcal{N}$. The second is a fairness loss term we introduce, which is computed over attributes in $\mathcal{S}$. λ is a parameter that may be used to balance the relative strength of the two terms. As in K-Means, this loss is computed over a given clustering; the task is thus to identify a clustering that minimizes O as much as possible. We now describe the details of the second term, and the intuitions behind its construction.

4.1 The Fairness Term in FairKM

While the K-Means term in the *FairKM* objective tries to ensure that the output clusters are coherent in the $\mathcal{N}$ attributes, the fairness term performs the complementary function of ensuring that the clusters manifest fair distributions of groups defined by sensitive attributes in $\mathcal{S}$. We outline the motivation and construction of the fairness term herein.

Attribute-Specific Deviation for a Cluster: Consider a single sensitive attribute S (e.g., gender) among the set of sensitive attributes $\mathcal{S}$. For each data object X, S may take on one value from a set of permissible values. Let s be one such value (e.g., female, for the choice of S as gender). For an ideally fair cluster C, one would expect that the fractional representation of s in C - the fraction of objects in C that take the value s for S - to be close to the fractional representation of s in $\mathcal{X}$. With our intent of generating clusters that are as fair as possible, we seek to generate clusterings such that the deviation between the fractional representations of s in C and $\mathcal{X}$ are minimized for each cluster C. For a given cluster C and a choice of value s for S, we model the deviation as simply the square of the absolute differences between the fractional representations in C and $\mathcal{X}$:

$$D_C^S(s) = \left(\frac{|\{X | X \in C \wedge X.S = s\}|}{|C|} - \frac{|\{X | X \in \mathcal{X} \wedge X.S = s\}|}{|\mathcal{X}|} \right)^2 \tag{2}$$

The deviation, when aggregated over all values of S, yields:

$$D_C^S = \begin{cases} \sum_{s \in Values(S)} D_C^S(s) & |C| \neq 0 \\ 0 & |C| = 0 \end{cases} \tag{3}$$

The above aggregation accounts for the fact that $D_C^S(s)$ is undefined when C is an empty cluster.

Domain Cardinality Normalization: Different sensitive attributes may have different numbers of permissible values (or domain cardinalities). For example, *race* and *gender* attributes typically take on much fewer values than *country of origin*. Those attributes with larger domains are likely to yield larger D_C^S scores because, (i) the deviations are harder to control within (small) clusters given the higher likely scatter, and (ii) there are larger numbers of $D_C^S(s)$ terms that add up to D_C^S. In order to ensure that attributes with larger domains do not dominate the fairness term, we normalize the deviation by the number of different values taken by an attribute, yielding ND_C^S, a normalized attribute-specific deviation:

$$ND_C^S = \frac{D_C^S}{|Values(S)|} \tag{4}$$

This is then summed up over all attributes in $\mathcal{S}$ to yield a single term for each cluster:

$$ND_C = \sum_{S \in \mathcal{S}} ND_C^S \tag{5}$$

Cluster Weighting: Observe that ND_C deviation loss would tend towards 0.0 for very large clusters, since they are obviously likely to reflect dataset-level distributions better; further, an empty cluster would also have $ND_C^S = 0$ by definition. Considering the above, a clustering loss term modelled as a simple sum over its clusters, $\left(\sum_{C \in \mathcal{C}} ND_C\right)$ or a cardinality weighted sum, $\left(\sum_{C \in \mathcal{C}} |C| \times ND_C\right)$, can both be driven towards 0.0 by keeping a lot of clusters empty, and distributing the dataset objects across very few clusters; the boundary case would be a single non-empty cluster. Indeed, this propensity towards the boundary condition is kept in check by the K-Means term; however, we would like our fairness term to drive the search towards more reasonable fair clustering configurations in lieu of simply reflecting a propensity towards highly skewed clustering configurations.

Towards achieving this, we weight each cluster's deviation by the square of it's fractional cardinality of the dataset. This leads to an overall loss term as follows:

$$\sum_{C \in \mathcal{C}} \left(\frac{|C|}{|\mathcal{X}|} \right)^2 \times ND_C \tag{6}$$

The squared term in the weighting enlarges the ND_C terms of larger clusters much more than smaller ones, making it unprofitable to create large clusters; this compensates for the propensity towards skewed clusters as embodied in the loss construction.

Overall Loss: The overall fairness loss term is thus:

$$deviation_{\mathcal{S}}(\mathcal{C}, \mathcal{X}) =$$

$$\sum_{C \in \mathcal{C}} \left(\frac{|C|}{|\mathcal{X}|} \right)^2 \times \sum_{S \in \mathcal{S}} \frac{\sum_{s \in Values(S)} \left(Fr_C^S(s) - Fr_{\mathcal{X}}^S(s) \right)^2}{|Values(S)|} \tag{7}$$

where $Fr_C^S(s)$ and $Fr_{\mathcal{X}}^S(s)$ are shorthands for the fractional representation of $S = s$ objects in C and $\mathcal{X}$ respectively.

4.2 The Optimization Approach

Having defined the objective function, we now outline the optimization approach. It is easy to observe that there are three sets of parameters, the clustering assignments for each data object in $\mathcal{X}$, the cluster prototypes that are used in the first term of the objective function, and the fractional representations, i.e., $Fr_C^S(s)$s, used in the fairness term. Unlike K-Means, given the more complex construction, it is harder to form a closed-form solution for the cluster assignments. Thus, from a given estimate of all three sets of parameters, we step over each data object $X \in \mathcal{X}$ in round-robin fashion, updating its cluster assignment, and making consequent changes in cluster prototypes and fractional representations. One set of round-robin updates forms an iteration, with multiple such iterations performed until convergence or until a maximum threshold of iterations is reached.

4.2.1 *Cluster Assignment Updates.*

At $\lambda = 0$, *FairKM* defaults to K-Means where the cluster assignments are determined only by proximity to the cluster prototype (over attributes in $\mathcal{N}$). At higher values of λ, *FairKM* cluster assignments are increasingly swayed by considerations of representational fairness of $\mathcal{S}$ attributes within clusters.

It may be noted that the cluster assignments are used in both the terms of the *FairKM* objective, in different ways. This makes a closed form estimation of cluster assignments harder to arrive at. This leads us to a round-robin approach of determining cluster assignments. When each X is considered, the cluster prototypes as well as the current cluster assignments of all other objects, i.e. $\mathcal{X} - \{X\}$, are kept unchanged. The cluster assignment for X is then estimated as:

$$Cluster(X) = \arg\min_{C} \; O_{\mathcal{C}+(X \in C)} \tag{8}$$

For the candidate object X, we evaluate the value of the objective function by changing X's cluster membership from the present one to each cluster, $C + (X \in C)$ indicating a corresponding change in the clustering configuration retaining all other objects' present cluster assignments. X is then assigned to the cluster for which the minimum value of O is achieved. While this may look as a simple step, implementing it naively is computationally expensive. However, easy optimizations are possible when one observes the dynamics of the change and how it operates across the two terms. We now outline a simple way of affecting the cluster assignment decision. First, let X's current cluster assignment be C'; the cluster assignment step can be equivalently written as:

$$Cluster(X) = \arg\min_{C} \delta(O)_{X \in C' \rightarrow X \in C} \tag{9}$$

where δO indicates the change in O when the respective cluster assignment change is carried out. This can be expanded into the changes in the two terms in the objective function as follows:

$$\delta(O)_{X \in C' \rightarrow X \in C} = \\ \delta(\text{K-Means term})_{X \in C' \rightarrow X \in C} + \lambda \times \delta(\text{deviation term})_{X \in C' \rightarrow X \in C} \tag{10}$$

We now detail the changes in the respective terms separately. **Change in K-Means Term:** We now outline the change in the K-Means term by moving X from C' to C. As may be obvious, this depends only on attributes in $\mathcal{N}$. We model the cluster prototypes as simply the average of the objects within the cluster. The change in the K-Means term is the sum of (i) the change in the K-Means term corresponding to C' brought about by the exclusion of X from it, and (ii) the change in the K-Means term corresponding to C brought about by the inclusion of X within it. We discuss them below.

Consider a single numeric attribute $N \in \mathcal{N}$, for simplicity. Through excluding X from C', the N attribute value of the cluster prototype of C' undergoes the following change:

$$C'.N \rightarrow \left[\left(C'.N - \frac{X.N}{|C'|}\right) \times \frac{|C'|}{|C' - 1|}\right] \tag{11}$$

where C' is overloaded to refer to the cluster and the cluster prototype (to avoid clutter), all values referring to those prior to exclusion of X. The term after the $\rightarrow$ stands for the N attribute value for the new cluster prototype. As indicated, it is computed by the removal of the contribution from X from the cluster prototype, followed by re-normalization, now that C' has one fewer object within it. The change in the K-Means term for N corresponding to C' is then as follows:

$$\delta_{X_{out}} KM(C', N) = \left(\sum_{X' \in C', X' \neq X} (X'.N - New(C'.N))^2\right) - \\ \left[\left(\sum_{X' \in C', X' \neq X} (X'.N - C'.N)^2\right) + (X.N - C'.N)^2\right] \tag{12}$$

where $New(C'.N)$ is the new estimate of $C'.N$ as outlined in Eq. 11. The first term corresponds to the K-Means loss in the new configuration (after exclusion of X), whereas the sum of the second and third terms correspond to that prior to exclusion of X. Analogous to the above, the new centroid computation for C and the change in the K-Means terms are outlined below:

$$C.N \rightarrow \left[\left(C.N \times \frac{|C|}{|C+1|}\right) + \frac{X.N}{C+1}\right] \tag{13}$$

$$\delta_{X_{in}} KM(C, N) = \left[\left(\sum_{X' \in C, X' \neq X} (X'.N - New(C.N))^2\right) + \\ (X.N - New(C.N))^2\right] - \left(\sum_{X' \in C, X' \neq X} (X'.N - C.N)^2\right) \tag{14}$$

It may be noticed that the computation of the changes above only involve X and other objects in C and C'. In particular, the other clusters and their objects do not come into play. So far, we have computed the changes for only one attribute N. The overall change in the K-Means term is simply the sum of these changes across all attributes in $\mathcal{N}$.

$$\delta(\text{K-Means term})_{X \in C' \rightarrow X \in C} = \\ \sum_{N \in \mathcal{N}} \left(\delta_{X_{out}} KM(C', N) + \delta_{X_{in}} KM(C, N)\right) \tag{15}$$

Change in Fairness Term: We now outline the construction of the change in the fairness term. As earlier, we start by considering a single cluster C^*, a single attribute S, and a single value s within it. The fairness term from Eq. 7 can be written as follows:

$$dev(C^*, S = s) = \frac{C^{*2} \times \left(\left(\frac{C^*_s}{C^*}\right)^2 + \left(\frac{\mathcal{X}_s}{\mathcal{X}}\right)^2 - 2\frac{C^*_s}{C^*}\frac{\mathcal{X}_s}{\mathcal{X}}\right)}{\mathcal{X}^2 \times |Values(S)|} \tag{16}$$

where each set (C^* and $\mathcal{X}$) is overloaded to represent both itself and its cardinality (to avoid notation clutter), and their suffixed versions (C^*_s and $\mathcal{X}_s$) are used to refer to their subsets containing their objects which take the value $S = s$. The above equation follows from the observation that $Fr^S_{C^*}(s) = \frac{C^*_S}{C^*}$ and analogously for $\mathcal{X}$. When an object changes clusters from C' to C, there is a change in the terms associated with both clusters, as in the previous case. The change in the origin cluster C' works out to be the follows:

$$\delta_{X_{out}} dev(C', S = s) = \frac{1}{\mathcal{X}^2 \times |Values(S)|} \times \left[\left(\frac{\mathcal{X}_s}{\mathcal{X}}\right)^2 (1 - 2C') + \\ \mathbb{I}(X.S = s)(1 - 2C'_s) - 2\left(\frac{\mathcal{X}_s}{\mathcal{X}}\right)\left(\mathbb{I}(X.S = s)(1 - C') - C'_s\right)\right] \tag{17}$$

where $\mathbb{I}(.)$ is an indicator function, and C' and C'_s denote the cardinalities before X is taken out of C'. We omit the derivation for space constraints. Intuitively, to nudge clusters towards fairness, we would like to incentivize removal of objects with $S = s$ from C' when C' is overpopulated with such objects (i.e., C'_s is high). This is evident in the $-(C'_s \times \mathbb{I}(X.S = s))$ component; when C'_s is high, removal of an object with $S = s$ entails a bigger reduction in the objective. The analogous change in the target cluster C, is as follows:

$$\delta_{X_{in}} dev(C, S = s) = \frac{1}{\mathcal{X}^2 \times |Values(S)|} \times \left[\left(\frac{\mathcal{X}_s}{\mathcal{X}}\right)^2 (1 + 2C) + \\ \mathbb{I}(X.S = s)(1 + 2C_s) - 2\left(\frac{\mathcal{X}_s}{\mathcal{X}}\right)\left(\mathbb{I}(X.S = s)(1 + C) + C_s\right)\right] \tag{18}$$

where C and C_s denote the cardinalities before X is inserted into C. Given that we are inserting X into C, the fairness intuition suggests that we should disincentivize addition of objects with s when C already has too many of such objects. This is reflected in the $(C_s \times \mathbb{I}(X.S = s))$ term; notice that this is exactly the same term as in the earlier case, but with a different sign.

Thus, the overall fairness term change is as follows:

$$\delta(\text{deviation term})_{X \in C' \to X \in C} = \sum_{S \in \mathcal{S}} \sum_{s \in Values(S)} \left(\delta_{X_{out}} dev(C', S = s) + \delta_{X_{in}} dev(C, S = s) \right) \tag{19}$$

This completes all the steps required for Eq. 9. Based on the change in the cluster assignment, the cluster prototypes and fractional representations are to be updated.

4.2.2 Cluster Prototype Updates. Once a new cluster has been finalized for X, the origin and target cluster prototypes are updated according to Eq. 12 and Eq. 14 respectively.

4.2.3 Fractional Representation Updates. The $Fr^S_{C'}(s)$s and $Fr^S_C(s)$s need to be updated to reflect the change in the cluster assignment of X. These are straightforward and given as follows:

$$\forall S \forall s \in Values(S), Fr^S_{C'}(s) = \begin{cases} \frac{C'_s - 1}{C' - 1} & \text{if } X.S = s \\ \frac{C'_s}{C' - 1} & \text{if } X.S \neq s \end{cases} \tag{20}$$

$$\forall S \forall s \in Values(S), Fr^S_{C}(s) = \begin{cases} \frac{C_s + 1}{C + 1} & \text{if } X.S = s \\ \frac{C_s}{C' + 1} & \text{if } X.S \neq s \end{cases} \tag{21}$$

where the C, C', C_s and C'_s values above are cardinalities of the respective sets prior to the update to X's cluster assignment.

Alg. 1 *FairKM*

```
Input. Dataset $\mathcal{X}$, attribute sets $\mathcal{S}$ and $\mathcal{N}$, number of clusters $k$
Hyper-parameters: Fairness Weighting $\lambda$
Output. Clustering $\mathcal{C}$
1. Initialize k clusters randomly
2. Set cluster prototypes as Cluster Centroids
3. while(not yet converged and max. iterations not reached)
4.     $\forall X \in \mathcal{X}$,
5.         Set Cluster(X) using Eq. 9 (and Eq. 10 through Eq. 19)
6.         Update cluster prototypes as outlined in Sec 4.2.2
7.         Re-estimate the $Fr^S_C(s)$ using Eq. 20 and Eq. 21
8. Return the current clustering assignments as $\mathcal{C}$
```

4.3 FairKM Algorithm

Having outlined the various steps, the *FairKM* algorithm can now be summarized in Algorithm 1. The method starts with a random initialization of clusterings (Step 1) and proceeds iteratively. Within each iteration, each object is considered in round-robin fashion, executing three steps in sequence: (i) updating the cluster assignment of X (Step 5), (ii) updating the cluster prototypes to reflect the change in cluster assignment of X (Step 6), and (iii) updating the fractional representations correspondingly (Step 7). The significant difference in construction from K-Means is due to the inter-dependency in cluster assignments; the cluster assignment for X depends on the current cluster assignments for all other objects $\mathcal{X} - \{X\}$, due to the construction of the *FairKM* objective as reflected in the update steps. The updates proceed as long as the clustering assignments have not converged or a pre-specified maximum number of iterations have not reached.

4.3.1 Complexity: The time complexity of *FairKM* is dominated by the cluster assignment updates. Within each iteration, for each X ($|\mathcal{X}|$ of them) and each cluster it could be re-assigned to (k of them), the deviation needs to be computed for both the (i) K-Means term, and the (ii) fairness term. First, considering the K-Means term, it may be noted that each other object in $\mathcal{X}$ would come into play once, either as a member of X's current cluster (in Eq. 12) or as a member of a potential cluster to which X may be assigned (in Eq. 14). This yields an overall complexity of each K-Means deviation computation being in $O(|\mathcal{X}||\mathcal{N}|)$. Second, considering the fairness deviation computation, it may be seen as a simple computation (Eq. 17 and 18) that can be completed in constant time. This computation needs to be performed for each attribute in $\mathcal{S}$ and each value of the attribute (consider m as the maximum number of values across attributes in $\mathcal{S}$), yielding a total complexity of $O(|\mathcal{S}|m)$ for each fairness update computation. With the updates needing to be computed for each new candidate cluster, the overall complexity of Step 5 would be $O(|\mathcal{X}||\mathcal{N}|k + |\mathcal{S}|mk)$. Step 6 is in $O(|\mathcal{X}||\mathcal{N}|)$ whereas Step 7 is simply in $O(|\mathcal{S}|m)$. With the above steps having to be performed for each X and for each iteration, the overall *FairKM* complexity works out to be in $O(|\mathcal{X}|^2|\mathcal{N}|kl + |\mathcal{X}||\mathcal{S}|mkl)$ where l is the number of iterations. While the quadratic dependency on the dataset size makes *FairKM* much slower than simple K-Means (which is linear on dataset size), *FairKM* compares very favorably against other fair clustering methods (e.g., exact fairlet decomposition [6] is NP-hard, and even the proposed approximation is super-quadratic) which are computationally intensive.

4.4 FairKM Extensions

We outline two extensions to the basic *FairKM* outlined earlier which was intended towards handling numeric non-sensitive attributes and multi-valued sensitive attributes.

4.4.1 Extension to Numeric Sensitive Attributes. *FairKM* is easily adaptable to numeric sensitive attributes (e.g., *age* for cases where that is appropriate). If all attributes in $\mathcal{S}$ are numeric, the fairness loss term in Eq. 7 would be written out as:

$$deviation_{\mathcal{S}}(\mathcal{C}, \mathcal{X}) = \sum_{C \in \mathcal{C}} \left(\frac{|C|}{|\mathcal{X}|} \right)^2 \times \sum_{S \in \mathcal{S}} (C.S - \mathcal{X}.S)^2 \tag{22}$$

where $C.S$ and $\mathcal{X}.S$ indicate the average value of the numeric attribute S across objects in C and $\mathcal{X}$ respectively. When there are a mix of multi-valued and numeric attributes, the inner term would take the form of Eq. 7 and Eq. 22 for multi-valued and numeric attributes respectively. These entail corresponding changes to the update equations which we do not describe here for brevity.

4.4.2 Extension to allow Sensitive Attribute Weighting. In certain scenarios, some sensitive attributes may need to be considered more important than others. This may be due to historical reasons based on a legacy of documented high discrimination on certain attributes, or due to visibility reasons where discrimination on certain attributes (e.g., *gender, race* and *sexual orientation*) being more visible than others (e.g., *country of origin*). The *FairKM* framework could easily be extended to allow for differential attribute-specific weighting by changing the deviation term to be as follows:

$$deviation_{\mathcal{S}}(\mathcal{C}, \mathcal{X}) = \sum_{C \in \mathcal{C}} \left(\frac{|C|}{|\mathcal{X}|}\right)^2 \times \sum_{S \in \mathcal{S}} w_S \times \frac{\sum_{s \in Values(S)} \left(Fr_C^S(s) - Fr_{\mathcal{X}}^S(s)\right)^2}{|Values(S)|} \tag{23}$$

Attributes that are more important for fairness considerations can then be assigned a higher weight, i.e. w_S, which would lead to their loss being amplified, thus incentivizing *FairKM* to focus more on them for fairness, consequently leading to a higher representational fairness over them, within the clusters in the output. The w_S terms would then also affect the update equations.

5 EXPERIMENTAL STUDY

We now detail our experimental study to gauge and quantify the effectiveness of *FairKM* in delivering good quality and fair clusterings against state-of-the-art baselines. We first outline the datasets in our experimental setup, followed by a description of the evaluation measures and baselines. This is then followed by our results and an analysis of the results.

5.1 Datasets

We use two real-world datasets in our empirical study. The datasets are chosen to cover very different domains, attributes and dataset sizes, to draw generalizable insights from the study. First, we use the popular *Adult* dataset from UCI repository [7]; this dataset is sometimes referenced as the *Census Income* dataset and contains information from the 1994 US Census. The dataset has 32561 instances, each instance represented using 13 attributes. Among the 13 attributes, 5 are chosen to form the set of sensitive attributes, $\mathcal{S}$. These are {*marital status*, *relationship status*, *race*, *gender*, *native country*}. The number of values taken by each of the sensitive attributes are shown in Table 3. The set of non-sensitive attributes, $\mathcal{N}$, pertain to *age*, *work class* (2 attributes), *education* (2 attributes), *occupation*, *fiscal information* (2 attributes) and *number of working hours*. The dataset has been widely used for predicting income as belonging to one of $> 50k\$$ or $<= 50k\$$. We first undersample the dataset to ensure parity across this income class attribute that we do not use in the clustering process. The total number of instances after undersampling is 15682. Second, we use a dataset[2] of 161 word problems from the domain of kinematics. Kinematics is the study of motion without considering the cause of motion. The problems in this dataset is categorized into various types as indicated in Table 2. The complexity of a word problem typically depends on the type. For example, Type 1 problems are easier to solve (in terms of the equations required) compared to Type 5 problems. Table 4 shows the number of problems of each of the above types in the dataset. Given such a dataset of word problems from kinematics domain, we are interested in the task of clustering the word problems such that the proportional representation of problems of a particular type in a cluster reflects its representation in the entire dataset. In the application scenario of automatic construction of multiple questionnaires (one from each cluster) from a question bank, the fair clustering task corresponds to ensuring that each questionnaire contains a reasonable mix of problem types. This ensures that there is minimal asymmetry between the different questionnaires generated by a clustering, in terms of overall hardness. For the fair clustering formulation, thus, the problem types form the set of 5 sensitive binary attributes, $\mathcal{S}$. The lexical representation of each word problem, as a 100 dimensional vector using Doc2Vec models [15], forms the set of numeric attributes in $\mathcal{N}$. Given our fairness consideration, we consider achieving a fair proportion of word problem types within each cluster that reflects their proportion across the dataset.

It may be noted that the *Adult* and *Kinematics* datasets come from different domains (Census and Word Problems/NLP respectively), have different sizes of non-sensitive attribute sets (8 and 100 attributes in $\mathcal{N}$ respectively), different kinds of sensitive attribute sets (multi-valued and binary respectively) and have widely varying sizes ($15k$ and 161 respectively). An empirical evaluation over such widely varying datasets, we expect, would inspire confidence in the generalizability of empirical results.

[2]https://github.com/savithaabraham/Datasets

5.2 Evaluation

Having defined the task of fair clustering in Section 3, it follows that a fair clustering algorithm would be expected to perform well on two sets of evaluation metrics, those that relate to clustering quality over $\mathcal{N}$ and those that relate to fairness over $\mathcal{S}$. We now outline such evaluation measures below, in separate subsections.

5.2.1 *Clustering Quality.*

These measure how well the clustering fares in generating clusters that are coherent on attributes in $\mathcal{N}$, and do not depend on attributes in $\mathcal{S}$. These could include:

- *Silhouette Score* **(SH):** Silhouette [19] measures the separatedness of clusters, and quantifies a clustering with a score in $[-1, +1]$, higher values indicating well-separated clusters.
- *Clustering Objective* **(CO):** Clustering objective functions such as those employed by K-Means [16] measure how much observations deviate from the centroids of the clusters they are assigned to, where lower values indicate coherent clusters. In particular, the K-Means objective function is:

$$\sum_{C \in \mathcal{C}} \sum_{X \in C} dist_{\mathcal{N}}(X, C) \tag{24}$$

  where C stands for both a cluster in the clustering $\mathcal{C}$ as well as the prototype object for the cluster, and $dist_{\mathcal{N}}(.,.)$ is the distance measure computed over attributes in $\mathcal{N}$.
- *Deviation from $\mathcal{S}$-blind Clusterings:* $\mathcal{S}$-blind clusterings may be thought of achieving the best possible clusters for the task when no fairness considerations are imposed. Thus, among two clusterings of similar fairness, that with lower deviation from $\mathcal{S}$-blind clusterings may be considered desirable. A fair clustering can be compared with a $\mathcal{S}$-blind clustering using the following two measures:
  - *Centroid-based Deviation* **(DevC):** Consider each clustering to be represented as a set of cluster centroids, one for each cluster within the clustering. The sum of pair-wise dot-products between centroid pairs, each pair constructed using one centroid from the fair clustering and one from the $\mathcal{S}$-blind clustering, would be a measure of deviation between the clusterings. Such measures have been used in generating disparate clusterings [11].
  - *Object pair-wise Deviation* **(DevO):** Consider each pair of objects from $\mathcal{X}$, and one clustering (either of $\mathcal{S}$-blind and fair); the objects may belong to either the *same* cluster or to *different* clusters. The fraction of object pairs from $\mathcal{X}$ where the same/different verdicts from the two clusterings disagree provide an intuitive measure of deviation between clusterings.

Type	Description
1:Horizontal Motion	The object involved is in a horizontal straight line motion.
2:Vertical motion with an initial velocity	The object is thrown straight up or down with a velocity.
3:Free fall	The object is in a free fall.
4:Horizontally projected	The object is projected horizontally from a height.
5:Two-dimensional	The body is projected with a velocity at an angle to the horizontal.

Table 2: Kinematics Word Problem Types

Attribute	No. of values
Marital status	7
Relationship status	6
Race	5
Gender	2
Native country	41

Table 3: Adult Dataset: Number of possible values for each sensitive attribute

Type	Count
1 - Horizontal motion	60
2 - Vertical motion with an initial velocity	36
3 - Free fall	15
4 - Horizontally projected	31
5 - Two-dimensional	19

Table 4: Kinematics Dataset: #Problems of each Type

5.2.2 Fairness. These measure the fairness of the clustering output from the (fair) clustering algorithm. Analogous to clustering quality measures that depend only on $\mathcal{N}$, the fairness measures we outline below depend only on $\mathcal{S}$ and are independent of $\mathcal{N}$. As outlined earlier, we quantify unfairness as the extent of deviation between representations of groups defined using attributes in $\mathcal{S}$ in the dataset and each cluster in the clustering output. Consider a multi-valued attribute $S \in \mathcal{S}$, which can take on t values. The normalized distribution of presence of each of the t values in $\mathcal{X}$ yields a t-length probability distribution vector $\mathcal{X}_S$. A similar probability distribution can then be computed for each cluster C in the clustering $\mathcal{C}$, denoted C_S. Different ways of measuring the cluster-specific deviations $\{\ldots, dev(C_S, \mathcal{X}_S), \ldots\}$ and aggregating them to a single number yield different quantifications of fairness, as below:

- *Average Euclidean* **(AE):** This measures the average of cluster-level deviations, deviations quantified using euclidean distance between representation vectors (i.e., $\mathcal{X}_S$ and C_Ss). Since clusters may not always be of uniform sizes, we use a cluster-cardinality weighted average.
$$AE_S = \frac{\sum_{C \in \mathcal{C}} |C| \times ED(C_S, \mathcal{X}_S)}{\sum_{C \in \mathcal{C}} |C|} \tag{25}$$
where $ED(.,.)$ denotes the euclidean distance.
- *Average Wasserstein* **(AW):** In this measure, the deviation is computed using Wasserstein distance in lieu of Euclidean, as used in [21], with other aspects remaining the same as above.
- *Max Euclidean* **(ME):** Often, just optimizing for average fairness across clusters is not enough since there could be a very skewed (small) cluster, whose effect may be obscured by other clusters. It is often the case that one or few clusters get picked from a clustering to be actioned upon. Thus, the maximum skew is of interest as an indicative upper bound on the unfairness the clustering could cause if *any* one of its clusters is chosen for further action.
- *Max Wasserstein* **(MW):** This uses Wasserstein instead of Euclidean, using the same formulation as *Max Euclidean.*

When there are multiple attributes in $\mathcal{S}$, as is often the case, the average of the above measures across attributes in $\mathcal{S}$ provides aggregate quantifications. As may be evident, the above constructions work only for categorical attributes; however, a similar set of measures can be readily devised for numeric attributes in $\mathcal{S}$. With our datasets containing only categorical attributes among $\mathcal{S}$, we do not outline the corresponding metrics for numeric attributes, though they follow naturally. We are unable to apply some popular fairness evaluation metrics such as *balance* [6] due to them being devised for binary attributes.

5.3 Baselines

We compare our approach against two baselines. The first is that of $\mathcal{S}$-blind K-Means clustering, that performs K-Means clustering on data using the attributes in $\mathcal{N}$ alone. This baseline is code-named K-Means ($\mathcal{N}$). K-Means ($\mathcal{N}$) will produce the most coherent clusters on $\mathcal{N}$ as its objective function just focuses on maximizing intra-cluster similarity and minimizing inter-cluster similarity over $\mathcal{N}$, unlike *FairKM* that has an additional fairness constraint which may result in compromising the coherence goal. Comparing the two enables us to evaluate the extent to which cluster coherence is traded off by *FairKM* in generating fairer clusters. The second baseline is the approach described in [22] which is a fair version of K-Means clustering for scenarios involving a single multi-valued sensitive attribute. We will refer to this baseline as *ZGYA* from here, based on the names of the authors. Since it is designed for a single multi-valued sensitive attribute and cannot handle multiple sensitive attributes within its formulation, we invoke *ZGYA* multiple times, separately for each attribute in $\mathcal{S}$. Each invocation is code-named *ZGYA*(S) where S is the sensitive attribute used in the invocation. We also report results for similar runs of *FairKM*, where we consider just one of the attributes in $\mathcal{S}$ as sensitive at a time. The comparative evaluation between *FairKM* and *ZGYA* enables studying the effectiveness of FairKM formulation over that of *ZGYA* in their relative effectiveness of trading off coherence for fairness.

5.4 Setting λ in *FairKM*

From Eq. 1, it may be seen that the K-Means term has a contribution from each object in $\mathcal{X}$, whereas the fairness term (Eq. 7) aggregates cluster level contributions. This brings a disparity in that the former has $|\mathcal{X}|/k$ times as many terms as the latter. Further, it may be noted that the fairness term aggregates deviations between cluster level fractional representations and dataset level fractional representation. The fractional representation being an average across objects in the cluster, each object can only influence $1/|C|$ of it, where $|C|$ is the cluster cardinality. On an average, across clusters, $|C| = |\mathcal{X}|/k$. Thus, the fairness term has $|\mathcal{X}|/k$ fewer terms, each of whom can be influenced by an object to a fraction of $1/(|\mathcal{X}|/k)$. To ensure that the terms are of reasonably similar sizes, so the clustering quality and fairness concerns be given similar weighting, the above observations suggest that λ be set to $(\frac{|\mathcal{X}|}{k})^2$. From our empirical observations, we have seen that the *FairKM* behavior varies smoothly around this setting. Based on the above heuristic, we set λ to 10^6 for the Adult dataset, and 10^3 for the Kinematics dataset, given their respective sizes. We will empirically analyze sensitivity to λ in Section 5.7. We set max iterations to 30 in *FairKM* instantiations.

5.5 Clustering Quality and Fairness

5.5.1 Evaluation Setup. In each of our datasets, there are five sensitive (i.e., $\mathcal{S}$) attributes. *FairKM* can be instantiated with all of them at once, and we do so with appropriate values of λ (10^6 or 10^3, as mentioned in Section 5.4). We perform 100 such instantiations, each with a different random seed, and measure the *clustering quality* and *fairness* evaluation measures (fairness measures computed separately for each attribute in $\mathcal{S}$ as well as the average across all attributes in $\mathcal{S}$) outlined in Section 5.2. We take the mean values across the 100 instantiations to arrive at a single robust value for each evaluation measure for *FairKM*. An analogous setting is used for our first baseline, the $\mathcal{S}$-blind K-Means (denoted K-Means ($\mathcal{N}$)), as well. Our second baseline, *ZGYA*, unlike *FairKM*, needs to be instantiated with one $\mathcal{S}$ attribute at a time. Given this setting, we adopt different mechanisms to compare *FairKM* against *ZGYA* across clustering quality and fairness evaluation measures. First, for *clustering quality*, we instantiate *ZGYA* separately with each attribute in $\mathcal{S}$ and compute an average value for each evaluation measure across random initializations as described previously. This yields one value for each evaluation measure for each attribute in $\mathcal{S}$, which we take an average of, and report as the *clustering quality* of *Avg. ZGYA*. Second, for *fairness*, we adopt a synthetic favorable setting for *ZGYA* to test *FairKM* against. For each attribute $S \in \mathcal{S}$, we consider the fairness metrics (AE, AW, ME, MW) obtained by the instantiation of *ZGYA* over *only* that attribute (averaged across random initializations, as earlier). This is compared to the fairness metrics obtained for S by the *FairKM* instantiation that considers *all* attributes in $\mathcal{S}$. In other words, for each $S \in \mathcal{S}$, we benchmark the single cross-$\mathcal{S}$ instantiation of *FairKM* against separate S-targeted instantiations of *ZGYA*. We also report the average across these separate comparisons across attributes in $\mathcal{S}$. For K-Means style clustering formulations, the number of clusters k, is an important parameter. We experiment with two values for k, viz., 5 and 15, for the Adult dataset, whereas we use $k = 5$ for the Kinematics dataset, given its much smaller size.

5.5.2 Clustering Quality. The clustering quality results appear in Table 5 (Adult dataset) and Table 7 (Kinematics dataset), with the direction against each evaluation measure indicating whether lower or higher values are more desirable. For clustering quality metrics that depend only on attributes in $\mathcal{N}$, we use K-Means ($\mathcal{N}$) as a reference point since that is expected to perform well, given that it does not need to heed to $\mathcal{S}$ and is not held accountable for fairness. Thus, *FairKM* is not expected to beat K-Means ($\mathcal{N}$); the lesser the degradation from K-Means ($\mathcal{N}$) on various clustering quality metrics, the better it may be regarded to be. We compare *FairKM* and *Avg. ZGYA* across the results tables, highlighting the better performer on each evaluation measure by boldfacing the appropriate value. On the Adult dataset (Table 5), it may be seen that *FairKM* performs better than *Avg. ZGYA* on seven out of eight combinations, with it being competitive with the latter on the eighth. *FairKM* is seen to score significantly better than *Avg. ZGYA* on clustering objective (CO) and silhoutte score (SH), with the gains on the deviation metrics (DevC and DevO) being more modest. It may be noted that *CO* and *SH* may be regarded as more reliable measures, since they evaluate the clustering directly. In contrast, *DevO* and *DevC* evaluate the deviation against reference K-Means ($\mathcal{N}$) clusterings; these deviation measures penalize deviations even if those be towards *other* good quality clusterings that may exist in the dataset. The trends from the Adult dataset hold good for the Kinematics dataset as well (see Table 7), confirming that the trends generalize well across datasets of widely varying character. Overall, our results indicate that *FairKM* is able to generate much better quality clusterings than *Avg. ZGYA*, when gauged on attributes in $\mathcal{N}$.

5.5.3 Fairness. The fairness evaluation measures for the Adult and Kinematics datasets appear in Tables 6 and 8 respectively; it may be noted that lower values are desirable on all evaluation measures, given that they all measure deviations. In these results, which include a synthetically favorable setting for *ZGYA* (as noted earlier), the top block indicates the average results across all attributes in $\mathcal{S}$, with the following result blocks detailing the results for the specific parameters in $\mathcal{S}$. The overarching summary of this evaluation suggests, as indicated in the top-blocks across the two tables, that *FairKM* surpasses the baselines with significant margins. The *% impr* column indicates the gain achieved by *FairKM* over the next best competitor. The percentage improvements recorded are around 35 + % on an average for the Adult dataset, whereas the corresponding figure is higher, at around 60 + % for the Kinematics dataset. We wish to specifically make a few observations from the results. *First*, the closest competitor to *FairKM* is *ZGYA* on the Kinematics dataset, whereas K-Means ($\mathcal{N}$) curiously outperforms *ZGYA* quite consistently on the Adult dataset. This indicates that *ZGYA* is likely more suited to settings where the number of values taken by the sensitive attribute is less. In our case, the Kinematics dataset has all binary attributes in $\mathcal{S}$, whereas Adult dataset has sensitive attributes that take as many as 41 different values. *Second*, while *FairKM* is designed to accommodate $\mathcal{S}$ attributes that take many different values, the fairness deviations appear to degrade, albeit at a much lower pace than *ZGYA*, as attributes take on very many values. This is indicated by the lower performance (with small margins) on the *native country* (41 values) attribute at $k = 5$ in Table 6. However, promisingly, it is able to utilize the additional flexibility that is provided by larger ks to ensure higher rates of fairness on them. As may be seen, *FairKM* recovers well to perform significantly better on *native country* at $k = 15$. This indicates that *FairKM* will benefit from a higher flexibility in cluster assignment (with higher k) when there are a number of (high cardinality) attributes to ensure fairness over. *Third*, the *FairKM* formulation targets to minimize overall fairness and does not specifically nudge it towards ensuring good performance on the *max* measures (ME and MW) that quantify the worst deviation across clusters. Thus, it's design allows to choose higher fairness in multiple clusters even at the expense of disadvantaging fairness in one or few clusters, which is indeed undesirable. The performance on ME and MW

Evaluation	k=5			k=15		
Measure	K-Means ($\mathcal{N}$)	Avg. ZGYA	FairKM	K-Means ($\mathcal{N}$)	Avg. ZGYA	FairKM
CO ↓	1120.9112	10791.8311	**1345.1688**	837.9785	4095.8366	**1235.2859**
SH ↑	0.7212	0.0557	**0.3918**	0.6076	0.0573	**0.3747**
DevC ↓	0.0	**8.4597**	8.4707	0.0	39.3615	**13.1244**
DevO ↓	0.0	0.0306	**0.0233**	0.0	0.0360	**0.0256**

Table 5: Clustering quality on Adult Dataset - FairKM vs. Average across $\{ZGYA(S)|S \in \mathcal{S}\}$, shown with K-Means($\mathcal{N}$).

$\mathcal{S}$ Attribute	Evaluation Measure	k=5				k=15			
		K-Means($\mathcal{N}$)	ZGYA(S)	FairKM	FairKM Impr(%)	K-Means($\mathcal{N}$)	ZGYA(S)	FairKM	FairKM Impr(%)
Mean across $\mathcal{S}$ Attributes	AE	0.0459	0.1201	**0.0278**	**39.5357**	0.0537	0.1289	**0.0295**	**45.0796**
	AW	0.0161	0.0370	**0.0087**	**45.7857**	0.0194	0.0398	**0.0094**	**51.7043**
	ME	0.2063	0.8729	**0.1457**	**29.4002**	0.2475	0.7810	**0.1542**	**37.6985**
	MW	0.0740	0.1235	**0.0502**	**32.0985**	0.0753	0.1262	**0.0542**	**28.0040**
Results for Each Sensitive Attribute in $\mathcal{S}$ below.									
Marital Status	AE	0.0792	0.0886	**0.0539**	**31.9408**	0.0853	0.1318	**0.0558**	**34.5263**
	AW	0.0182	0.0159	**0.0132**	**16.5650**	0.0191	0.0258	**0.0136**	**28.4239**
	ME	0.3055	0.7356	**0.2578**	**15.6087**	0.3572	0.6365	**0.2607**	**27.0042**
	MW	**0.0573**	0.0890	0.0592	-3.3881	**0.0566**	0.0952	0.0604	-6.6317
Rel. Status	AE	0.0711	0.1743	**0.0486**	**31.5656**	0.0808	0.1903	**0.0500**	**38.1517**
	AW	0.0197	0.0371	**0.0146**	**25.8744**	0.0219	0.0429	**0.0150**	**31.3346**
	ME	0.3331	0.7796	**0.2717**	**18.4487**	0.3823	0.7804	**0.2777**	**27.3667**
	MW	**0.0732**	0.1205	0.0760	-3.8026	**0.0750**	0.1439	0.0776	-3.4770
Race	AE	0.0163	0.0564	**0.0066**	**59.2251**	0.0168	0.0647	**0.0079**	**53.0164**
	AW	0.0053	0.0154	**0.0023**	**55.9473**	0.0055	0.0162	**0.0028**	**48.9813**
	ME	0.0385	1.0085	**0.0266**	**30.8822**	0.0565	1.2175	**0.0336**	**40.6276**
	MW	0.0126	0.1159	**0.0092**	**27.3039**	0.0165	0.1142	**0.0115**	**30.2523**
Gender	AE	0.0529	0.2535	**0.0183**	**65.3039**	0.0711	0.2256	**0.0208**	**70.7472**
	AW	0.0370	0.1153	**0.0130**	**64.9210**	0.0499	0.1122	**0.0147**	**70.4913**
	ME	0.3324	0.9793	**0.1487**	**55.2713**	0.4028	1.0201	**0.1697**	**57.8731**
	MW	0.2254	0.2568	**0.1051**	**53.3681**	0.2262	0.2671	**0.1200**	**46.9680**
Native Country	AE	**0.0101**	0.0276	0.0113	-11.2331	0.0146	0.0323	**0.0130**	**10.9108**
	AW	**0.0005**	0.0013	0.0006	-15.2027	0.0007	0.0015	**0.0006**	**4.7201**
	ME	**0.0221**	0.8612	0.0236	-6.4585	0.0385	0.2506	**0.0292**	**24.1555**
	MW	**0.0012**	0.0354	0.0016	-25.8608	0.0020	0.0107	**0.0015**	**25.6292**

Table 6: Fairness evaluation on Adult Dataset - $\mathcal{S}$-blind K-Means, Single invocation of *FairKM* on all $\mathcal{S}$ attributes, Separate Invocations of ZGYA on each attribute in $\mathcal{S}$. (Note: This is a synthetic favorable setting for ZGYA, to stress test *FairKM* against ZGYA).

Evaluation	K-Means ($\mathcal{N}$)	Avg. ZGYA	FairKM
CO ↓	145.6441	164.4703	**148.1003**
SH ↑	0.0390	-0.0001	**0.0149**
DevC ↓	0.0	1.1844	**1.1241**
DevO ↓	0.0	**0.0032**	0.0038

Table 7: Clustering quality on Kinematics Dataset - *FairKM* vs. Average across $\{ZGYA(S)|S \in \mathcal{S}\}$, shown with K-Means($\mathcal{N}$).

suggest that such trends are not widely prevalent, with *FairKM* recording reasonable gains on ME and ME. However, cases such as *marital status* in Table 6 and Type-3 in Table 8 suggest that is a direction in which *FairKM* could improve. *Finally*, the overall summary from Tables 6 and 8 suggest that *FairKM* delivers much fairer clusters on $\mathcal{S}$ attributes, and records significant gains over the baselines, in our empirical evaluation.

5.6 *FairKM* vs. *ZGYA*

Having compared *FairKM* against *ZGYA* for fairness in a synthetic setting that was favorable to the latter in the previous section, we now do a more direct comparison here. In particular, we consider comparing the *FairKM* and *ZGYA* instantiations with each sensitive attribute separately, which offers a more level

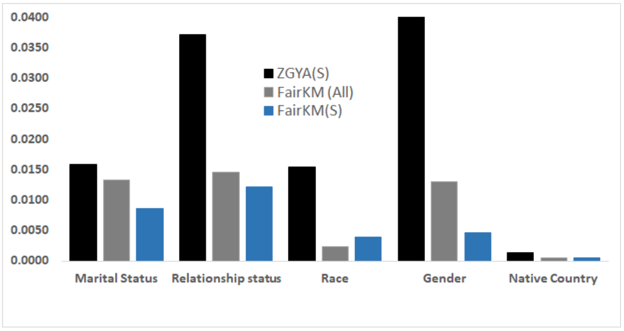


Figure 1: Adult Dataset: AW Comparison

setting. Figure 1 illustrates the comparison on the AW evaluation measure over the Adult dataset for each $\mathcal{S}$ attribute with *ZGYA(S)* and *FairKM(S)* values shown separated by the *FairKM (All)* value in between them; all these are values obtained with $k = 5$. The *FairKM (All)* is simply *FairKM* instantiated with all attributes in $\mathcal{S}$, which was used in the comparison in the previous section. As may be seen, with *FairKM(S)* focusing on just the chosen attribute (as opposed to *FairKM (All)* that needs to spread attention across all attributes in $\mathcal{S}$), *FairKM(S)* is able to achieve better values for AW. Thus, *FairKM(S)* is seen to beat *ZYGA(S)* by larger margins than *FairKM (All)*, as expected. The *Race* attribute shows

$\mathcal{S}$ Attribute	Metric	K-Means ($\mathcal{N}$)	ZGYA (S)	FairKM	FairKM Impr(%)
Mean across $\mathcal{S}$ Attributes	AE	0.1704	0.1183	**0.0172**	**85.4311**
	AW	0.1021	0.0766	**0.0120**	**84.3660**
	ME	0.3744	0.2571	**0.1488**	**42.1364**
	MW	0.2083	0.1676	**0.0852**	**49.1420**
	Results for Each Sensitive Attribute in $\mathcal{S}$ below.				
Type-1	AE	0.2567	0.1821	**0.0148**	**91.8775**
	AW	0.1289	0.1000	**0.0103**	**89.7246**
	ME	0.4909	0.3502	**0.1673**	**52.2397**
	MW	0.2828	0.2321	**0.1004**	**56.7159**
Type-2	AE	0.2145	0.1481	**0.0163**	**88.9722**
	AW	0.1213	0.0994	**0.0113**	**88.6729**
	ME	0.5116	0.3398	**0.1600**	**52.9166**
	MW	0.2149	0.1931	**0.0888**	**54.0235**
Type-3	AE	0.0759	0.0604	**0.0178**	**70.5473**
	AW	0.0535	0.0427	**0.0123**	**71.2578**
	ME	0.1935	**0.1270**	0.1527	-20.2176
	MW	0.1206	0.0898	**0.0754**	**16.0235**
Type-4	AE	0.1631	0.1009	**0.0152**	**84.9649**
	AW	0.1079	0.0708	**0.0107**	**84.9541**
	ME	0.3605	0.2410	**0.1263**	**47.5836**
	MW	0.2103	0.1662	**0.0770**	**53.6570**
Type-5	AE	0.1415	0.0999	**0.0221**	**77.8973**
	AW	0.0989	0.0703	**0.0154**	**78.0243**
	ME	0.3155	0.2273	**0.1375**	**39.5175**
	MW	0.2128	0.1569	**0.0846**	**46.1075**

Table 8: Fairness evaluation on Kinematics Dataset - $\mathcal{S}$-blind K-Means, Single invocation of *FairKM* on all $\mathcal{S}$ attributes, Separate Invocations of ZGYA on each attribute in $\mathcal{S}$. (Note: This is a synthetic favorable setting for ZGYA, to stress test *FairKM* against ZGYA).

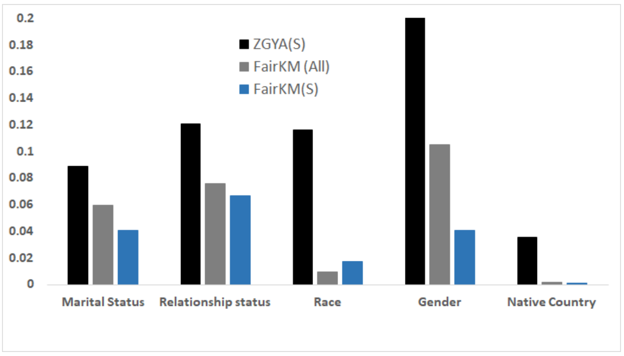


Figure 2: Adult Dataset: MW Comparison

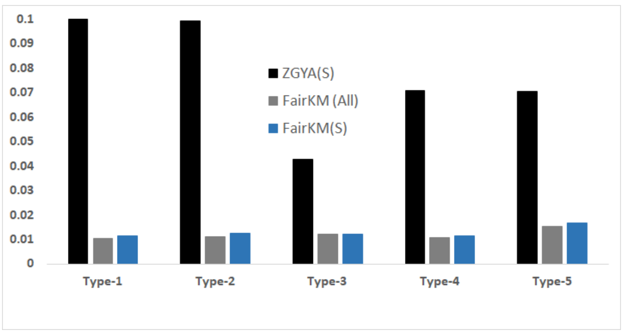


Figure 3: Kinematics Dataset: AW Comparison

a different trend, with *FairKM(S)* recording a slightly higher AW than *FairKM(S)*. While we believe this is likely to be due to an

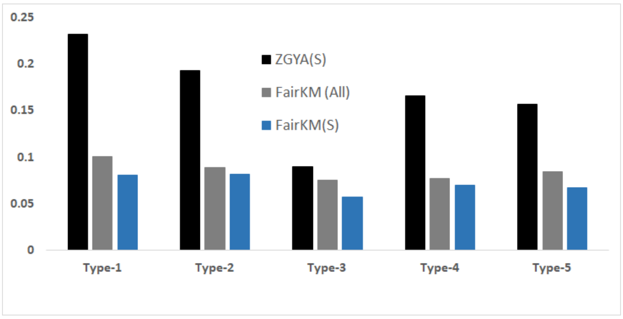


Figure 4: Kinematics Dataset: MW Comparison

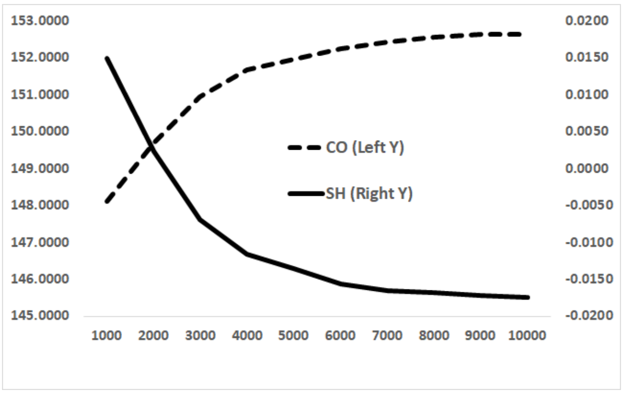


Figure 5: Kinematics Dataset: (CO and SH) vs. λ

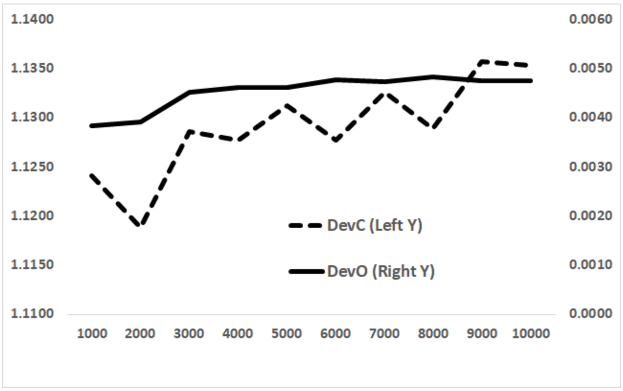


Figure 6: Kinematics Dataset: (DevC and DevO) vs. λ

unusually high skew in the *race* attribute where 87% of objects take the same single value, this warrants further investigation. Figure 2 presents the corresponding chart for MW evaluation measure, and offers similar high-level trends as was observed for AW. The corresponding charts for the Kinematics dataset appear in Figures 3 and 4 respectively. Over the much smaller Kinematics dataset, the gains by *FairKM(S)* over *FairKM (All)* are more pronounced in MW with both techniques recording reasonably similar AW numbers. The observed trends were seen to hold for AE and ME evaluation measures as well, those charts excluded for brevity. To summarize the findings across the datasets, it may be seen that *FairKM(S)* may be seen to beat the *ZGYA(S)* baseline with larger margins than *FairKM (All)* on an average, as desired.

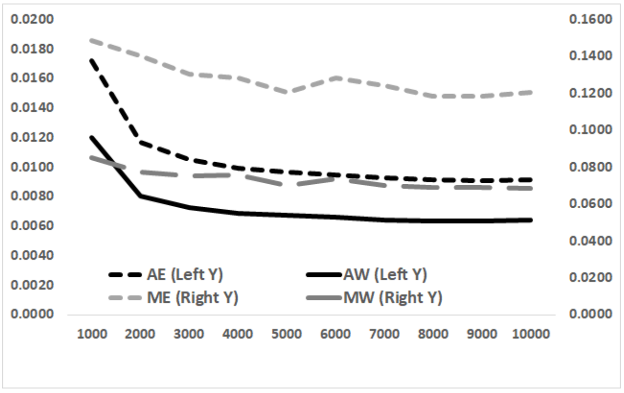


Figure 7: Kinematics Dataset: Fairness Metrics vs. λ

5.7 *FairKM* Sensitivity to λ

We now study *FairKM*'s sensitivity to it's only parameter λ, the weight for the fairness term. With increasing λ, we expect *FairKM* to fare better on fairness measures with corresponding degradations in the clustering quality measures. The vice versa is expected to hold with decreasing λ. We observed such desired trends across Adult and Kinematics datasets, with changes being slower and steadier for the larger Adult dataset. This is on expected lines with the number of parameters such as clustering assignments being larger on the Adult dataset. In the interest of focusing on the smaller dataset, we outline the changes with λ on clustering quality and fairness measures on the Kinematics dataset, when λ is varied from 1000 to 10000. The variations on the CO and SH measures are illustrated in Figure 5, whereas the variations on DevC and DevO are plotted in Figure 6. We use both sides of the Y-axis to plot the measures which widely vary in terms of their ranges; the axis used is indicated in the legend. As may be seen from them, CO, SH and DevO record slow and steady degradation (the Y-axis is stretched to highlight the region of the change; it may be noted that the quantum of change is very limited) with increasing λ. The degradation in DevC, however, is more jittery, while the direction of change remains on expected lines. The fairness deviation measures are plotted against varying λ (once again, on both Y-axes) in Figure 7. They record gradual but steady improvements (being deviations, they are better when low) with increasing λ, on expected lines. Overall, it may be seen that *FairKM* moves steadily but gradually towards fairness with increasing λ, as desired.

6 CONCLUSIONS AND FUTURE WORK

We considered the problem of ensuring representational fairness in clustering for scenarios comprising multiple sensitive attributes. We proposed a novel clustering method, *FairKM*, to accomplish the fair clustering task. In particular, *FairKM* comprises the optimization of the classical clustering objective in tandem with a novel fairness loss term, towards achieving a trade-off between clustering quality (on the non-sensitive attributes) and cluster fairness (on the sensitive attributes). We outline a series of evaluation metrics for both the above criteria, and perform a rigorous empirical evaluation of *FairKM* over two real-world datasets of widely varying character, pitching *FairKM* against other available methods. Our empirical evaluation illustrates that *FairKM* outperforms the baseline *ZGYA* even over synthetic settings that are artificially favorable to the latter. This illustrates the effectiveness of the *FairKM* formulation.

6.1 Future Work

We are exploring three directions of future work towards enhancing *FairKM*. First, we are studying the performance trends of *FairKM* with increasing number of sensitive attributes as well as increasing number of values per sensitive attribute. Second, drawing cue from the observation in Section 5.6, we are looking at how *FairKM* can be improved to ensure good performance even on attributes with highly skewed distributions. Third, the main computational bottleneck in *FairKM* is the cluster centroid update while doing cluster assignments. We are considering approximation heuristics such as mini-batch updates where centroid updates are done only once every mini-batch of clustering assignment updates, to speed up *FairKM* for scalability.

Acknowledgements: The second author was partially supported by SPARC (P620). The authors would also like to thank Krishna M. Sivalingam (IIT Madras) for his constant encouragement.

REFERENCES

[1] Sara Ahmadian, Alessandro Epasto, Ravi Kumar, and Mohammad Mahdian. 2019. Clustering without Over-Representation. *arXiv:1905.12753* (2019).
[2] Aris Anagnostopoulos, Luca Becchetti, Matteo Böhm, Adriano Fazzone, Stefano Leonardi, Cristina Menghini, and Chris Schwiegelshohn. 2019. Principal Fairness: Removing Bias via Projections. *arXiv:1905.13651* (2019).
[3] Arturs Backurs, Piotr Indyk, Krzysztof Onak, Baruch Schieber, Ali Vakilian, and Tal Wagner. 2019. Scalable fair clustering. *arXiv:1902.03519* (2019).
[4] Suman K Bera, Deeparnab Chakrabarty, and Maryam Negahbani. 2019. Fair algorithms for clustering. *arXiv:1901.02393* (2019).
[5] Xingyu Chen, Brandon Fain, Charles Lyu, and Kamesh Munagala. 2019. Proportionally Fair Clustering. *arXiv:1905.03674* (2019).
[6] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, and Sergei Vassilvitskii. 2017. Fair clustering through fairlets. In *NIPS*. 5029–5037.
[7] Dheeru Dua and Casey Graff. 2017. UCI machine learning repository (2017). *URL http://archive. ics. uci. edu/ml* (2017).
[8] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. 2012. Fairness through awareness. In *Proceedings of the 3rd innovations in theoretical computer science conference*. ACM, 214–226.
[9] Anil K Jain. 2010. Data clustering: 50 years beyond K-means. *Pattern recognition letters* 31, 8 (2010), 651–666.
[10] Anil K Jain, Richard C Dubes, et al. 1988. *Algorithms for clustering data*. Vol. 6. Prentice hall Englewood Cliffs, NJ.
[11] Prateek Jain, Raghu Meka, and Inderjit S Dhillon. 2008. Simultaneous unsupervised learning of disparate clusterings. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 1, 3 (2008), 195–210.
[12] Faisal Kamiran and Toon Calders. 2009. Classifying without discriminating. In *International Conference on Computer, Control and Communication*. 1–6.
[13] Matthäus Kleindessner, Pranjal Awasthi, and Jamie Morgenstern. 2019. Fair k-center clustering for data summarization. *arXiv:1901.08628* (2019).
[14] Matthäus Kleindessner, Samira Samadi, Pranjal Awasthi, and Jamie Morgenstern. 2019. Guarantees for spectral clustering with fairness constraints. *arXiv:1901.08668* (2019).
[15] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*. 1188–1196.
[16] James MacQueen et al. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA, 281–297.
[17] Matt Olfat and Anil Aswani. 2019. Convex formulations for fair principal component analysis. In *AAAI*, Vol. 33. 663–670.
[18] Clemens Rösner and Melanie Schmidt. 2018. Privacy preserving clustering with constraints. *arXiv:1802.02497* (2018).
[19] Peter J Rousseeuw. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20 (1987), 53–65.
[20] Melanie Schmidt, Chris Schwiegelshohn, and Christian Sohler. 2018. Fair coresets and streaming algorithms for fair k-means clustering. *arXiv:1812.10854* (2018).
[21] Bokun Wang and Ian Davidson. 2019. Towards Fair Deep Clustering With Multi-State Protected Variables. *arXiv:1901.10053* (2019).
[22] Imtiaz Masud Ziko, Eric Granger, Jing Yuan, and Ismail Ben Ayed. 2019. Clustering with Fairness Constraints: A Flexible and Scalable Approach. *arXiv:1906.08207* (2019).

Ontology-Based RDF Integration of Heterogeneous Data

Maxime Buron[1] François Goasdoué[2] Ioana Manolescu[1] Marie-Laure Mugnier[3]
[1]Inria and LIX (UMR 7161, CNRS and Ecole polytechnique), France
[2]Univ. Rennes, CNRS, IRISA, France [3]Univ. Montpellier, LIRMM, Inria, France

ABSTRACT

The proliferation of heterogeneous data sources in many application contexts brings an urgent need for expressive and efficient data integration mechanisms. There are strong advantages to using RDF graphs as the integration format: being schemaless, they allow for flexible integration of data from heterogeneous sources; RDF graphs can be interpreted with the help of an ontology, describing application semantics; last but not least, RDF enables joint querying of the data and the ontology.
To address this need, we formalize *RDF Integration Systems (RIS)*, Ontology Based-Data Access mediators, that go beyond the state of the art in the ability to expose, integrate and flexibly query data from heterogeneous sources through GLAV (global-local-as-view) mappings. We devise several *query answering strategies*, based on an innovative integration of LAV view-based rewriting and a form of mapping saturation. Our experiments show that one of these strategies brings strong performance advantages, resulting from a balanced use of mapping saturation and query reformulation.

1 INTRODUCTION

The proliferation of digital data sources across all application domains brings a new urgency to the need for tools which allow to query heterogeneous data (relational, JSON, key-values, graphs etc.) in a flexible fashion. Traditional data integration systems fall into two classes: *data warehousing*, where all data source content is materialized in a single centralized source, respectively, *mediation*, where data remains in their original stores and all data can be queried through a single module called *mediator*. Data warehousing simplifies query evaluation, but requires potentially costly maintenance operations when the content of data sources changes; mediation does not suffer from these drawbacks, but requires more intricate query evaluation algorithms to distribute the work between the sources and the mediator.
Below, we classify prior mediator-based approaches according to two main dimensions, and illustrate this classification in Table 1. Note that we also include in this table theoretical frameworks that did not necessarily lead to implementations.
A first dimension concerns the **data model and query language** provided by the mediator to its applications.
(*i*) The earliest goal of a mediator system was to mimic a single, integrated database. Thus the mediator supports one data model and its query language, e.g., relational and SQL, or XML and XPath/XQuery. More recent polystore systems support side-by-side different (data model, query language) pairs. These database-style mediators appear in the row we label **DB** in Table 1.
(*ii*) Mediators studied in knowledge representation and management research provide a view of the data sources as a set of classes and relationships, also endowed with a set of semantic

		Mappings		
		GAV	**LAV**	**GLAV**
Model	**DB**	[22, 24, 27]	[4, 5, 22, 38]	[19]
	CQ	[40, 41, 43]	[1, 28, 30, 36]	[18]
	SPARQL-data	[11, 17, 32, 34]	[45]	[21]
	SPARQL	[16, 33, 44]		this work

Table 1: Outline of the positioning of our work.

constraints, or ontology. In such systems, users formulate conjunctive (relational) queries; answering them involves not only evaluation over the data (as done in DB mediators), but also reasoning on the data with the help of ontologies. This mediation approach is also commonly termed *Ontology-Based Data Access* (OBDA) [41], with ontologies expressed in Description Logics (DL, in short). Work following this approach are listed in the row we label **CQ** in Table 1.
(*iii*) RDF [47] is naturally suited as an integration model, thanks to its flexibility, its wide adoption in the Open Data community, its close relationship with ontology languages such as RDFS and OWL, and the presence of its associated standard SPARQL query language. Accordingly, several mediators from the CQ group have been extended to support RDF as an integration model and SPARQL query answering. However, while SPARQL allows *querying the data together with the ontology*, e.g., "find the properties of node *n*, as well the classes to which the values of these properties belong", a DL-based mediation approach shares with all logic-based query languages, e.g., Datalog, SQL etc., the inability to do so. RDF mediators which support SPARQL but limited to querying the data only (not the ontology) appear in the row we label **SPARQL-data** in Table 1.
(*iv*) Recent RDF mediators lift this limitation to support joint querying of the data and ontology; we list them in the **SPARQL** row in Table 1.

A second dimension is **how the source (or local) schemas are connected to the global (integration) schema**, using *mappings* [23]. There are three types of mappings, each corresponding to a column in Table 1. The simplest mappings define each element of the global schema, e.g., each relation (if the global schema is relational), as a view over the local schemas; this is known as Global-As-View, or **GAV** in short. In a GAV system, a query over the global (virtual) schema is easily transformed into a query over the local schemas, by *unfolding* each global schema relation, i.e., replacing it with its definition. In contrast, Local-As-View (**LAV**) mappings define elements of the local schemas as views over the global one. Query answering in this context requires *rewriting the query with the views* describing the local sources [31]. Global-Local-As-View (**GLAV**) data integration generalizes both GAV and LAV. A GLAV mapping pairs a query q_1 over one or several local schemas to a query q_2 over the global schema, having the same answer variables. The semantics is that for each answer of q_1, the integration system exposes the data comprised in a corresponding answer of q_2. *GLAV maximizes flexibility*, or, equivalently, *integration expressive power*: unlike LAV, a GLAV mapping may expose only part of a given source's data, and may combine data from several sources; unlike GAV, a GLAV mapping may include joins or complex expressions over the global schema.

In this work, we study **GLAV mediation supporting SPARQL queries over the data and the ontology**. We pick GLAV for its highest expressive power, RDF for its wide adoption, and aim at querying the data and the ontology in order to fully benefit from the flexibility and expressivity of RDF. As Table 1 shows, our system is *the first capable of integrating multiple data sources through GLAV mappings, for SPARQL querying over the data and the ontology*; further, it supports *heterogeneous* data sources (of different data models). A benefit of our using GLAV is the ability to support a form of *incomplete information*, naturally present in RDF through the so-called *blank nodes*, in the virtual RDF graph exposed by the mediator (see Section 3.1).

Our closest competitors only support GAV mappings, even though some support more expressive ontologies and/or queries [16, 33, 44]. Formal OBDA frameworks based on GLAV mappings have been defined, e.g., [18], without concretely deployed systems. A technique for simulating GLAV mappings through GAV ones under certain conditions is suggested in [21], however this solution has many drawbacks; we defer a detailed discussion to Section 6.

Contributions and novelty The contributions we make in this work are as follows.

(1). RIS Formalism We formally define **RDF Integration Systems** (RIS, in short), OBDA mediators capable of exposing data from heterogeneous sources of virtually any data model through GLAV mappings, under the form of an RDF graph endowed with an RDFS ontology. We formalize the problem of BGP (basic graph pattern) RDF query answering over the RDF data and ontology exposed in such systems.

(2). Novel RIS query answering techniques We describe several RIS query answering methods based on transforming *mappings into LAV view definitions*, and on reducing query answering to rewriting it using views. Our first method combines known techniques; the other two methods are novel, and rely on a form of *mapping saturation*. We show that a smart decomposition of reasoning between offline precomputation and query time makes one of these methods much faster than the others.

The paper is organized as follows. Section 2 recalls a set of preliminary notions we build upon. Then, Section 3 defines our RIS and formalizes RIS query answering. Section 4 describes RIS query answering methods. Section 5 presents our experiments, then we discuss related work and conclude.

2 PRELIMINARIES

We present the basics of the RDF graph data model (Section 2.1), of RDF entailment used to make explicit the implicit information RDF graphs encode (Section 2.2), and how RDF graphs can be queried using the widely-considered SPARQL Basic Graph Pattern queries (Section 2.3).

Then, we recall two techniques, namely *query reformulation* (Section 2.4) and *view-based query rewriting* (Section 2.5), which will serve as building blocks for our query answering techniques.

2.1 RDF Graphs

We consider three pairwise disjoint sets of values: $\mathscr{I}$ of IRIs (resource identifiers), $\mathscr{L}$ of literals (constants) and $\mathscr{B}$ of blank nodes modeling unknown IRIs or literals, a.k.a. *labelled nulls* [3, 29]. A *(well-formed) triple* belongs to $(\mathscr{I} \cup \mathscr{B}) \times \mathscr{I} \times (\mathscr{L} \cup \mathscr{I} \cup \mathscr{B})$, and an *RDF graph* G is a set of (well-formed) triples. A triple $(\mathsf{s}, \mathsf{p}, \mathsf{o})$ states that its *subject* s has the *property* p with the *object* value o [47]. We denote by $\mathrm{Val}(G)$ the set of all values (IRIs, blank nodes and literals) occurring in an RDF graph G, and by $\mathrm{Bl}(G)$ its set of blank nodes. In triples, we use _:b (possibly with indices) to denote blank nodes, and quoted strings to denote literals.

Schema triples	Notation
Subclass	$(\mathsf{s}, \prec_{sc}, \mathsf{o})$
Subproperty	$(\mathsf{s}, \prec_{sp}, \mathsf{o})$
Domain typing	$(\mathsf{s}, \hookleftarrow_{d}, \mathsf{o})$
Range typing	$(\mathsf{s}, \hookrightarrow_{r}, \mathsf{o})$
Data triples	Notation
Class fact	$(\mathsf{s}, \tau, \mathsf{o})$
Property fact	$(\mathsf{s}, \mathsf{p}, \mathsf{o})$ s.t. $\mathsf{p} \notin \{\tau, \prec_{sc}, \prec_{sp}, \hookleftarrow_{d}, \hookrightarrow_{r}\}$

Table 2: RDF triples.

Within an RDF graph, we distinguish **data** triples from **schema** ones. The former describe data (either attach a type, or a class, to a resource, or state the value of a certain data property of a resource). The latter state ontological constraints using RDF Schema (RDFS), which relate classes and properties: **subclass** (specialization relation between types), **subproperty** (specialization of a binary relation), typing of the **domain** (first attribute) of a property, respectively, **range** (typing of the second attribute) of a property. Table 2 introduces short notations we adopt for these schema properties.

From now on, we denote by $\mathscr{I}_{\mathrm{rdf}}$ the *reserved* IRIs from the RDF standard, e.g., the properties τ, $\prec_{sc}$, $\prec_{sp}$, $\hookleftarrow_{d}$, $\hookrightarrow_{r}$ shown in Table 2. The rest of the IRIs are application-dependent classes and properties, which are said *user-defined* and denoted by $\mathscr{I}_{\mathrm{user}}$. Hence, $\mathscr{I}_{\mathrm{user}} = \mathscr{I} \setminus \mathscr{I}_{\mathrm{rdf}}$.

We will consider RDF graphs with *RDFS ontologies* made of schema triples of the four above flavours. More precisely:

Definition 2.1 (RDFS ontology). An *ontology triple* is a schema triple whose subject and object are user-defined IRIs from $\mathscr{I}_{\mathrm{user}}$. An *RDFS ontology* (or ontology in short) is a set of ontology triples. Ontology O *is the ontology of an RDF graph* G if O is the set of schema triples of G.

Above, ontology triples are not allowed over blank nodes. This is only to simplify the presentation; we could have allowed them, and handled them as in [29]. More importantly, we forbid ontology triples from altering the common semantics of RDF itself. For instance, we do not allow $(\hookleftarrow_{d}, \prec_{sp}, \hookrightarrow_{r})$, which would impose that the range of every property shares all the types of the property's domain! This second restriction can be seen as common-sense; it underlies most ontological formalisms, in particular description logics [8] thus the W3C's Web Ontology Language (OWL), Datalog± [15] and existential rules [39], etc.

Example 2.2 (Running example, based on [12]). Consider the following RDF graph:

$G_{\mathrm{ex}} = \{$(:worksFor, $\hookleftarrow_{d}$, :Person), (:worksFor, $\hookrightarrow_{r}$, :Org),
(:PubAdmin, $\prec_{sc}$, :Org), (:Comp, $\prec_{sc}$, :Org),
(:NatComp, $\prec_{sc}$, :Comp), (:hiredBy, $\prec_{sp}$, :worksFor)
(:ceoOf, $\prec_{sp}$, :worksFor), (:ceoOf, $\hookrightarrow_{r}$, :Comp),
(:p$_1$, :ceoOf, _:b_c), (_:b_c, τ, :NatComp),
(:p$_2$, :hiredBy, :a), (:a, τ, :PubAdmin)$\}$

The ontology of G_{ex}, i.e., the first eight schema triples, states that people work for organizations, some of which are public administrations or companies. Further, national companies are a kind of companies. Being hired by or being CEO of an organization are two ways of working for it; in the latter case, this organization is a company. The facts of G_{ex}, i.e., the four remaining data triples, state that :p$_1$ is CEO of some unknown company represented by the blank node _:b_c, which is a national company, and :p$_2$ is hired by the public administration :a.

	Rule [48]	Entailment rule	
$\mathcal{R}$	`rdfs5`	$(\mathsf{p}_1, \prec_{sp}, \mathsf{p}_2), (\mathsf{p}_2, \prec_{sp}, \mathsf{p}_3) \rightarrow (\mathsf{p}_1, \prec_{sp}, \mathsf{p}_3)$	$\mathcal{R}_c$
	`rdfs11`	$(\mathsf{s}, \prec_{sc}, \mathsf{o}), (\mathsf{o}, \prec_{sc}, \mathsf{o}_1) \rightarrow (\mathsf{s}, \prec_{sc}, \mathsf{o}_1)$	
	`ext1`	$(\mathsf{p}, \hookleftarrow_d, \mathsf{o}), (\mathsf{o}, \prec_{sc}, \mathsf{o}_1) \rightarrow (\mathsf{p}, \hookleftarrow_d, \mathsf{o}_1)$	
	`ext2`	$(\mathsf{p}, \hookrightarrow_r, \mathsf{o}), (\mathsf{o}, \prec_{sc}, \mathsf{o}_1) \rightarrow (\mathsf{p}, \hookrightarrow_r, \mathsf{o}_1)$	
	`ext3`	$(\mathsf{p}, \prec_{sp}, \mathsf{p}_1), (\mathsf{p}_1, \hookleftarrow_d, \mathsf{o}) \rightarrow (\mathsf{p}, \hookleftarrow_d, \mathsf{o})$	
	`ext4`	$(\mathsf{p}, \prec_{sp}, \mathsf{p}_1), (\mathsf{p}_1, \hookrightarrow_r, \mathsf{o}) \rightarrow (\mathsf{p}, \hookrightarrow_r, \mathsf{o})$	
	`rdfs2`	$(\mathsf{p}, \hookleftarrow_d, \mathsf{o}), (\mathsf{s}_1, \mathsf{p}, \mathsf{o}_1) \rightarrow (\mathsf{s}_1, \tau, \mathsf{o})$	$\mathcal{R}_a$
	`rdfs3`	$(\mathsf{p}, \hookrightarrow_r, \mathsf{o}), (\mathsf{s}_1, \mathsf{p}, \mathsf{o}_1) \rightarrow (\mathsf{o}_1, \tau, \mathsf{o})$	
	`rdfs7`	$(\mathsf{p}_1, \prec_{sp}, \mathsf{p}_2), (\mathsf{s}, \mathsf{p}_1, \mathsf{o}) \rightarrow (\mathsf{s}, \mathsf{p}_2, \mathsf{o})$	
	`rdfs9`	$(\mathsf{s}, \prec_{sc}, \mathsf{o}), (\mathsf{s}_1, \tau, \mathsf{s}) \rightarrow (\mathsf{s}_1, \tau, \mathsf{o})$	

Table 3: Sample RDFS entailment rules.

2.2 RDF Entailment Rules

An *entailment rule* r has the form body(r) $\rightarrow$ head(r), where body(r) and head(r) are RDF graphs, respectively called *body* and *head* of the rule r. In this work, we consider the **RDFS entailment rules** $\mathcal{R}$ shown in Table 3, which are the most frequently used; in the table, all values except RDF reserved IRIs are blank nodes. For instance, rule `rdfs5` reads: whenever in an RDF graph, a property p_1 is a subproperty of a property p_2, and further p_2 is a subproperty of p_3 (body of `rdfs5`), it follows that p_1 is a subproperty of p_3 (head of `rdfs5`). Similarly, rule `rdfs7` states that if p_1 is a subproperty of p_2 and a resource s has the value o for p_1, then s also has o as a value for p_2. The triples $(\mathsf{p}_1, \prec_{sp}, \mathsf{p}_3)$ and $(\mathsf{s}, \mathsf{p}_2, \mathsf{o})$ in the above examples are called **implicit**, i.e., they hold in a graph thanks to the entailment rules, even if they may not be **explicit**ly present in the graph. Following [12], we view $\mathcal{R}$ as partitioned into two subsets: the rules $\mathcal{R}_c$ lead to implicit schema triples, while rules $\mathcal{R}_a$ lead to implicit data triples[1]. The **direct entailment** of an RDF graph G with a set of RDF entailment rules $\mathcal{R}$, denoted by $C_{G,\mathcal{R}}$, is the set of implicit triples resulting from rule applications that use solely the explicit triples of G. For instance, the rule `rdfs9` applied to the graph G_{ex}, which comprises (:NatComp, $\prec_{sc}$, :Comp), (_:b_c, τ, :NatComp), leads to the implicit triple (_:b_c, τ, :Comp). This triple belongs to $C_{G_{\text{ex}},\mathcal{R}_a}$ (hence also to $C_{G_{\text{ex}},\mathcal{R}}$). The **saturation** of an RDF graph allows materializing its semantics, by iteratively augmenting it with the triples it entails using entailment rules, until reaching a fixpoint; this process is finite [48]. Formally:

Definition 2.3 (RDF graph saturation). Let G be an RDF graph and $\mathcal{R}$ a set of entailment rules. We recursively define a sequence $(G_i)_{i\in\mathbb{N}}$ of RDF graphs as follows: $G_0 = G$, and $G_{i+1} = G_i \cup C_{G_i,\mathcal{R}}$ for $i \geq 0$. The *saturation* of G w.r.t. $\mathcal{R}$, denoted $G^{\mathcal{R}}$, is G_n for n the smallest integer such that $G_n = G_{n+1}$.

Example 2.4 (Saturation). The saturation of G_{ex} w.r.t. the set $\mathcal{R}$ of RDFS entailment rules shown in Table 3 is attained after the following *two* saturation steps:

$(G_{\text{ex}})_1 = G_{\text{ex}} \cup$
{(:NatComp, $\prec_{sc}$, :Org),
(:hiredBy, $\hookleftarrow_d$, :Person), (:hiredBy, $\hookrightarrow_r$, :Org),
(:ceoOf, $\hookleftarrow_d$, :Person), (:ceoOf, $\hookrightarrow_r$, :Org),
(:p$_1$, :worksFor, _:b_c), (_:b_c, τ, :Comp),
(:p$_2$, :worksFor, :a), (:a, τ, :Org)}

$(G_{\text{ex}})_2 = (G_{\text{ex}})_1 \cup$
{(:p$_1$, τ, :Person), (:p$_2$, τ, :Person), (_:b_c, τ, :Org)}

2.3 Basic Graph Pattern Queries

A popular RDF query dialect consists of **basic graph pattern queries**, or **BGPQs**, in short. Let $\mathscr{V}$ be a set of variable symbols, disjoint from $\mathscr{I} \cup \mathscr{B} \cup \mathscr{L}$. A basic graph pattern (BGP) is a set of *triple patterns* (triples in short) belonging to $(\mathscr{I} \cup \mathscr{B} \cup \mathscr{V}) \times (\mathscr{I} \cup \mathscr{V}) \times (\mathscr{I} \cup \mathscr{B} \cup \mathscr{L} \cup \mathscr{V})$. For a BGP P, we denote by Var(P) the set of variables occurring in P, by Bl(P) its set of blank nodes, and by Val(P) its set of values (IRIs, blank nodes, literals and variables).

Definition 2.5 (BGP Queries). A *BGP query* q is of the form $q(\bar{x}) \leftarrow P$, where P is a BGP (also denoted by body(q)), and $\bar{x} \subseteq$ Var(P) are the answer variables of q.

To ease the presentation, and without loss of generality, we consider BGPQs *without* blank nodes, as it is well-known that these can be replaced by non-answer variables [46].

For query answering based on query reformulation (see Section 2.4), it is convenient to slightly generalize BGPQs into **partially instantiated BGPQs** [12, 29]. Starting from a BGPQ q, partial instantiation may replace *some* variables with values from $\mathscr{I} \cup \mathscr{L} \cup \mathscr{B}$, as specified by a substitution σ. Due to σ, and in contrast with standard BGPQs, some answer variables of the resulting query q_σ can be bound:

Example 2.6 (Partially instantiated BGPQ). Consider the BGPQ asking for *who is working for which kind of company* $q(x, y) \leftarrow (x, \text{:worksFor}, z), (z, \tau, y), (y, \prec_{sc}, \text{:Comp})$ and the substitution $\sigma = \{x \mapsto \text{:p}_1\}$. The corresponding partially instantiated BGPQ is: $q(\text{:p}_1, y) \leftarrow (\text{:p}_1, \text{:worksFor}, z), (z, \tau, y), (y, \prec_{sc}, \text{:Comp})$. In it, the first answer variable has been bound to :p$_1$.

For simplicity, below we use the term "query" to designate either a standard BGPQ or a partially instantiated BPGQ.

The semantics of a BGPQ on an RDF graph is defined through standard **homomorphisms** from the query body to the queried graph. We recall that a homomorphism from a BGP P to an RDF graph G is a function φ from Val(P) to Val(G) such that for any triple $(s, p, o) \in P$, the triple $(\varphi(s), \varphi(p), \varphi(o))$ is in G, with φ the identity on IRIs and literals. Next, we distinguish **query evaluation**, whose result is just based on the explicit triples of the graph, i.e., on BGP-to-RDF graph homomorphisms, from **query answering** that also accounts for the implicit graph triples, resulting from entailment. Formally:

Definition 2.7 (Evaluation and answering). The *answer set* to a BGPQ q on an RDF graph G w.r.t. a set $\mathcal{R}$ of RDF entailment rules is: $q(G, \mathcal{R}) = \{\varphi(\bar{x}) \mid \varphi \textit{ homomorphism from } \text{body}(q) \textit{ to } G^{\mathcal{R}}\}$. If $\bar{x} = \emptyset$, q is a *Boolean* query, in which case q is false when $q(G, \mathcal{R}) = \emptyset$ and true when $q(G, \mathcal{R}) = \{\langle\rangle\}$, i.e., the answer to q is the empty tuple.

The *evaluation* of q on G, denoted $q(G, \emptyset)$ or $q(G)$ in short, is obtained from homomorphisms from body(q) to G alone (not $G^{\mathcal{R}}$). It can be seen as a particular case of query answering when $\mathcal{R} = \emptyset$.

Example 2.8 (Evaluation and answering). Consider again the BGPQ q from the preceding example. Its evaluation on G_{ex} is empty because G_{ex} has no explicit :worksFor assertion, while its answer set on G_{ex} w.r.t. $\mathcal{R}$ is $\{\langle \text{:p}_1, \text{:NatComp}\rangle\}$ because :p$_1$ being CEO of _:b_c, :p$_1$ implicitly works for it, and _:b_c is explicitly a company of the particular type :NatComp.

The above notions and notations naturally extend to *unions* of (partially instantiated) BGPQs, or **UBGPQ**s in short.

We end this section by pointing out that many RDF data management systems use *saturation-based query answering*, which directly follows the definition of query answering: an RDF graph G is first saturated with the set $\mathcal{R}$ of entailment rules, so that the

[1]In the notations $\mathcal{R}_c$ and $\mathcal{R}_a$, c and a respectively stand for "constraint triples" (called schema triples here) and "assertion triples" (data triples).

answer set to an incoming query q is obtained through query evaluation as $q(G^{\mathcal{R}})$.

2.4 Query Reformulation

Reformulation-based query answering is an alternative technique to the widely adopted saturation-based query answering. It consists in *reformulating* a query using $\mathcal{R}$, so that evaluating the reformulated query on G yields the answer set to the original query on G w.r.t. $\mathcal{R}$. Intuitively, reformulation injects the ontological knowledge into the query, just as saturation injects it into the RDF graph. We rely here on the very recent algorithm from [12], which takes into account all the entailment rules from Table 3. The process is decomposed into *two steps* according to the partition of $\mathcal{R}$ into $\mathcal{R}_a$ and $\mathcal{R}_c$.
(i) The first step reformulates a BGPQ q w.r.t. an RDFS ontology O and the set of rules $\mathcal{R}_c$ into a UBGPQ, say Q_c, which is guaranteed not to contain ontology triples. Intuitively, this step generates new BGPQs obtained from q by instantiating variables that query the ontology with all their possible bindings; for instance, y in a query triple $(y, \prec_{sc}, \text{:Comp})$ is bound to the IRIs of all explicit and implicit subclasses of :Comp in O. This step, alone, is sound and complete w.r.t. $\mathcal{R}_c$ for query answering, i.e., for any graph G with ontology O, $q(G, \mathcal{R}_c) = Q_c(G)$.
(ii) The second step reformulates Q_c w.r.t. O and $\mathcal{R}_a$, and outputs a UBGPQ, say $Q_{c,a}$. This step, alone, is sound and complete w.r.t. $\mathcal{R}_a$ for query answering, i.e., for any graph G with ontology O, $Q_c(G, \mathcal{R}_a) = Q_{c,a}(G)$. Furthermore, a key property is that $q(G, \mathcal{R}) = Q_c(G, \mathcal{R}_a)$, i.e., only $\mathcal{R}_a$ needs to be considered to answer Q_c with respect to the entire set of rules $\mathcal{R}$. This is the fundamental reason why the sucessive application of these two reformulation steps leads to a sound and complete reformulation-based query answering technique: $q(G, \mathcal{R}) = Q_{c,a}(G)$.

Example 2.9 (Two-step reformulation). Consider the query $q(x, y) \leftarrow (x, \text{:worksFor}, z), (z, \tau, y), (y, \prec_{sc}, \text{:Comp})$ from the preceding example and the ontology O in Example 2.2. The first reformulation step instantiates the triple $(y, \prec_{sc}, \text{:Comp})$ on O, leading to:
$Q_c = q(x, \text{:NatComp}) \leftarrow (x, \text{:worksFor}, z), (z, \tau, \text{:NatComp})$.
Then, Q_c is reformulated into $Q_{c,a}$ =
$q(x, \text{:NatComp}) \leftarrow (x, \text{:worksFor}, z), (z, \tau, \text{:NatComp}) \cup$
$q(x, \text{:NatComp}) \leftarrow (x, \text{:hiredby}, z), (z, \tau, \text{:NatComp}) \cup$
$q(x, \text{:NatComp}) \leftarrow (x, \text{:ceoOf}, z), (z, \tau, \text{:NatComp})$
by specializing :worksFor according to its subproperties in O. It can be checked that $Q_{c,a}(G_{\text{ex}}) = q(G_{\text{ex}}, \mathcal{R}) = q(G_{\text{ex}}^{\mathcal{R}}) = \{\langle \text{:p}_1, \text{:NatComp}\rangle\}$, obtained here from the third BGPQ in $Q_{c,a}$.

2.5 Query Rewriting-based Data Integration

We recall now the basics of relational view-based query rewriting (Section 2.5.1), which has been extensively studied [23, 31]. Then we present a generalization of the notion of views as mappings [35] (Section 2.5.2).

2.5.1 View-based (LAV) Data Integration.

An integration system $\mathcal{I}$ is made of a *global schema* S (a set of relations) and a set $\mathcal{V}$ of *views*. An instance of $\mathcal{I}$ assigns a set of tuples to each relation of S and to each view of $\mathcal{V}$. The data stored in a view is called its *extension*. Further, to each view V is associated a query $V(\bar{x})$:- $\psi(\bar{x})$ over the global schema S, specifying how its data fits into S. Accordingly, this framework is called local-as-view (LAV) data integration. For instance, let S consist of three relations *Emp(eID, name, dID), Dept(dID, cID, country), Salary(eID,amount)*, where *eID*, *dID* and *cID* are respectively identifiers for employees, departments and companies.
Consider the views V_1*(eID, name, country) :- Emp(eID, name, dID), Dept(dID, "IBM", country)* providing the names of IBM employees and where they work, and V_2*(eID, amount) :- Emp(eID, name, "R&D"), Salary(eID,amount)*, which indicates the salaries of employees in R&D departments. Typically, no single view is expected to bring *all* information of a given kind; for instance, V_1 brings *some* IBM employees, but other views may bring others, e.g., V_2, possibly overlapping with V_1; this is called the "Open World Assumption" (OWA).
In an OWA setting, we are interested in *certain answers* [31], i.e., those that are sure to be part of the query result, knowing the data present in the views. Such answers can be computed by *rewriting* a query over S, into one over the views $\mathcal{V}$; evaluating the rewriting over the view extensions produces the answers.
Ideally, a rewriting should be equivalent to the query over S, i.e., compute exactly the same answers. However, depending on the views and queries, such a rewriting may not exist. For instance, the query $q(n, a)$:- $Emp(e, n, d), Dept(d, c, \text{"France"}), Salary(e, a)$ does not have an equivalent rewriting using V_1 and V_2, because V_1 only provides IBM employees working in France, while V_2 only has salaries of employees of R&D departments. A *maximally contained rewriting* brings all the query answers that can be obtained through the given set of views; the rewriting may be not be equivalent to q (but just contained in q). In our example, $q_r(n, a)$:- $V_1(e, n, \text{"France"}), V_2(e, a)$ is a maximally contained rewriting of q; it returns employees of French IBM R&D departments with their salary, clearly a subset of q answers.
A remarkable result holds for *(unions of) conjunctive queries ((U)CQs)*, *conjunctive views* (views V such that the associated query $V(\bar{x})$:- $\psi(\bar{x})$ is a CQ) and *rewritings that are UCQs*: any maximally contained rewriting computes exactly the certain answers [2]; we will build upon this result for answering queries in our RDF integration systems.

2.5.2 GLAV Data Integration.

The above setting has been generalized to views that are not necessarily stored as such, but just queries over some *underlying data source*. For instance, assuming a data source D holds the relations *Person(eID, name)* and *Contract(eID, dID, country)* (see Figure 1) with people and their work contracts at IBM, the view V_1 from the above example may be *defined on* D by the query V_1^D over the D schema shown in the figure (note that V_1^D hides the department from system $\mathcal{I}$); V_1^D provides the extension of V_1. Similarly, view V_2 may be defined as a query over some data source (or sources).

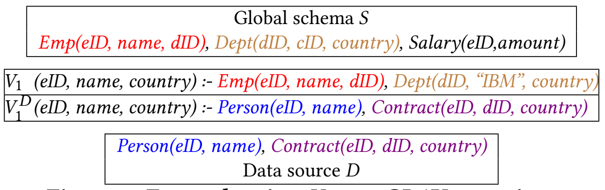

Global schema S
Emp(eID, name, dID), Dept(dID, cID, country), Salary(eID,amount)
V_1 *(eID, name, country) :- Emp(eID, name, dID), Dept(dID, "IBM", country)*
V_1^D*(eID, name, country) :- Person(eID, name), Contract(eID, dID, country)*
Person(eID, name), Contract(eID, dID, country)
Data source D

Figure 1: Example: view V_1 as a GLAV mapping.

Query rewriting is unchanged, whether the views are stored or defined by source queries. In the latter case, to obtain answers, a view-based rewriting needs to be *unfolded*, replacing every occurence of a view symbol V with the body of the source query defining that view. Executing the resulting query (potentially over different data sources) computes the answers. This integration setting, which considers views as intermediaries between sources and the integration schema, has been called "global-local-as-view" (GLAV) [26]. An association of a query q_1 over the data sources and another query q_2 over the global schema, both with the same answer variables, e.g., $q_1 = V_1^D$ and $q_2 = V_1$ above, is commonly called a *GLAV mapping* (denoted $q_1(\bar{x}) \rightsquigarrow q_2(\bar{x})$).

Historically, two restrictions of GLAV mappings have been investigated. First, *global-as-view* (or GAV) mappings define global schema relations as views over the local schemas. Specifically, a GAV mapping $q_1(\bar{x}) \rightsquigarrow q_2(\bar{x})$, q_2 defines a single element of the global schema (hence $body(q_2)$ is restricted to a single atom if q_2 is a CQ, or a single triple pattern if q_2 is a BGPQ) and its variables are exactly $\bar{x}$. Second, *local-as-view* (LAV) mappings express elements of the local schema as views over the global schema, similarly to the views described in Section 2.5.1.

Importantly, unlike GAV mappings, GLAV ones do not require all variables of q_2 to be answer variables (e.g., *dID* in V_1 in Figure 1); this makes integration more powerful. For example, suppose that ⟨1,*"John Doe"*,*"France"*⟩ is an answer to V_1^D above. Then, V_1 exposes this tuple in the global schema as: *Emp*(1,*"John Doe"*,x), *Dept*(x,*"IBM"*,"France"), stating that John Doe works for a department x located in France. Here, x is an existential variable (called "labeled null" in [3]); the GLAV mapping *states the existence* of such a department in the global schema, even if its identifier is unknown (because it is not provided by V_1). Therefore, John Doe is a certain answer to a query asking for all employees in IBM departments, based on the above GLAV mapping. This answer cannot be found using GAV mappings.

3 PROBLEM STATEMENT

In this section, we first formalize the notion of *RDF integration system* (Section 3.1). Then, we state the associated query answering problem (Section 3.2), for which Section 4 provides solutions.

3.1 RDF integration system (RIS)

In an RDF integration system (RIS in short), data from heterogeneous sources, each of which may have its own data model and query language, is integrated into an RDF graph, consisting of an (RDFS) ontology and of data triples derived from the sources by means of GLAV-style mappings. Mappings allow (*i*) specifying the data made available from the sources, and (*ii*) organizing it according to the RIS ontology.

Definition 3.1 (RIS mappings and extensions).
A *RIS mapping* m is of the form $m = q_1(\bar{x}) \rightsquigarrow q_2(\bar{x})$ where q_1 and q_2 are two queries with the same answer variables, and q_2 is a BGPQ whose body contains only triples of the forms:

- $(\mathsf{s}, p, \mathsf{o})$ where $p \in \mathscr{I}_{\text{user}}$,
- (s, τ, C) where $C \in \mathscr{I}_{\text{user}}$.

The *body* of m is q_1 and its *head* is q_2. The *extension* of m is the set of tuples $\text{ext}(m) = \{V_m(\delta(v_1), \dots, \delta(v_n)) \mid \langle v_1, \dots, v_n \rangle \in q_1(D)\}$, where $q_1(D)$ is the answer set of q_1 on the data source D that m integrates and δ is a function that maps source values to RDF values, i.e., IRIs, blank nodes and literals.

Intuitively, m specifies that the result of query q_1 on D transformed in RDF, i.e., the extension of m, is exposed to the RIS as the result of the (BGP) query q_2.

Example 3.2 (Mappings). Consider the two mappings:
m_1 with head $q_2(x) \leftarrow (x, \text{:ceoOf}, y), (y, \tau, \text{:NatComp})$ and
m_2 with head $q_2(x, y) \leftarrow (x, \text{:hiredBy}, y), (y, \tau, \text{:PubAdmin})$.
Suppose that the body of m_1 returns $\langle p^{D_1} \rangle$ as its results, and that the δ function maps the value $p_1^{D_1}$ from the data source D_1 to the IRI :p$_1$. Then, the extension of m_1 is: $\text{ext}(m_1) = \{V_{m_1}(\text{:p}_1)\}$. Further, suppose that the body of m_2 returns $\langle p_2^{D_2}, a^{D_2} \rangle$, and that δ maps the values $p_2^{D_2}, a^{D_2}$ from the data source D_2 to the IRIs :p$_2$, :a. Then, the extension of m_2 is: $\text{ext}(m_2) = \{V_{m_2}(\text{:p}_2, \text{:a})\}$.

Given a set of RIS mappings $\mathcal{M}$, the *extent* of $\mathcal{M}$ is the union of the mappings' extensions, i.e., $\mathcal{E} = \bigcup_{m \in \mathcal{M}}$ *ext(m)*, and we denote by $\text{Val}(\mathcal{E})$ the set of values occurring in $\mathcal{E}$. We can now define the RIS data triples *induced* by some mappings and an extent thereof. These are *all the data which is exposed (can be queried) through a RIS.*

Definition 3.3 (RIS data triples). Given a set $\mathcal{M}$ of RIS mappings and an extent $\mathcal{E}$ of $\mathcal{M}$, the *RIS data triples* induced by $\mathcal{M}$ and $\mathcal{E}$ form an RDF graph defined as follows:

$$G_{\mathcal{E}}^{\mathcal{M}} = \bigcup_{m = q_1(\bar{x}) \rightsquigarrow q_2(\bar{x}) \in \mathcal{M}} \{\text{bgp2rdf}(\text{body}(q_2)_{[\bar{x} \leftarrow \bar{t}]})) \mid V_m(\bar{t}) \in \mathcal{E}\}$$

where

- $\text{body}(q_2)_{[\bar{x} \leftarrow \bar{t}]}$ is the BGP $\text{body}(q_2)$ in which the answer variables $\bar{x}$ are bound to the values in the tuple $V_m(\bar{t})$, part of $\mathcal{E}$;
- $\text{bgp2rdf}(\cdot)$ is a function that transforms a BGP into an RDF graph, by replacing each variable with a fresh blank node.

Observe that, because we use GLAV mappings, RIS data triples may include fresh blank nodes, as exemplified below; these correspond to the existential variables allowed in GLAV mappings, as discussed at the end of Section 2.5.2.

Example 3.4. Reusing the mappings from Example 3.2, let $\mathcal{M} = \{m_1, m_2\}$ and its extent $\mathcal{E} = \{V_{m_1}(\text{:p}_1), V_{m_2}(\text{:p}_2, \text{:a})\}$. The RIS data triples they lead to are:

$$G_{\mathcal{E}}^{\mathcal{M}} = \{(\text{:p}_1, \text{:ceoOf}, \_\text{:}b_c), (\_\text{:}b_c, \tau, \text{:NatComp}), (\text{:p}_2, \text{:hiredBy}, \text{:a}), (\text{:a}, \tau, \text{:PubAdmin})\}$$

In particular, the first and second triples contain the blank node $\_\text{:}b_c$, introduced by bgp2rdf instead of the variable y in the head (query q_2) of m_1. Importantly, only non-answer variables in a mapping head lead to blank nodes introduced this way: by Def. 3.3, answer variables (here x for m_1 and x, y for m_2) are replaced with values from $V_m(\bar{t})$, thus from $\text{Val}(\mathcal{E})$.

Finally, we **define a RIS** as a tuple $S = \langle O, \mathcal{R}, \mathcal{M}, \mathcal{E} \rangle$ stating that S allows to access (query), with the reasoning power given by the set $\mathcal{R}$ of RDFS entailment rules, the RDF graph comprising the ontology O and the data triples induced by the set of mappings $\mathcal{M}$ and their extent $\mathcal{E}$.

3.2 Query answering problem

The problem we consider is answering BGPQs in a RIS. We define certain answers in a RIS setting as follows:

Definition 3.5 (Certain answer set). The *certain answer set* of a BGPQ q on a RIS $S = \langle O, \mathcal{R}, \mathcal{M}, \mathcal{E} \rangle$ is:
$\text{cert}(q, S) = \{\varphi(\bar{x}) \mid \varphi$ *homomorphism from* $\text{body}(q)$ *to* $(O \cup G_{\mathcal{E}}^{\mathcal{M}})^{\mathcal{R}}\}$
where $\varphi(\bar{x})$ comprises only values from $\text{Val}(\mathcal{E})$.

The certain answer set $\text{cert}(q, S)$ is thus the subset of $q(O \cup G_{\mathcal{E}}^{\mathcal{M}}, \mathcal{R})$ restricted to tuples fully built from source values, i.e., we exclude tuples with blank nodes introduced by the mappings (see Def. 3.3). Note, however, that blank nodes can be exploited to answer queries, as shown below.

Example 3.6 (Certain answers). Consider the RIS S made of the ontology O of G_{ex} in Example 2.2, the set $\mathcal{R}$ of entailment rules shown in Table 3, and the set of mappings $\mathcal{M}$ together with the extent $\mathcal{E}$ from Example 3.4.

Let $q(x, y) \leftarrow (x, \text{:worksFor}, y), (y, \tau, \text{:Comp})$ be the query asking "who works for which company", while the query $q'(x) \leftarrow (x, \text{:worksFor}, y), (y, \tau, \text{:Comp})$ asks "who works for some company". The only difference between them is that y is an answer variable in q and not in q'. The certain answer set of q is $\emptyset$,

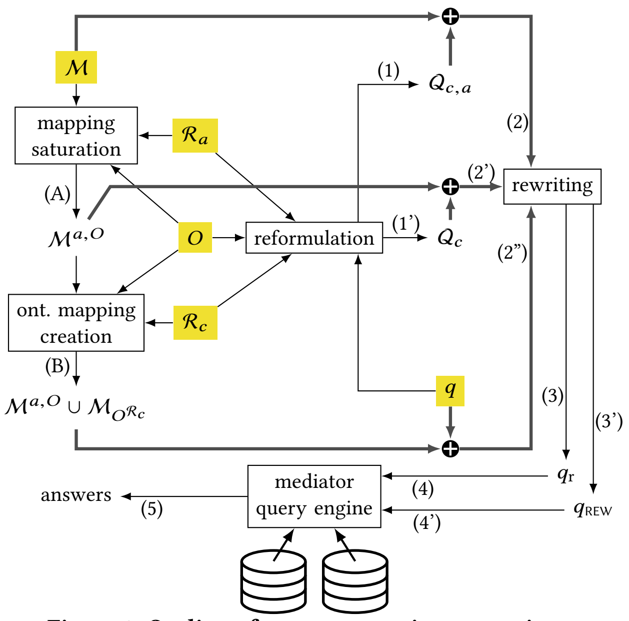


Figure 2: Outline of query answering strategies.

while the certain answer set of q' is $\{\langle :\text{p}_1 \rangle\}$. This answer results from the RIS data triples $(:\text{p}_1, :\text{worksFor}, \_:b_c)$, $(\_:b_c, \tau, :\text{Comp})$, which are entailed from:

- the $G_{\mathcal{E}}^{\mathcal{M}}$ triples $(:\text{p}_1, :\text{ceoOf}, \_:b_c)$, $(\_:b_c, \tau, :\text{NatComp})$, with the blank node $\_:b_c$ discussed in Example 3.4, and:
- either the O triples $(:\text{ceoOf}, \prec_{sp}, :\text{worksFor})$, $(:\text{ceoOf}, \hookrightarrow_r, :\text{Comp})$ together with the $\mathcal{R}$ rules `rdfs3`, `rdfs7`, or the O triples $(:\text{ceoOf}, \prec_{sp}, :\text{worksFor})$, $(:\text{NatComp}, \prec_{sc}, :\text{Comp})$ together with the $\mathcal{R}$ rules `rdfs3`, `rdfs9`.

The query q has no answer because it requires a value not available from the source: the company for which $:\text{p}_1$ works; the RIS only knows the existence of such value through the blank node $\_:b_c$ begotten by bgp2rdf in its data triples. In contrast, q' allows finding out that $:\text{p}_1$ works for (as CEO of) some (national) company, even though the mapping m_1 (the only one involving companies) does not expose the company IRI through the RIS.

The problem we study in the next section is:

Problem 1. *Given a RIS S, compute the certain answer set of a BGPQ q on S, i.e.,* cert(q, S).

4 QUERY ANSWERING IN A RIS

Since we adopt a mediator-style approach, the RIS data triples $G_{\mathcal{E}}^{\mathcal{M}}$ are not materialised, hence the saturation of $O \cup G_{\mathcal{E}}^{\mathcal{M}}$ cannot be computed to answer queries as defined above. Instead, queries are rewritten in terms of the remote heterogeneous sources, based on the RIS ontology O, reasoning power $\mathcal{R}$ and mappings $\mathcal{M}$. We present three query answering strategies, which differ in how the ontological reasoning is incorporated: we may have *all*, *some* or *no* reasoning performed at query time, as outlined in Figure 2.

All reasoning at query time The first strategy will be detailed in Section 4.1. First, it reduces the RIS query answering problem to *standard query evaluation* in an RDF data management system, by reformulating (step (1) in Figure 2) the query q based on the RIS ontology O and entailment rules $\mathcal{R} = \mathcal{R}_c \cup \mathcal{R}_a$. The obtained reformulated query $Q_{c,a}$ thus yields the expected certain answers when evaluated on the RIS data triples (recall Section 2.4), provided that answers with blank nodes introduced by the *bgp2rdf* function are discarded (recall Section 3.2). Since these data triples are not materialized, the RDF query evaluation problem is in turn reduced to *relational view-based query answering*, by rewriting $Q_{c,a}$ using the RIS GLAV mappings $\mathcal{M}$ seen as LAV views (step (2)). This produces a relational rewriting q_r over the mappings extension (step (3)), whose evaluation with a mediator query engine provides the desired certain answers (steps (4) and (5)).

Some reasoning at query time The second strategy (detailed in Section 4.2) is a main contribution of this paper. First, it reduces the RIS query answering problem to *saturation-based query answering* by reformulating (step (1')) the query q based on O and $\mathcal{R}_c$ only (not $\mathcal{R} = \mathcal{R}_c \cup \mathcal{R}_a$ as above). The obtained reformulation Q_c thus yields the expected certain answer set when evaluated on the RIS data triples *saturated with* $\mathcal{R}_a$ (recall Section 2.4), again provided that the answers with blank nodes introduced by the *bgp2rdf* function are discarded (as above). Since these triples are not materialized in a RIS, hence cannot be saturated with $\mathcal{R}_a$, the saturation-based query answering problem is in turn reduced to *relational view-based query answering*, by rewriting Q_c using the RIS GLAV mappings *saturated O and $\mathcal{R}_a$*, seen as LAV views. These saturated mappings, denoted $\mathcal{M}^{a,O}$, are obtained (step (A)) from the original ones by adding to their head queries (q_2) all the implicit data triples they model w.r.t. O and $\mathcal{R}_a$. Then, the partially reformulated query Q_c is rewritten using $\mathcal{M}^{a,O}$ (step (2')) and the resulting query (step (3)) is evaluated as in the first strategy (steps (4) and (5)). Importantly, mappings are saturated offline, and need to be updated only when some mapping changes. This limits both the reasoning effort at query time *and* the complexity of the reformulated query to rewrite, hence the rewriting time needed to obtain a rewriting q_r over the data sources, as our experiments show (Section 5).

No reasoning at query time Finally, the third strategy (detailed in Section 4.3) reduces the RIS query answering problem directly to view-based query answering. Here, the mappings are saturated offline as above (step (A)), in order to model all explicit and implicit RIS data triples. Also, these mappings are complemented with a set of mappings, noted $\mathcal{M}_{O^{\mathcal{R}_c}}$ (step (B)), comprising all the explicit and implicit RIS schema triples w.r.t. O and $\mathcal{R}$; since only $\mathcal{R}_c$ rules entail new schema triples (Table 3), $O^{\mathcal{R}}$ is actually equal to $O^{\mathcal{R}_c}$. This second set of mappings is also computed offline, and only needs to be updated when the ontology changes. A query q just needs to be rewritten based on the above mappings $\mathcal{M}^{a,O} \cup \mathcal{M}_{O^{\mathcal{R}_c}}$ seen as LAV views (step (2")), in order to obtain, as above, a rewriting q_{REW} over the data sources (step(3'), followed by the evaluation steps (4') and (5)).

Before going into the technical details of the above strategies, we introduce a set of simple functions. The *bgp2ca* function transforms a BGP into a conjunction of atoms with ternary predicate T (standing for "triple") as follows: $bgp2ca(\{(\text{s}_1, \text{p}_1, \text{o}_1), \ldots, (\text{s}_n, \text{p}_n, \text{o}_n)\}) = T(\text{s}_1, \text{p}_1, \text{o}_1) \wedge \cdots \wedge T(\text{s}_n, \text{p}_n, \text{o}_n)$. The *bgpq2cq* function transforms a BGPQ $q(\bar{x}) \leftarrow body(q)$ into a CQ $q(\bar{x}) \leftarrow bgp2ca(body(q))$. Finally, the function *ubgpq2ucq* function transforms a UBGPQ $\bigcup_{i=1}^{n} q_i(\bar{x}_i)$ into a UCQ by applying the above *bgpq2cq* function to each of its q_i.

4.1 Rewriting Fully-Reformulated Queries using Mappings as Views: REW-CA

Based on [12], the first step of this strategy, (1) in Figure 2, reformulates a query q w.r.t. O and $\mathcal{R} = \mathcal{R}_c \cup \mathcal{R}_a$ into a query $Q_{c,a}$. This allows obtaining the certain answers directly from the RIS data triples, and not from their saturation after they have been augmented with O (recall Definition 3.5). Indeed, the correctness of the reformulation ensures that the certain answers of q on the RIS S correspond precisely to those of $Q_{c,a}$ asked on S when disregarding O and $\mathcal{R}$, as formally expressed in the next lemma. Of course, this still does not provide a concrete solution to obtain the desired certain answers using standard query evaluation, since the RIS data triples $G^{\mathcal{M}}_{\mathcal{E}}$ are not materialized.

Lemma 4.1. *Let $S = \langle O, \mathcal{R}, \mathcal{M}, \mathcal{E}\rangle$ be a RIS, q be a BGPQ and $Q_{c,a}$ its UBGPQ reformulation w.r.t. $O, \mathcal{R} = \mathcal{R}_c \cup \mathcal{R}_a$ using [12]. Then:*

$$\text{cert}(q, S) = \text{cert}(Q_{c,a}, \langle \emptyset, \emptyset, \mathcal{M}, \mathcal{E}\rangle)$$

The proof of this and our following claims can be found in [13]. Recall that the RIS data triples are defined from the mappings $\mathcal{M}$ by, for every mapping $m = q_1(\bar{x}) \rightsquigarrow q_2(\bar{x}) \in \mathcal{M}$, (i) evaluating the mapping body $q_1(\bar{x})$ on the data source to produce its extension $\text{ext}(m) \in \mathcal{E}$, and then (ii) instantiating the mapping head $q_2(\bar{x})$ with its extension. At the same time, this is also how the instance of a data integration system based on LAV views and their extensions is defined in a relational setting (Section 2.5.1)! Based on this analogy, we *recast the RIS query answering problem* of the above Lemma, into a *relational view-based query answering* one. To this aim, we treat our mappings as LAV views:

Definition 4.2 (Mappings as relational LAV views). Let $m = q_1(\bar{x}) \rightsquigarrow q_2(\bar{x})$ be a mapping. Its corresponding relational LAV view is: $V_m(\bar{x}) \leftarrow \mathit{bgp2ca}(\text{body}(q_2))$.

Example 4.3. The relational LAV views corresponding to the mappings m_1, m_2 from Example 3.2 are:

- $V_{m_1}(x) \leftarrow T(x, \text{:ceoOf}, y), T(y, \tau, \text{:NatComp})$
- $V_{m_2}(x, y) \leftarrow T(x, \text{:hiredBy}, y), T(y, \tau, \text{:PubAdmin})$

We denote the set of views derived from all the mappings $\mathcal{M}$ by Views($\mathcal{M}$). Crucially, the extent $\mathcal{E}$ of the mapping set $\mathcal{M}$ is also an extent for the corresponding set of views Views($\mathcal{M}$).

Based on the above Lemma 4.1, treating mappings and their extent as relational LAV views and their extent, and seeing (U)BGPQs as (U)CQs with the help of the functions introduced in the beginning of Section 4, we reduce the RIS query answering problem to view-based query answering:

Theorem 4.4 (REW-CA correctness). *Let $S = \langle O, \mathcal{R}, \mathcal{M}, \mathcal{E}\rangle$ be a RIS and q be a BGPQ. Let $Q_{c,a}$ be the reformulation of q w.r.t. O and $\mathcal{R}$ using [12]. Then:*

$$\text{cert}(q, S) = \text{cert}(\mathit{ubgpq2ucq}(Q_{c,a}), \text{Views}(\mathcal{M}), \mathcal{E})$$

where $\text{cert}(\mathit{ubgpq2ucq}(Q_{c,a}), \text{Views}(\mathcal{M}), \mathcal{E})$ *denotes the certain answer set of* $\mathit{ubgpq2ucq}(Q_{c,a})$ *over* Views($\mathcal{M}$) *and* $\mathcal{E}$.

Importantly, this provides an effective solution to RIS query answering problem by using state-of-the-art view-based query rewriting techniques [31], in particular for step (2) in Figure 2.

Example 4.5 (REW-CA query answering). Consider again the RIS in Example 3.6 and the query $q(x, y) \leftarrow (x, y, z), (z, \tau, t), (y, \prec_{sp}, \text{:worksFor}), (t, \prec_{sc}, \text{:Comp}), (x, \text{:worksFor}, a), (a, \tau, \text{:PubAdmin})$ asking "who works for some public administration, and what working relationship he/she has with some company". Its UBGPQ reformulation, seen as a UCQ, is shown in Figure 3. Its maximally-contained rewriting based on the views obtained from the RIS mappings is: $q_r(x, \text{:ceoOf}) \leftarrow V_{m_1}(x), V_{m_2}(x, y)$, obtained from the second CQ in the above union. This becomes clear when the views are replaced by their bodies: $q(x, \text{:ceoOf}) \leftarrow T(x, \text{:ceoOf}, y_1), T(y_1, \tau, \text{:NatComp}), T(x, \text{:hiredBy}, y_2), T(y_2, \tau, \text{:PubAdmin})$. Note that the other CQs cannot be rewritten given the available views. With the current RIS, this rewriting yields an empty certain answer set to q, i.e., $\text{cert}(q, S) = \emptyset$, because the extent of the mappings, hence of the views, is: $\mathcal{E} = \{V_{m_1}(\text{:p}_1), V_{m_2}(\text{:p}_2, \text{:a})\}$. However, if we add $V_{m_2}(\text{:p}_1, \text{:a})$ to $\mathcal{E}$, then $\text{cert}(q, S) = \{\langle \text{:p}_1, \text{:ceoOf}\rangle\}$.

$$\begin{aligned}
Q_{c,a} = \quad & q(x, \text{:ceoOf}) \leftarrow && T(x, \text{:ceoOf}, z), T(z, \tau, \text{:NatComp}),\\
& && T(x, \text{:worksFor}, a), T(a, \tau, \text{:PubAdmin})\\
\cup\ & q(x, \text{:ceoOf}) \leftarrow && T(x, \text{:ceoOf}, z), T(z, \tau, \text{:NatComp}),\\
& && T(x, \text{:hiredBy}, a), T(a, \tau, \text{:PubAdmin})\\
\cup\ & q(x, \text{:ceoOf}) \leftarrow && T(x, \text{:ceoOf}, z), T(z, \tau, \text{:NatComp}),\\
& && T(x, \text{:ceoOf}, a), T(a, \tau, \text{:PubAdmin})\\
\cup\ & q(x, \text{:hiredBy}) \leftarrow && T(x, \text{:hiredBy}, z), T(z, \tau, \text{:NatComp}),\\
& && T(x, \text{:worksFor}, a), T(a, \tau, \text{:PubAdmin})\\
\cup\ & q(x, \text{:hiredBy}) \leftarrow && T(x, \text{:hiredBy}, z), T(z, \tau, \text{:NatComp}),\\
& && T(x, \text{:hiredBy}, a), T(a, \tau, \text{:PubAdmin})\\
\cup\ & q(x, \text{:hiredBy}) \leftarrow && T(x, \text{:hiredBy}, z), T(z, \tau, \text{:NatComp}),\\
& && T(x, \text{:ceoOf}, a), T(a, \tau, \text{:PubAdmin})
\end{aligned}$$

Figure 3: Sample reformulation for Example 4.5.

4.2 Rewriting Partially-Reformulated Queries using Saturated Mappings as Views: REW-C

In constrast with the REW-CA strategy that performs *all* the reasoning w.r.t. O and $\mathcal{R} = \mathcal{R}_c \cup \mathcal{R}_a$ at query time, our second strategy called REW-C splits the reasoning work between offline preprocessing and query time.

The first step of this strategy, labeled (1') in Figure 2, reformulates a query q using [12], but *solely* w.r.t. $O, \mathcal{R}_c$, producing a UBGPQ denoted Q_c. From the correctness of this reformulation step, and the fact that only $\mathcal{R}_a$ needs to be considered to answer Q_c with respect to the entire set of rules $\mathcal{R}$ (recall Section 2.4), the certain answer set of q asked on the RIS S is exactly the certain answer set of Q_c asked on S when disregarding $\mathcal{R}_c$. Formally:

Lemma 4.6. *Let $S = \langle O, \mathcal{R}, \mathcal{M}, \mathcal{E}\rangle$ be a RIS, q be a BGPQ and Q_c its reformulation w.r.t. $O, \mathcal{R}_c$ [12]. Then:*

$$\text{cert}(q, S) = \text{cert}(Q_c, \langle O, \mathcal{R}_a, \mathcal{M}, \mathcal{E}\rangle)$$

In other words, the desired answer set could be obtained by evaluating Q_c on the RIS data triples $G^{\mathcal{M}}_{\mathcal{E}}$ saturated by $\mathcal{R}_a$. Again, since the RIS data triples are not materialized, this does not provide a concrete solution. To account for the impact of the ontology O and the entailment rules $\mathcal{R}$ on these "virtual" data triples, we rely on *BGPQ saturation* [25]: given a BGPQ q, O and $\mathcal{R}$, the *saturation* $q^{\mathcal{R},O}$ *is q augmented with* all *the triples q implicitly asks for, given the ontology O and the rules $\mathcal{R}$*. BGPQ saturation is exemplified below:

Example 4.7 (BGPQ saturation). Consider the ontology O of G_{ex} and the query $q(x) \leftarrow (x, \text{:hiredBy}, y), (y, \tau, \text{:NatComp})$ asking who has been hired by a national company. Its saturation w.r.t. $\mathcal{R}_a, O$ is: $q^{\mathcal{R}_a,O}(x) \leftarrow \text{body}(q), (x, \text{:worksFor}, y), (x, \tau, \text{:Person}), (y, \tau, \text{:Comp}), (y, \tau, \text{:Org})$.

We use BGPQ saturation to saturate the RIS mapping heads w.r.t. $\mathcal{R}_a, O$, so that the *saturated* mappings together with $\mathcal{E}$ model the *saturated RIS data triples w.r.t.* $\mathcal{R}_a, O$. To compute $q^{\mathcal{R}_a,O}$ we (1) saturate body(q) $\cup\ O$ using $\mathcal{R}_a$, then (2) add to the body of q all triples thus inferred.

Definition 4.8 (Mappings saturation). The saturation of a set $\mathcal{M}$ of RIS mappings w.r.t. entailment rules $\mathcal{R}_a$ and ontology O is:

$$\mathcal{M}^{a,O} = \bigcup_{m \in \mathcal{M}} \{q_1(\bar{x}) \rightsquigarrow q_2^{\mathcal{R}_a,O}(\bar{x}) \mid m = q_1(\bar{x}) \rightsquigarrow q_2(\bar{x})\}$$

We saturate mappings offline, and just need to update them when O or the mapping heads change.

Example 4.9 (Saturated mappings). Consider the RIS of Example 3.6, the mapping heads in $\mathcal{M}^{a,O}$ are (added implicit triples are in blue):

$$\begin{aligned}
m_1: \quad & q_2^{\mathcal{R}_a,O}(x) \leftarrow && (x, \text{:ceoOf}, y), (y, \tau, \text{:NatComp}) \\
& && (x, \text{:worksFor}, y), (y, \tau, \text{:Comp}) \\
& && (x, \tau\text{:Person}), (y, \tau, \text{:Org}) \\
m_2: \quad & q_2^{\mathcal{R}_a,O}(x, y) \leftarrow && (x, \text{:hiredBy}, y), (y, \tau, \text{:PubAdmin}) \\
& && (x, \text{:worksFor}, y), (y, \tau, \text{:Org}) \\
& && (x, \tau, \text{:Person})
\end{aligned}$$

From the above Lemma and the use of saturated RIS mappings instead of the original ones, we show:

Lemma 4.10. *Let $S = \langle O, \mathcal{R}, \mathcal{M}, \mathcal{E}\rangle$ be a RIS, q be a BGPQ and $\mathcal{Q}_c$ its reformulation w.r.t. $O, \mathcal{R}_c$ [12]. Then:*

$$\text{cert}(q, S) = \text{cert}(\mathcal{Q}_c, \langle \emptyset, \emptyset, \mathcal{M}^{a,O}, \mathcal{E}\rangle)$$

This result allows solving the RIS query answering problem by relational view-based query rewriting (step (2') in Figure 2):

Theorem 4.11 (REW-C correctness). *Let $S = \langle O, \mathcal{R}, \mathcal{M}, \mathcal{E}\rangle$ be a RIS, q be a BGPQ and $\mathcal{Q}_c$ its reformulation w.r.t. $O, \mathcal{R}_c$.Then:*

$$\text{cert}(q, S) = \text{cert}(ubgpq2ucq(\mathcal{Q}_c), \text{Views}(\mathcal{M}^{a,O}), \mathcal{E})$$

Example 4.12 (REW-CA). Consider again the RIS in Example 3.6 and the query q of Example 4.5. Its reformulation $\mathcal{Q}_c$ w.r.t. $O, \mathcal{R}_c$, seen as a UCQ, is:

$$\begin{aligned}
& q(x, \text{:ceoOf}) \leftarrow && T(x, \text{:ceoOf}, z), T(z, \tau, \text{:NatComp}), \\
& && T(x, \text{:worksFor}, a), T(a, \tau, \text{:PubAdmin}) \\
\cup\ & q(x, \text{:hiredBy}) \leftarrow && T(x, \text{:hiredBy}, z), T(z, \tau, \text{:NatComp}), \\
& && T(x, \text{:worksFor}, a), T(a, \tau, \text{:PubAdmin})
\end{aligned}$$

This reformulation is therefore rewritten using the RIS views as: $q_r(x, \text{:ceoOf}) \leftarrow V_{m_1}(x), V_{m_2}(x, y)$. It is obtained from the first CQ in the above; the second one has no rewriting based on the available RIS views. We remark that this rewriting is equivalent to the one obtained in Example 4.5, hence yields the same answers.

4.3 Rewriting Queries using Saturated Mappings and Ontology Mappings as Views: REW

This strategy does not reason at query time at all. Instead, it rewrites a query q based on the saturated RIS mappings $\mathcal{M}^{a,O}$ as above, *and* on a specific set of ontology mappings we build to model the *saturated RIS ontology as a data source*:

Definition 4.13 (Ontology mappings). The set of *ontology mappings* for a RIS ontology O is:

$$\mathcal{M}_{O^c} = \bigcup_{x \in \{\prec_{sc}, \prec_{sp}, \hookleftarrow_d, \hookrightarrow_r\}} \{m_x \mid m_x = q_1(\text{s}, \text{o}) \rightsquigarrow q_2(\text{s}, \text{o})\}$$

with $q_2(\text{s}, \text{o}) \leftarrow (\text{s}, x, \text{o})$. The *extension* of an ontology mapping m_x is $\text{ext}(m_x) = \{V_{m_x}(\text{s}, \text{o}) \mid (\text{s}, x, \text{o}) \in O^{\mathcal{R}_c}\}$. The *extent* of $\mathcal{M}_{O^c}$ is denoted $\mathcal{E}_{O^c}$.

We compute ontology mappings offline, and only need to update them when the ontology changes. The ontology mapping extensions $\mathcal{E}_{O^c}$ store all the explicit and implicit RIS ontology triples (recall from Section 2.2 that only $\mathcal{R}_c$ lead to such triples). Importantly, this leads to the observation that a query triple that refers to the ontology (schema) can be evaluated on the ontology mapping extensions alone. Formally:

$$\begin{aligned}
& q(x, \text{:ceoOf}) \leftarrow && V_{m_1}(x), V_{m_{\prec_{sp}}}(\text{:ceoOf}, \text{:worksFor}), \\
& && V_{m_{\prec_{sc}}}(\text{:NatComp}, \text{:Comp}), V_{m_2}(x, a) \\
\cup\ & q(x, \text{:ceoOf}) \leftarrow && V_{m_1}(x), V_{m_{\prec_{sp}}}(\text{:ceoOf}, \text{:worksFor}), \\
& && V_{m_{\prec_{sc}}}(\text{:Comp}, \text{:Comp}), V_{m_2}(x, a) \\
\cup\ & q(x, \text{:ceoOf}) \leftarrow && V_{m_1}(x), V_{m_{\prec_{sp}}}(\text{:ceoOf}, \text{:worksFor}), \\
& && V_{m_{\prec_{sc}}}(\text{:Org}, \text{:Comp}), V_{m_2}(x, a) \\
\cup\ & q(x, \text{:worksFor}) \leftarrow && V_{m_1}(x), V_{m_{\prec_{sp}}}(\text{:worksFor}, \text{:worksFor}), \\
& && V_{m_{\prec_{sc}}}(\text{:NatComp}, \text{:Comp}), V_{m_2}(x, a) \\
\cup\ & q(x, \text{:worksFor}) \leftarrow && V_{m_1}(x), V_{m_{\prec_{sp}}}(\text{:worksFor}, \text{:worksFor}), \\
& && V_{m_{\prec_{sc}}}(\text{:Comp}, \text{:Comp}), V_{m_2}(x, a) \\
\cup\ & q(x, \text{:worksFor}) \leftarrow && V_{m_1}(x), V_{m_{\prec_{sp}}}(\text{:worksFor}, \text{:worksFor}), \\
& && V_{m_{\prec_{sc}}}(\text{:Org}, \text{:Comp}), V_{m_2}(x, a) \\
\cup\ & q(x, \text{:hiredBy}) \leftarrow && V_{m_2}(x, z), V_{m_{\prec_{sp}}}(\text{:hiredBy}, \text{:worksFor}), \\
& && V_{m_{\prec_{sc}}}(\text{:PubAdmin}, \text{:Comp}), V_{m_2}(x, a) \\
\cup\ & q(x, \text{:hiredBy}) \leftarrow && V_{m_2}(x, z), V_{m_{\prec_{sp}}}(\text{:hiredBy}, \text{:worksFor}), \\
& && V_{m_{\prec_{sc}}}(\text{:Org}, \text{:Comp}), V_{m_2}(x, a) \\
\cup\ & q(x, \text{:worksFor}) \leftarrow && V_{m_2}(x, z), V_{m_{\prec_{sp}}}(\text{:worksFor}, \text{:worksFor}), \\
& && V_{m_{\prec_{sc}}}(\text{:PubAdmin}, \text{:Comp}), V_{m_2}(x, a) \\
\cup\ & q(x, \text{:worksFor}) \leftarrow && V_{m_2}(x, z), V_{m_{\prec_{sp}}}(\text{:worksFor}, \text{:worksFor}), \\
& && V_{m_{\prec_{sc}}}(\text{:Org}, \text{:Comp}), V_{m_2}(x, a) \\
\cup\ & \textstyle\bigcup_{\text{r} \in \{\prec_{sc}, \prec_{sp}, \hookleftarrow_d, \hookrightarrow_r\}} q(x, \text{r}) \leftarrow && V_{m_\text{r}}(x, z), V_{m_2}(v, z), \\
& && V_{m_{\prec_{sp}}}(\text{r}, \text{:worksFor}), \\
& && V_{m_{\prec_{sc}}}(\text{:PubAdmin}, \text{:Comp}), \\
& && V_{m_2}(x, a) \\
& \qquad \cup\ q(x, \text{r}) \leftarrow && V_{m_\text{r}}(x, z), V_{m_2}(v, z), \\
& && V_{m_{\prec_{sp}}}(\text{r}, \text{:worksFor}), \\
& && V_{m_{\prec_{sc}}}(\text{:Org}, \text{:Comp}), \\
& && V_{m_2}(x, a)
\end{aligned}$$

Figure 4: Sample rewriting for Example 4.17.

Lemma 4.14. *Let $S = \langle O, \mathcal{R}, \mathcal{M}, \mathcal{E}\rangle$ be a RIS and q be a BGPQ. Then:*

$$\text{cert}(q, S) = \text{cert}(q, \langle O, \mathcal{R}_a, \mathcal{M}_{O^c} \cup \mathcal{M}, \mathcal{E}_{O^c} \cup \mathcal{E}\rangle)$$

This lemma effectively "pushes" $\mathcal{R}_c$ reasoning in the set of mappings (to which we add $\mathcal{M}_{O^c}$) and the extent (to which we add $\mathcal{E}_{O^c}$). Next, we rely (as we did for REW-CA) on mappings saturation with $O, \mathcal{R}_a$ to also push $\mathcal{R}_a$ reasoning in the mappings, leading to:

Lemma 4.15. *Let $S = \langle O, \mathcal{R}, \mathcal{M}, \mathcal{E}\rangle$ be a RIS and q be a BGPQ. Then:*

$$\text{cert}(q, S) = \text{cert}(q, \langle \emptyset, \emptyset, \mathcal{M}_{O^c} \cup \mathcal{M}^{a,O}, \mathcal{E}_{O^c} \cup \mathcal{E}\rangle)$$

This allows to reduce RIS query answering to relational view-based query rewriting (step (2") in Figure 2):

Theorem 4.16 (REW correctness). *Let $S = \langle O, \mathcal{R}, \mathcal{M}, \mathcal{E}\rangle$ be a RIS and q be a BGPQ. Then:*

$$\text{cert}(q, S) = \text{cert}(bgpq2cq(q), \text{Views}(\mathcal{M}_{O^c} \cup \mathcal{M}^{a,O}), \mathcal{E}_{O^c} \cup \mathcal{E})$$

Example 4.17 (REW). Consider again the RIS in Example 3.6 and the query q of Example 4.5 seen as a CQ:

$$\begin{aligned}
q(x, y) \leftarrow\ & T(x, y, z), T(z, \tau, t), T(y, \prec_{sp}, \text{:worksFor}), \\
& T(t, \prec_{sc}, \text{:Comp}), T(x, \text{:worksFor}, a), \\
& T(a, \tau, \text{:PubAdmin})
\end{aligned}$$

Its maximally-contained rewriting q_{REW} based on the views obtained from the RIS *saturated* mappings and *ontology* mappings appears in Figure 4. This rewriting is much larger than the ones of the two preceding techniques: this is due to the ontology mappings. If we assume that $\mathcal{E}$ also contains $V_{m_2}(\text{:p}_1, \text{:a})$, as we did in

Example 4.5, we obtain again $\text{cert}(q, S) = \{\langle :p_1, :\text{ceoOf}\rangle\}$, which results from the evaluation of the first CQ in the UCQ rewriting; the other CQs yield empty results because some required $\prec_{sc}$ or $\prec_{sp}$ contraints are not found in the views built from the RIS ontology mappings.

How do our strategies compare? Since they are all correct, they lead to the same RIS certain answer set, however they do not necessarily compute the same view-based rewritings. Indeed, REW considers the additional set $\mathcal{M}_{O^c}$ of ontology mappings. Hence, for queries over the ontology, i.e., featuring in a property position $\prec_{sc}$, $\prec_{sp}$, $\hookleftarrow_d$, $\hookrightarrow_r$, or a variable, a REW rewriting is larger than a REW-CA or REW-C rewriting and, to be answered, requires the additional ontology source. In contrast, REW-CA and REW-C yield logically equivalent rewritings; we minimize them both to avoid possible redundancies, thus they become identical (up to variable renaming). Hence, REW-CA and REW-C do not differ in how these rewritings are evaluated. Instead, they differ in how the rewritings are *computed*, or, equivalently, on *the distribution of the reasoning effort on the data and mappings, across various query answering stages.* As our experiments show, given the computational complexity of view-based query rewriting [42], this difference has a significant impact on their performance.

5 EXPERIMENTAL EVALUATION

We now describe our experiments with RIS query answering. In addition to our strategies based on query rewriting, we include in our comparison a simple **alternative strategy, based on materialization and denoted MAT**. Offline (before answering queries), this strategy materializes the RIS data triples and saturates them with the rule set $\mathcal{R}$. The materialization is stored and saturated in an **RDF data management system (RDFDB, in short)**. Then, MAT query answering amounts to query evaluation on the saturated materialization. Therefore, MAT query answering can be seen as a lower bound for query answering through other strategies.

5.1 Experimental settings

Software Our platform is developed in Java 1.8, as follows. Our RDFDB is OntoSQL[2], a Java platform providing efficient RDF storage, saturation, and query evaluation on top of an RDBMS [14, 29], relying on Postgres v9.6. To save space, OntoSQL encodes IRIs and literals into integers, and a dictionary table which allows going from one to the other. It stores all resources of a certain type in a one-attribute table, and all (subject, object) pairs for each property (including RDFS schema properties) in a table; the tables are indexed. OntoSQL is used in the MAT strategy, and it also provides the RDF query reformulation algorithm [12].
We rely on the Graal engine [9] for **view-based query rewriting**. Graal is a Java toolkit dedicated to query answering algorithms in knowledge bases with existential rules (a.k.a. tuple-generating dependencies). Since the relational view $V_m(\bar{x}) \leftarrow bgp2ca(\text{body}(q_2))$ corresponding to a GLAV mapping m (recall Def. 4.2) can be seen as a specific existential rule of the form $V_m(\bar{x}) \rightarrow bgp2ca(\text{body}(q_2))$, the query reformulation algorithm of Graal can be used to rewrite the UCQ translation of a BGPQ with respect to a set of RIS mappings. To **execute queries against heterogeneous data sources**, we use Tatooine [4, 10], a Java-based mediator (or polystore) system, capable both of pushing queries in underlying data sources and (unlike other polystores, e.g., [24]) of evaluating joins within the mediator engine. Query rewritings produced by Graal are unfolded into queries on the data sources (using the q_1 parts of the mappings, see Section 2.5.2) and passed to Tatooine. We implemented the RIS query answering methods described here in Java 1.8 on top of these tools.
Hardware We used servers with 2,7 GHz Intel Core i7 processors and 160 GB of RAM, running CentOs Linux 7.5.

[2]https://ontosql.inria.fr

5.2 Experimental scenarios

RDF Integration Systems (RIS) used Our first interest was to study scalability of RIS query answering, in particular in the *relational* setting studied in many prior works. To achieve this, we used the BSBM benchmark relational data generator[3] to build databases consisting of 10 relations named producer, product, offer, review etc. Using two different benchmark scale factors, we obtained a data source DS_1 of 154.054 tuples across the relations, respectively, DS_2 of 7.843.660 tuples; both are stored in Postgres. We used two RDFS ontologies O_1 respectively O_2, containing, first, subclass hierarchies of 151 (resp. 2011) product types, which come with DS_1, respectively, DS_2. To O_1 and O_2, we add a natural RDFS ontology for BSBM composed of 26 classes and 36 properties, used in 40 subclass, 32 subproperty, 42 domain and 16 range statements.
Relational-sources RIS We devised two sets $\mathcal{M}_1, \mathcal{M}_2$ of 307, respectively, 3863 mappings, which expose the relational data from DS_1, respectively, DS_2 as RDF graphs. The relatively high number of mappings is because: (*i*) each product type (of which there are many, and their number scales up with the BSBM data size) appears in the head of a mapping, enabling fine-grained and high-coverage exposure of the data in the integration graph; (*ii*) we also generated more complex GLAV mappings, partially exposing the results of join queries over the BSBM data; interestingly, these mappings expose incomplete knowledge, in the style of Example 3.4.
The mapping sets lead to the **RIS graphs of** $2.0 \cdot 10^6$, respectively, $108 \cdot 10^6$ triples. Their **saturated** versions comprise respectively $3.4 \cdot 10^6$ and $185 \cdot 10^6$ triples. Our first two RIS are thus: $S_1 = \langle O_1, \mathcal{R}, \mathcal{M}_1, \mathcal{E}_1\rangle$ and $S_2 = \langle O_2, \mathcal{R}, \mathcal{M}_2, \mathcal{E}_2\rangle$, where $\mathcal{E}_i$ for i in $\{1, 2\}$ are the extents resulting from DS_i and $\mathcal{M}_i$.
Heterogeneous-sources RIS Second, going beyond relational-sources OBDA [16, 17, 44], our architecture extends to *heterogeneous data sources*. For that, we converted a third (33%) of DS_1, DS_2 into JSON documents, and stored them into MongoDB, leading to the JSON data sources denoted $DS_{j,1}, DS_{j,2}$; the relational sources $DS_{r,1}, DS_{r_2}$ store the remaining (relational) data. Conceptually, for i in $\{1, 2\}$, the extension based on $DS_{r,i}$ and extension based on $DS_{j,i}$ form a partition of $\mathcal{E}_i$. We devise a set of **JSON-to-RDF mappings** to expose $DS_{j,1}$ and $DS_{j,2}$ into RDF, and denote $\mathcal{M}_3$ the set of mappings exposing $DS_{r,1}$ and $DS_{j,1}$, together, as an RDF graph; similarly, the mappings $\mathcal{M}_4$ expose $DS_{r,2}$ and $DS_{j,2}$ as RDF. Our last two RIS are thus: $S_3 = \langle O_1, \mathcal{R}, \mathcal{M}_3, \mathcal{E}_3\rangle$ and $S_4 = \langle O_2, \mathcal{R}, \mathcal{M}_4, \mathcal{E}_4\rangle$, where $\mathcal{E}_3$ is the extent of $\mathcal{M}_3$ based on $DS_{r,1}$ and $DS_{j,1}$, while $\mathcal{E}_4$ is the extent of $\mathcal{M}_4$ based on $DS_{r,2}$ and $DS_{j,2}$. *The RIS data and ontology triples of S_1 and S_3 are identical; thus, the difference between these two RIS is only due to the heterogeneity of their underlying data sources.* The same holds for S_2 and S_4.
Queries We devised a set of 28 BGP queries having from 1 to 11 triple patterns (5.5 on average), of varied selectivity (they return between 2 and $330 \cdot 10^3$ results in S_1 and S_3 and between 2 and $4.4 \cdot 10^6$ results in S_2 and S_4); 6 among them query the data *and* the ontology (recall Example 2.6), a capability which most

[3]https://downloads.sourceforge.net/project/bsbmtools/bsbmtools/bsbmtools-0.2

RIS		Q01	Q01a	Q01b	Q02	Q02a	Q02b	Q02c	Q03	Q04	Q07	Q07a	Q09	Q10	Q13
all	N_{TRI}	5	5	5	6	6	6	6	5	2	3	3	1	3	4
S_1, S_3	$\lvert\mathcal{Q}_{c,a}\rvert$	7	21	175	21	49	147	1225	525	1	5	19	7	670	28
S_1, S_3	N_{ANS}	1272	4376	22738	16	56	174	1342	19	91	2	3	5617	9	13190
S_2, S_4	$\lvert\mathcal{Q}_{c,a}\rvert$	21	175	1407	63	147	525	1225	4375	1	5	19	7	9350	84
S_2, S_4	N_{ANS}	15514	111793	863729	124	598	1058	1570	5	4487	2	3	299902	10	167760

RIS		Q13a	Q13b	Q14	Q16	Q19	Q19a	Q20	Q20a	Q20b	Q20c	Q21	Q22	Q22a	Q23
all	N_{TRI}	4	4	3	4	9	9	11	11	11	11	3	4	4	7
S_1, S_3	$\lvert\mathcal{Q}_{c,a}\rvert$	84	700	1	25	63	147	21	63	525	1225	670	2	40	192
S_1, S_3	N_{ANS}	43157	330142	56200	8114	2015	3515	0	236	2312	7564	1085	28	434	25803
S_2, S_4	$\lvert\mathcal{Q}_{c,a}\rvert$	5628	5628	1	201	525	1225	63	525	1225	4221	9350	40	520	192
S_2, S_4	N_{ANS}	4416946	10049829	2998948	249004	39826	60834	904	7818	10486	51988	37176	1528	18588	1329887

Table 4: Characteristics of the queries used in our experiments.

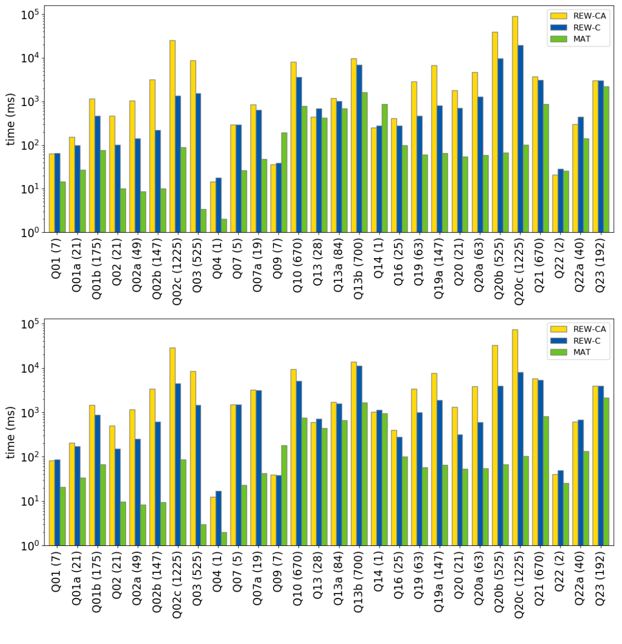


Figure 5: Query answering times on the smaller RIS S_1 (top, relational sources) and S_3 (bottom, heterogeneous sources).

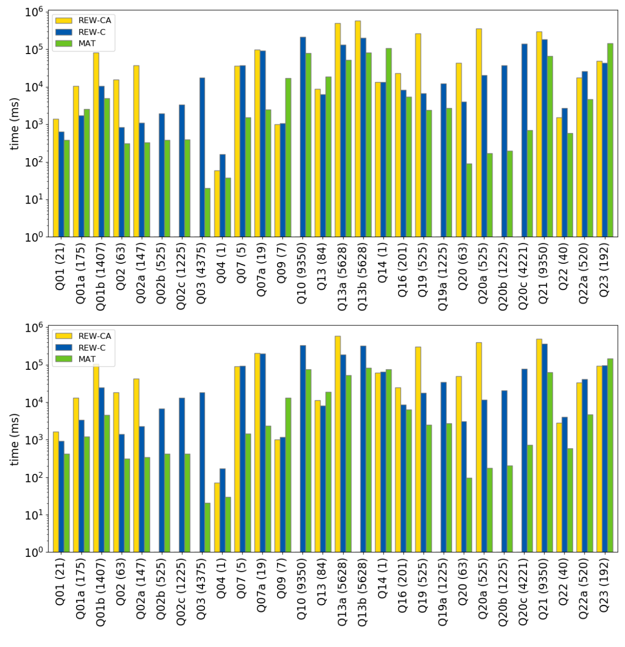


Figure 6: Query answering times on the larger RIS S_2 (top, relational sources) and S_4 (bottom, heterogeneous sources).

competitor systems lack (see Section 6). Table 4 reports three query properties impacting query answering performance: the number of induced triples (N_{TRI}), the number of BGPQ reformulations on the ontology ($|\mathcal{Q}_{c,a}|$, ranging from 1 to 1225; this strongly determines the performance of answering such large union queries, recall Example 4.5), and its number of answers (N_{ANS}) on the two RIS groups (S_1, S_3 and S_2, S_4). To further study the impact of the ontology on query evaluation complexity, we created *query families* denoted Q_X, Q_{Xa} etc. by replacing the classes and properties appearing in Q_X with their super classes or super properties in the ontology. In such a family, Q_X *is the most selective, and queries are sorted in the increasing order of their number of reformulations.*

Our ontologies, mappings, queries, and experimental details are available online[4].

5.3 Query answering performance

REW inefficiency We have conducted experiments[4] using our six queries on ontological triples showing, as in Example 4.17 and Figure 4, an explosion of the size of the rewriting (number of CQs), compared to the rewriting produced by the two other approaches. *On queries (also) over the ontology,* as explained in Section 4.3, we noted that the size of the rewriting produced by REW is larger (by a multiplicative factor of 29 to 74 in S_1 and S_3, and of 33 to 969 in S_2 and S_4) than the rewritings of the two other strategies, which led to an explosion of the time spent minimizing the rewriting, and made REW overall unfeasible; the details of these tests can be found online[4]. *On queries that do not carry over the ontology,* REW produces the same rewritings as the other methods. Thus, we do not report further REW performance below.

Query answering time comparison Figure 5 depicts the query answering times, on the smaller RIS, of REW-CA, REW-C and MAT. *The size of (number of BGPQs in) the reformulation of each query w.r.t. $\mathcal{R}$, $|\mathcal{Q}_{c,a}|$ appears in parentheses after the query name, in the labels along the x axis.* Given that S_1, S_3 have the same RIS data triples, the MAT strategy coincides among these two RIS. Figure 6 shows the corresponding times for the largest RIS S_2 and S_4; the same observations apply. Note the logarithmic time axes.

A first observation is that *our query set is quite diverse*; their answering times range from a few to more than 10^5 ms.

As expected, query answering in MAT *is the fastest in most cases,* since it has no reasoning work to do at query answering time. However, it required, for S_1, S_3, $1.2 \cdot 10^5$ ms to build the materialization and $1.49 \cdot 10^5$ ms more to saturate it, whereas for S_2, S_4, these times are 14h46 ($5.31 \cdot 10^7$ ms), respectively, 1h28 ($5.28 \cdot 10^6$

[4]Experiment web site: https://gitlab.inria.fr/mburon/org/blob/master/projects/het2onto-benchmark/bsbm/

ms). Not only these are *orders of magnitude more than all query answering times*; recall also that materializing $G_{\mathcal{E}}^{\mathcal{M}}$ requires maintaining it when the underlying data changes, and its saturation $(G_{\mathcal{E}}^{\mathcal{M}} \cup O)^{\mathcal{R}}$ needs a second level of maintenance. Thus, MAT is not practical when data sources change. We were surprised to see REW-C and REW-CA somehow faster than MAT for queries Q_{09} and Q_{14}. Answering these queries through MAT within OntoSQL leads to producing many results that involve *mapping-generated blank nodes*, tuples which should not appear in our certain answers, as per Definition 3.5. We remove such tuples in post-processing mode, which leads to a performance overhead for MAT. REW-C and REW-CA, in contrast, are answered by evaluating rewritings, and do not have to apply such a result pruning. It remains to be seen if this pruning could be pushed in an RDFDB; note that not *all* answers including blank nodes should be pruned, only those whose blank nodes are due to mappings.

In each scenario, we observe that REW-C is faster or takes as long as REW-CA. Since the two approaches produce the same rewritings, the difference is due to steps before the step (3) in Figure 2. It turns out it is due to the rewriting time, which in turn strongly depends on the size of the reformulation it receives as input. In REW-C, the reformulations w.r.t. $\mathcal{R}_c$ are of size 1 (no union, just one BGP) for queries on data triples only, and never exceed 64 in S_1 and S_3 and 200 in S_2 and S_4, whereas, in REW-CA the reformulation sizes are much larger. REW-C is most often faster than REW-CA, by up to two orders of magnitude e.g., for Q_{02a}, Q_{19} and Q_{20a} on S_2, the latter two on S_4 etc. One order of magnitude speed-up is noticeable even on the smaller RIS S_1, S_3 (Figure 5) for Q_{02a}. As a consequence, REW-C completes successfully in all scenarios we study, whereas REW-CA fails to complete for many queries with timeout set to 10min (missing yellow bars in Figure 6), in close correlation with the increased number of reformulations.

Scaling in the data size As stated in Section 5.2, there is a *scale factor of about 50* between S_1, S_3 on one hand, and S_2, S_4 on the other. Figures 5 and 6 show that the query answering times generally grow by less than 50, when moving from S_1 to S_2, and from S_3 to S_4. This is mostly due to the good scalability of PostgreSQL (in the all-relational RIS), Tatooine (itself building on PostgreSQL and MongoDB, in the heterogeneous RIS), and OntoSQL (for MAT). As discussed above, computation steps we implemented outside these systems are strongly impacted by the *mappings, ontology and query*; intelligently distributing the reasoning effort, as REW-C does, avoids the heavy performance penalties that from which REW-CA and REW sometimes suffer.

Impact of heterogeneity REW-CA and REW-C incur a (modest) overhead when combining data from PostgreSQL and MongoDB (heterogeneous RIS) w.r.t. the relational-sources RIS. Part of this is due to the cost of marshalling data across system boundaries; the rest is due to imperfect optimization within Tatooine. Overall, the comparison demonstrates that RIS query answering is feasible and quite efficient even on heterogeneous data sources.

5.4 Experiment conclusion

In a setting where the data, ontology and mappings do not change, MAT is an efficient and robust query answering technique, at a rather high cost to materialize and saturate the RIS instance. In contrast, in a dynamic setting, REW-C *smartly combines partial reformulation and view-based query rewriting to efficiently compute query answers.* The changes it requires when the ontology and mappings change (basically re-saturating mapping heads) are light and likely to be very fast. Thus, we conclude that REW-C is the best query answering strategy for dynamic RIS.

6 RELATED WORK AND CONCLUSION

Ontologies have been used to integrate relational or heterogeneous data sources in mediators [49] with LAV views based on description logics [1, 37] or their combination with Datalog [28, 30]. Semantics have been used at the integration level since e.g., [20] for SGML and soon after for RDF [6, 7]; data is considered represented and stored in a flexible object-oriented model, thus no mappings are used.

Our work follows the OBDA paradigm introduced in [41]. This paradigm was conceived to enhance access to relational data by mappings to an ontology expressed in a dialect of the DL-Lite description logic family (typically DL-Lite$_{\mathcal{R}}$ underpinning the OWL 2 QL profile of the W3C ontological language OWL 2). Mature DL-based systems include Mastro[5] [17] and Ontop[6] [16, 43]. Another notable OBDA system, namely Ultrawrap^{OBDA} [44], is based on an extension of RDFS to inverse and transitive properties. All these systems rely on GAV mappings.

Compared to these, our main novelty is to handle GLAV mappings and provide query answering algorithms for the resulting novel RIS setting. Note that formal OBDA frameworks with GLAV mappings have long been defined, e.g., in [18], but not put into practice. Regarding the other components of OBDA, we consider a simpler ontological language than existing OBDA systems, but support BGPQs on both data and ontological triples, a feature hardly found in these systems (an exception is [33]).

As explained in the introduction, GLAV mappings maximize the expressive power of the integration system. In particular, they allow to expose a form of *incomplete* information (recall Example 3.6). To some extent, GLAV mappings may be simulated by GAV mappings provided with so-called Skolem functions on answer variables, as suggested for instance in [21]. To illustrate, consider the GLAV mapping $m_1 = q_1(\bar{x}) \rightsquigarrow q_2(\bar{x})$ with head $q_2(x) \leftarrow (x, \text{:ceoOf}, y), (y, \tau, \text{:NatComp})$ from Example 3.2. The non-answer variable y could be replaced by a Skolem function $f(x)$, which would yield two GAV mappings, namely $m_{1_1} = q_1(\bar{x}) \rightsquigarrow q_{2_1}(\bar{x})$ and $m_{1_2} = q_1(\bar{x}) \rightsquigarrow q_{2_2}(\bar{x})$, with respective head $q_{2_1}(\bar{x}) \leftarrow (x, \text{:ceoOf}, f(x))$ and $q_{2_2}(\bar{x}) \leftarrow (f(x), \tau, \text{:NatComp})$. Note that Skolem functions would have to produce syntactically correct RDF values in a materialization scenario. Still in a materialisation scenario, query answering would require some post-processing to prevent the values built by the Skolem functions to be accepted as answers, while in a query rewriting sceSkolemnario functional values would also have to be dealt with in a special way, which in particular prevents to use off-the-shelf view-based query rewriting algorithms. Hence, value invention would be simulated here at the price of technically more complex mappings and processing. Second, the break-up of GLAV mappings into several GAV mappings would lead to higher conceptual complexity since intrinsically connected triples, as those associated with $(x, \text{:ceoOf}, y)$ and $(y, \tau, \text{:NatComp})$ in the example, could not be exposed together by a single mapping. Last but not least, query rewriting would be considerably slowed down and would produce highly redundant rewritings, as demonstrated in the seminal paper [42].

Our mapping saturation (Definition 4.8) is inspired by a query saturation technique introduced in [25] to compute least general generalizations of BGPQs under RDFS background knowledge.

[5] http://obdasystems.com/mastro/
[6] https://ontop.inf.unibz.it/

It can be seen as a generalization to GLAV mappings of the $\mathcal{T}$-mapping technique introduced in [43] (and further developed in [44]) to optimize query rewriting in a classical OBDA context. The $\mathcal{T}$-mapping technique consists of completing the original set of GAV mappings with new ones, encapsulating information inferred from the DL ontology. For instance, given a GAV mapping $m = q_1(x) \rightsquigarrow q_2(x) \leftarrow C(x)$ with C a class, and a DL constraint specifying that C is a subclass of D, a new mapping $m' = q_1(x) \rightsquigarrow q_2'(x) \leftarrow D(x)$ is created by composing m and the DL constraint. On this example, we would saturate the head of m into $q_2(x) \leftarrow C(x) \wedge D(x)$, which is semantically equivalent to adding the mapping m'. However, when mappings are GLAV and not GAV, one cannot simply add new mappings. For instance, consider the GLAV mapping $m_1 = q_1(\bar{x}) \rightsquigarrow q_2(\bar{x})$ with head $q_2(x) \leftarrow (x, \text{:ceoOf}, y), (y, \tau, \text{:NatComp})$; given the entailment rule `rdfs9` and the ontological triple $(\text{:NatComp}, \prec_{sc}, \text{:Comp})$, the saturation adds the triple $(y, \tau, \text{:Comp})$ to the body of q_2; creating instead a new mapping of the form $m_1' = q_1(\bar{x}) \rightsquigarrow q_2'(\bar{x})$ with head $q_2'(x) \leftarrow (y, \tau, \text{:Comp})$ would be unsatisfactory as y in m_1' should correspond to the same object as y in m_1.
Our mapping saturation technique could be extended to more general entailment rules, in which the head of the rules may include blank nodes that are not in their body, possibly shared by several triples. This is part of our future research agenda.

Acknowledgements: This work is supported by the Inria Project Lab iCoda and the ANR project CQFD (ANR-18-CE23-0003).

REFERENCES

[1] Nada Abdallah, François Goasdoué, and Marie-Christine Rousset. 2009. DL-LITER in the Light of Propositional Logic for Decentralized Data Management. In *IJCAI*.

[2] Serge Abiteboul and Oliver M. Duschka. 1998. Complexity of Answering Queries Using Materialized Views. ACM Press.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

[4] Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis. 2019. Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue. In *SIGMOD*. https://hal.inria.fr/hal-02070827

[5] Bernd Amann, Catriel Beeri, Irini Fundulaki, and Michel Scholl. 2002. Querying XML Sources Using an Ontology-Based Mediator. In *CoopIS*. Springer Berlin Heidelberg.

[6] Bernd Amann and Irini Fundulaki. 1999. Integrating Ontologies and Thesauri to Build RDF Schemas. In *ECDL*.

[7] Bernd Amann, Irini Fundulaki, and Michel Scholl. 2000. Integrating ontologies and thesauri for RDF schema creation and metadata querying. *Int. J. on Digital Libraries* 3, 3 (2000).

[8] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider (Eds.). 2003. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press.

[9] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, Swan Rocher, and Clément Sipieter. 2015. Graal: A Toolkit for Query Answering with Existential Rules. In *RuleML*.

[10] Raphaël Bonaque, Tien Duc Cao, Bogdan Cautis, François Goasdoué, Javier Letelier, Ioana Manolescu, Oscar Mendoza, Swen Ribeiro, Xavier Tannier, and Michaël Thomazo. 2016. Mixed-instance querying: a lightweight integration architecture for data journalism. In *VLDB*. https://hal.inria.fr/hal-01321201

[11] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Julien Corman, and Guohui Xiao. 2018. A Generalized Framework for Ontology-Based Data Access. In *AI*IA*.

[12] Maxime Buron, François Goasdoué, Ioana Manolescu, and Marie-Laure Mugnier. 2019. Reformulation-Based Query Answering for RDF Graphs with RDFS Ontologies. In *ESWC*.

[13] Maxime Buron, François Goasdoué, Ioana Manolescu, and Marie-Laure Mugnier. 2018. Rewriting-Based Query Answering for Semantic Data Integration Systems. In *BDA (informal publication only)*. https://hal.archives-ouvertes.fr/hal-01927282

[14] Damian Bursztyn, François Goasdoué, and Ioana Manolescu. 2015. Optimizing Reformulation-based Query Answering in RDF. In *EDBT*.

[15] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. 2009. A general Datalog-based framework for tractable query answering over ontologies. In *PODS*.

[16] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. 2017. Ontop: Answering SPARQL queries over relational databases. *Semantic Web* 8, 3 (2017).

[17] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. 2011. The MASTRO system for ontology-based data access. *Semantic Web* 2, 1 (2011).

[18] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Riccardo Rosati, and Marco Ruzzi. 2009. Using OWL in Data Integration. In *Semantic Web Information Management – A Model-Based Perspective.* Springer.

[19] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2012. Query Processing under GLAV Mappings for Relational and Graph Databases. *PVLDB* 6, 2 (2012).

[20] Vassilis Christophides, Martin Doerr, and Irini Fundulaki. 1997. A Semantic Network Approach to Semi-Structured Documents Repositories. In *ECDL*.

[21] Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. 2018. Using Ontologies for Semantic Data Integration. In *A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years.* Vol. 31. Springer, Cham.

[22] A. Deutsch and V. Tannen. 2003. MARS: A System for Publishing XML from Mixed and Redundant Storage.. In *VLDB*.

[23] AnHai Doan, Alon Halevy, and Zachary G. Ives. 2012. *Principles of Data Integration.* Morgan Kaufmann, Waltham, MA.

[24] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik. 2015. The BigDAWG Polystore System. *SIGMOD* 44, 2 (2015).

[25] Sara El Hassad, François Goasdoué, and Hélène Jaudoin. 2017. Learning Commonalities in SPARQL. In *ISWC*.

[26] Marc Friedman, Alon Y. Levy, and Todd D. Millstein. 1999. Navigational Plans for Data Integration. In *I3 workshop@IJCAI*.

[27] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, Vasilis Vassalos, and Jennifer Widom. 1997. The TSIMMIS Approach to Mediation: Data Models and Languages. *J. Intell. Inf. Syst.* 8, 2 (1997).

[28] François Goasdoué, Véronique Lattès, and Marie-Christine Rousset. 2000. The Use of CARIN Language and Algorithms for Information Integration: The PICSEL System. *Int. J. Cooperative Inf. Syst.* 9, 4 (2000).

[29] François Goasdoué, Ioana Manolescu, and Alexandra Roatis. 2013. Efficient query answering against dynamic RDF databases. In *EDBT*.

[30] François Goasdoué and Marie-Christine Rousset. 2004. Answering queries using views: A KRDB perspective for the semantic Web. *ACM TOIT* 4, 3 (2004).

[31] Alon Y. Halevy. 2001. Answering Queries Using Views: A Survey. *The VLDB Journal* 10, 4 (Dec. 2001).

[32] Dag Hovland, Roman Kontchakov, Martin G. Skjæveland, Arild Waaler, and Michael Zakharyaschev. 2017. Ontology-Based Data Access to Slegge. In *ISWC*.

[33] Roman Kontchakov, Martin Rezk, Mariano Rodríguez-Muro, Guohui Xiao, and Michael Zakharyaschev. 2014. Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime. In *ISWC*.

[34] Davide Lanti, Guohui Xiao, and Diego Calvanese. 2017. Cost-Driven Ontology-Based Data Access. In *ISWC*.

[35] Maurizio Lenzerini. 2002. Data Integration: A Theoretical Perspective. In *PODS*.

[36] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. 1996. Querying Heterogeneous Information Sources Using Source Descriptions. In *VLDB*.

[37] Alon Y. Levy, Divesh Srivastava, and Thomas Kirk. 1995. Data Model and Query Evaluation in Global Information Systems. *J. Intell. Inf. Syst.* 5, 2 (1995).

[38] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. 2001. Answering XML Queries on Heterogeneous Data Sources. In *VLDB*.

[39] Marie-Laure Mugnier. 2011. Ontological Query Answering with Existential Rules. In *RR*.

[40] Floriana Di Pinto, Domenico Lembo, Maurizio Lenzerini, Riccardo Mancini, Antonella Poggi, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. 2013. Optimizing query rewriting in ontology-based data access. In *EDBT*.

[41] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. 2008. Linking Data to Ontologies. *J. Data Semantics* 10 (2008).

[42] Rachel Pottinger and Alon Y. Halevy. 2001. MiniCon: A scalable algorithm for answering queries using views. *VLDB J.* 10 (2001).

[43] Mariano Rodriguez-Muro, Roman Kontchakov, and Michael Zakharyaschev. 2013. Ontology-Based Data Access: Ontop of Databases. In *ISWC*.

[44] Juan F. Sequeda, Marcelo Arenas, and Daniel P. Miranker. 2014. OBDA: Query Rewriting or Materialization? In Practice, Both!. In *ISWC*.

[45] Grégory Smits, Olivier Pivert, Hélène Jaudoin, and François Paulus. 2014. AGGREGO SEARCH: Interactive Keyword Query Construction. In *EDBT*.

[46] W3C. 2013. SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/

[47] W3C. 2014. RDF 1.1 Concepts and Abstract Syntax. https://www.w3.org/TR/rdf11-concepts/

[48] W3C. 2014. RDF 1.1 Semantics. https://www.w3.org/TR/rdf11-mt/#rdfs-entailment

[49] Gio Wiederhold. 1992. Mediators in the Architecture of Future Information Systems. *IEEE Computer* 25, 3 (1992).

Manually Detecting Errors for Data Cleaning Using Adaptive Crowdsourcing Strategies

Haojun Zhang
University of Wisconsin-Madison
hzhang0418@cs.wisc.edu

Chengliang Chai
Tsinghua University
chaicl15@mails.tsinghua.edu.cn

AnHai Doan
University of Wisconsin-Madison
anhai@cs.wisc.edu

Paraschos Koutris
University of Wisconsin-Madison
paris@cs.wisc.edu

Esteban Arcaute
Facebook AI
esteban@fb.com

ABSTRACT

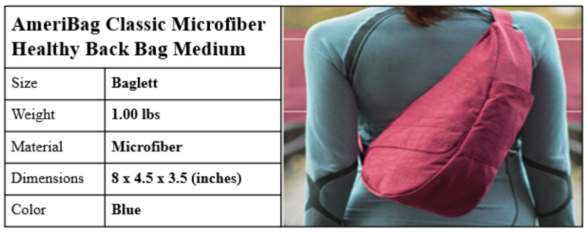
Current work to detect data errors often uses (semi-)automatic solutions. In this paper, however, we argue that there are many real-world scenarios where users have to detect data errors *completely manually*, and that more attention should be devoted to this problem. We then study one instance of this problem in depth. Specifically, we focus on the problem of *manually verifying the values of a target attribute*, and shows that the current best solution in industry, which uses crowdsourcing, has significant limitations. We develop a new solution that addresses the above limitations. Our solution can find a much more accurate ranking of the data values in terms of their difficulties for crowdsourcing, can help domain experts debug this ranking, and can handle ambiguous values for which no golden answers exist. Importantly, our solution provides a unified framework that allows users to easily express and solve a broad range of optimization problems for crowdsourcing, to balance between cost and accuracy. Finally, we describe extensive experiments with three real-world data sets that demonstrate the utility and promise of our solution approach.

1 INTRODUCTION

Data cleaning has received significant recent attention (e.g., [5, 7, 10, 46]), due to the explosion of data science applications, which often need data cleaning before analysis can be carried out. Most recent data cleaning works focus on detecting and repairing data errors [5, 7, 10, 46] (e.g., outliers, incorrect values, duplicate tuples, and constraint violations). In this paper we focus on *detecting* errors.

To detect data errors, current work often employs semi-automatic solutions, which use machine learning or hand-crafted data quality rules (e.g., "age must be between 18 and 80" and "any employee in NYC earns no less than any non-NYC employee at the same level" [5]). In certain cases the user can be involved, e.g., to provide feedback to the solutions or verify that the data instances reported by the solutions are indeed errors.

In practice, however, *there are many scenarios where users still have to detect data errors completely manually*. First, to detect data errors we often need to extract the values of certain attributes. Such extraction can be very difficult for today algorithms, but much easier for human users. This often happens when an attribute value is buried in a picture or text.



AmeriBag Classic Microfiber Healthy Back Bag Medium	
Size	Baglett
Weight	1.00 lbs
Material	Microfiber
Dimensions	8 x 4.5 x 3.5 (inches)
Color	Blue

Figure 1: An example of manual error detection.

Example 1.1. Consider the product in Figure 1. A data quality rule is "the value of attribute color *should be consistent with the color of the product in the picture". There is no algorithm today that can reliably extract the color of the product from the picture. Here the picture shows not just the product, a bag, but also a woman wearing a bag, making the extraction of the bag's color even more difficult. A human user however can quickly detect that the bag's color in the picture is red. This is inconsistent with the value of attribute* color *in the text, which is blue, suggesting a data error.*

Even if an attribute's values are present (so no extraction is required), it can still be difficult for algorithms to judge if those values are correct. For example, there is no good algorithm today to detect if a given URL is indeed the correct URL for a given business (especially where multiple fake URLs exist for a business). So detecting incorrect business URLs (e.g., to clean business listings) is still done largely by human users. Another example is verifying if the category of a product is "athletic (man)", which typically requires a human user to read the product description, examine the picture, etc.

Finally, an algorithmic solution may exist (e.g., extracting a person's gender from a picture can be done reliably today using deep learning [24]), but the business may have no one qualified to develop, debug, and run it. Or there is someone qualified, but developing and debugging the algorithm would take weeks, whereas the cleaning work must be done within days. In such cases, businesses often resort to detecting data errors completely manually, using human users.

In this paper we consider manually detecting data errors. As a first step, we consider the common setting in which *users must manually check the correctness of the values of a target attribute* (e.g., color, category). This problem is often called *manual data validation* [5, 7, 10, 46]. It is pervasive, yet no published work has addressed it in depth, as far as we can tell.

Using in-house experts to do manual data validation is not practical for large amounts of data: it takes too long and is not a good use of their limited time. So companies often use *crowdsourcing*, where the crowd can be for instance contractors or

Mechanical Turk workers. A common solution formulates each error detection as a question, sends it to the crowd, solicits k answers (e.g., $k = 5$), then takes a majority vote. For example, if three out of five workers answer "no" to the question "is the color of this product indeed blue?" for the product in Figure 1, then we can report that product as potentially having a data error.

The above solution is conceptually simple, but inefficient. Intuitively, different data values pose different levels of difficulties to human users. For example, most people know the color "blue", and so can answer questions about this color with high accuracy. But fewer people know the color "chartreuse". So we may want to solicit fewer answers for questions involving "blue" (e.g., 3 answers per question), but more answers for questions involving "chartreuse" (e.g., 7). Such crowdsourcing strategies, which are sensitive to the difficulties of different data regions, can significantly reduce the crowdsourcing cost while achieving the same level of error detection accuracy.

Indeed many companies have now employed such *adaptive crowdsourcing strategies*. A very common solution (e.g., employed at WalmartLabs, Facebook, Johnson Controls, and elsewhere) works as follows:

(1) Compute a ranking K of the data values (in decreasingly order of their difficulties),
(2) Examine K to assign to each data value v a number of answers n_v (such that a data value placed higher in K is assigned a higher number of answers), then
(3) Solicit n_v answers for each question with value v.

Obtaining the ranking K is important for many purposes. For example, after crowdsourcing to obtain answers for all questions, companies often take a sample and manually check the accuracies of the questions in the sample, for quality assurance purposes. The ranking K allows them to bias the sample, e.g., intentionally sample more items with values in the top-10 of K, to check how well crowdsourcing works for these difficult values. Example 4.1 lists other usages of ranking K.

While the above solution has been quite popular in industry, it has several important limitations. In this paper we address those and significantly advance this line of research in several ways.

First, obtaining *a good ranking of the data values* in terms of their difficulties is critical. To do so, the above solution estimates the difficulty score of a data value to be *the average worker accuracy* for a sample set S of questions with that value. To estimate these scores accurately, the size of sample set S must be quite large. But this incurs a lot of domain expert's effort, because he or she must label all data instances in S (as having data errors or not).

Here we show that we do not need a large sample S. Our key idea is that if the average time it takes for a worker to answer questions is high, or if the disagreement among workers is high, then those also indicate that a data value is likely to be difficult. Consequently, *we use all three factors (i.e., worker accuracy, average answer time, and worker disagreement) to directly rank the data values, using a machine learning approach.* We show how to minimize the domain expert's effort, by iteratively expanding sample S and stopping when a convergence condition is met.

Second, once the ranking has been created, domain experts often want to examine, debug, and modify it. To address this problem, *we develop a solution to help a domain expert debug the ranking.* Specifically, he or she can request explanations on why a data value v is considered difficult. Among others, our solution can explain that v is not difficult, but appears so due to spammers, low-quality workers, or careless mistakes from the workers; or that v is indeed difficult, because the value is hard to understand (e.g., "chartreuse"), or the item description is incomplete, or the description has confusing/conflict information, etc.

Third, the existing solution considers only the problem of *minimizing the crowdsourcing cost while achieving the same detection accuracy* (as the baseline solution of soliciting the same number of answers regardless of the data value). We show that in practice, users want to consider a far broader range of problems. Examples include minimizing cost given that the accuracy exceeds a threshold, maximizing accuracy given a budget on the cost, improving the overall accuracy of a set of data items having difficult values, and more. *We develop a unified framework that allows users to easily express and solve a broad range of such optimization problems*, all of which find crowdsourcing strategies that adapt to the data value difficulties.

Finally, the existing solution assumes *golden answers* exist for the questions with each data value (otherwise the worker accuracy for that value cannot be computed). In practice, surprisingly, we found that there are many cases where there are no such golden answers. For example, a product description may show the picture of a bag in sand color, with the value for attribute "color" being "desert sand". So the question for the crowd is "is the color of this product 'desert sand'?". But nobody knows what "desert sand" means. There is no such color. Or more accurately, this is an *ambiguous* color invented by the marketing team. As such, there is no correct, i.e., golden answer to the above question. (In our experiments, 2/3 of workers answer yes, and the rest answer no.) Clearly, this problem of *ambiguous values* must be addressed, before the above adaptive crowdsourcing solution can be applied. In this paper we develop a simple but effective solution to this problem.

Contributions: To summarize, in this paper we make several fundamental contributions to the problem of manually detecting errors for data cleaning:

- We argue that the above problem is pervasive, and needs more attention. As far as we can tell, this is the first work that studies this problem in depth.
- We focus on the problem of manually verifying the values of a target attribute, and shows that the current best solution has significant limitations.
- We develop a new solution that addresses the above limitations and significantly advances the state of the art. Our solution can find a much more accurate ranking of the data values, can help domain experts debug this ranking by providing explanations on why a data value is considered difficult, and can handle ambiguous values for which no golden answers exist. Importantly, our solution provides a unified framework that allows users to easily express and solve a broad range of optimization problems.
- We describe extensive experiments with three real-world data sets that demonstrate the utility and promise of our solution approach.

2 PROBLEM DEFINITION

We now describe the problem of manual detection of data errors considered in this paper.

Data Items, Attributes, and Values: For manually detecting data errors, many problem types exist. As a start, in this paper we will consider the problem of manually verifying the correctness of the categorical values of a target attribute. Specifically, let $D = \{d_1, \ldots, d_n\}$ be a set of data items, such as books, papers, products, etc. We assume that each item is encoded as a tuple of attribute-value pairs, i.e., $d_i = \langle a_1 = v_{i1}, \ldots, a_m = v_{im} \rangle$. For example, a product may be encoded as $\langle category = shirt, gender = male, color = blue \rangle$. We will use $a_j(d_i)$ to refer to the attribute a_j of d_i.

We assume that each attribute $a_j(d_i)$ has a set of correct values $V_j^*(d_i)$. For example, a course about discrete math is suitable for students from both mathematics and computer science departments, therefore its subject contains at least two values: mathematics and computer science. We say that $a_j(d_i)$ is correct if and only if its value v_{ij} is in $V_j^*(d_i)$.

Further, we assume that all attributes of each data item d_i are correct, except one attribute, which is referred to as the *target attribute* and whose values we will need to verify. Without loss of generality, we assume that the target attribute is the last attribute a_m.

Manual Validation of the Target Attribute: Let c_i be the context of d_i, defined as "all other attributes and their values": $c_i = \langle a_1 = v_{i1}, \ldots, a_{m-1} = v_{i(m-1)} \rangle$.

Our problem is to verify $a_m(d_i)$ for all items d_i in D. For each item d_i, verifying whether the value of a_m is correct is equivalent to answering the following question q_i: "is the value of $a_m(d_i)$ indeed v_{im}, given the context c_i and any other background knowledge B that the worker may have?" (we discuss examples of background knowledge B below).

Then the problem of verifying the target attribute a_m for items in D can be translated into answering the set of questions $Q = \{q_1, \ldots, q_n\}$, where the answer for each question q_i is yes or no. If the answer is yes, then our confidence that $a_m(d_i)$ is correct is increased. If the answer is no, then it is likely that there is a data error in d_i (in practice, the error may not be in a_m, but the error must exist because $a_m(d_i)$ is inconsistent with c_i). In this case, d_i is sent for further verification by an expert.

Suppose we have golden answers for all questions in Q, then for any solution to the above validation problem, we can define its overall accuracy to be the fraction of questions whose answers are correct, i.e., n_0/n, where n_0 is the number of questions whose answers match the golden answers, and $n = |Q|$.

Current Manual Solutions: Today, such questions are often answered manually, on a GUI, by an expert or a small set of experts, e.g., data analysts at an e-retailer, data scientists in an R&D group. To give the expert the maximal context information, a question will typically display the entire description of the item, e.g., all attribute-value pairs (see Figure 1).

If the expert still cannot decide after examining these attribute-value pairs, he or she may try to find more information, e.g., examining the same product at a different e-retailer, looking for any new information that can help answer the question. For the question "is the color of this product chartreuse?", the expert may have to first look up the meaning of "chartreuse" on the Web, and so on. We refer to such externally acquired knowledge as *the background knowledge B*.

Clearly, this manual solution is tedious and time consuming. As a result, many real-world applications have turned instead to crowdsourcing to verify attribute values.

Current Crowdsourcing Solutions: The simplest crowdsourcing (CS) solution solicits k answers from crowd workers for each question $q \in Q$, then combines these answers using majority voting to obtain a final answer for q.

Note that we can combine worker answers using more sophisticated strategies, e.g., estimating each worker's accuracy, then taking a weighted sum [19, 22, 27, 39]. Many real-world applications, however, still use majority voting, which is easy to understand, debug, and maintain. This is especially important when there is a high personnel turnover. Further, the application may need to contract with a crowdsourcing company and this may be the only solution being offered by that company. Finally, as far as we know, there is no published conclusive evidence yet that more sophisticated strategies to combine answers work much better in practice. Thus, in this paper we will focus on the above majority-voting solution to verify attribute values, leaving more sophisticated solutions for future research.

The above CS solution, while faster than manual solutions, can incur high monetary costs, especially if the application wants high accuracy for crowdsourcing.

Example 2.1. Suppose an e-retailer A must verify the attributes of 50K newly arrived products. To ensure that product details on its Web pages are error-free as much as possible, A wants crowdsourcing to have an accuracy of at least 95%. To reach this accuracy, soliciting 3 answers per question is often insufficient, A would need to solicit 5, 7, or more answers. Assuming 3 cents per answer, if A solicits 5, 7, or 9 answers per question, crowdsourcing 5 attributes of 50K products costs $37.5K, $52.5K, and $67.5K, respectively.

Thus, it is important that we develop solutions to minimize the crowdsourcing cost, while achieving the same verification accuracy. As discussed in the introduction, intuitively, different data regions often have different degrees of difficulty for human verification. So if we can estimate these difficulty levels, we can adjust the degree of redundancy (i.e., the number of answers solicited for each question in a region). For example, a set of products D can be split into data regions where all products with the same color form a region. Then for each question in the region with "red" color, we only need to solicit 3 answers, say; whereas for each question in the region with the "acid yellow" color, which is more difficult, we would solicit 7 answers.

Indeed many companies have now employed such *adaptive CS strategies.* A very common solution works as follows:

(1) Compute a ranking of the data regions (in decreasingly order of their difficulties),
(2) Examine the ranking to assign to each region a number of answers (such that a region placed higher in the ranking is assigned a higher number of answers), then
(3) Solicit that number of answers for each question in the region.

It is important that this solution outputs both a ranking and a crowdsourcing plan (which specifies how many answers to solicit for each question in a data region). Outputting a ranking serves many important purposes, as discussed in the introduction (see also Example 4.1).

The above solution is appealing, but has significant limitations. (1) The ranking that it computes is often inaccurate, because the solution uses only the average worker accuracy to find the ranking. (2) Domain experts often cannot debug the ranking, e.g., understand why a data region is considered difficult. (3) The task of assigning to each data region a number of answers is often done in an ad-hoc "eyeballing" way, by examining where the data region is in the ranking. (4) It is difficult to express and solve a broad range of optimization problems regarding crowdsourcing costs and accuracy, even though users often have such needs. (5) Finally, this solution cannot handle "ambiguous" data values (e.g., "desert sand"), for which there are no golden answers.

In the rest of this paper we introduce our solution, called VChecker, which addresses the above limitations.

3 RANKING THE DATA REGIONS

In VChecker, we first obtain a ranking of the data regions, in decreasing order of their difficulties. In this section we discuss how we split the data into different regions, then rank them. (The next two sections describe how to debug the ranking, then how to use it to formulate and solve a broad range of optimization problems, to find good crowdsourcing plans.)

3.1 Splitting Data into Regions

We consider scenarios where for each data instance d_i, the difficulty in verifying the target attribute a_m only depends on its value v_{im}. Such scenarios are common in practice. For instance, for products such as the one described in Figure 1, the difficulty of verifying the attribute color only depends on its value (e.g., red, blue, acid yellow, etc.).

In such cases, the expert will split the data such that all questions with the same target attribute value form a region (because all such questions share the same difficulty level). Formally, let $D = \{d_1, \ldots, d_n\}$ be the set of data instances, a_m be the target attribute, $Q = \{q_1, \ldots, q_n\}$ be the set of questions "is the value of attribute a_m of instance d_i indeed v_{im}?", and $V = \{v_1, \ldots, v_r\}$ be the set of all values of a_m for instances in D. Then we can split the set of questions Q into r sets such that all questions (and only these questions) in a set Q_i share the same value for attribute a_m. We refer to each such set as a data region. In general, it is not always possible to so simply split the data into regions. This raises the interesting problem of how to help the expert do so, which we leave for future work.

3.2 Learning to Rank the Regions

To rank the data regions, a common solution in industry is to compute for each region an *average worker accuracy*, then rank the regions in increasing worker accuracy (thus in decreasing difficulties).

Specifically, let Q_i be a data region, i.e., the set of questions (in Q) with the same value v_i for the target attribute a_m. The current solution assumes that all crowd workers have the same probability of answering any question in Q_i correctly (a reasonable assumption in many real-world scenarios). It then takes this probability to be the average worker accuracy for Q_i, denoted as $a(Q_i)$.

To estimate $a(Q_i)$, the solution randomly takes a set x of questions in Q_i, solicits y answers from the crowd for each question, then computes $a(Q_i)$ as the fraction of xy answers that are correct. To determine answers' correctness, the solution uses the golden answers to the x questions, as provided by an expert. Finally, the solution ranks the data regions in increasing order of the computed average worker accuracy $a(Q_i)$.

While conceptually simple, this solution is limited in several important ways. First, it provides no way to determine x and y. If they are set to large values, then we waste a lot of crowdsourcing money and expert time (to provide golden answers). If they are set to small values, then it is difficult to estimate $a(Q_i)$ *accurately*. Second, it fails to exploit extra information that can help better

V = {black, red, iris, lavender, chartreuse}
(a)

$G = \{\langle \text{black}, f_1 \rangle, \langle \text{red}, f_2 \rangle, \langle \text{chartreuse}, f_5 \rangle\}$
(c)

$$F = \{f_1 = \langle 1, 2.3, 0 \rangle, \; f_2 = \langle 1, 2.4, 0.05 \rangle, \; f_3 = \langle 0.7, 5.1, 0.1 \rangle, \; f_4 = \langle 0.6, 8.4, 0.1 \rangle, \; f_5 = \langle 0.7, 11.2, 0.15 \rangle\}$$
(b)

{chartreuse} $\geq$ {black, red}
(d)

$$S = \{(f_5, 1), (f_1, 2), (f_2, 2)\}$$
(e)

Figure 2: Creating training data for SVM Rank.

rank the data regions. Finally, the solution does not provide any way to solicit and incorporate knowledge from the expert, even though he or she often has such knowledge about the difficulties of the various data regions.

Key Ideas of Our Solution: Our solution exploits three key ideas. First, we observe that if a value is difficult, it often takes a worker longer to provide an answer (e.g., for a question involving the value "acid yellow", he or she may need to consult the Web to understand its meaning before being able to answer the question). It also often causes more disagreement among the workers. As a result, we capture and exploit these two types of information and use them together with the worker accuracy to learn to rank the values in decreasing order of their difficulties.

Second, to learn the ranking, we ask the expert to provide training data in the form of (v_i, v_j) such that v_i is ranked more difficult than v_j. We also allow the expert to debug the ranking and manually edit it if necessary (see Section 4). Thus, our solution provides a natural way for the expert to provide domain knowledge about the difficulties of the data regions.

Finally, to minimize the crowdsourcing and expert cost, we develop a solution in which we iteratively explore larger values for x (the number of questions sampled per value) and y (the number of answers solicited per question), and stop when a condition is met. We now describe the above ideas in detail.

1. Defining the Problem of Ranking the Values: Let $V = \{v_1, \ldots, v_r\}$ be the set of all values of the target attribute for all data instances in D. Our goal is to find a total ranking K of the values in V, such that v_i being ranked higher than v_j means that v_i is judged more difficult than v_j.

2. Learning the Ranking: For each value $v_i \in V$, we start by sampling x questions from the corresponding region Q_i, then solicit y answers for each question from the crowd (we explain later how to select x and y). This produces a total of xy answers.

Next, we create a feature vector $f_i = \langle a_i, t_i, e_i \rangle$, where a_i is the worker accuracy for v_i, computed as the fraction of xy answers that are correct. To determine if an answer is correct, the expert must provide the golden answers for the x questions. t_i is the time it takes for a worker to answer the questions, averaged over the xy answers. Finally, e_i is the disagreement among the workers in answering the questions, measured as $1 - |N_{yes} - N_{no}|/(xy)$, where N_{yes} and N_{no} are the total numbers of yes/no answers from the workers, respectively (and $N_{yes} + N_{no} = xy$).

Example 3.1. Consider the five colors in set V in Figure 2.a. Figure 2.b shows five feature vectors created for these colors, after sampling x questions from each color region and soliciting y answers from the crowd for each question.

At this point, we have obtained a set of feature vectors $F = \{f_1, \ldots, f_r\}$, one for each value. We now learn to rank the values, using these feature vectors. To do so, we use SVM Rank, a well-known ML algorithm that can be used to rank examples [38].

To use SVM Rank, we create training data as follows. First, we randomly sample a set G of feature vectors (FVs) from F. Next, we need to rank the FVs in G (in terms of the difficulty of their corresponding values). Abusing notation, we use "$f_i \geq f_j$" to indicate that FV f_i is ranked the same or higher than FV f_j (i.e., the value corresponding to f_i is the same or more difficult than that of f_j).

Ideally, we want to create a total ranking on G, i.e., for any pair $(f_i, f_j) \in G \times G$, either $f_i \geq f_j$ or $f_j \geq f_i$, then use this total ranking as training data. However, creating a total ranking is very expensive and often quite difficult for the expert, so we ask him or her to create only a partial ranking. Specifically, the expert merely divides G into two groups U and V based on his or her domain knowledge such that for any $f_i \in U$ and $f_j \in V$, $f_i \geq f_j$.

Then for each $f_i \in U$, we create a training example $(f_i, 1)$. Similarly, for each $f_j \in V$, we create a training example $(f_j, 2)$. Here we assume that an example associated with rank 1 is more difficult than any example associated with rank 2. We output the set S of all these examples as the training data for SVM Rank.

Example 3.2. Continuing with Example 3.1, suppose we have selected the three colors black, red, and chartreuse for creating the training data (see Figure 2.c). Suppose the expert specifies that chartreuse is considered more difficult than both black and red (Figure 2.d). Then we can create the training set S in Figure 2.e for SVM Rank.

SVM Rank then uses S to learn a regression model that assigns a score to each example, such that the higher the score, the higher the example is ranked. Finally, we apply SVM Rank to FVs in F to compute for each FV a score and use these scores to rank the FVs. This produces a total ranking K for the values in V, such that a higher ranked value is said to be more difficult than a lower-ranked one.

The expert can then optionally examine, debug, and edit the ranking K, as we discuss later in Section 4.

3. Determining Parameters x and y: Recall that for each value v_i we take x questions from the set of questions with that value Q_i, then solicit y answers per question from the crowd. We now discuss how to set x and y. Our solution is to start with the smallest x and y, iteratively increase them, computing rankings along the way, then stop when these rankings have "converged". This way, we hope to minimize the cost of the expert (who needs to answer $x|V|$ questions and the crowd (who needs to answer $xy|V|$ questions).

Specifically, we start with (2,2), i.e., $x = 2$ and $y = 2$ (the smallest values that allow us to meaningfully compute feature vectors), and compute the ranking of the values $K(2, 2)$, as described earlier. Next, we increase y to consider (2,3), and compute $K(2, 3)$. Then we consider (3,3) and compute $K(3, 3)$, and so on. To compute a new ranking, say $K(3, 3)$, using SVM Rank, the expert needs to label, i.e., provide golden answers to the new questions, and we need to solicit crowd answers for these questions. But we do not have to create any additional training data.

We use the Spearman score [48], which ranges from 1 to -1, to measure the correlation between any two rankings. Consider three consecutive rankings $K(x_{n-2}, y_{n-2})$, $K(x_{n-1}, y_{n-1})$, $K(x_n, y_n)$. If it is the case that the score Spearman$(K(x_{n-2}, y_{n-2}), K(x_{n-1}, y_{n-1}))$ and the score Spearman$(K(x_{n-1}, y_{n-1}), K(x_n, y_n))$ are both exceeding a pre-specified threshold, or if x_n and y_n reach pre-specified maximal values, then we stop, returning $K(x_n, y_n)$ as the desired ranking. Algorithm 1 shows the pseudo code of the entire process to rank the data regions.

4 DEBUGGING THE RANKING

Recall from the previous section that at the start, we enlist the expert and the crowd workers to create a ranking K of the values in V, in decreasing order of difficulties. In practice, it turns out that the ranking K can be used for many important purposes.

Example 4.1. The ranking K can be used to re-calibrate the worker accuracies of the values (see Section 5.2). It can be used in formulating optimization problems, e.g., a user may want to focus on the top-10 most difficult values in K and try to maximize the average accuracy of those values (see Section 5.1). Finally, K can also be used in quality assurance (QA). For example, after we have crowdsourced to obtain answers for all questions, we may want to take a sample and manually check the accuracies of the questions in the sample, for QA purposes. The ranking K allows us to bias the sample, e.g., intentionally sample items with values in the top-10 of K, to check how well the crowdsourcing process works for these difficult values.

As a result, *it is important to make the ranking K as accurate as possible.* Once K has been created (see Section 4), the expert often wants to examine, debug, and modify it. Currently, however, there is no debugging support. To address this problem, as a first step, in this paper we will develop a way to generate explanations, which can help the expert debug K.

Specifically, given a value v placed high in the ranking K, indicating that it is difficult, the expert can ask for an explanation on why v is judged difficult by the system.

Example 4.2. When asked why "acid yellow" is judged difficult, our system may return explanations that state that this value is actually not difficult, but appears to be difficult due to spammers and low-quality (i.e., bad) workers who gave many incorrect answers. Or the system may return explanations that state that the value is indeed difficult because it is unfamiliar to many workers. Other explanations may include "the product description contains incomplete or confusing information" and "the description is hard to understand", among others.

Clearly, such explanations can significantly help the expert understand and debug the ranking K. To generate such explanations, we first develop a model M on how a crowd worker answers a question. Next, we analyze M to create a taxonomy T of possible explanations. Finally, we develop a procedure that, when given a value v, analyzes answers solicited from the crowd to identify the most likely explanations in T for v. We now discuss these steps in more details.

Developing a User Model for Answering Questions: There are many possible ways to model how a worker answers our questions. For this paper we use the following simple yet reasonable model. Suppose a worker U has to answer a question q, which has a value v for the target attribute and a context c (which is the rest of the description of the data item). Then U first tries to understand v. Next, U tries to understand c. Finally, U determines if v and c are consistent, returning "yes" or "no" if

Algorithm 1 Learning to Rank the Data Regions

```
Input: $Q$: set of questions, $V$: set of values, $x_{\max}$: max num of sampled questions,
    $y_{\max}$: max num of answers to be collected per sampled question, $(x_0, y_0)$: initial
    value for $(x, y)$, $\epsilon$: convergence threshold for ranking, $n_0$: number of training
    examples for SVM Rank
Output: A ranking $K$ of values
 1: $V_t \leftarrow$ Randomly sample $n_0$ values from $V$
 2: $O \leftarrow$ CreatePartialOrder($V_t$)
 3: $P \leftarrow$ GenerateConfigs($x_0, y_0, x_{\max}, y_{\max}$)
 4: $Q_s, A_s, T_s, G_s \leftarrow \emptyset$
 5: $(x_c, y_c) \leftarrow (0, 0)$ // current $(x, y)$
 6: $L \leftarrow [\ ]$ // list of rankings
 7: for $(x, y) \in P$ do
 8:     Sample $x - x_c$ new questions per value and add them to $Q_s$
 9:     Collects needed answers and time data and add them to $A_s$
        and $T_s$
10:     Find golden answers for newly sampled questions and add
        them to $G_s$
11:     Compute set of features $F$ from $A_s, T_s, G_s$
12:     $K(x, y) \leftarrow$ SVMRank values in $V$ using $F$, $O$
13:     $(x_c, y_c) \leftarrow (x, y)$
14:     Append $K(x, y)$ to $L$
15:     if IsRankingConverged($L$, $\epsilon$) then
16:         break
17:     end if
18: end for
19: $K \leftarrow K(x_c, y_c)$
20: $W \leftarrow$ Improve estimated worker accuracy using $K$ in Equation 4
21: return $K$, $W$

22: procedure CreatePartialOrder($V_t$)
23:     The expert partitions $V_t$ into two groups $U$, $V$ such that each
        value in $U$ is more difficult than each value in $V$
24:     $O \leftarrow \emptyset$
25:     for $v \in U$ do
26:         Add $(v, 1)$ into $O$
27:     end for
28:     for $v \in V$ do
29:         Add $(v, 2)$ into $O$
30:     end for
31:     return $O$
32: end procedure

33: procedure GenerateConfigs($x_0, y_0, x_{\max}, y_{\max}$)
34:     $P \leftarrow [(x_0, y_0)]$
35:     $n = \min(x_{\max} - x_0, y_{\max} - y_0)$
36:     for $i = 1, 2, \ldots, n$ do
37:         Append $(x_0 + i - 1, y_0 + i)$ and $(x_0 + i, y_0 + i)$ to $P$
38:     end for
39:     if $x_0 + n == x_{\max}$ then
40:         $m = y_{\max} - y_0 - n$
41:         for $i = 1, 2, \ldots, m$ do
42:             Append $(x_0 + n, y_0 + n + i)$ to $P$
43:         end for
44:     else if $y_0 + n == y_{\max}$ then
45:         $m = x_{\max} - x_0 - n$
46:         for $i = 1, 2, \ldots, m$ do
47:             Append $(x_0 + n + i, y_0 + n)$ to $P$
48:         end for
49:     end if
50:     return $P$
51: end procedure

52: procedure IsRankingConverged($L$, $\epsilon$)
53:     if $len(L) < 3$ then
54:         return False
55:     else
56:         Let $K_{n-2}, K_{n-1}, K_n$ be the last three rankings in $L$
57:         $s_1 \leftarrow$ Spearman($K_{n-2}, K_{n-1}$)
58:         $s_2 \leftarrow$ Spearman($K_{n-1}, K_n$)
59:         if $s_1 \geq \epsilon$ and $s_2 \geq \epsilon$ then
60:             return True
61:         else
62:             return False
63:         end if
64:     end if
65: end procedure
```

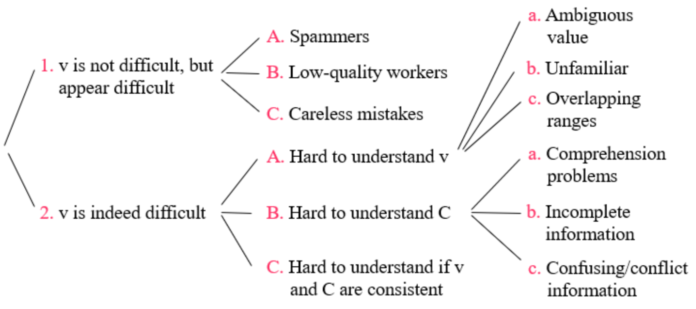


Figure 3: A taxonomy of explanations.

Algorithm 2 Generating Explanations

```
Input: $v$: the value to be explained
Output: $E_v$: the set of possible explanations for value $v$
 1: Collect data $S = \{Q_x, A, W\}$ where $Q_x$ is all $x|V|$ questions sam-
    pled in difficulty estimation stage, $A$ is all answers and their time
    stats for questions in $Q_x$, $W$ is the set of all workers for $A$
 2: Compute accuracy $\alpha$ and average time $t$ for $A$
 3: for each $w \in W$ do
 4:     Apply a procedure on $R_w$ using $\alpha$, $t$ to classify $w$ as spammers,
        low-quality workers, or regular workers
 5: end for
 6: Let $W_v$ be the set of workers who answer at least one question with
    value $v$
 7: $E_w \leftarrow$ GenWorkerExplanations($W_v$)
 8: Update $S$ to $S^+$ by removing answers and the time stats from spam-
    mers and low-quality workers
 9: Apply a procedure on $R_v$ using $S^+$, $\alpha$, $t$ to classify the nature $N_v$
    of value $v$ (i.e., ambiguous, unfamiliar, overlapping)
10: $E_v \leftarrow$ GenValueExplanations($N_v$)
11: Let $Q_v$ be the set of questions in $Q_x$ with value $v$
12: for each $q \in Q_v$ do
13:     Apply a procedure on $R_q$ using $\alpha$, $t$ to classify the nature of $q$
        (i.e., comprehension, incomplete, confusing/conflict)
14: end for
15: Let $N_q$ be the set of natures for questions in $Q_v$
16: $E_q \leftarrow$ GenQuestionExplanations($N_q$)
17: $E_v \leftarrow E_w \cup E_v \cup E_q$
18: return $E_v$
```

U can make this determination with high confidence. Otherwise U returns the answer ("yes" or "no") judged most likely.

Creating a Taxonomy of Explanations: Analyzing the above user model produces the taxonomy of explanations in Figure 3. Observe that the explanations fall into several clean groups. The first group concerns the *workers:* if they are spammers, bad workers, etc., then the value may not be difficult but will appear difficult. The second group concerns the *nature of the value v itself:* is it ambiguous, unfamiliar, etc? If so, it may explain why v is ranked difficult. The final group concerns the *nature of the question/the description of the data item/the context.* Is the description understandable (e.g., in English)? Is it complete? As we will see below, we can develop solutions to explore each of the above groups of explanations.

Generating Explanations for a Value v: Given a value v, we now seek to generate explanations for why v is difficult. We refer to each node in the above taxonomy (see Figure 3) as an *explanation.* Our solution will return a set of such explanations (in future work we will explore ranking them). To do so, the solution proceeds in the following steps (see Algorithm 2 for the pseudo code).

(1) Collect data S: We first collect data that can be analyzed to generate explanations. This data S consists of Q_x, the set of all questions generated for the difficulty score estimation process, A, the set of all answers solicited for questions in Q_x, and W, the set of all workers who have given at least one answer in A.

(2) Use S to classify the workers: We use a rule-based procedure to classify workers in W into spammers, bad workers, and regular workers. For example, we classify a worker w as a spammer if w's accuracy is significantly lower than the average accuracy, and w's response time is much faster than the average response time (as computed from data S).

(3) Generate explanations regarding the nature of the workers: Next, we identify likely explanations regarding the nature of the workers in the taxonomy T. For example, if a certain percentage of workers that have answered questions involving v are spammers, then we will identify node "1.A (Spammers)" of T as an explanation.

(4) Update data S into S^+: Next, we remove the data involving the spammers and bad workers from S, so that we can work with more accurate statistics in subsequent steps.

(5) Use S^+ to classify the nature of value v: Similar to Step 2, here we use a rule-based procedure to analyze S+, to classify the value v as ambiguous, unfamiliar, etc. For example, if the average worker accuracy for v is high and the average response time is low, then we determine that v is not unfamiliar.

(6) Generate explanations regarding the nature of value v: Again, similar to Step 3, we identity explanations in taxonomy T that involve the nature of value v. This step is straightforward.

(7) Use S^+ to classify the nature of the questions and generate explanations: We proceed similarly to Steps 2-3. For example, if a certain percentage of the questions involving v is confusing, then we will identify node "2.B.c" of T as an explanation. Finally, we return all identified explanations as the set of explanations for value v.

It is important to note that our rule-based procedures for the above steps have been created, only once. They are not dependent on the particular application domain. However, the rules employed do use various parameters (e.g., thresholds). These parameters are set based on analyzing the data S (but can also be tuned by the domain expert).

5 FINDING GOOD PLANS

We now discuss finding good crowdsourcing plans. We begin by considering the *types of problems* that the user wants to solve. As discussed in Section 1, a common baseline crowdsourcing (CS) plan is to solicit t_b answers per question, then take the majority vote to be the final answer. The existing solution has considered a single problem: minimize the total CS cost while keeping the accuracy the same as that of the baseline plan.

In practice, however, we observe that *users often want to express a wide range of other CS problems*. Examples include minimizing cost given that the accuracy exceeds a threshold, maximizing accuracy given a budget on the cost, improving the overall accuracy of a set of data items having difficult values, and more.

As a result, in this section we develop a unified framework in which users can easily express a variety of such CS problems. Some of these problems make use of the ranking K (e.g., maximizing the average accuracy of the values in the top-5 of K). Next, we show how to solve these problems using integer linear programming (ILP). Our solutions often involve the average worker accuracy per data value. Finally, we show how to use the ranking K to improve our estimations of these average worker accuracies.

5.1 Expressing Crowdsourcing Problems

Let $D = \{d_1, \ldots, d_n\}$ be a set of data items to be validated. Let $V = \{v_1, \ldots, v_r\}$ be the set of values for the target attribute of the items in D. We define a crowdsourcing plan p to be a tuple $\langle\langle v_1, t_1\rangle, \ldots, \langle v_r, t_r\rangle\rangle$, where for each question involving the value v_i, plan p will solicit t_i answers from the crowd ($i \in [1, r]$).

Let $S \subseteq V$ be a set of values. We define $acc(S, p)$ to be the accuracy of plan p for the values in S, i.e., the fraction of questions with value $v \in S$ that receive a correct (aggregated) answer when p is executed. We define $cost(S, p)$ to be the total number of answers solicited from the crowd for the questions with value $v \in S$.

We can now define a general CS problem template as follows *"Given a set of plans P and a set of values $\mathcal{S}$, return the plan that maximizes or minimizes an objective O, subject to a constraint C, where O and C involve P and $\mathcal{S}$, and optionally a ranking K of values".* In this paper we consider the following concrete CS problems that follow the above template.

Finding Plans That Outperform a Baseline Plan: In many scenarios there exists already a baseline plan p_b. The user however wants a plan p that is better than p_b in some aspects. While numerous problem variations exist, in this paper we consider the following variations:

T_1: Minimize cost while achieving the same accuracy

Return the plan p that minimizes $cost(V, p)$, subject to constraints $acc(V, p) \geq acc(V, p_b)$ and $cost(V, p) \leq cost(V, p_b)$. This is the problem considered by PBA [14].

T_2: Maximize accuracy while keeping the same cost

Return the plan p that maximizes $acc(V, p)$, subject to constraints $cost(V, p) \leq cost(V, p_b)$ and $acc(V, p) \geq acc(V, p_b)$.

T_3: Maximize the individual accuracy

In many cases the overall accuracy $acc(V, p)$ can be high, say 95%, yet certain individual accuracies (e.g., $acc(v, p)$ for certain v-s) may be quite low, say 60%. For example, the overall accuracy for color verification can be 95%. Yet the accuracy for "chartreuse" is only 60%.

In such cases, the user often wants to improve the accuracies of the values *across the board* as much as possible, while keeping the overall accuracy at least as high as that of p_b and keeping the overall cost at most as high as that of p_b. To do this, the user can try to solve the following problem: Return the plan p that maximizes $min_{v_i \in V} acc(v_i, p)$, subject to $acc(V, p) \geq acc(V, p_b)$ and $cost(V, p) \leq cost(V, p_b)$. Intuitively, if a plan increases $min_{v_i \in V} acc(v_i, p)$, then it would increase the accuracies of all individual values.

Solving problems $T_1 - T_3$ for only a subset of values

The above problems $T_1 - T_3$ consider all values in V. In certain cases, the user may be interested in optimizing for only a subset of values $S \subseteq V$, such as the top 10 most difficult values, according to the ranking K. In such cases, we can formulate problems similar to $T_1 - T_3$, but replace V with S where appropriate.

Finding Plans That Satisfy General Constraints: In certain cases, the user does not have a baseline plan p_b to compare against. Instead, he or she just wants to find an "optimal" plan that satisfies certain constraints about cost and accuracy. Many variations exist. In this paper we consider the following:

T_4: Minimize cost while keeping accuracy above a threshold

Return the plan p that minimizes $cost(V,p)$, subject to constraint $acc(V,p) \geq \alpha$.

T_5: Maximize accuracy while keeping cost below a threshold

Return the plan p that maximizes $acc(V,p)$, subject to constraint $cost(V,p) \leq \beta$.

Solving problems $T_4 - T_5$ for only a subset of values

Again, in certain cases, the user may be interested in optimizing for only a subset of values $S \subseteq V$. In such cases, we can formulate problems similar to $T_4 - T_5$, but replace V with S where appropriate.

5.2 Solving Crowdsourcing Problems

We have described how users can express a variety of CS problems for detecting data errors. We now discuss how to solve them. The main idea is to formulate them as integer linear programming (ILP) optimization problems, solve these problems to find an optimal CS plan $p* = \langle\langle v_1, t_1\rangle, \ldots, \langle v_r, t_r\rangle\rangle$, then execute $p*$.

In what follows we discuss how to carry out the above idea for problem type T_1, then briefly discuss problem types $T_2 - T_5$. Recall that in problem type T_1, we want to find the plan p that minimizes $cost(V,p)$, subject to constraints $acc(V,p) \geq acc(V,p_b)$ and $cost(V,p) \leq cost(V,p_b)$. We now discuss how to estimate the quantities $cost(V,p)$, $cost(V,p_b)$, $acc(V,p_b)$, and $acc(V,p)$.

Estimating $cost(V,p)$ and $cost(V,p_b)$: It is straightforward to compute $cost(V,p_b)$, the crowdsourcing cost of the baseline solution. Recall that V is the set of values. Suppose each value v_i has n_i questions, then the total number of questions is $\sum_{i=1}^{r} n_i$. Since t_b answers need to be collected per question, $cost(V,p_b) = t_b \sum_{i=1}^{r} n_i$.

To compute $cost(V,p)$, recall that in our solution, for each value v_i, we have sampled x questions and collected y answers per sampled question. If plan p states that t_i answers will be collected for each remaining question, then the cost spent on value v_i will be $t_i(n_i - x) + xy$. Then the total cost on V can be computed as $cost(V,p) = \sum_{i=1}^{r}(t_i(n_i - x) + xy)$.

However, we cannot use t_i's as variables in the resulting ILP optimization problem (because constraints involving them will not be linear). To handle this problem, we use a set of indicator variables to represent t_i. Specifically, suppose $t_{\min}$ and $t_{\max}$ are the min and max number of answers to be collected per question (these two values are pre-specified; $t_{\min}, t_{\max}$ need to be odd positive integers since majority vote is used for aggregation). Let $A = \{t_{\min}, t_{\min} + 2, \ldots, t_{\max}\}$. Clearly, all t_i's are in A. To represent t_i, for each $j \in A$ we create an indicator variable h_{ij}. That is, if $j = t_i$, then $h_{ij} = 1$; otherwise $h_{ij} = 0$ for all $j \neq t_i$. We have $t_i = \sum_{j \in A} j h_{ij}$ and $cost(V,p) = \sum_{i=1}^{r}((\sum_{j\in A} j h_{ij})(n_i - x) + xy)$. As we will see shortly, our ILP formulation uses this formula for $cost(V,p)$.

Estimating $acc(V,p_b)$: Let m_i be the number of questions with value v_i whose aggregated answers are correct, then $acc(V,p_b) = (\sum_{i=1}^{r} m_i)/(\sum_{i=1}^{r} n_i)$, where n_i is the number of questions for value v_i.

To estimate m_i, for each question q with value v_i we need to compute the probability that q's aggregated answer is correct, which depends on the number of collected answers. Recall that we assume that all questions with value v_i have the same difficulty and workers are i.i.d. (i.e., identically independently distributed) for each value. When we collect the same number of answers per question for a value, the aggregated answers of those questions will have the same probability of being correct.

We define $f_{i,t}$ as the probability that for any question q with value v_i, q's aggregated answer is correct when t answers are collected per question. So if the baseline approach collects t_b answers per question, then $m_i = n_i f_{i,t_b}$, where n_i is the number of questions in region i (for value v_i). We now describe how to compute $f_{i,t}$ for any i and t.

To compute $f_{i,t}$, we use the worker accuracy a_i for value v_i. Since we assume that workers are i.i.d. for value v_i, when t answers are collected for question q in region i, these answers are independent and each answer has the probability a_i of being correct. So the number of correct answers follows the binomial distribution $B(t, a_i)$. Since we use majority voting, q's aggregated answer is correct if and only if more than half of the collected answers of q are correct. So we can compute

$$f_{i,t} = \sum_{j=\lceil t/2 \rceil}^{t} \binom{t}{j} a_i^j (1 - a_i)^{t-j} \tag{1}$$

For each value v_i, we compute f_{i,t_b} using (1), then estimate m_i's and $cost(V,p_b)$ as described above. (At the end of this subsection we will describe how we use the rankings from Section 4 to adjust a_i's for all v_i's.)

Estimating $acc(V,p)$: Recall that $A = \{t_{\min}, t_{\min} + 2, \ldots, t_{\max}\}$ and $f_{i,t}$ is the probability that for any question q with value v_i, q's aggregated answer is correct when t answers are collected per question. Using the indicator variables described earlier, the expected probability that q's aggregated answer is correct can be estimated as $\sum_{j\in A} h_{ij} f_{i,j}$. Then the overall accuracy of our approach is computed as $acc(V,p) = \frac{\sum_{i=1}^{r}(x+(n_i-x)(\sum_{j\in A} h_{ij} f_{i,j}))}{\sum_{i=1}^{r} n_i}$.

Formulating T_1 as an ILP Problem: We now can formulate problem T_1 as the following ILP problem:

$$\begin{aligned}
\underset{\substack{h_{ij} \forall j \in A,\\ i=1,2,\ldots r}}{\text{minimize}} \quad & \sum_{i=1}^{r}(xy + (\sum_{j\in A} j h_{ij})(n_i - x)) \\
\text{subject to} \quad & \frac{\sum_{i=1}^{r}(x + (n_i - x)(\sum_{j\in A} h_{ij} f_{i,j}))}{\sum_{i=1}^{r} n_i} \geq \alpha_b \\
& \sum_{i=1}^{r}(xy + (\sum_{j\in A} j h_{ij})(n_i - x)) \leq t_b \sum_{i=1}^{r} n_i \\
& \sum_{j\in A} h_{ij} = 1 \quad \forall i = 1, 2, \ldots, r \\
& h_{ij} \in \{0, 1\} \quad \forall j \in A, \forall i = 1, 2, \ldots, r
\end{aligned} \tag{2}$$

The objective function is the total number of answers to be collected, which should be minimized. The first constraint ensures that the overall accuracy is same or better than that of the baseline approach (here α_b is $acc(V,p_b)$). The second constraint ensures that the total cost is no more than that of the baseline. The third constraint ensures that for each value, only one indicator variable is equal to 1. Finally, we solve the above ILP problem using the Gurobi solver [1], and return any solution found to the user, as the crowdsourcing plan p to be executed. We have

Proposition 5.1. *Let t_{min} and t_{max} be the minimal and maximal number of answers that the user wants to solicit for each question. Let t_b be the number of answers that the baseline solution solicit for each question, and x be the number of questions that we sample per value for the difficulty estimation step. If $t_{min} \leq t_b \leq t_{max}$ and $t_b \geq x$, then Equation 2 always has at least one solution.*

Solving Problems $T_2 - T_5$: So far we have discussed solving problem T_1. Problems T_2, T_4, and T_5 can be transformed similarly. For T_2, we only need to change the objective to maximize $acc(V, p)$ (which is the left side part of first constraint in problem T_1). For T_4 (or T_5), we only need to replace the estimated baseline accuracy (or cost) with the given accuracy (or cost) threshold from problem T_1 (or T_2) and remove the unnecessary constraint on cost (or accuracy).

Solving T_3, which maximizes $\min_{v_i \in V} acc(v_i, p)$, is a bit more involved. Let z be the minimum value accuracy among values in V, then the objective function of the transformed optimization is simply to maximize z, and it must add a constraint for each value v_i in V to ensure the accuracy of v_i is at least z (Constraint 3 in Equation 3). Therefore, we formulate problem T_3 as

$$\begin{aligned}
\underset{\substack{z, h_{ij} \forall j \in A,\\ i=1,2,\ldots r}}{\text{maximize}} \quad & z \\
\text{subject to} \quad & \frac{\sum_{i=1}^{r}(x + (n_i - x)(\sum_{j \in A} h_{ij} f_{i,j}))}{\sum_{i=1}^{r} n_i} \geq \alpha_b \\
& \sum_{i=1}^{r}(xy + (\sum_{j \in A} j h_{ij})(n_i - x)) \leq t_b \sum_{i=1}^{r} n_i \\
& \frac{x + (n_i - x)(\sum_{j \in A} h_{ij} f_{i,j})}{n_i} \geq z \qquad \forall v_i \in V \\
& \sum_{j \in A} h_{ij} = 1 \quad \forall i = 1, 2, \ldots, r \\
& h_{ij} \in \{0, 1\} \quad \forall j \in A, \forall i = 1, 2, \ldots, r
\end{aligned} \tag{3}$$

We have described how to solve problems $T_1 - T_5$ in the cases where they involve the set of all values V. It is easy to see that these problems can be solved in a similar fashion if they involve only a subset of values $S \subseteq V$.

Using the Ranking to Adjust Worker Accuracies: Recall that for each value $v_i \in V$, we have obtained xy answers from the crowd, and have estimated the worker accuracy for v_i as a_i, the fraction of the xy answers that are correct.

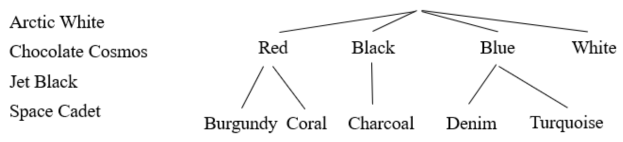
However, a_i is often not a good estimation of the true worker accuracy for v_i, because the set of xy answers is often small (e.g., $x = 4$ and $y = 5$ for 20 answers total). Thus, we seek to improve these estimations, using the ranking K. Our key idea is that if v_i is ranked higher than v_j, thus being perceived as being more difficult, then the worker accuracy for v_i should be no higher than that of v_j. If this is not the case, then we can adjust such worker accuracies so that they become more consistent with the ranking K.

Specifically, suppose K assigns to each value $v_i \in V$ a rank $k_i \in [1, r]$, where a smaller k_i indicates a value closer to the top of the ranked list. Then we model the task of improving the worker accuracies a_i's as the following optimization problem:

$$\begin{aligned}
\underset{z_1, z_2, \ldots, z_r}{\text{minimize}} \quad & \sum_{i=1}^{r}(z_i - a_i)^2 \\
\text{subject to} \quad & 0 \leq z_i \leq z_j \leq 1 \quad \forall i, j \in [1, r] \text{ s.t. } k_i < k_j
\end{aligned} \tag{4}$$

Here z_i is the improved worker accuracy for value v_i, and the cost function is the sum of the squares of $z_i - a_i$ (also known as L_2 cost function). Its constraint ensures that each value has the same or less worker accuracy than any easier value. This model is a simple Isotonic Regression problem, which always has a solution. It can be efficiently solved in $O(r)$ time [23], where r is the number of values. We solve it using Gurobi [1]. We then set $a_i = z_i$ and use a_i's as the worker accuracies in formulating ILP problems, as discussed earlier in this section.

Table 1: An example of handling ambiguous colors.



6 MANAGING AMBIGUOUS VALUES

As discussed in Section 1, in practice, there are many cases when the value for the target attribute is inherently ambiguous, such as "desert sand" and "arctic white". In such cases even the expert has trouble determining what should be the correct answer to the question, let alone asking the crowd workers. Such cases are surprisingly common, and no existing work has addressed them, as far as we can tell.

In this paper we provide a simple yet effective solution to this problem, based on what we have seen working well in industry. Briefly, we ask the expert E to first create a taxonomy Z of only unambiguous values, such as the one in Figure 1. Then the expert E examines each value v in V (the set of all values for the target attribute in the data set D). If E judges v to be inherently ambiguous, E should map v to a value $m(v)$ in Z.

Example 6.1. Table 1 shows a set of values (on the left side of the figure) that are ambiguous. The expert can map "Arctic White" to node "White" in the taxonomy, "Chocolate Cosmos" to "Burgundy", and so on.

A question such as "is the color of this product indeed chocolate cosmos?" is then transformed into "is the color of this product indeed burgundy?", which is unambiguous for crowd workers to answer.

7 EMPIRICAL EVALUATION

We now evaluate our solution. First, we crawled online sources to obtain the three datasets shown under "Datasets A" in Table 2. Their schemas are shown at the top of the table, with the target attribute underlined. Column "# Items" lists the number of data items in each dataset, and column "# Values" lists the number of values for the target attribute.

Since it would be too expensive to crowdsource all items in all datasets, we downsample all three datasets (using stratified sampling in which for each value of the target attribute, we randomly retain only 20% of the data items with that value). The new datasets are listed under "Datasets B" in the same table. Our experiments with real crowd workers are performed on these new datasets. We used Amazon Mechanical Turk for crowdsourcing, and used common turker qualifications, such as allowing only turkers with at least 100 approved HITs and 95% approval rate.

7.1 Learning to Rank

We first examine the performance of learning to rank. Recall that for each dataset, we sample x questions for each value, and then solicit y answers for each question. Thus, the expert must provide golden answers for $x|V|$ questions (where V is the set of

Table 2: Datasets for our experiments.

Products (title, description, picture, price, color)
Courses (title, description, department, #credits, subject)
Apparel (title, description, style, size, picture, category)

	Datasets A		Datasets B	
	# Items	# Values	# Items	# Values
Products	10,869	63	2,131	57
Courses	7,583	148	1,395	133
Apparel	3,480	12	690	11

Table 3: Evaluating the quality of the rankings.

	WAK			VChecker		
	Precision	Recall	F_1	Precision	Recall	F_1
Products	71.97	61.33	66.22	68.64	68.47	68.56
Courses	74.81	51.46	60.98	66.71	64.46	65.57
Apparel	76.67	63.89	69.70	72.22	72.22	72.22

Table 4: Evaluating the generated explanations.

Values	Explanations by VChecker	Explanations by Expert	# Compatible Explanations
P.Denim	2Ab,2Ac,2B,2C	2A,2C	3 {2Ab,2Ac,2C}
P.Brown	1,2A,2Ab,2B,2C	1C,2A,2Bc,2C	5 {1,2A,2Ab,2B,2C}
P.Turquoise	2A,2B,2C	2Ab,2Ac,2Bc,2C	3 {2A,2B,2C}
C.German	2A,2B,2C	2A,2Bc,2C	3 {2A,2B,2C}
C.Zoology	2A,2C	2A,2B,2C	2 {2A,2C}
C.Dance	1A,1Ab,2A,2C	1A,2C	3 {1A,1Ab,2C}
A.Tanks	2Aa,2B,2Ba,2Bb,2C	2A,2B,2C	5 {2Aa,2B,2Ba,2Bb,2C}
A.Underwear	2A,2B,2C	2Bc,2C	2 {2B,2C}
A.Socks	2A,2C	2C	1 {2C}

Table 5: VChecker vs. the UCS baseline solution.

Dataset	t_b	Cost			Accuracy	
		UCS	VChecker	Reduction	UCS	VChecker
Products	3	6,393	4,435	30.6	96.10	95.53
	5	10,655	5,961	44.1	96.89	96.03
	7	14,917	7,585	49.2	97.27	96.18
	9	19,179	8,941	53.4	97.49	96.32
Courses	3	4,185	4,063	2.9	95.94	96.08
	5	6,975	5,393	22.7	96.83	97.18
	7	9,765	6,639	32.0	97.17	97.60
	9	12,555	7,715	38.6	97.56	97.69
Apparel	3	2,070	2,016	2.6	97.60	97.62
	5	3,450	2,384	30.9	98.03	97.82
	7	4,830	2,866	40.7	98.36	97.97
	9	6,210	3,202	48.4	98.84	98.02

all values for the target attribute), and the crowd must provide $xy|V|$ answers. So it is highly desirable that we minimize these two quantities, to minimize the workload of the expert and the crowd workers.

For our current three datasets, (x, y) are $(4, 5)$, $(5, 5)$, $(5, 5)$ for Products, Courses, and Apparel, respectively. Our iterative expansion process (to find x and y) converged for Products and Courses. These results suggest that indeed VChecker spends relatively little expert and crowd effort to compute the difficulty scores.

Next, we examine the quality of the ranking K of the values that we have obtained. To do so, we need a "golden" ranking $K*$. We obtain $K*$ as follows. First, for each value v, we collect A_v, the set of all answers obtained from the crowd for all questions involving v. Since we have obtained at least 9 answers for each question, this is usually a large number (in the hundreds). Next, we have identified the correct answer for all questions in our datasets, so we can compute the worker accuracy for v as the fraction of answers in A_v that is correct. Since A_v is a large set of answers, we take this worker accuracy to be the golden worker accuracy. Finally, we sort the values in decreasing order of these golden accuracies, to obtain a golden rank $K*$ of the values, in decreasing order of difficulties.

We now compare ranking K with $K*$. Direct comparison turns out to be difficult, because we often have two values v_i and v_j such that v_i is ranked above v_j in K and the reverse applies in $K*$, yet their difficulty scores differ by less than 0.01, say. In such cases, where the difficulty scores differ by less than a small ϵ threshold, we say the two values are not comparable. We translate ranking $K*$ into a set of $S(K*)$ of all (v_i, v_j) pairs that are comparable, and translate ranking K into a similar set $S(K)$.

Table 3 compares these sets. Consider the last three cells of the first row (the cells under "VChecker"). The cell under "Precision" is 68.64%, meaning 68.64% of pairs in $S(K)$ appear in $S(K*)$. The cell under "Recall" means 68.47% of pairs in $S(K*)$ appear in $S(K)$. These two numbers produce a F_1 score of 68.56%. Thus, for Products, the ranking K approximates the golden rank $K*$ quite well, with high precision and recall (though there is still room for improvement). Similar results are shown for Courses and Apparel.

Recall that the current popular solution in industry uses the average worker accuracy to rank the data values. Table 3 shows that the ranking produced by this solution is worse than the VChecker ranking (see the first three columns of the table, under "WAK", shorthand for "worker accuracy-based ranking", which show lower F_1 values). This result suggests that VChecker is indeed able to exploit additional information such as the response time and the worker disagreement to obtain a better ranking of value difficulties than the current existing solution.

7.2 Generating Explanations

To evaluate our explanation generator, for each dataset we select 3 values in the top part of the ranking K, then ask for their explanations. For comparison purposes, we also ask a domain expert to manually generate explanations, after carefully examining all the answers solicited from the crowd.

Table 4 lists the explanations for VChecker vs those generated by the experts. "2Ab" for example is the explanation at node "2.A.b" in the taxonomy of explanations in Figure 3 ("v is indeed difficult because it is unfamiliar"). The table shows that the two sets of explanations share large overlaps (see the last column of the table), suggesting that VChecker is effective in generating explanations to explain why a value is considered difficult.

7.3 Finding Good Crowdsourcing Plans

We now show that VChecker can find good crowdsourcing plans for a variety of problem types. Table 5 compares VChecker to the baseline plan of soliciting the same number t_b of answers for each data value. We call this plan UCS, shorthand for "uniform crowdsourcing".

To explain, consider the third row of this table. It shows that for dataset Products, if UCS solicits $t_b = 3$ answers per question, then it incurs a total crowdsourcing cost of 6,393 answers. If

Table 6: VChecker vs UCS in solving problem T_3.

Dataset	Min Value Accuracy		Avg Value Accuracy	
	UCS	VChecker	UCS	VChecker
Products	68.45	83.33	92.34	96.11
Courses	72.45	74.68	96.11	96.81
Apparel	92.23	95.58	97.46	98.30

we solve the CS problem T_1 (as described in Section 5.1; we experiment with other CS problem types below) to find a better CS plan, which would minimize this cost while keeping accuracy at least equal or better than that of UCS, then the cost of this new plan (listed under column "VChecker") is 4,435. This produces a reduction of 30.6% in cost. The last two cells of this row show that the accuracies of UCS and VChecker are comparable (96.1 vs 95.53) [1]. (We obtained these accuracy numbers by executing both plans on Amazon Mechanical Turk.) Subsequent rows are similar, but for different values of t_b.

The table shows that VChecker can significantly reduce the cost of the baseline solution UCS, by 22.7-53.4% in all cases, except two cases where the reduction is a more modest 2.6% and 2.9%. It also shows that the accuracy of VChecker is comparable to that of UCS (with the difference in the range [-1.17%, 0.43%]).

Solving Other Types of CS Problems: Earlier we have shown how VChecker solves CS problems of type T_1. We now show that VChecker is effective in helping users solve other types of CS problems.

In Section 5.1 we discuss problem T_3, where the user wants to improve the accuracies of the values *across the board* as much as possible, while keeping the overall accuracy at least as high as that of the baseline plan p_b and keeping the overall cost at most as high as that of p_b. The goal is to return the plan p that maximizes $min_{v_i \in V} acc(v_i, p)$, subject to $acc(V, p) \geq acc(V, p_b)$ and $cost(V, p) \leq cost(V, p_b)$.

Table 6 shows how well VChecker performs for this problem. The column "UCS" shows the minimal value accuracy (i.e., the lowest accuracy among those of all values) when it solicits 3 answers for each question. The column "VChecker" shows that VChecker is able to improve this minimal accuracy significantly, while keeping the cost no higher than the cost of UCS. The last two columns show that even the average value accuracy (i.e., averaged over all values) of VChecker is higher than that of UCS.

In Section 5.1 we also discuss the problem of maximizing the accuracy of k most difficult values, as taken from the ranking K. Table 7 shows that VChecker is effective for solving this problem, improving the accuracy of the top 5 most difficult values per dataset significantly.

7.4 Additional Experiments

Sensitivity Analysis: In the current VChecker system we set $x_{max} = 5$ and $y_{max} = 5$, meaning that the iterative exploration process (see Section 3.2) never goes beyond these values. Figure 4 shows how iterative exploration is sensitive to varying these values. It shows that this process converges between 3 and 6 for all three datasets, suggesting that setting the values to 5 is a reasonable choice.

[1]When solving the ILP problem, we specified the constraint that VChecker has the same or better accuracy than UCS. When executing the found plan on Mechanical Turk, however, this constraint may not hold, due to spammers, careless workers, etc. Nevertheless, our experiments show that the accuracies of VChecker and UCS differ by a very small range.

Table 7: Maximizing accuracy of 5 most difficult values.

Dataset	Values	Value Accuracy		
		UCS	VChecker	% Improved
Products	Coral	68.45	83.33	14.88
	Denim	81.07	100.00	18.93
	Taupe	76.96	100.00	23.04
	Brown	95.55	97.92	2.37
	Camel	98.52	100.00	1.48
Courses	La Follette School of Public Affairs (PUB AFFR)	78.04	87.50	9.46
	Agronomy (AGRONOMY)	96.13	100.00	3.87
	Geological Engineering (G L E)	93.45	100.00	6.55
	Civil and Environmental Engineering (CIV ENGR)	97.11	100.00	2.89
	German (GERMAN)	87.22	96.90	9.68
Apparel	Underwear	98.43	100.00	1.57
	Tanks	92.23	100.00	7.77
	Pants	94.72	96.85	2.13
	Socks	96.87	96.90	0.03
	Swimwear	99.34	97.35	-1.99

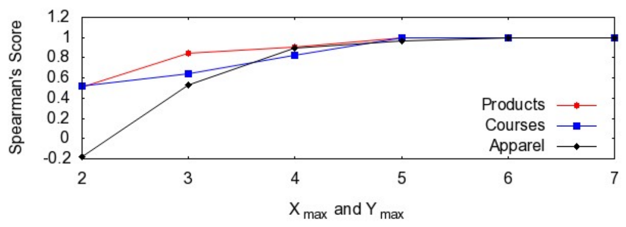


Figure 4: Convergence in iterative exploration.

Managing Ambiguous Values: Finally, we briefly discuss examples of managing ambiguous values. In our experiments it turns out that Products has ambiguous values. Specifically, it has a total of 173 values, 110 of which are considered ambiguous and have to be mapped to 63 values in a taxonomy of unambiguous values. Examples of such mappings include Arctic White mapped to White, Fluorescent Orange mapped to Orange Red, and Saddle Brown mapped to Brown. This clearly suggests that managing such ambiguous values is critical in real-world verification of attribute values.

8 RELATED WORK

Data cleaning has received enormous attention (e.g., [2–4, 6, 9, 11, 12, 16, 32–37, 40–45, 50]). See [5, 7, 10, 46] for recent tutorials, surveys, and books. However, as far as we can tell, no published work has examined the problem of manually detecting data errors in categorical attributes, as we do in this paper.

In recent years, crowdsourcing (CS) has received significant attention and has also been applied to many data cleaning problems (e.g., [8, 14, 15, 20, 22, 25, 28, 30, 31, 47, 49]). Among these, the work [14] also discusses the idea of adapting crowdsourcing strategies to the difficulties of data regions. However, it considers this idea in the setting of crowdsourcing for active learning. Further, it does not consider learning to rank the data regions, nor debugging the ranking, as we do this paper.

A critical challenge in CS is that the quality of workers varies. Researchers have proposed many methods to differentiate workers, such as filtering out spammers [25, 47], measuring the reliability and quality of workers [19, 22, 27, 39], and finding the right group of workers for a given task [17]. These methods usually assume that all the questions are of the same difficulty. In contrast, VChecker utilizes the difficulty heterogeneity among the questions while assuming that all workers have the same quality.

VChecker uses majority voting to aggregate the collected answers of each question. Many other aggregation methods have been proposed [19, 22, 27, 39]. They usually assign higher weights to answers from workers with good quality, then perform weighted aggregation. Many build probabilistic models [22, 39] to iteratively update the estimation of worker quality and weights. However, as far as we can tell, there is no published work yet showing conclusive evidence that these methods can achieve higher accuracy than majority voting, especially when we can only collect a small number of answers per question due to limited budget, as in our setting here.

Researchers also propose other methods to reduce CS cost, e.g., early-stopping strategies [13, 21, 29]. They stop collecting more answers for a question when they realize that collecting more answers will not change the aggregated answer. Such methods can also be used in VChecker to further reduce our cost, when we use the best plan returned by VChecker to crowdsource all the questions.

Finally, most CS works only collect answers from the crowd. [26] also collects the self-reported confidence from workers to improve the accuracy of aggregated answers. However, they also notice that workers have the tendency to overestimate or underestimate their confidence. Recently [18] proposes to collect the time spent by workers to measure CS effort. VChecker also collects the response times, but use these (and other data) to estimate question difficulty.

9 CONCLUSIONS

Detecting data errors completely manually is a ubiquitous problem in data cleaning, yet it has not received much attention. In this paper we have shown that the current common solution of crowdsourcing the above problem using the same number of answers per question can be improved by detecting the difficulties of data regions, then adjusting the number of answers required for each region based on its difficulty. We showed that current work using this idea has several significant limitations. We proposed VChecker, a novel solution to address these limitations, and described extensive experiments with three real-world data sets that demonstrate the promise of our solution. For future work, we plan to improve VChecker in multiple ways, including developing solutions to partition the input data into regions, better solutions to estimate and rank the data regions' difficulties, and better solutions to generate explanations for domain experts.

REFERENCES

[1] [n.d.]. Gurobi. http://www.gurobi.com/.
[2] Z. Abedjan et al. 2016. Detecting Data Errors: Where are we and what needs to be done? *PVLDB* 9, 12 (2016), 993–1004.
[3] A. Arasu et al. 2011. Towards a Domain Independent Platform for Data Cleaning. *IEEE Data Eng. Bull.* 34, 3 (2011), 43–50.
[4] S. Chaudhuri et al. 2006. Data Debugger: An Operator-Centric Approach for Data Quality Solutions. *IEEE Data Eng. Bull.* 29, 2 (2006), 60–66.
[5] Xu Chu et al. 2016. Data Cleaning: Overview and Emerging Challenges. In *SIGMOD.*
[6] X. Chu et al. 2016. Distributed Data Deduplication. In *VLDB.*
[7] X. Chu and I. F. Ilyas. 2016. Qualitative Data Cleaning. *PVLDB* 9, 13 (2016).
[8] S. Das et al. 2017. Falcon: Scaling Up Hands-Off Crowdsourced Entity Matching to Build Cloud Services. In *SIGMOD.*
[9] A. Das Sarma et al. 2012. An automatic blocking mechanism for large-scale de-duplication tasks. In *CIKM.*
[10] T. Dasu and T. Johnson. 2003. *Exploratory Data Mining and Data Cleaning.* John Wiley.
[11] X. Dong et al. 2010. Global Detection of Complex Copying Relationships Between Sources. *PVLDB* 3, 1 (2010), 1358–1369.
[12] V. Efthymiou et al. 2015. Parallel Meta-blocking: Realizing Scalable Entity Resolution over Large, Heterogeneous Data. In *Big Data.*
[13] Aditya G. Parameswaran et al. 2012. CrowdScreen: algorithms for filtering data with humans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012.* 361–372.
[14] B. Mozafari et al. 2014. Scaling Up Crowd-Sourcing to Very Large Datasets: A Case for Active Learning. *PVLDB* 8, 2 (2014), 125–136.
[15] C. Gokhale et al. 2014. Corleone: hands-off crowdsourcing for entity matching. In *SIGMOD.* 601–612.
[16] D. Haas et al. 2015. Wisteria: Nurturing Scalable Data Cleaning Infrastructure. *PVLDB* 8, 12 (2015), 2004–2007.
[17] H. Li et al. 2014. The wisdom of minority: discovering and targeting the right group of workers for crowdsourcing. In *WWW.* 165–176.
[18] J. Cheng et al. 2015. Measuring Crowdsourcing Effort with Error-Time Curves. In *ACM CHI.* 1365–1374.
[19] J. Whitehill et al. 2009. Whose vote should count more: Optimal integration of labels from labelers of unknown expertise. In *Advances in neural information processing systems.* 2035–2043.
[20] J. Wang et al. 2012. CrowdER: Crowdsourcing Entity Resolution. *PVLDB* 5, 11 (2012), 1483–1494.
[21] L. Mo et al. 2013. *Optimizing task assignment for crowdsourcing environments.* Technical Report. Citeseer.
[22] P. Ipeirotis et al. 2010. Quality management on amazon mechanical turk. In *ACM SIGKDD workshop on human computation.* ACM, 64–67.
[23] P. Mair et al. 2009. Isotone optimization in R: pool-adjacent-violators algorithm (PAVA) and active set methods. *Journal of statistical software* 32, 5 (2009), 1–24.
[24] Q. Deng et al. 2015. Deep learning for gender recognition. In *ICCCS.* 206–209.
[25] S. Jagabathula et al. 2014. Reputation-based Worker Filtering in Crowdsourcing. In *Advances in Neural Information Processing Systems 27.* 2492–2500.
[26] S. Oyama et al. 2013. Accurate Integration of Crowdsourced Labels Using Workers' Self-reported Confidence Scores. In *IJCAI.*
[27] V. Raykar et al. 2009. Supervised learning from multiple experts: whom to trust when everyone lies a bit. In *ICML.* 889–896.
[28] X. Chu et al. 2015. KATARA: A Data Cleaning System Powered by Knowledge Bases and Crowdsourcing. In *SIGMOD.* 1247–1261.
[29] X. Liu et al. 2012. CDAS: A Crowdsourcing Data Analytics System. *PVLDB* 5, 10 (2012), 1040–1051.
[30] Y. Amsterdamer et al. 2013. CrowdMiner: Mining association rules from the crowd. *PVLDB* 6, 12 (2013), 1250–1253.
[31] Y. Tong et al. 2014. CrowdCleaner: Data cleaning for multi-version data on the web via crowdsourcing. In *IEEE.* 1182–1185.
[32] Z. Khayyat et al. 2015. BigDansing: A System for Big Data Cleansing. In *SIGMOD.* 1215–1230.
[33] M. J. Franklin et al. 2011. CrowdDB: answering queries with crowdsourcing. In *SIGMOD.*
[34] J. Freire et al. 2016. Exploring What not to Clean in Urban Data: A Study Using New York City Taxi Trips. *IEEE Data Eng. Bull.* 39, 2 (2016), 63–77.
[35] Helena Galhardas et al. 2001. Declarative Data Cleaning: Language, Model, and Algorithms. In *VLDB.*
[36] D. Haas et al. 2016. CLAMShell: Speeding up Crowds for Low-latency Data Labeling. In *VLDB.*
[37] J. Heer et al. 2015. Predictive Interaction for Data Transformation. In *CIDR.*
[38] T. Joachims. 2002. Optimizing search engines using clickthrough data. In *SIGKDD.* 133–142.
[39] A. Khan and H. Garcia-Molina. 2017. CrowdDQS: Dynamic Question Selection in Crowdsourcing Systems. In *SIGMOD.*
[40] L. Kolb et al. 2011. Parallel Sorted Neighborhood Blocking with MapReduce. In *BTW.*
[41] S. Krishnan et al. 2016. ActiveClean: Interactive Data Cleaning For Statistical Modeling. *PVLDB* 9, 12 (2016).
[42] A. Marcus et al. 2011. Crowdsourced databases: Query processing with people. In *CIDR.*
[43] B. Mozafari et al. 2014. Scaling Up Crowd-Sourcing to Very Large Datasets: A Case for Active Learning. In *VLDB.*
[44] A. G. Parameswaran and N. Polyzotis. 2011. Answering Queries using Humans, Algorithms and Databases. In *CIDR.*
[45] H. Park and J. Widom. 2013. Query Optimization over Crowdsourced Data. In *VLDB.*
[46] E. Rahm and H. H. Do. 2000. Data Cleaning: Problems and Current Approaches. *IEEE Data Eng. Bull.* 23, 4 (2000).
[47] V. Raykar and S. Yu. 2012. Eliminating Spammers and Ranking Annotators for Crowdsourced Labeling Tasks. *Journal of Machine Learning Research* 13 (2012), 491–518.
[48] C. Spearman. 1987. The Proof and Measurement of Association between Two Things. *The American Journal of Psychology* 100, 3/4 (1987), 441–471.
[49] V. Verroios et al. 2017. Waldo: An Adaptive Human Interface for Crowd Entity Resolution. In *SIGMOD.*
[50] J. Wang et al. 2013. Leveraging Transitive Relations for Crowdsourced Joins. In *SIGMOD.*

Publishing Video Data with Indistinguishable Objects

Han Wang
Illinois Institute of Technology
hwang185@hawk.iit.edu

Yuan Hong
Illinois Institute of Technology
yuan.hong@iit.edu

Yu Kong
Rochester Institute of Technology
yu.kong@rit.edu

Jaideep Vaidya
Rutgers University
jsvaidya@rbs.rutgers.edu

ABSTRACT

Millions of videos are ubiquitously generated and shared everyday. Releasing videos would be greatly beneficial to social interactions and the community but may result in severe privacy concerns. To the best of our knowledge, most of the existing privacy preserving techniques for video data focus on detecting and blurring the sensitive regions in the video. Such simple privacy models have two major limitations: (1) they cannot quantify and bound the privacy risks, and (2) they cannot address the inferences drawn from the background knowledge on the involved objects in the videos. In this paper, we first define a novel privacy notion *ϵ-Object Indistinguishability* for all the predefined sensitive objects (e.g., humans and vehicles) in the video, and then propose a video sanitization technique VERRO that randomly generates utility-driven synthetic videos with indistinguishable objects. Therefore, all the objects can be well protected in the generated utility-driven synthetic videos which can be disclosed to any untrusted video recipient. We have conducted extensive experiments on three real videos captured for pedestrians on the streets. The experimental results demonstrate that the generated synthetic videos lie close to the original video for retaining good utility while ensuring rigorous privacy guarantee.

1 INTRODUCTION

Millions of videos are ubiquitously generated and shared everyday via video surveillance devices, traffic cameras, smart phones, among others. Sharing such video data would greatly benefit human interactions and the community. For instance, surveillance cameras in buildings capture possible threats to the corporate assets (such videos are shared for analysis in many cases [44]). Traffic videos contribute to monitoring the street traffic and traffic analysis applications such as vehicle counting and traffic flow analysis [3] as well as pedestrian behavior analysis.

However, these scenarios have often raised severe privacy concerns since human faces, bodies, identities, activities and other sensitive information can be recorded in such videos [8, 44]. Thousands of vehicles are involved in a traffic monitoring video, and drivers may not be willing to share their vehicle plate, make, model, locations and trajectories [2]. In addition, video surveillance systems monitor specific areas of interests with the following goals: law enforcement, personal safety, resource planning, and security of assets [44]. While ensuring safety and deterrence, it may also compromise the privacy of innocent individuals. Thus, privacy preserving solutions for videos have attracted significant interests recently.

To the best of our knowledge, most of the existing privacy preserving techniques for addressing the privacy concerns in video data (e.g., [1, 6, 20, 24, 42]) focus on detecting and blurring the sensitive regions in the videos (e.g., faces and bodies). Such simple privacy models have two major limitations: (1) they cannot quantify the bound of privacy risks, and (2) they cannot address the inferences drawn from the background knowledge on the involved objects. For instance, the video recipient may have known that an individual lives near the scene, and he/she usually wears red clothes as well as likes running at a specific side of the street. In this case, even if all the humans can be detected and blurred before video disclosure, they can be readily re-identified by the adversary with the above background knowledge.

ϵ-Object Indistiguishability. To tackle such critical limitations, we define a novel privacy notion for protecting the objects (and the corresponding individuals) in the videos – "*ϵ-Object Indistinguishability*", which is extended from the emerging *differential privacy* in local setting [4, 10, 16]. Specifically, in the past decade, the notion of differential privacy has emerged essentially as the de facto privacy standard for bounding the privacy risks while sanitizing different data [7, 13, 25]. Adversaries cannot infer if a certain individual is included in the input or not from the noisy aggregated result (perturbed by a trusted aggregator) regardless of their background knowledge [13]. More recently, local differential privacy (LDP) models have been proposed to privately perturb data by each individual such that the collected (random) data from different individuals can be *indistinguishable.* Inspired by the LDP models, our privacy notion also ensures *indistinguishability* for all the objects in the randomized output video, and thus the perturbed video can be safe to be disclosed to any untrusted video recipient.

Recall that videos differ from many other data (e.g., statistical databases [13], location data [32], and search logs [26]). A local video may include numerous objects corresponding to multiple different individuals, e.g., many pedestrians are recorded in a single video, and many vehicles are recorded in the same video. A video includes the "local data" of many individuals (e.g., humans) which will be shared to the untrusted recipients via the video owner. Thus, the primary difference between *ϵ-Object Indistinguishability* and the original definition of LDP [4, 10, 16] is that the video owner locally perturbs data for all the objects rather than letting the objects execute perturbation (see Figure 1).

Contributions. With the privacy notion of *ϵ-Object Indistinguishability*, we propose a video sanitization technique that randomly generates a synthetic video by the video owner (e.g., the agency which captures the video) while ensuring *ϵ-Object Indistinguishability* and good utility.

Specifically, we design a novel random response scheme (by optimizing the RAPPOR [16]) that randomly generates different objects in the video by maximizing the utility of random response

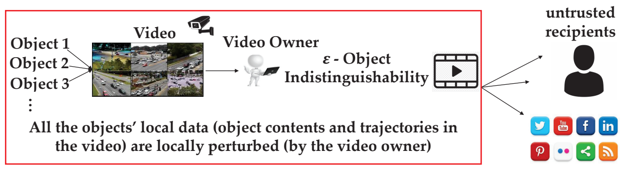

Object 1
Object 2
Object 3
Video
Video Owner
ε - Object Indistinguishability
untrusted recipients
All the objects' local data (object contents and trajectories in the video) are locally perturbed (by the video owner)

Figure 1: VERRO: Ensuring *Object Indistinguishability* in the Video Data Sanitization

[16] applied to the objects. Thus, we name our proposed technique as *"Video with Randomly Responded Objects* (VERRO)". As a result, we boost the utility of VERRO in two folds: (1) for each object, optimizing its random response in different frames, and (2) interpolating the trajectories of objects in the video [17] (without additional privacy leakage [14], see Section 4). Thus, the synthetic video can be disclosed to any untrusted recipient. Finally, we summarize the major contributions of this paper as below:

- To the best of our knowledge, we define the first rigorous privacy notion for all the sensitive objects (predefined by the video owner) in the video data, which ensures that all the objects are *indistinguishable* in the randomized output video against arbitrary background knowledge.
- We propose a novel video sanitization technique VERRO that randomly generates utility-driven synthetic videos in which any two sensitive objects are ϵ-*Object Indistinguishable.*[1] The video owner can also specify its privacy budget ϵ for all the objects in its video.
- The proposed novel random response scheme (by optimizing RAPPOR [16]) in VERRO that optimally picks frames of the video to randomly generate objects (while satisfying indistinguishability). The utility of the synthetic video is further improved using computer vision techniques.
- We have conducted extensive experiments on real videos to validate the performance of VERRO. The experimental results demonstrate that VERRO can effectively generate private synthetic videos with high utility.

The remainder of this paper is organized as follows. Section 2 introduces some preliminaries. Section 3 illustrates the first phase of VERRO and analyzes the privacy guarantee. Section 4 presents the second phase of VERRO (for further boosting the utility) and its privacy guarantee. Section 5 gives discussions for VERRO. Section 6 demonstrates the experimental results. Section 7 and 8 present the literature and conclusions.

2 PROBLEM FORMULATION

In this section, we first describe the adversary model, then define our privacy notion, and finally provide a general overview of our proposed approach.

2.1 Adversary Model

Denote a video as $\mathcal{V}$ which is captured by a video owner, e.g., a hospital or a company equipped with CCTV surveillance, and an agency which captures the video on the street. Video $\mathcal{V}$ (all the frames) includes a set of n sensitive objects $\mathbb{O} = \{O_1, O_2, \cdots, O_n\}$ (e.g., humans and vehicles). Assume that the video owner would like to share $\mathcal{V}$ to an external party for analysis (viz. the adversary). To ensure privacy, our proposed VERRO (randomly) generates a synthetic video $\mathcal{V}^*$ which is close to $\mathcal{V}$, such that:

- Each sensitive object in all the frames satisfies ϵ-*Object Indistinguishability* – the adversary cannot distinguish any two objects from the output synthetic video $\mathcal{V}^*$ with arbitrary background knowledge.
- The synthetic video $\mathcal{V}^*$ retains good utility (close to $\mathcal{V}$).

In VERRO, we assume that the adversaries can possess arbitrary background knowledge on each object (e.g., object contents, trajectories, at-scene times, and gathering groups of objects). To retain the output utility, VERRO does not change the background scene(s), but the privacy model can break the linkage between each object and the background scene(s) via *indistinguishability.*

With privacy guarantee for all the objects (making them indistinguishable), VERRO regularly generates synthetic videos for videos including sensitive objects w.r.t. multiple individuals (e.g., pedestrians and vehicles). In case that a video includes only one sensitive object, the adversary still cannot re-identify the object (see Section 5). In addition, VERRO only addresses the visual privacy concerns, assuming that the adversary cannot identify objects from the audio or audio is not captured (e.g., traffic monitoring and video surveillance).

2.2 Privacy Notion

2.2.1 Traditional Privacy Model. Video $\mathcal{V}$ includes multiple sensitive objects $O_1, \ldots, O_n$, which can be detected and tracked in all the frames [48, 49]. Specifically, it first detects all the sensitive objects in each frame with the detection algorithms (e.g., HOG for human [51] and SVM for vehicles [22]). Each detected object can be accurately tracked with the same ID if they highly overlap in multiple frames.

The traditional privacy models are defined to blur all the detected objects [1, 6, 20, 24, 42]. An alternative solution could be replacing the detected objects with "synthetic objects" [28, 43]. Each object can be replaced by a unique synthetic object: for instance, a red synthetic human and a purple synthetic human can be used to represent two different pedestrians in all the frames involving them. Then, the inferences and re-identification visually from the objects can be greatly mitigated.

2.2.2 ϵ-Object Indistinguishability for Sensitive Objects. Recall that only replacing the objects with synthetic objects in the video cannot address the re-identification based on the adversaries' background knowledge (as discussed in Section 1). Thus, we need to ensure *indistinguishability* for not only objects themselves (can be achieved by synthetic objects) but also their moving trajectories [50] in the video.

To this end, inspired from the indistinguishability provided by the ϵ-LDP, we define a novel privacy notion ϵ-*Object Indistinguishability* by considering each object's trajectory in the video (coordinates at different frames) as its "local data". Specifically, in the standard LDP definition [4, 10, 16], there are a set of users, each of which has its own data. After each user locally perturbs its data, the obfuscated output can be directly disclosed to any untrusted recipient/aggregator, where the randomized data collected from any two different users are indistinguishable [10, 16]. Migrating the LDP model to the objects in any video $\mathcal{V}$, we define the ϵ-*Object Indistinguishability* as below:

Definition 2.1 (ϵ-Object Indistinguishability). A randomization algorithm $\mathcal{A}$ satisfies ϵ-*Object Indistinguishability*, if and only if

[1]As formally defined in Definition 2.1, ϵ-*Object Indistinguishability* ensures a similar privacy guarantee as ϵ-*local differential privacy* [4, 10, 16].

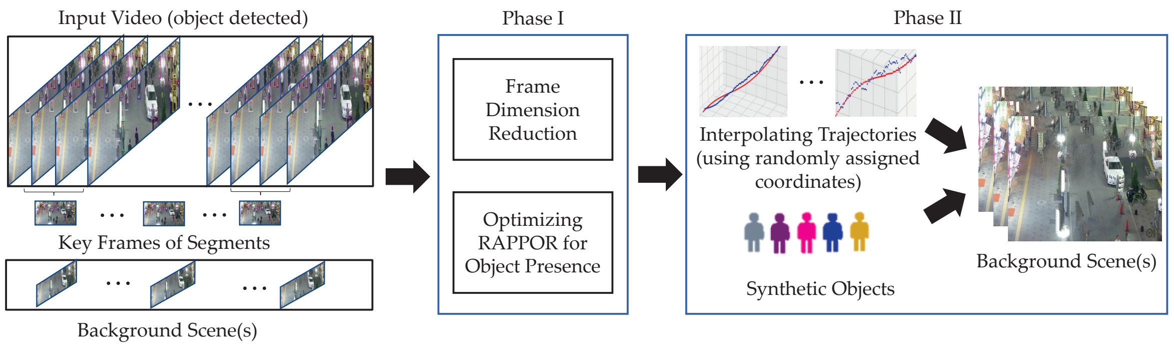

Input Video (object detected)
Key Frames of Segments
Background Scene(s)
Phase I
Frame Dimension Reduction
Optimizing RAPPOR for Object Presence
Phase II
Interpolating Trajectories (using randomly assigned coordinates)
Synthetic Objects
Background Scene(s)

Figure 2: VERRO for Utility-Driven Synthetic Video Generation with Object Indistinguishability

for any two input objects $O_i, O_j \in \mathbb{O}$ in the input video $\mathcal{V}$, and for any output object of $\mathcal{A}$ in the synthetic video $\mathcal{V}^*$ (denoted as y), we have $Pr[\mathcal{A}(O_i) = y] \le e^{\epsilon} \cdot Pr[\mathcal{A}(O_j) = y]$.

Similar to ϵ-LDP [16], *ϵ-Object Indistinguishability* also focuses on the indistinguishability of randomizing any two objects, rather than the indistinguishability of randomizing any two neighboring inputs (whether any object is included or not included in the input) in traditional differential privacy [13]. Privacy budget ϵ decides the degree of indistinguishability (identical to LDP [16]).

Definition 2.1 guarantees that the randomly perturbed output of any two objects in $\mathcal{V}$ (both the object contents and the trajectories in all the frames) are ϵ-indistinguishable in $\mathcal{V}^*$. It also ensures *plausible deniability* for every object [5]. Since *ϵ-Object Indistinguishability* also requires all the objects to be visually indistinguishable (object contents), VERRO randomly assigns synthetic objects (e.g., the same shape but different colors) to replace the original distinct objects while generating the synthetic video $\mathcal{V}^*$. The synthetic objects are generated and placed by considering the distance of the object to the camera (e.g., the synthetic object size is larger if getting closer to the camera) [31].

2.3 VERRO Framework

The major components of VERRO (see Figure 2) consist of:

(1) **Preprocesssing**: all the objects are detected and tracked, and background scene (for each frame) is extracted using computer vision techniques [11, 48, 49].
(2) **Phase I**: for each object, its presence or absence in different frames/segments of the video are randomly generated (by random response) to be indistinguishable. Before executing random response, VERRO reduces the frame dimension in the video by detecting the key frames. Then, the utility can be improved by allocating optimal budgets for different dimensions. Furthermore, we also formulate a utility maximizing random response problem (optimizing RAPPOR [16]) to retain the optimal object presence information after Phase I. Note that Phase I satisfies *ϵ-Object Indistinguishability*: all the objects' presence in all the frames are indistinguishable (see details in Section 3).
(3) **Phase II**: with the randomly generated presence/absence information for each object, VERRO generates the synthetic video by inserting the synthetic objects into the video (background scene(s)). Specifically, the coordinates (where to insert the synthetic objects) are assigned, and computer vision techniques are applied to interpolate object moving trajectories between two assigned coordinates in the synthetic video. We also shown that Phase II does not leak any additional information (as a post-processing step [14]), and then VERRO satisfies *ϵ-Object Indistinguishability* (see details in Section 4).

3 PHASE I: OPTIMAL OBJECT PRESENCE

As the "local data" of each object (e.g., a pedestrian or vehicle) in the video $\mathcal{V}$, the object trajectory includes its presence or absence information in each frame and the coordinates in the frame (if present). In this section, we illustrate the Phase I of VERRO that first generates indistinguishable object presence.

3.1 Poor Utility with Random Response

We first define a bit vector for each object to indicate if such object is included in different frames or not:

Definition 3.1 (Object Presence Vector). Given video $\mathcal{V}$ which includes m different frames $F_1, \dots, F_m$ and n distinct objects $\mathbb{O} = \{O_1, ..., O_n\}$, whether each object $O_i, i \in [1, n]$ is present in frame $F_k, k \in [1, m]$ or not (all m frames) can form a bit vector: $B_i = (b_i^1, .., b_i^m) \in \{0, 1\}^m$ for object O_i.

It has been proven that a classic randomized response (RR) technique (e.g., RAPPOR [4, 16]) can be adapted to ensure ϵ-LDP for locally randomizing bit vectors. Similarly, a naive solution of ensuring *ϵ-Object Indistinguishability* for the object presence vectors is to directly the random response mechanism (*we will discuss how to optimize the utility in Section 3.2 and 3.3*). For each object $O_i \in \mathbb{O}, i \in [1, n]$, if object O_i exists in frame F_k, we set $b_i^k = 1, k \in [1, m]$. Otherwise, $b_i^k = 0$ holds in the vector B_i. Then, we flip one bit in vector $B_i, i \in [1, n]$ with a certain probability to report the true value. Then, all the perturbed bits in the object presence vector B_i can be combined as the output object presence vector for object O_i. Thus, the vectors $B_1, \dots, B_n$ (of all the objects) can be indistinguishable. Algorithm 1 shows the details of directly applying random response for object presence.

Algorithm 1 Random Response for Object Presence [16]

```
1: detect all the objects $\mathbb{O} = \{O_1, \dots, O_n\}$ in $\mathcal{V}$
2: for each $O_i, i \in [1, n]$ do
3:     collect the object presence vector $B_i = (b_i^1, .., b_i^m)$ in $\mathcal{V}$
4:     for each frame $F_k, k \in [1, m]$ do
5:         equally allocate budget $\epsilon/m$ to frame $F_k$
6:         random response for bit $b_i^k$ with the probability $\frac{e^{\epsilon/m}}{1+e^{\epsilon/m}}$
7:     end for
8:     $B_i \leftarrow (b_i^1, \dots, b_i^m)$
9: end for
10: Return $\forall i \in [1, n]$, $B_i$
```

Theorem 3.2. *Algorithm 1 randomly generates object presence vectors for objects with ϵ-Object Indistinguishability.*

Proof. *ϵ-Object Indistinguishability* can be proven by following the proof of ϵ-LDP with random response [16]. Given the object presence vectors $B_i = \{b_i^1, \dots, b_i^m\}$ and $B_j = \{b_j^1, \dots, b_j^m\}$ of any two objects $O_i, O_j \in \mathbb{O}$, for any possible output m-bit vector $y = (y^1, \dots, y^m)$, we have:

$$\frac{Pr[\mathcal{A}(B_i) = y]}{Pr[\mathcal{A}(B_j) = y]} = \frac{Pr(b_i^1 = y^1)}{Pr(b_j^1 = y^1)} \cdots \frac{Pr(b_i^m = y^m)}{Pr(b_j^m = y^m)} \tag{1}$$

Since each bit is allocated with an equal privacy budget ϵ/m, the flipping probability would be $\frac{e^{\epsilon/m}}{1+e^{\epsilon/m}}$ [16]. For $k \in [1, m]$, if $b_i^k = b_j^k$ (either 0 or 1), then $\frac{Pr(b_i^k = y^k)}{Pr(b_j^k = y^k)}$ always equals 1. If $b_i^k \neq b_j^k$ and $b_i^k = y_k$, thus we have:

$$\frac{Pr(b_i^k = y^k)}{Pr(b_j^k = y^k)} = \frac{e^{\frac{\epsilon}{m}}}{1 + e^{\frac{\epsilon}{m}}} \cdot (1 + e^{\frac{\epsilon}{m}}) = e^{\frac{\epsilon}{m}} \tag{2}$$

Similarly, if $b_i^k \neq b_j^k$ and $b_j^k = y_k$, we have $\frac{Pr(b_i^k = y^k)}{Pr(b_j^k = y^k)} = e^{-\epsilon/m}$. Then, we have $\forall k \in [1, m]$, $\frac{Pr(b_i^k = y^k)}{Pr(b_j^k = y^k)} \leq e^{\epsilon/m}$ (equals one of 1, $e^{\epsilon/m}$ and $e^{-\epsilon/m}$). Combining all m bits, we have:

$$\frac{Pr[\mathcal{A}(B_i) = y]}{Pr[\mathcal{A}(B_j) = y]} \leq e^{\epsilon} \tag{3}$$

Thus, the generated presence bit vectors satisfy *ϵ-Object Indistinguishability*. This completes the proof. □

Poor Utility. Although Algorithm 1 satisfies *ϵ-Object Indistinguishability*, the utility of synthetic video would be extremely low since the total number of frames in a video m can be thousands or more, and then the allocated budget for each frame would be negligible. It destroys the utility of random response (i.e., RAPPOR [16]). For instance, a vehicle occurs in 100 frames out of a 1000-frame video, then the privacy budget for each frame is $\epsilon/1000$, which makes the flipping probability close to 0.5. Then, each of the 1000 frames would have 50% probability to include the vehicle (and other vehicles), then the objects in the video are too random (extremely low utility at this time). Thus, we explore an alternative solution for the video data in Section 3.2 and 3.3.

3.2 Dimension Reduction in the Video

Recall that the limited utility in Algorithm 1 results from the high dimensions in the video (considering each frame as a dimension). Most existing LDP techniques (e.g., RAPPOR [16], LDPMiner [40], and PLDP [9]) have reduced the dimension (e.g., bloom filter reduces the bits dimension for RAPPOR [16], top k frequent items reduces the dimension of items in LDPMiner [40], and Johnson-Lindenstrauss transform reduces the dimension of location data [9]). In videos, since difference between two consecutive frames is very small, we extract the key frames [12, 19, 30] out of m frames from $\mathcal{V}$ to reduce dimension in VERRO.

3.2.1 Key Frame Extration.

In computer vision, many existing key frame extraction algorithms have been proposed based on the boundary method [19], motion analysis [12], clustering [30], among others. Since algorithms based on clustering has been shown to generate more accurate results [30], we integrate it into VERRO for dimension reduction. The basic idea is to divide the video into several groups of similar frames.

The algorithm [39] first transforms each pixel RGB value to construct the HSV (hue, saturation, value) histogram for each frame, and then calculates the pixel distribution in terms of hue, saturation, value, respectively. Each cluster is initialized with a new frame, and expanded by adding new consecutive frames which are similar to the existing frames (measured by the HSV histograms). After the clustering, each cluster includes a group of consecutive frames, which can be considered as a segment of the video. Finally, a key frame can be extracted from each cluster/segment. The details are illustrated in Algorithm 2.

As a result, the key frame can be utilized to represent every segment. Then, the m-bit object presence vectors (for all the objects) can be reduced to ℓ-bit vectors. For instance, key frames $\mathcal{F}_1, \dots, \mathcal{F}_\ell$ (where ℓ denotes the number of key frames, and $\ell \ll m$ in general) are extracted from $\mathcal{V}$. Object O_i's presence vector B_i can be reduced to $B_i' = (kb_i^1, \cdots, kb_i^\ell)$.

Algorithm 2 Segmentation and Key Frame Extraction

```
1:  initialize the first segment $S_1 = F_1$, segment index $i = 1$
2:  equally partition $H, S, V$ value ranges to $h$, $s$ and $v$ parts
3:  for each frame $F_k$, $k \in [2, m]$ do
4:      for each part $\hat{h}, \hat{s}, \hat{v}$ in H, S, V do
5:          construct the histograms $H(\hat{h})$, $S(\hat{s})$, $V(\hat{v})$ in frame $F_k$
6:      end for
7:      $Sim_H(F_k, S_i) = \sum_{\hat{h}=1}^{h} \min\{H(\hat{h}), S_i[H(\hat{h})]\}$
8:      $Sim_S(F_k, S_i) = \sum_{\hat{s}=1}^{s} \min\{S(\hat{s}), S_i[S(\hat{s})]\}$
9:      $Sim_V(F_k, S_i) = \sum_{\hat{v}=1}^{v} \min\{V(\hat{v}), S_i[V(\hat{v})]\}$
        $\{\alpha, \beta, \gamma$: weights for H, S, V; similarity threshold: $\tau\}$
10:     if $(\alpha \cdot SI_H + \beta \cdot SI_V + \gamma \cdot SI_S) \geq \tau$ then
11:         $S_i \leftarrow S_i \cup F_k$
12:     else
13:         $i = i + 1$ and initialize a new segment $S_i$
14:         $S_i \leftarrow S_i \cup F_k$
15:     end if
16: end for
17: for each segment $S_i$ do
18:     compute the maximum frame entropy $Entropy(F)$:
19:     max $\{-\alpha \cdot \sum_{\hat{h}=1}^{h}[H(\hat{h}) \log H(\hat{h})] - \beta \cdot \sum_{\hat{s}=1}^{s}[S(\hat{s}) \log S(\hat{s})] - \gamma \cdot \sum_{\hat{v}=1}^{v}[V(\hat{v}) \log V(\hat{v})]\}$
20:     extract the key frame with maximum entropy $\mathcal{F}_i$ in $S_i$
21: end for
22: return all the segments and key frames
```

3.2.2 Random Response.

After dimension reduction, random response can be implemented based on the RAPPOR framework [16] for each object. Each bit kb_i^k in ℓ-bit vector of object O_i is randomly flipped into 0 or 1 using the following rules:

$$kb_i^k = \begin{cases} kb_i^k, \text{ with the probability of } (1-f) \\ 1, \text{ with the probability of } \frac{f}{2} \\ 0, \text{ with the probability of } \frac{f}{2} \end{cases} \tag{4}$$

Theorem 3.3. *The random response (with rules in Equation 4) $\ell \log(\frac{2-f}{f})$-Object Indistinguishability.*

Proof. Again, object indistinguishability can be proven by following the proof of LDP [16]. Specifically, the RAPPOR [16] satisfies $2h \log(\frac{2-f}{f})$-LDP with the output size of the hash function in the bloom filter h and the flipping probability f. Maximum difference sizes are $2h$ between two input values. Thus, the random response (with rules in Equation 4) make ϵ equal to $\ell \log(\frac{2-f}{f})$ since size difference in any two presence vector is at most ℓ (by replacing the encoded bit vectors of bloom filter as the object presence vectors in RAPPOR [16]), which satisfies $\ell \log(\frac{2-f}{f})$-*Object Indistinguishability.*

□

3.3 Optimizing RAPPOR for Object Presence

Although ℓ is far less than m, the number of key frames ℓ may still be large depending on the background scene(s), activity motion and light density. To solve this, we can further reduce the dimension by choosing a subset of key frames out of ℓ key frames to allocate the privacy budget. Indeed, determining whether each key frame is picked for allocating the privacy budget or not can be formulated as an optimization problem (*maximizing the utility of generating the synthetic video using the random object presence vectors in Phase II*).

3.3.1 Optimization Problem. For each key frame $\mathcal{F}_k, k \in [1,\ell]$, we define a binary variable $x_k \in \{0,1\}$ to represent if key frame $\mathcal{F}_k$ is picked for budget allocation or not. Then, the total number of picked key frames is referred as $\sum_{k=1}^{\ell} x_k$. Per the Theorem 3.3, we have the random response satisfies $\sum_{k=1}^{\ell} x_k \log(\frac{2-f}{f})$-*Object Indistinguishability.*

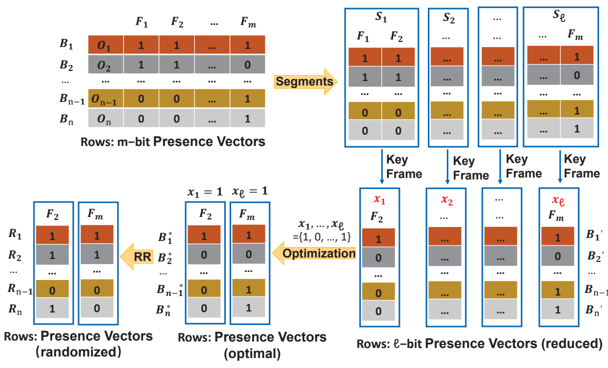

Segments
Rows: m-bit Presence Vectors
Key Frame
Optimization
RR
Rows: Presence Vectors (randomized)
Rows: Presence Vectors (optimal)
Rows: ℓ-bit Presence Vectors (reduced)

Figure 3: Dimension Reduction, Utility Maximization and Random Response

An example for dimension reduction, utility maximization and random response is given in Figure 3. Considering the n objects $O_1, \dots, O_n$, after dimension reduction, all the n object presence vectors are reduced to n different ℓ-bit vectors. Our goal is to accurately retain more objects in the video, thus we aim at minimizing the distance between $\forall i \in [1,n], B_i'$ (extracted from $\mathcal{V}$) and $\forall i \in [1,n], R_i$ (denoted as the ℓ-bit vectors by applying random response to $\forall i \in [1,n], B_i'$).

Specifically, since $\forall i \in [1,n], R_i$ are randomized bit vectors (the kth entry in all the vectors are 0 if $x_k = 0$), we should measure the difference between the expectation $\forall i \in [1,n], E(R_i) = E[(R_i^1, \dots, R_i^{\ell})]$ and $B_i' = (kb_i^1, \dots, kb_i^{\ell})$.

We first learn the expectation of R_i^k (the kth entry in R_i). If $x_k = 0$, then $\forall i \in [1,n], R_i^k = 0$ hold. Thus, we have:

$$E(R_i^k) = x_k \cdot [Pr(R_i^k = 1) \cdot 1 + Pr(R_i^k = 0) \cdot 0] \tag{5}$$

There are two cases for R_i^k (in case of $x_k = 1$):

(1) If $kb_i^k = 1$, per Equation 4, we have $E[R_i^k] = 1 \cdot [(1-f) \cdot 1 + \frac{f}{2} \cdot 0 + \frac{f}{2} \cdot 1]$.

(2) If $kb_i^k = 0$, we have $E[R_i^k] = 1 \cdot [(1-f) \cdot 0 + \frac{f}{2} \cdot 0 + \frac{f}{2} \cdot 1]$.

Thus, the expectation can be summarized as following:

$$\begin{cases} E(R_i^k) = \frac{f}{2}, \text{ if } x_k = 1 \text{ and } kb_i^k = 0 \\ E(R_i^k) = 1 - \frac{f}{2}, \text{ if } x_k = 1 \text{ and } kb_i^k = 1 \\ E(R_i^k) = 0, \text{ if } x_k = 0 \text{ and } kb_i^k = 0 \text{ or } 1 \end{cases} \tag{6}$$

The objective function can be formulated as:

$$\min: \sum_{k=1}^{\ell} [x_k \sum_{i=1}^{n} |E(R_i^k) - kb_i^k|] \tag{7}$$

Furthermore, for accurately interpolating the objects in different frames in Phase II, the number of key frames picked for budget allocation should be no less than 2. Therefore, we formulate the optimization problem as below:

$$\begin{aligned} &\min: \sum_{k=1}^{\ell} [x_k \sum_{i=1}^{n} |E(R_i^k) - kb_i^k|] \\ &s.t. \begin{cases} 2 \le \sum_{k=1}^{\ell} x_k \le \ell \\ \forall k \in [1,\ell], x_k \in \{0,1\} \end{cases} \end{aligned} \tag{8}$$

Detailing expectation $E(R_i^k)$ with the flipping probability, the optimization problem can be converted to:

$$\begin{aligned} &\min: \sum_{k=1}^{\ell} (x_k | \frac{n \cdot f}{2} - f \cdot \sum_{i=1}^{n} kb_i^k |) \\ &s.t. \begin{cases} 2 \le \sum_{k=1}^{\ell} x_k \le \ell \\ \forall k \in [1,\ell], x_k \in \{0,1\} \end{cases} \end{aligned} \tag{9}$$

3.3.2 Complexity and Solver. Since f and $\forall k \in [1,\ell], \forall i \in [1,n], kb_i^k$ are constants, $\forall k \in [1,\ell], |\frac{n \cdot f}{2} - f \cdot \sum_{i=1}^{n} kb_i^k|$ are constants. Then, Equation 9 is a binary integer programming (BIP) problem. Although solving the BIP problems can be NP-hard [27], we can approximately solve Equation 9 using linear programming (LP) since the objective function and the constraints are *linear*: (1) letting the binary variable $\forall k \in [1,\ell], x_k$ be continuous in $[0,1]$, (2) solving the problem using standard LP solvers (e.g., the Simplex algorithm), and (3) in the optimal solution of the LP problem, $\forall k \in [1,\ell]$, if $x_k \in [0, 0.5)$, we assign $x_k = 0$; if $x_k \in [0.5, 1]$ we assign $x_k = 1$ as the approximated optimal solution of the BIP problem.

3.3.3 Addressing Possible Privacy Leakage in Optimization. Compared to randomly picking a number of key frames for budget allocation, computing the optimal frames for budget allocation may result in some minor privacy leakage since the total number of objects in the kth key frame $\sum_{i=1}^{n} kb_i^k, k \in [1,\ell]$ (which is used in the optimization) might be different. Such privacy leakage is generally minor due to a small sensitivity Δ of the object count in each frame (e.g., $\Delta = 1$ for protecting the presence/absence of each object in every frame). Thus, it can be addressed by injecting a small amount of generic Laplace noise $Lap(\frac{\Delta}{\epsilon'})$ into $\sum_{i=1}^{n} kb_i^k, k \in [1,\ell]$ before formulating the optimization problem. Although adding such small amount of noise may slightly deviate the optimality, this could guarantee end-to-end indistinguishability (differential privacy). Since such privacy guarantee is well studied in literature [13], we do not discuss it in this paper due to space limitation.

3.4 Privacy Guarantee

After solving the optimization problem, as shown in Figure 3, each of the picked key frames will be allocated with a privacy budget $\epsilon/\sum_{k=1}^{\ell} x_k$. In the meanwhile, VERRO utilizes the optimal solution $\forall k \in [1, \ell], x_k$ n to derive the optimal presence vectors ($\sum_{k=1}^{\ell} x_k$-bit), denoted as $B_1^*, \dots, B_n^*$. Next, random response is applied to $B_1^*, \dots, B_n^*$ to generate output presence vectors $R_1, \dots, R_n$.

Theorem 3.4. *Phase I satisfies ϵ-Object Indistinguishability.*

Proof. Phase I derives the presence bit vectors B_i^* and B_j^* for any two objects O_i and O_j after the optimization. Then, random response is applied to B_i^* and B_j^* and generate random vectors R_i and R_j. Per Theorem 3.3, Phase I satisfies ϵ-*Object Indistinguishability* where $\epsilon = \sum_{k=1}^{\ell} x_k \ln \frac{2-f}{f}$ (note that the privacy guarantee for utility maximization has been discussed in Section 3.3.3). □

It is worth noting that the presence of objects in the remaining $(m - \sum_{k=1}^{\ell} x_k)$ frames and the coordinates of the objects in all m frames in the synthetic video $\mathcal{V}^*$ will be generated in Phase II.

4 PHASE II: VIDEO GENERATION

In this section, we illustrate the details of Phase II.

4.1 Background Scene(s)

As discussed in Section 2, video preprocessing includes detecting/tracking objects and background scene(s) extraction. While removing objects from digital images (e.g., each frame of a video), the pixels within the objects are missing in the frame and need to be reconstructed for the background scene(s). In VERRO, we utilize an efficient algorithm [11] to fill the blank area by considering both texture and structure.

First, the quality of the output image/frame highly depends on the order of filling different parts of the blank areas. The algorithm provides a filling strategy by prioritizing them using the combination of the continuation of strong edges and high-confidence surrounded pixels. The priority is computed for every border patch, with distinct patches for each pixel on the boundary of the blank areas. Then, we always start filling at the border pixels with the highest priority.

Second, while filling the pixel p, the algorithm places it at the centroid of a patch with certain size (e.g., 3×3). Then, we traverse all the background pixels, and the centroid pixel of the most similar patch from the source background region will be filled in p, where the similarity is measured by the sum of squared errors. Some reconstructed background scenes are demonstrated in Section 6.

4.2 Randomly Generating Object Coordinates

Phase I generates indistinguishable presence information (in different frames) for all the objects. Next, we need to insert synthetic objects into the background scene (each frame) to generate the synthetic video $\mathcal{V}^*$. Specifically, we denote all the frames in the synthetic video $\mathcal{V}^*$ as $\{F_1^*, \dots, F_m^*\}$, and the frames in $\mathcal{V}^*$ corresponding to the original key frames as $\{\mathcal{F}_1^*, \dots, \mathcal{F}_\ell^*\}$. We then discuss different cases of generating coordinates for the objects in each frame.

4.2.1 $R_i = \emptyset$. If all the entries in any object presence vector are 0, such random vector output R_i would result in object loss (the synthetic video will lose one object), and it is unnecessary to identify the coordinates for them in this case. We have evaluated such utility loss in Section 6, and most of the objects can be retained by VERRO in practice.

4.2.2 $R_i \neq \emptyset$. If there exists at least one non zero entry in R_i, then an object will be inserted to the synthetic video $\mathcal{V}^*$. A critical and challenging question is that where to insert the object. We employ the coordinates of all the objects in the original video $\mathcal{V}$ as "Candidate Coordinates" to generate the coordinates in each frame of the synthetic video.

Specifically, in each key frame of the synthetic video $\forall k \in [1, \ell], \mathcal{F}_k^*$, the number of objects inserted into key frame $\mathcal{F}_k^*$ is $\sum_{i=1}^{n} R_i^k$ (derived in Phase I). Denoting the number of objects in the kth key frame of $\mathcal{V}$ as $c_k, k \in [1, \ell]$ where $c_k = 0$ if $x_k = 0$, we thus have:

- *Sufficient candidate coordinates*: if $\sum_{i=1}^{n} R_i^k \le c_k$, the number of required objects in $\mathcal{F}_k^*$ is no greater than the number of candidate coordinates in $\mathcal{F}_k$. Then, VERRO randomly picks $\sum_{i=1}^{n} R_i^k$ out of c_k candidate coordinates for $\sum_{i=1}^{n} R_i^k$ different objects in the background scene (frame $\mathcal{F}_k^*$). Please see the left example in Figure 4.
- *Insufficient candidate coordinates*: if $\sum_{i=1}^{n} R_i^k > c_k$, the number of required objects in $\mathcal{F}_k^*$ is greater than the number of candidate coordinates in $\mathcal{F}_k$. For instance, in the right example in Figure 4, we expand the set of candidate coordinates by adding the candidate coordinates in $\mathcal{F}_k$'s neighboring frames in the same segment. Then, VERRO randomly picks $\sum_{i=1}^{n} R_i^k$ out of c_k' candidate coordinates (c_k' is expanded from c_k where $c_k < \sum_{i=1}^{n} R_i^k \le c_k'$) to insert $\sum_{i=1}^{n} R_i^k$ different objects into the background scene (frame $\mathcal{F}_k^*$).

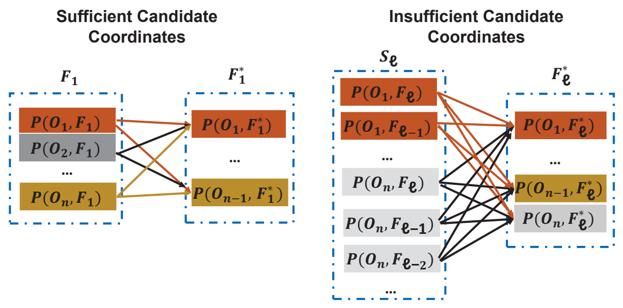


Figure 4: Random Coordinates Assignment (before Interpolation)

After assigning coordinates to the key frames (where $R_i^k = 1$), we obtain at least 1 frame with the corresponding coordinates for any O_i (if the corresponding object is retained in the synthetic video) – the retained object has been assigned with coordinates in at least two frames in almost all the cases in our experiments in Section 6. With such randomly assigned coordinates in some key frames, we can interpolate the coordinates in other frames (out of m frames in total) between such key frames. For instance, given coordinates in two key frames F_1 and F_{10} for object O_i, then its coordinates between F_1 and F_{10} can be estimated. In literature, there are many interpolation methods for moving object trajectories data, such as nearest neighbor interpolation [21] and Lagrange interpolation [17]. In VERRO, we adopt the Lagrange interpolation to estimate such trajectories with the randomly generated positions.

Finally, after interpolation, we define the first frame in which any object first occurs as "head" and the frame where such object last occurs as "end" in the interpolated trajectory. The head and end generally involve such object on the border of the frame. Thus, the interpolation terminates as each object's head and end are identified on the border of the frame (*objects do not occur in all the frames in general*).

Theorem 4.1. *VERRO (Phase I and Phase II) satisfies ϵ-Object Indistinguishability.*

Proof. Given any two objects O_i and O_j, their randomly generated presence vectors R_i and R_j are proven to be *ϵ-Object Indistinguishable* (after Phase I). We now examine the randomly assigned coordinates in the key frames and two full interpolated trajectories in the synthetic video $\mathcal{V}^*$.

Specifically, given any output presence vector y and any output trajectory $t = \{t_1, \dots, t_m\}$ in $\mathcal{V}^*$, for simplicity of notation, we also denote the trajectories of O_i and O_j in $\mathcal{V}^*$ as $O_i = \{T_i^1, \dots, T_i^m\}$ and $O_j = \{T_j^1, \dots, T_j^m\}$, respectively.

$$\frac{Pr[\mathcal{A}(O_i) = t]}{Pr[\mathcal{A}(O_j) = t]}$$
$$= \frac{Pr[\mathcal{A}(B_i') = y]}{Pr[\mathcal{A}(B_j') = y]} \cdot \frac{Pr[\mathcal{A}(T_i^1) = t_1]}{Pr[\mathcal{A}(T_j^1) = t_1]} \cdots \frac{Pr[\mathcal{A}(T_i^m) = t_m]}{Pr[\mathcal{A}(T_j^m) = t_m]}$$

On one hand, we have $\frac{Pr[\mathcal{A}(B_i')=y]}{Pr[\mathcal{A}(B_j')=y]]} \leq e^\epsilon$ (Phase I). On the other hand, if $\forall k \in [1, m], R_i^k = R_j^k = 1$, two objects are present in the same frame F_k (and F_k^*). In this case, since the same randomization is applied to O_i and O_j to pick the coordinates from the same set of candidates, we have $\forall k \in [1, m], Pr[\mathcal{A}(T_i^k) = t_k] = Pr[\mathcal{A}(T_j^k) = t_k]$. If $\forall k \in [1, m], R_i^k = R_j^k = 0$ (the coordinates are interpolated from the coordinates randomly assigned in the previous case [14]), we also have $\forall k \in [1, m], Pr[\mathcal{A}(T_i^k) = t_k] = Pr[\mathcal{A}(T_j^k) = t_k]$.

To sum up the above three cases, we have:

$$\frac{Pr[\mathcal{A}(O_i) = t]}{Pr[\mathcal{A}(O_j) = t]} \leq e^\epsilon \tag{10}$$

where $\epsilon = \sum_{k=1}^{\ell} x_k \log(\frac{2-f}{f})$, as analyzed in Theorem 3.3 and Section 3.3. This completes the proof. □

Finally, we summarize the procedures and privacy guarantee in VERRO. Given an video, the presence of objects in all the frames are indistinguishable via random response. Then, adversaries cannot identify specific objects by the frame presences with any background knowledge. Furthermore, we randomly generate synthetic positions of objects. Therefore, we claim that any object in the input $\mathcal{V}$ can possibly generate any object in the output $\mathcal{V}^*$ (with random response in Phase I and random coordinates assignment in Phase II).

5 DISCUSSION

Distributed Framework: LDP techniques [4, 16] are deployed in distributed setting where each user perturbs its local data to share. Our object-based privacy model ensures indistinguishability at the object level where all the "distributed" local data can be perturbed by a "local agent" (aka. video owner) and shared as $\mathcal{V}^*$ to untrusted recipients.

Different video owners can also share their perturbed videos to any untrusted recipient (all the objects in each video are still well protected). Note that VERRO does not ensure video level indistinguishability (all the videos are indistinguishable). We will investigate the utility of the video level indistinguishability in practice and explore the LDP solutions in the future.

Noise Cancellation: in VERRO, objects and their trajectories are generated in the sanitized video. Thus, the individual noises resulted from random response for all the objects may not be directly canceled in the output video. Indeed, after random response and random coordinates assignment, there exists trajectories in the sanitized video which are close to the original trajectories (as shown in Figure 6-8 in our experiments). Also, such noise can be cancelled in data aggregation applications [9] (e.g., object counting, as shown in Figure 12 and 13).

Multiple Object Types: in this paper, we use pedestrians and vehicles as concrete examples to show their indistinguishability in the publishable synthetic video. It is worth noting that other objects can also be protected with the defined privacy notion in VERRO by replacing the detecting algorithms and synthetic objects. Furthermore, if any video includes multiple types of objects (e.g., pedestrians and vehicles), VERRO can generate the synthetic video for different types of objects, respectively. For instance, it first randomly generates pedestrians, and then randomly generates the vehicles. All the pedestrians are *ϵ-Object Indistinguishable* while all the vehicles are *ϵ-Object Indistinguishable*, assuming that it does not leak additional information across different object types (as all the objects have been replaced with random synthetic objects in the same type).

Protection for One-Object Video: VERRO can generate synthetic videos in which all the objects are ϵ-indistinguishable. In case that the video includes only one sensitive object, VERRO can also protect such object against re-identification. In existing LDP techniques [4, 16], if only one user perturbs its Object data and discloses it to the untrusted aggregator, the original data cannot be identified from its perturbed data. Similar to such works (e.g., RAPPOR [16]), the objects and the trajectories cannot be identified from the perturbed presence in the synthetic video even if the adversary has arbitrary background knowledge on the presence of individuals at specific times.

Imperfect Background Scene(s): as discussed in Section 4, background scene(s) is extracted from the original video. The reconstructed scene may not be as perfect as the original frame (e.g., human/vehicle silhouette or duplicated/blurred region may occur). Thus, imperfect background scene(s) may leak some privacy about "there exists some object in the silhouette or blurred regions in the original video". However, adversaries cannot infer that "who is in that region or which object is in that region" since all the objects are indistinguishable from end to end.

System Deployment: the proposed VERRO can be implemented as an application, and deployed as a component to generate utility-driven synthetic videos by processing the videos captured by each camera (e.g., in the surveillance system, integrated with the traffic monitoring facilities, in smart phones or other mobile devices) where *ϵ-Object Indistinguishability* can be guaranteed.

6 EXPERIMENTS

In this section, we present the performance evaluations.

6.1 Experimental Setup

We conduct our experiments on three real videos in the repository of multiple object tracking benchmark[2]. To benchmark the results, we choose three pedestrian videos, two videos are captured by static cameras while the third video is recorded by a moving camera (where multiple background scenes are extracted):

(1) MOT16-01 (people walking around a large square, denoted as "MOT01") [35]: 23 distinct pedestrians are sensitive objects in 450 frames (static camera).
(2) MOT16-03 (pedestrians on the street at night, denoted as "MOT03") [35]: 148 distinct pedestrians are sensitive objects in 1,500 frames (static camera).
(3) MOT16-06 (street scene from a moving platform, denoted as "MOT06") [35]: 221 distinct pedestrians are sensitive objects in 1,194 frames (moving camera).

Table 1: Characteristics of Experimental Videos

Video	Resolution	Frame #	Objects	Camera
MOT16-01	1920 × 1080	450	23	static
MOT16-03	1920 × 1080	1,500	148	static
MOT16-06	640 × 480	1,194	221	moving

We implement the detecting/tracking algorithm [48, 49] to identify all the objects (pedestrians). Objects are detected in each frame, and the same object is marked with the same ID in the entire video. Computer vision technique [11] is also utilized to extract/reconstruct the background scene(s) from the input video $\mathcal{V}$. All the programs are implemented in Python 3.6.4 with the OpenCV 3.4.0 library and tested on an HP PC with Intel Core i7-7700 CPU 3.60GHz and 32G RAM.

6.2 Generic Utility Evaluation

We first evaluate the utility of our synthetic videos. The proposed VERRO is a two-phase LDP approach. In Phase I, it randomly generates the object presence in all the frames of the synthetic video ("1" or "0"). In Phase II, we interpolate the trajectories. Thus, we evaluate two different types of utility: (1) the retained utility after Phase I (Random Response), and (2) the utility of synthetic video after Phase II.

6.2.1 Utility for Phase I. Phase I generates "presence bit vectors" for all the objects with frame dimension reduction, optimization ("OPT") and random response ("RR"). Some objects might not be included in the key frames, and/or might not be generated in the random response. Then, such objects cannot be generated in the synthetic video (all the entries in the corresponding vectors are 0) since they cannot be interpolated without any object presence in Phase I (also treated as noise). Thus, we evaluate the count of distinct objects (pedestrians) in Phase I.

First, Table 2 shows some results after detecting key frames for frame dimension reduction. In video MOT01, there are 22 key frames, and 19 out of 23 objects are present in the key frames. In video MOT03, 52 key frames are extracted, and 124 out of 148 objects are present in such key frames. In video MOT06, 191 out of 221 objects are captured in the identified 48 key frames. We can observe that frame dimension reduction results in less utility loss (retaining $\sim$ 80% distinct objects).

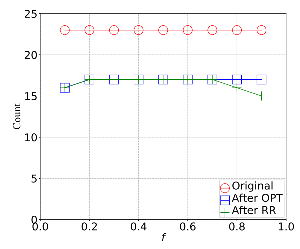
Figure 5(a), 5(c) and 5(e) present the count of distinct objects in original video, after optimization ("OPT"), and random response ("RR"). We set the flipping probability f from 0.1 to 0.9 for random response. In Figure 5(a), approximately 17 distinct objects can be retained in 10 key frames (optimized). f only slightly affects the optimization: the count of distinct objects increases a little bit

[2]https://motchallenge.net/

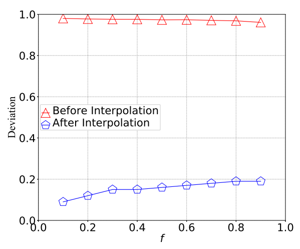
Table 2: Distinct Objects after Key Frame Extraction

Video	Frame #	Objects #	Key Frame #	Remaining #
MOT01	450	23	22	19
MOT03	1,500	148	52	124
MOT06	1,194	221	48	191

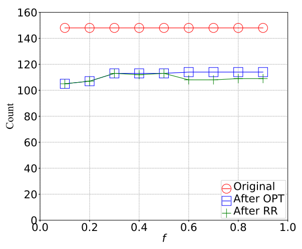
as f grows. To evaluate how f affects the random response, we can observe that one or two objects are not randomly generated in RR as f grows to a large flipping probability (e.g., 0.8). This matches the fact that higher f results in worse utility in random response (Theorem 3.2) – such utility loss is indeed minor in our experiments. In addition, we can draw similar observations in Figure 5(c) and 5(e) where the utility loss of random response is even less for videos MOT03 and MOT06. Thus, Phase I retains a high percent of distinct objects via their random presence vectors, which means less side effect introduced by RR (this facilitates the interpolation in Phase II for boosting utility).



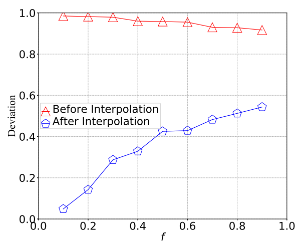
(a) Count of Distinct Ojects



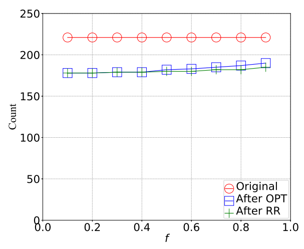
(b) Deviation of Object Trajectories



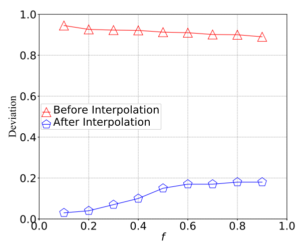
(c) Count of Distinct Ojects



(d) Deviation of Object Trajectories



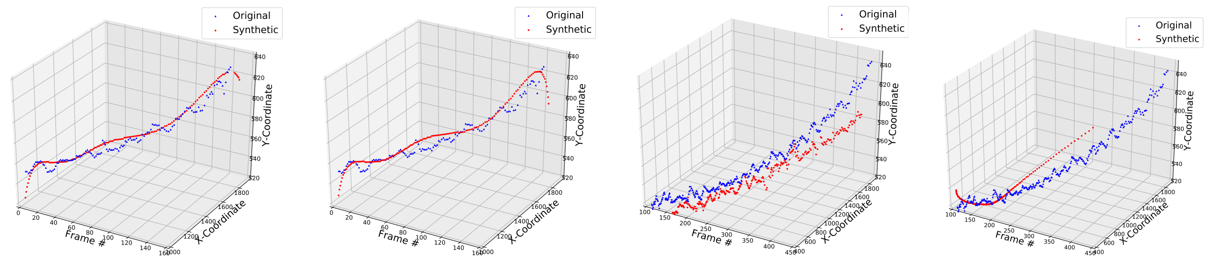
(e) Count of Distinct Ojects



(f) Deviation of Object Trajectories

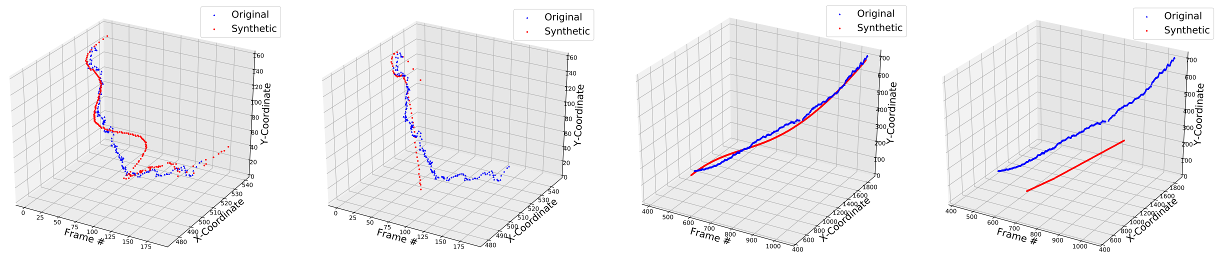
Figure 5: Utility Evaluation of Phase I & II of MOT01 (MOT03 and MOT06)

6.2.2 Utility for Phase II. Since the synthetic video generated in Phase II includes the synthetic objects at the same scene, the corresponding synthetic object of each original object (e.g., pedestrian) may have different coordinates in the same frame.



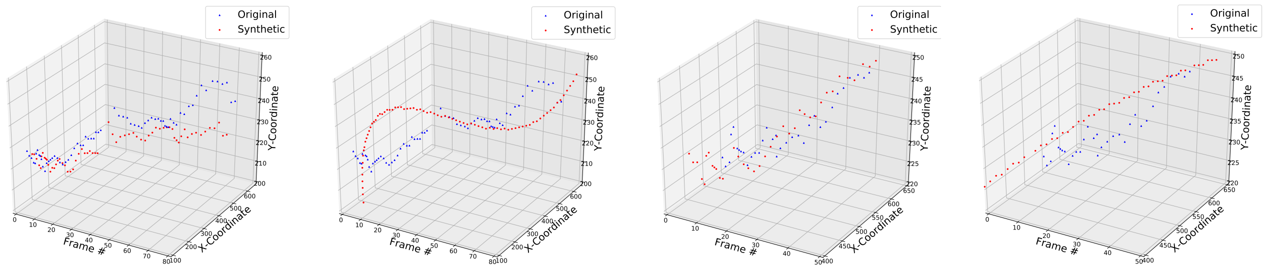
(a) Object #2 (f=0.1) (b) Object #2 (f=0.9) (c) Object #9 (f=0.1) (d) Object #9 (f=0.9)

Figure 6: Trajectories of Two Randomly Selected Objects in MOT01



(a) Object #35 (f=0.1) (b) Object #35 (f=0.9) (c) Object #105 (f=0.1) (d) Object #105 (f=0.9)

Figure 7: Trajectories of Two Randomly Selected Objects in MOT03



(a) Object #5 (f=0.1) (b) Object #5 (f=0.9) (c) Object #165 (f=0.1) (d) Object #165 (f=0.9)

Figure 8: Trajectories of Two Randomly Selected Objects in MOT06

All the coordinates in different frames may form a trajectory in the synthetic video. Thus, we also measure the deviation for the trajectories of all the objects in the original video and synthetic video: $\sum_{i=1}^{n}\sum_{k=1}^{m}\frac{P(O_i,F_k)-P(O_i,F_k^*)}{P(O_i,F_k)}$, where $P(O_i, F_k)$ and $P(O_i, F_k^*)$ are the center coordinates of object O_i in the kth frame of the input video and the synthetic video.

In Figure 5(b), 5(d) and 5(f), we can observe that the deviation before Phase II is higher than 0.9, since each object is only generated in a few frames. The deviation of trajectories increases as the flipping probability f gets larger since more flips occur more frequently (e.g., "0" to "1", or vice-versa). In such three figures, after Phase II, the deviation can be significantly reduced (e.g., in $[0.1, 0.2]$ for video MOT01, in $[0.02, 0.2]$ for video MOT06).

More specifically, we randomly select two objects (e.g., pedestrians) from each of the three videos, and extract their trajectories in the original video $\mathcal{V}$. In addition, we also extract their corresponding trajectories in the synthetic video $\mathcal{V}^*$. Figure 6, 7 and 8 demonstrate the trajectories of those objects in the input videos and synthetic videos, where 3-dimensional axes refer to the frame ID and coordinates (X, Y) in videos. As $f = 0.1$, the trajectories of the objects lie closer to the original ones (compared to $f = 0.9$). It is worth noting that any object (pedestrian) in the original video can generate the corresponding trajectory of any object (e.g., the plotted trajectories corresponding to Object #2 and Object #9 in Figure 6). This is ensured by the ϵ-indistinguishable presence bit vectors randomly generated from all the objects in VERRO.

6.3 Visual & Aggregated Results

We also randomly pick a frame from each of the three experimental videos, and present the generated background scenes and the corresponding frames in the synthetic videos. For video MOT01, Figure 9(a) shows the input frame and the detected objects in the frame. Also, we use a background interpolation algorithm [11] to fill the missing pixels (after removing all the detected objection), as shown in Figure 9(b). Similarly, a randomly picked frame (with the detected objects) and the generated background

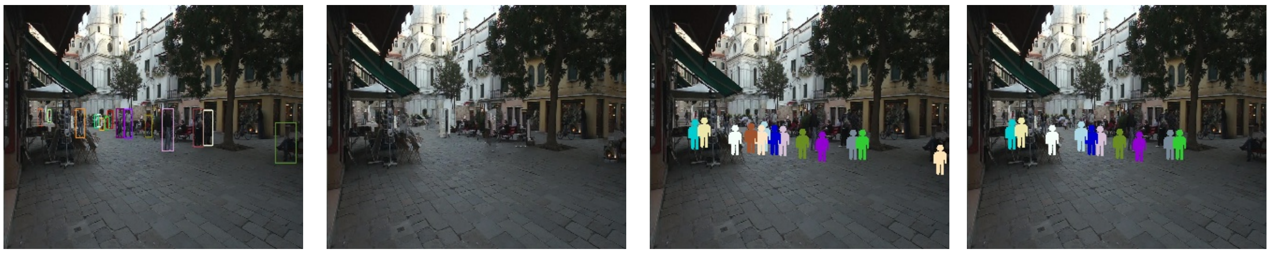

(a) Frame 8 (b) Background Scene (c) Synthetic Frame (f=0.1) (d) Synthetic Frame (f=0.9)

Figure 9: Representative Frames in MOT01 and the Generated Synthetic Video

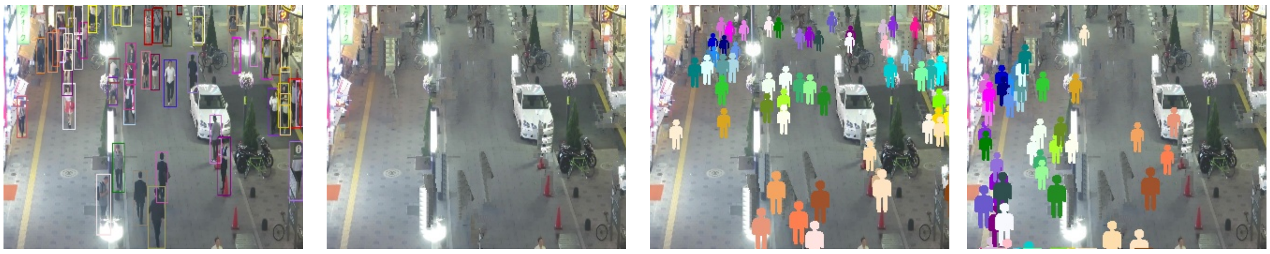

(a) Frame 134 (b) Background Scene (c) Synthetic Frame (f=0.1) (d) Synthetic Frame (f=0.9)

Figure 10: Representative Frames in MOT03 and the Generated Synthetic Video

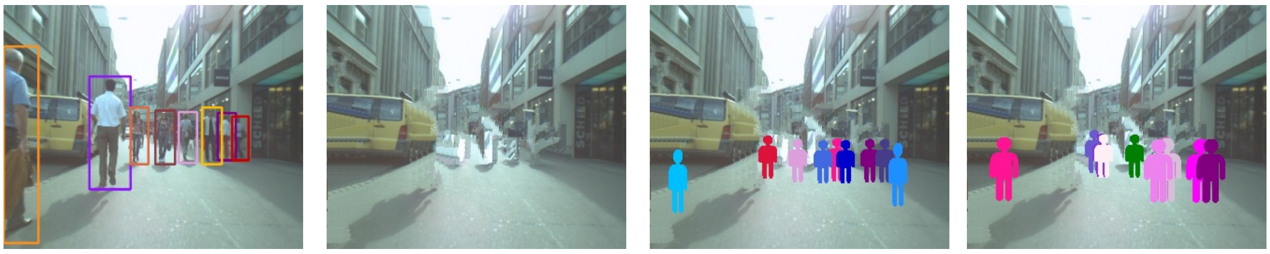

(a) Frame 216 (b) Background Scene (c) Synthetic Frame (f=0.1) (d) Synthetic Frame (f=0.9)

Figure 11: Representative Frames in MOT06 and the Generated Synthetic Video

scenes in MOT03 and MOT06 are given in the first two subfigures of Figure 10 and 11. Some human silhouettes still exist in the background scenes. Clearly, the silhouettes cannot be associated to any objects in the synthetic video (as shown in Figure 10(c), 10(d), 11(c) and 11(d)). This confirms the discussion for imperfect background scene in Section 5.

In the synthetic videos, we use different colors for different synthetic objects. Compare to $f = 0.1$ (shown in Figure 9(c), 10(c) and 11(c)), $f = 0.9$ would lead to more coordinates/trajectory deviation (as shown in Figure 9(d), 10(d) and 11(d)). However, accurate count of objects (pedestrians) can be retained in the synthetic frames even if the flipping probability f is specified as 0.9 (small privacy bound). Thus, we can still use such synthetic videos to function specific application based on the count of objects, e.g., head counting and crowd density [23, 34]. To confirm such observation, we also detect and count all the pedestrians in each frame of the synthetic videos ($f = 0.1$ and $f = 0.9$).

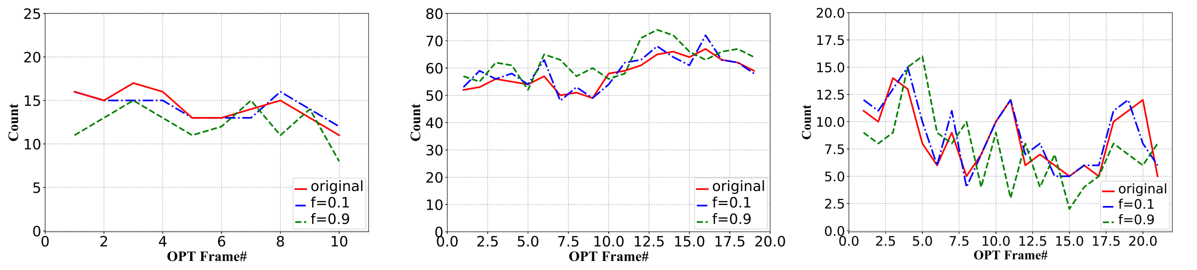
Figure 12 shows the pedestrian counts in the (optimized) key frames (after Phase I). The aggregated result lies very close to the original result when f is small. When f goes larger, the aggregated result is slightly more fluctuated, and more objects are generated in the frames. Figure 13 demonstrates the aggregated counts of pedestrians in each frame (after Phase II). Note that many objects (with the coordinates outside the frames; not between the “head” and “end”) are suppressed in Phase II, making the object counts in different frames more accurate. Note that if multiple cameras capture more videos (e.g., surveillance or traffic monitoring cameras for the smart city) for joint analysis, the noise can be further cancelled in the applications.

6.4 Overheads

We evaluate the overheads of VERRO. Table 3 presents the runtime of the two phases and the required bandwidth for sending the synthetic videos to an untrusted recipient.

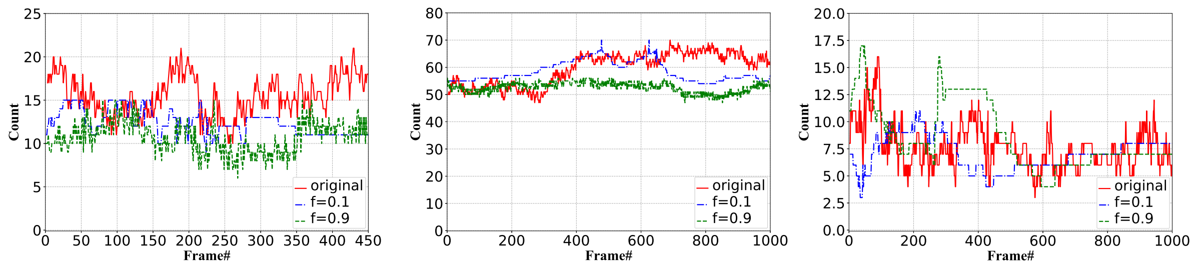
Table 3: Computational and Communication Overheads

Video	Phase I (Sec)	Phase II (Sec)	Bandwidth (MB)
MOT01	0.89	34.78	9.58
MOT03	1.56	36.12	16.6
MOT06	1.57	43.12	19.4



(a) MOT01 (b) MOT03 (c) MOT06

Figure 12: Object Counts in the Optimized Key Frames (by each frame)



(a) MOT01 (b) MOT03 (c) MOT06

Figure 13: Object Counts in the Synthetic Videos (by each frame)

The computational cost increases as the count of distinct objects increases (MOT01 has the least pedestrians while MOT06 has the most pedestrians). The results reflect a *sublinear* increase trend, which enables VERRO to be scaled to generate synthetic videos for longer videos (with more frames). In addition, although MOT06 has a lower resolution (less pixels) than MOT01 and MOT03, it is captured by a moving camera. Since more background scenes have to be interpolated, it requires longer runtime (but still efficient). Note that the runtime for object detecting and background scene(s) generation (1-2 minutes in our experiments) can be considered as computational costs for preprocessing.

Finally, the communication overhead for sharing three synthetic videos is almost identical to the original video size.

7 RELATED WORK

In the context of privacy preserving video publishing, many solutions have been proposed in literature (e.g., [6, 20, 41, 42, 44]). Saini et al. [41] have categorized such works in terms of the sensitive attributes obfuscated in the sanitization. These sensitive attributes include the evidence types *bodies, what*(activity), *where* (location where the video is recorded) and *when* (time when the video is recorded). In general, most of these works employ a *detect* and *blur* policy for only body attributes [6, 20, 42, 44] and some of them [15, 41, 47] aims at preserving the privacy against other three implicit inference channels.[3]

Specifically, these techniques often leverage computer vision techniques [20, 29] to first *detect* faces and/or other sensitive regions in the video frames and then *obscure* them. However, such *detect-and-protect* solutions have some limitations. For instance, the *detect-and-protect* techniques cannot formally quantify and bound the privacy leakage. In addition, blurred regions might still be reconstructed by deep learning methods [33, 37]. Last but not least, these techniques often use naive measures for quantifying the privacy loss in videos. For instance, in [20, 36], if faces are present, then it is considered as complete privacy loss, otherwise no privacy loss is reported. Fan [18] applied Laplace noise to randomly perturb the pixels in an image to ensure differential privacy for protecting specific regions of an image. However, the quality of the image is significantly deviated in the sanitized results. Our proposed privacy notion and the VERRO technique have addressed all the above limitations.

On the other hand, in the context of privacy preserving data publishing, the notion of differential privacy has emerged as a standard specification during past decade. This strong notion of privacy was first proposed by Dwork [13] to guarantee *indistinguishability* in the published data against an adversary armed with arbitrary background knowledge. Although differential privacy has been widely used to sanitize and release data in statistical databases [13], numeric data [45], location data [38], and search logs [25], to the best of our knowledge, no attempt has yet been made to benefit from differential privacy in video databases. Furthermore, to fully utilize differential privacy for sanitizing videos, we have defined our privacy notion based on a recently proposed locally implemented notion of differential privacy in which individuals in the videos (i.e., as objects) can directly interact with the sanitized result to ensure trustworthiness and fine-grained privacy. The emerging local differential privacy (LDP) models [4, 10, 16] have been utilized in a wide variety of applications (e.g., heavy hitters or histogram construction [4, 16], and frequent itemset mining [46]), but cannot be directly applicable to local video perturbation. VERRO complements the literature with strong privacy protection for (local) objects in the video against arbitrary background knowledge.

[3]The synthetic videos generated by VERRO can preserve the information of "where and when the videos are captured" while ensuring indistinguishability of objects (the linkage between every object and such inference channels can be broken to avoid leakage in the disclosure of the background scene).

8 CONCLUSION

Privacy concerns arise in considerable number of real world videos. To the best of our knowledge, we take the first cut to pursue indistinguishability for objects in the video by defining a novel privacy notion *ϵ-Object Indistinguishability*. We propose a two-phase video sanitization technique VERRO that locally perturbs all the objects in the video and generates a utility-driven synthetic video with indistinguishable objects, which can be directly shared to any untrusted recipient. In the synthetic videos, not only the object contents (e.g., different humans, and vehicle make/model/color), but also their moving trajectories in the video (e.g., a series of coordinates in different frames) can be effectively protected since every synthetic object and its trajectory can be possibly generated from any object in the original video. Experiments performed on real videos have validated the effectiveness and efficiency of VERRO. In the future, we will comprehensively study the utility of the synthetic videos in more application scenarios, and explore rigorous protection for objects which can be tracked in multiple videos.

9 ACKNOWLEDGEMENTS

This work is partially supported by the National Science Foundation (NSF) under awards CNS-1745894 and CNS-1564034, and the National Institutes of Health (NIH) under awards R01GM118574 and R35GM134927. The authors would like to thank the anonymous reviewers for their constructive comments.

REFERENCES

[1] 2012. YouTube~Official~Blog~2012
[2] 2019. https://ops.fhwa.dot.gov/trafficanalysistools/ngsim.htm
[3] Bruno Abreu, Luis Botelho, and et.al. 2000. Video-based multi-agent traffic surveillance system. In *Intelligent Vehicles Symposium.* 457–462.
[4] Raef Bassily and Adam Smith. 2015. Local, private, efficient protocols for succinct histograms. In *Symposium on Theory of Computing.* 127–135.
[5] Vincent Bindschaedler, Reza Shokri, and Carl A Gunter. 2017. Plausible deniability for privacy-preserving data synthesis. *VLDB* (2017), 481–492.
[6] Michael Boyle, Christopher Edwards, and Saul Greenberg. 2000. The Effects of Filtered Video on Awareness and Privacy. In *CSCW.* 1–10.
[7] Yang Cao, Masatoshi Yoshikawa, Yonghui Xiao, and Li Xiong. 2017. Quantifying Differential Privacy under Temporal Correlations. In *ICDE.* 821–832.
[8] Paula Carrillo, Hari Kalva, and Spyros Magliveras. 2008. Compression independent object encryption for ensuring privacy in video surveillance. In *Multimedia and Expo.* 273–276.
[9] Rui Chen, Haoran Li, A Kai Qin, Shiva P. Kasiviswanathan, and Hongxia Jin. 2016. Private spatial data aggregation in the local setting. In *ICDE.* 289–300.
[10] Graham Cormode, Somesh Jha, Tejas Kulkarni, Ninghui Li, Divesh Srivastava, and Tianhao Wang. 2018. Privacy at scale: Local differential privacy in practice. In *Management of Data.* 1655–1658.
[11] Antonio Criminisi, Patrick Pérez, and Kentaro Toyama. 2004. Region filling and object removal by exemplar-based image inpainting. *IEEE Transactions on image processing* 13, 9 (2004), 1200–1212.
[12] Ajay Divakaran, Regunathan Radhakrishnan, and Kadir A Peker. 2002. Motion activity-based extraction of key-frames from video shots. In *ICIP.* 932–935.
[13] C. Dwork. 2011. Differential privacy. *Encyclopedia of Cryptography and Security* (2011), 338–340.
[14] C. Dwork, A. Roth, et al. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science* (2014), 211–407.
[15] A. Erdelyi, T. Barat, P. Valet, T. Winkler, and B. Rinner. 2014. Adaptive cartooning for privacy protection in camera networks. In *AVSS.* 44–49.
[16] Ú. Erlingsson, V. Pihur, and A. Korolova. 2014. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *CCS.* 1054–1067.
[17] Fariba Fahroo and I Michael Ross. 2002. Direct trajectory optimization by a Chebyshev pseudospectral method. *Journal of Guidance, Control, and Dynamics* 25, 1 (2002), 160–166.
[18] L. Fan. 2018. Image Pixelization with Differential Privacy. In *DBSec.* 148–162.
[19] Huamin Feng, Wei Fang, Sen Liu, and Yong Fang. 2005. A new general framework for shot boundary detection and key-frame extraction. In *Multimedia information retrieval.* 121–126.
[20] Douglas A Fidaleo, Hoang-Anh Nguyen, and Mohan Trivedi. 2004. The networked sensor tapestry (NeST): a privacy enhanced software architecture for interactive analysis of data in video-sensor networks. In *Video surveillance & sensor networks Workshop.* 46–53.
[21] Elias Frentzos, Kostas Gratsias, Nikos Pelekis, and Yannis Theodoridis. 2007. Algorithms for nearest neighbor search on moving object trajectories. *Geoinformatica* 11, 2 (2007), 159–193.
[22] Feng Han, Ying Shan, Ryan Cekander, Harpreet S Sawhney, and Rakesh Kumar. 2006. A two-stage approach to people and vehicle detection with HOG-based SVM. In *Performance Metrics for Intelligent Systems.* 133–140.
[23] Marcus Handte, Muhammad Umer Iqbal, and et. al. 2014. Crowd Density Estimation for Public Transport Vehicles. In *EDBT/ICDT Workshops.* 315–322.
[24] Steven Hill, Zhimin Zhou, Lawrence Saul, and Hovav Shacham. 2016. On the (in) effectiveness of mosaicing and blurring as tools for document redaction. *Privacy Enhancing Technologies* 4 (2016), 403–417.
[25] Yuan Hong, Jaideep Vaidya, Haibing Lu, Panagiotis Karras, and Sanjay Goel. 2015. Collaborative Search Log Sanitization: Toward Differential Privacy and Boosted Utility. *IEEE Trans. Dependable Sec. Comput.* 12, 5 (2015), 504–518.
[26] Yuan Hong, Jaideep Vaidya, Haibing Lu, and Mingrui Wu. 2012. Differentially private search log sanitization with optimal output utility. In *15th International Conference on Extending Database Technology.* 50–61.
[27] Ravindran Kannan and Clyde L Monma. 1978. On the computational complexity of integer programming problems. In *Optimization and Operations Research.* 161–172.
[28] Kevin Karsch, Varsha Hedau, David A. Forsyth, and Derek Hoiem. 2011. Rendering synthetic objects into legacy photographs. *Trans. Graph.* (2011), 157.
[29] Takashi Koshimizu, Tomoji Toriyama, and Noboru Babaguchi. 2006. Factors on the sense of privacy in video surveillance. In *Continuous archival and retrival of personal experences.* 35–44.
[30] Sanjay K Kuanar, Rameswar Panda, and Ananda S Chowdhury. 2013. Video key frame extraction through dynamic Delaunay clustering with a structural constraint. *Visual Communication and Image Representation* (2013), 1212–1227.
[31] Xuan Li, Kunfeng Wang, Yonglin Tian, Lan Yan, Fang Deng, and Fei-Yue Wang. 2019. The ParallelEye Dataset: A Large Collection of Virtual Images for Traffic Vision Research. *Intelligent Transportation Systems* (2019), 2072–2084.
[32] Bingyu Liu, Shangyu Xie, Han Wang, Yuan Hong, Xuegang Ban, and Meisam Mohammady. 2019. VTDP: Privately Sanitizing Fine-grained Vehicle Trajectory Data with Boosted Utility. *IEEE Transactions on Dependable and Secure Computing* (2019), 1–1.
[33] Richard McPherson, Reza Shokri, and Vitaly Shmatikov. 2016. Defeating image obfuscation with deep learning. *arXiv preprint arXiv:1609.00408* (2016).
[34] Frank McSherry and Kunal Talwar. 2007. Mechanism Design via Differential Privacy. In *FOCS.* 94–103.
[35] A. Milan, Laura Leal-Taixé, Ian Reid, Stefan Roth, and Konrad Schindler. 2016. MOT16: A benchmark for multi-object tracking. *CoRR* (2016).
[36] S. Moncrieff, S. Venkatesh, and G. West. 2008. Dynamic privacy assessment in a smart house environment using multimodal sensing. *TOMM* (2008), 10.
[37] Seong Joon Oh, Rodrigo Benenson, Mario Fritz, and Bernt Schiele. 2016. Faceless person recognition: Privacy implications in social media. In *European Conference on Computer Vision.* 19–35.
[38] Lu Ou, Zheng Qin, Shaolin Liao, Yuan Hong, and Xiaohua Jia. 2018. Releasing Correlated Trajectories: Towards High Utility and Optimal Differential Privacy. *IEEE Transactions on Dependable and Secure Computing* (2018), 1–1.
[39] Lei Pan, Xiao-Jun Wu, and Yuan-Yuan You. 2005. Video shot segmentation and key frame extraction based on clustering. *Infrared and Laser Engineering* 34, 3 (2005), 341.
[40] Zhan Qin, Yin Yang, Ting Yu, Issa Khalil, Xiaokui Xiao, and Kui Ren. 2016. Heavy hitter estimation over set-valued data with local differential privacy. In *CCS.* ACM, 192–203.
[41] Mukesh Saini, Pradeep K. Atrey, Sharad Mehrotra, and Mohan Kankanhalli. 2014. W3-privacy: understanding what, when, and where inference channels in multi-camera surveillance video. *Multimedia Tools and Applications* 68, 1 (2014), 135–158. https://doi.org/10.1007/s11042-012-1207-9
[42] A. Senior, S. Pankanti, A. Hampapur, L. Brown, Ying-Li Tian, A. Ekin, J. Connell, Chiao Fe Shu, and M. Lu. 2005. Enabling video privacy through computer vision. *Security Privacy* (2005), 50–57.
[43] Alexander Toshev, Ameesh Makadia, and Kostas Daniilidis. 2009. Shape-based Object Recognition in Videos Using 3D Synthetic Object Models. In *CVPR.*
[44] M. Upmanyu, A. M. Namboodiri, K. Srinathan, and C. V. Jawahar. 2009. Efficient privacy preserving video surveillance. In *Computer Vision.* 1639–1646.
[45] Jaideep Vaidya, Basit Shafiq, Anirban Basu, and Yuan Hong. 2013. Differentially Private Naive Bayes Classification. In *2013 IEEE/WIC/ACM International Conferences on Web Intelligence.* 571–576.
[46] Tianhao Wang, Ninghui Li, and Somesh Jha. 2018. Locally differentially private frequent itemset mining. In *SP.* 127–143.
[47] T. Winkler and B. Rinner. 2013. Sensor-level security and privacy protection by embedding video content analysis. In *DSP.* 1–6.
[48] Nicolai Wojke and Alex Bewley. 2018. Deep cosine metric learning for person re-identification. In *WACV.* 748–756.
[49] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. 2017. Simple online and realtime tracking with a deep association metric. In *ICIP.* 3645–3649.
[50] Roman Yarovoy, Francesco Bonchi, Laks VS Lakshmanan, and Wendy Hui Wang. 2009. Anonymizing moving objects: How to hide a mob in a crowd?. In *Extending Database Technology.* 72–83.
[51] Qiang Zhu, Mei-Chen Yeh, Kwang-Ting Cheng, and Shai Avidan. 2006. Fast human detection using a cascade of histograms of oriented gradients. In *CVPR.* 1491–1498.

PrefDiv: Efficient Algorithms for Effective Top-k Result Diversification

Xiaoyu Ge
University of Pittsburgh
xiaoyu@cs.pitt.edu

Panos K. Chrysanthis
University of Pittsburgh
panos@cs.pitt.edu

ABSTRACT

The ever-increasing supply of data is bringing renewed attention to result diversification, a technique usually studied with result relevance for a given task. Together, they produce a subset of results that are relevant to the user query and contain less redundant information. In this work, we formulate an extended version of the result diversification problem, considering three objectives—relevance, diversity, and coverage—and present a novel approach and algorithms that produce better-diversified results. Our approach takes a large set of possible answers generated from a user query and outputs a representative subset of results that are highly ranked according to the preference of the user. The data items contained in the representative set are diverse, such that each item is different from the rest and provides good coverage of the underlying aspects of the original results. Our approach also suggests a set of appropriate parameters for each user query to achieve a balance between our conflicting objectives and is efficient enough to ensure an interactive experience. We study the complexity of our algorithms and experimentally evaluate them in terms of normalized relevance, coverage, and execution time. Our evaluation indicates a speedup of up to 159x, and outperforms the state-of-the-art algorithms on multiple fronts.

1 INTRODUCTION

Motivation With the exponential increase in the amount of data being generated every second, the term "Big Data" that is adopted to represent the challenge of large-scale data processing is currently mentioned frequently in everyday life [20]. This reflects the fact that people are increasingly reliant on using data as an integral part of their daily activities (e.g., decisions and collaborations).

The challenge of scalable data processing can be examined from two viewpoints. Traditionally, scalability has been seen from a *systems point of view*, where challenges can be attributed to an increasing rate of data on the one hand, and network bandwidth, processing power, and storage limitation on the other hand. Scalability can also be viewed from a *human point of view* [23]. Given the exponential volume of data, the challenge here is how to avoid overwhelming users with irrelevant results.

Query personalization is a well-known technique for dealing with scalability challenges from a human point of view, which often happens at two different levels:

- *Ranking* – Ranking techniques utilize user *preferences* with the aim of providing the most relevant results to the users (e.g., [33]). These techniques can be distinguished as *quantitative-based*, *qualitative-based*, or *hybrid*, based on the type of user preferences that they can support.
- *Diversification* – Diversification techniques aim to reduce the amount of redundant information in the results. These techniques typically group data in sets that are most "dissimilar" with each other (e.g., [3, 12]).

Since highly-ranked items can be similar, result diversification [31] has recently drawn significant attention as a technique to facilitate applications such as keyword search, recommendation systems, and online shopping. The key idea of result diversification is to output a subset of representative results from the original in an informative way, since the user most probably will not view results beyond a small number. This requires the representative top-k results to be relevant, diverse, and maintain good coverage of the original answers (i.e., able to cover different underlying aspects of the original results). One thing to note is that the definitions of both relevance and diversity are subjective; thus, they can vary depending on the query and the user.

Goal In this paper, we present an approach to efficiently compute the representative result set for arbitrary top-k queries under user-definable relevance and diversity definitions. We name this as the *Diversified Top-k (DT-k)* problem.

Challenges Below, we will illustrate the challenges to our proposed approach by means of three examples.

Example 1.1. Assume a tourist who is currently visiting Athens wants to find an affordable restaurant with great taste. So she visits a publicly available database that contains the relation RESTAURANT (Name, Food Type, Cost, Score), where *Name* indicates the official name of the restaurant; *Food Type* indicates the type of food (e.g., Greek, Japanese, Chinese); *Cost* is the average expense per person, and *Score* is a numeric number between 1 and 10 that indicates the quality of the food and services offered at the restaurant. To find the ideal place for dinner, she, therefore, enters the following SQL-like query:

*SELECT * FROM RESTAURANT*
WHERE Score $\geq$ 6 *AND Cost* $\leq$ 20
ORDER BY Cost ASC;

However, these kinds of queries may produce thousands of results, among which the top 5 and bottom 5 results are listed in Table 1. The problem is that users are typically only interested in seeing a small portion of these results, not to mention many of these results are, in fact, redundant (e.g., differ only in the name). Simply fetching a certain top number (e.g., top 5) of results does not help improve their usefulness. Instead, the user might be better served with the right amount of diverse (i.e., dissimilar) items from the original answer with good coverage of different aspects (e.g., Food Type, Cost, Score). Furthermore, among those representative subsets with good diversity and coverage, the one that is most relevant to the user's interest should be preferred, such that the relevance refers to criteria that can be used to rank the answers. These may be obtained by interoperating the SQL-like query itself (e.g., through the "Order By" predicates), or derived from external user profiles (e.g., query histories, crowdsourcing).

One immediate challenge raised is how to define diversity, which clearly changes based on the user and the query being performed. In our work, we associate diversity with the similarity between pairs of answers (i.e., data items). To address this challenge, we propose a tunable definition that can be adjusted with a set of diversity thresholds DIV. Each threshold div in DIV is a real number between $[0, 1]$, which specifies the threshold between "similar" and "dissimilar" data items with respect to the normalized distance given by the specified distance measure (e.g., Euclidean, Manhattan, and Hamming) and attributes. $|DIV| = 0$ results in the traditional top-k query, while more diversity thresholds with higher values increase the diversity of the result set.

Name	Food Type	Cost	Score
McDonald	Fast Food	8	7
KFC	Fast Food	8	7
Burger King	Fast Food	8	7
Arby's	Fast Food	8	7
Oinomageireio H Epirus	Greek	8	9
......			
Scala Vinoteca	Greek	20	9
Ta Karamanlidika tou Fani	Greek	20	10
A Little Taste of Home	Greek	20	9
Liondi Traditional Greek	Greek	20	9
Dio Dekares i Oka	Greek	20	9

Table 1: Top-5 and bottom-5 tuples with respect to the cost.

Name	Food Type	Cost	Score
McDonald	Fast Food	8	7
Beer Garden Ritterburg	German	8	9
Nolan	Japanese	9	8
Oinomageireio H Epirus	Greek	10	10
Dosirak	Korean	12	6

Table 2: Top-5 tuples based on cost that are diversified respect to attributes "Food Type" and "Score".

Example 1.2. With the above diversity parameter, the previous sample query in Example 1.1 could be expanded accordingly:

*SELECT * FROM* RESTAURANT
WHERE Score ≥ 6 *AND* (*Cost* ≤ 20)
ORDER BY Cost *DESC*
DIVERSE BY div = 0.2 *ON* 'Food Type' (Hamming)
AND div = 0.3 *ON* 'Score' (Euclidean) *LIMIT* 5;

where Food Type and Score are the attributes on which the diversity is calculated, and Hamming and Euclidean are the corresponding distance measures. The idea here is to generate a set of results that follow the diversity constraints *DIV* specified within the query [1]. The result of the above query is illustrated in Table 2. Although the above example produces some compelling results with its information representative subset, it could be difficult to see how the coverage is contributing differently to the results than the dissimilarity. To illustrate the importance of the coverage, let us consider a simple example:

Example 1.3. Consider the nodes in Figures 1 and 2. In these two figures, each node represents an item in the dataset, and an edge exists between a pair of nodes iff the similarity between these two nodes are close enough according to some pre-defined threshold. On the one hand, in Figure 2, a set of dissimilar items $\{v_5, v_4\}$ is selected. However, only $\{v_1, v_5, v_4\}$ are considered to be covered by $\{v_5, v_4\}$, as $\{v_2, v_3\}$ are not connected with either v_5 or v_4. On the other hand, in Figure 1, a single vertex v_1 is connected to all four vertices, hence achieving 100% coverage. In this case, one can see that vertex v_1 better represents the entire graph when compared with $\{v_5, v_4\}$, thus indicating coverage is another valuable aspect to the quality of the representative results.

The above two examples (i.e., Example 1.2 and 1.3) illustrate the key advantages and desired features of an effective approach that provides a meaningful and representative subset of the original query results. First, the representative subset is relevant to the intention of the query and contains items that would be ranked highly in the original results. Second, the chosen representative items are diverse, each contributing additional novelty to the answer. Third, the representative items are selected in a way that most items in the original answers are reachable with a small distance (i.e., change) from one of the representative answers. Clearly, simply applying ranking, diversification, or clustering on the original result sets could not achieve the above properties. Thus, techniques that clearly consider multiple aspects of the representative results are needed to address this challenge.

[1]Note that our PrefDiv algorithms take the set of diversity constraints *DIV* as one of their inputs, and it is up to the design of the actual system that integrates the PrefDiv to determine how *DIV* will be integrated with its user query.

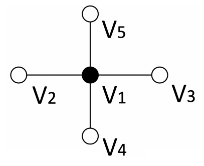


Figure 1: Single vertex v_1 with 100% coverage.

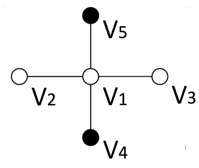


Figure 2: A set of vertices $\{v_4, v_5\}$ with 60% coverage.

Unfortunately, as we will discuss in more detail in Section 2.2.3, finding the optimal solution that maximizes both the "relevance" and "diversity" is an NP-Hard problem by itself, let alone with the addition of the other aspect "coverage" that should also be considered when producing the representative results.

Our Approach To overcome these challenges, we propose an extremely efficient online algorithm, called *Preferential Diversity* (PrefDiv) [13] , for producing representative result sets with sufficient relevance, diversity, and coverage of the original answers. PrefDiv is a top-k bounded general diversification approach that can be applied to any existing relevance ranking model and datasets to retrieve a diversity-aware top-k representative subset of results. PrefDiv starts to construct the representative result set with the k most relevant results (according to the ranking method), then gradually refine this representative set by eliminating pairs of items that do not satisfy the constraints specified by the set of diversity thresholds *DIV*. This is achieved by identifying pairs of items in the representative result set that violate one or more diversity thresholds, and then, among the two items contained in the pair, one with lower relevance will be replaced with an item from the database that improves diversity and coverage. In the end, PrefDiv produces k representative results balanced between relevance, diversity, and coverage.

To the best of our knowledge, PrefDiv is the first general approach to deliver representative results that explicitly consider relevance, diversity, and coverage with an interactive speed that is independent of the underlying database and data set.

However, in order to optimize multiple conflicting objectives such as relevance and diversity, a common approach taken by most diversification algorithms, including our PrefDiv, is to utilize a number of tunable parameters. This could be a major drawback for an algorithm, because with the increase of the number of required parameters, the complexity of the algorithm increases as well, making it more difficult to use in real-world scenarios.

In this paper, we extend and present a family of PrefDiv algorithms based on the vanilla PrefDiv. These includes two novel algorithms that automatically determine: 1) the corresponding diversity thresholds $DIV = \{div_1, div_2, ..., div_n\}$ given the set of diversity constraints Ψ, and 2) the tunable parameters A that balance the trade-off between the relevance and diversity, respectively.

Contributions To achieve the solution as described above, this paper makes the following contributions.

- We formulate the *Diversified Top-k (DT-k)* problem, provide a theoretical analysis of its complexity, and show NP-hardness results. (Section 2)
- We provide a detailed description of the design of PrefDiv, which is an efficient online result diversification algorithm.

Table 3: LIST OF NOTATIONS USED IN THE PAPER

Symbol	Explanation
R_Q	a set of initial query results
R	a set of representative results
k	the number of item in the result-set
$\mathcal{L}$	the number of iterations to obtain R with PrefDiv
Ψ	a set of diversity constraints
ψ	the diversity constraint
div	a diversity threshold
div_{opt}	an optimal diversity threshold
Δ	a set of dimensions
A	a relevance parameter
v	a self-adjustable relevance parameter
I_x	the intensity value of item x
$U(x)$	a utility function that produces the I_x
$sim_\Psi(x_i, x_j)$	x_i and x_j are similar w.r.t Ψ
$dissim_\Psi(x_i, x_j)$	x_i and x_j are dissimilar w.r.t Ψ

- We introduce the concept of Relevance Proportionality (RP), that dynamically balances the trade-off between relevance and diversity during the retrieval of the top-k representative results based on the given query and dataset.
- We propose a novel greedy algorithm that automatically finds the optimal diversity threshold, which maximizes the coverage of the representative result set produced by the PrefDiv.
- We perform extensive experimental evaluations with two real-world datasets, Cameras [10] and Foursquare [7]. Our experimental results show that PrefDiv and its optimizations outperform the most similar state-of-the-art competitors, as suggested in [34], by a significant margin. Compared to other alternatives (discussed in Section 5), our algorithm achieves up to 159x speedup and produces a representative subset that better covers the original answers with a negligible performance decrease in relevance. (Section 4)

2 PROBLEM FORMULATION

In our work, we assume that the database DB is composed of N data items over a D-dimensional space $(d_1, d_2, ..., d_D)$, where each dimension $d \in D$ can be either numerical or categorical attributes. Note that the above assumption enables us to handle any data type (e.g., structured, semi-structured, unstructured) as long as they are vectorized. The user specifies a query Q that aims to retrieve a set of k representative items from DB over a subset of dimensions S, such that $S \leq D$. The goal here is to produce a set of k items that maximizes the *relevance* while ensuring the *diversity* (i.e., ensuring each item is diverse with respect to one another). Below, we will first provide the necessary background and basic concepts of our problem, and then present our problem definition and analyze its complexity. The list of symbols used in the following sections of the paper is shown in Table 3.

2.1 Background

2.1.1 Relevance. The relevance of R represents the degree of the relevancy of each data item $x \in R$ and is typically represented with a utility function $U(x)$ that measures the "goodness" of each data item with respect to certain metrics.

Definition 1. *Relevance* – Given a database DB and a utility function $U(x)$, the relevance is measured as the outcome of $U(x)$, which is an *intensity value* $I_x \in (0, 1) \subset \mathbb{R}$ for item $x \in$ DB that is used to express the degree of benefit from retrieving x.

A higher intensity value indicates that a data item is more desired than those items with a lower intensity value.

The intensity value (i.e., relevance score) enables database systems to produce a total order of each data item in a given data set and thus allow the extraction of top-k data items. This is simply achieved by retrieving k data items with the highest intensity value.

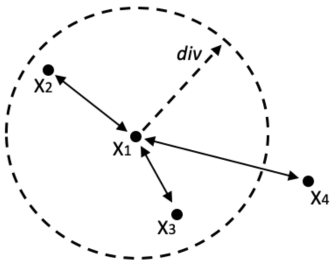


Figure 3: Illustration of similarity and dissimilarity.

2.1.2 Diversity. In our work, the diversity of a set of data items R is achieved by enforcing each pair of data items in R to be *dissimilar* with respect to each other, such that two data items x_i, and x_j are said to be dissimilar if for a given set of user-specified diversity constraints Ψ, x_i and x_j satisfy all constraints $\psi \in \Psi$. Formally, we can define a diversity constraint as follows:

Definition 2. *Diversity Constraint* – For a given pair of items x_i and x_j, a set of attributes Δ, a distance threshold div, and a distance function $dist(x_i, x_j, \Delta)$ that measures the distance between x_i and x_j with respect to the set of dimensions specified in Δ. A diversity constraint ψ is satisfied iff $dist(x_i, x_j, \Delta) > div$.

Based on the above definition of diversity constraints, we now define dissimilarity as:

Definition 3. *Dissimilarity* – Let X be a set of data items. For a given set of diversity constraints Ψ, two items x_i and $x_j \in X$ are dissimilar to each other, denoted as $dissim_\Psi(x_i, x_j)$, if they satisfy each diversity constraint ψ in Ψ.

Consequently, the similarity can simply be defined as the opposite of the dissimilarity, such that:

Definition 4. *Similarity* – Let X be a set of data items. For a given set of diversity constraints Ψ, two items x_i and $x_j \in X$ are similar to each other, denoted as $sim_\Psi(x_i, x_j)$, if they fail to satisfy at least one diversity constraint in Ψ.

Figure 3 illustrates the concept of similarity and dissimilarity with four 2-dimensional data items $x_1, .., x_4$, and a single diversity constraint that requires the Euclidean distance between each data object with respect to both dimensions (i.e., $\Delta = d_1, d_2$) to be at least div apart. Let us take point x_1 as an example. According to Definition 4, points $\{x_2, x_3\}$ are similar to x_1, since $dist(x_2, x_1, \Delta) \leq div$ and $dist(x_3, x_1, \Delta) \leq div$. In contrast, x_4 is dissimilar with respect to x_1, as $dist(x_4, x_1, \Delta) > div$.

2.1.3 Coverage. As pointed out in the previous literature [11], the coverage is another aspect that is important to the quality of the representative results. Since the size of the representative results is very restricted compared to the original answers, having a set of representative results with good coverage increases the chance for the user to get meaningful information from the selected representation items. Furthermore, coverage enables the system to organize all the answers in a cluster-like fashion, where each original answer of query Q can still be retrieved by "zoom-in" into one of the representative items. Such that the "zoom-in" operation will reveal all answers that are "similar" to the selected representative item. The actual implementation of this "zoom-in" operation has been well discussed in [11], thus it is omitted from the discussion of this paper.

Clearly, the coverage is defined completely based on the definition of the similarity and thus related heavily to the diversity constraints when the number of representative results is fixed to a certain number k. When k is fixed, a set of more relaxed diversity constraints (i.e., with higher diversity threshold) will help the representative set include more original answers into its coverage, and a set of stricter diversity constraints will certainly decrease the coverage of the representative set. In particular, given the definition of similarity, if item x_j satisfies $sim_\Psi(x_i, x_j)$, x_j is said to be covered by the item x_i. Consequently, we can define the coverage of a set of items as follows:

Definition 5. *Coverage* – Given a set of original answers R_Q and a representative result set R, where $R \subseteq R_Q$, the coverage of R corresponds to the percentage of items in R_Q that satisfies $sim_\Psi(x_i, x_j)$, such that $x_i \in R$ and $x_j \in R_Q$.

2.2 Diversified Top-k (DT-k) Problem

Based on the above discussions and definitions, we name our problem the *Diversified Top-k (DT-k)* problem.

2.2.1 Problem Formulation. Consider a database DB that consists of N data items distributed over a multi-dimensional space with mixed numeric and categorical dimensions. Given a query Q and its corresponding initial results set R_Q over DB, the desired result cardinality of k, a utility function $U(x)$, and a set of diversity constraints Ψ, the solution of DT-k produces a k-sized representative subset R from the original results R_Q, whose *relevance*, according to $U(x)$ is maximum, while satisfying the set of diversity constraints Ψ.

We name the above k-sized subset of representative results as Diversified Top-k (DT-k) set.

2.2.2 Problem Complexity. Finding the optimal DT-k Set for the Diversified Top-k problem is computationally hard, which can be shown by mapping it to the well-known *Maximum-weight Independent Set* problem [1]. We can achieve the mapping by forming a graph of G that corresponds to the original results R_Q. Each data item x_i in R_Q maps to a vertex v_i in G. An edge e is added between two vertices v_i and v_j if the distance between these two vertices is close enough such that not all diversity constraints are satisfied, and the intensity value I_{x_i} of an item x_i represents the weights of the corresponding vertex in G. Some tractable solutions have been proposed in the literature [18, 21], but these solutions require either a very specific type of graph (e.g., Outerstring graphs) or have strict restrictions (e.g., sparsity, outcome degree of each vertex). Thus, they are not practical in our environment.

2.2.3 Secondary Objective. As discussed above, coverage is another important aspect of result diversification, which is dependent completely on the diversity threshold specified inside each diversity constraint. Given that diversity constraints are typically defined by the user, this may lead to sub-optimal results if the user fails to define reasonable constraints. Consequently, our secondary objective is to address this challenge by automatically adapting the diversity constraints based on the type of the query being performed and the initial result set. Later, in Section 3.4, we will present a general optimization that helps determine the most suitable diversity constraints for different user queries.

3 PREFDIV ALGORITHMS

In this section, we introduce our solution to the Diversified Top-k problem. First, we start with the discussion of a naive approach to the problem and then propose our solution to this problem, namely, *Preferential Diversity* (PrefDiv) algorithm. Finally, we discuss some optimizations that improve the effectiveness of our proposed PrefDiv algorithm and reduce its number of tunable parameters.

3.1 Naive Solution

Before we discuss our solutions, one naive solution to the Diversified Top-k problem work as follows: given a new user query Q, a k, a set of initial results $R_Q = \{x_1, ..., x_t\}$, a utility function $U(x)$ and a set of diversity constraints Ψ, for each item in R_Q of q, we first compute and sort each item in R_Q according to the intensity value computed by the $U(x)$. We pick the item $x_i \in R_Q$ with the highest intensity value; for each remaining items x_j in R_Q, we mark them as "Eliminated" if they are similar to the x_i (i.e., $sim_\Psi(x_i, x_j)$). We then add x_i into the final result set R and remove x_i from R_Q. Afterwards, a new unmarked item with the highest intensity value will be picked from R_Q, and the previous steps will be repeated until either $|R| = k$ or all remaining items in $|R_Q|$ are marked as "Eliminated".

This naive solution is a greedy approach that will eventually produce a set of items that satisfy all diversity constraints with relatively high-intensity values. Clearly, the naive solution is computationally expensive, especially when the size of R_Q is large. Furthermore, it does not guarantee the resulting set to contain at least k items. However, we use this naive solution as a foundation and propose an efficient online solution that achieves better performance with much less computational cost.

ALGORITHM 1: PrefDiv

```
Require:
 1: Initial result set $R_Q$, result cardinality $k$, relevance
    parameter $A$, a set of diversity constraints $\Psi$
Ensure:
 2: One subset $R$ of $R_Q$
 3: $T \leftarrow \emptyset$
 4: while exists unexamined items in $R_Q$ and $|R| < k$ do
 5:   $T \leftarrow$ Pick $k$ items with highest intensity from $R_Q$
 6:   for all $x_i \in T$ do
 7:     if $Dissim_\Psi(x_i, x_j) : \forall x_j \in R$ then
 8:       $R \leftarrow R \cup x_i$
 9:     else
10:       Mark $x_i$ as "redundant"
11:   while number of promoted items in $R$ from $T < A * k$ do
12:     $R \leftarrow R \cup x_{max}$, s.t., $x_{max}$ is marked &
    $\forall x_j \in T, I_{x_{max}} \geq I_{x_j}$
13:     $T \leftarrow T - x_{max}$
14:   $A \leftarrow A/2$
15:   $R_Q = R_Q - T$
16: Return $R$
```

3.2 Preferential Diversity

Our Preferential Diversity algorithm is an online solution for the DT-k problem. As discussed in the previous section, finding the optimal solution to the DT-k problem is computationally expensive. Thus we chose a greedy approach in the PrefDiv design. To maximize the efficiency of PrefDiv, we need to develop it as an online algorithm that accesses database tuples (i.e., items) incrementally. The main idea underlying PrefDiv is minimizing as much as possible the number of data items being examined.

PrefDiv builds the DT-k set R by starting with a set of k highest ranked data item (with respect to the relevance score/intensity value), and then gradually replacing items that fail the diversity constraints with slightly less relevant but diverse items outside of R that satisfy the diversity constraints. This process continues until all items in R satisfy the specified diversity constraints.

One potential issue is that relevant items in the DT-k set tend to be similar to each other. Thus strictly enforcing diversity constraints may eliminate too many items that are highly beneficial to the user. To address this issue, we propose a *relevance parameter A* that allows PrefDiv to produce representative results with *partial* diversity. When $A = 1$, R would simply be the top k items from the initial set, i.e., the items with the k highest intensity values. When $A = 0$, R contains k dissimilar items from the initial set. When A is between 0 and 1 and given that PrefDiv is an iterative algorithm, considering k objectives each iteration, the final result will have at least $A * k$ items from every iteration, and in each iteration A will be divided by half. For example, when $A = 0.5$ and $k = 20$, the first iteration will select at least $20 * 0.5$ items for the final result set, the second iteration will select at least $20*(0.5*0.5)$ items, and so on. With this parameter, the user is able to control the trade-off between relevance *vs.* diversity by enabling partial diversity whenever necessary.

As illustrated in Algorithm 1, the basic logic of PrefDiv is as follows: PrefDiv takes as input, a set of initial results R_Q sorted according to the descending of their intensity value, the desired result cardinality of k, partial diversity parameter A, and a set of diversity constraints Ψ. It outputs a DT-k set that represents the original answers R_Q. In each iteration, PrefDiv fetches and removes k items with the highest intensity value from R_Q and places them in a temporary set T. Each of the items in T is then compared with items currently in R, such that any item in T that fails to satisfy all diversity constraints with respect to all items in R will be marked as “Redundant"; else, it will be added into R immediately. This process will continue until all items in T are either moved into R or marked as “Redundant". Once all k items fetched in the current iteration have been examined, PrefDiv will check if a sufficient number of items were moved into R according to parameter A. In case the number is not sufficient, the difference will be covered by the highest-ranked items (with respect to intensity value) that are marked as “Redundant" in the current iteration. Afterward, the above iteration will be repeated until k representative items are produced ($|R| = k$).

Time Complexity According to the above discussion, we can observe that the worst-case complexity of PrefDiv is $O(kN)$, since each of the N unlabeled items will be compared at most $k - 1$ times with the items that are currently in the result set before being included or discarded from the final result. Fortunately, as the size of k is usually a small number, PrefDiv should typically behave as a linear algorithm. Furthermore, as we will show in our empirical studies (Section 4), depending on the diversity constraints, PrefDiv typically does not need to examine all original items in R_Q. That is, a very small set of the item would be sufficient enough to produce R if Ψ are appropriately defined.

3.3 Relevance Proportionality

From the above discussions, it should be clear that having a good balance between relevance and diversity is important to the quality of the representative result set. In PrefDiv, we have introduced the relevance parameter A to enable the partial diversity, which helps preserve the relevance of the representative results. Our empirical study shows that such a parameter does help improve the quality of the result set R. However, it is up to the user to define A for any query, and this may increase user efforts when using our algorithm. This motivated us to introduce a new self-adjusted parameter v to replace the manual relevance parameter A, which led to a new variation of PrefDiv called *Preferneral Diversity with Proportional Relevance* (PrefDiv-PR). As illustrated in Algorithm 2, the idea here is to automatically compute the right amount of items that should be promoted into the final result set based on the proportion of the relevance of each iteration.

In particular, v adapts to the aggregated intensity value of all items in each iteration, and can be computed as follows: Assume a given set of original results R_Q, a DT-k subset $R \subseteq R_Q$ and a number of iterations $\mathcal{L}$ needed for PrefDiv to obtain the result set R. For each iteration ℓ., s.t. $\ell < \mathcal{L}$, a set of items with the highest intensity value from the remaining items of R_Q are reserved into a separated set B_ℓ, s.t. $B_\ell \subseteq R_Q$ and $|B_\ell| = k$. The v_ℓ of an

ALGORITHM 2: PrefDiv-PR

```
Require:
 1: Initial result set $R_Q$, result cardinality $k$, a set of diversity
    constraints $\Psi$
Ensure:
 2: One subset $R$ of $R_Q$
 3: $\mathcal{L} \leftarrow 0$
 4: $T \leftarrow \emptyset$
 5: while exists unexamined items in $R_Q$ and $|R| < k$ do
 6:     $T \leftarrow$ Pick $k$ items with highest intensity from $R_Q$
 7:     for all $x_j \in T$ do
 8:         if $Dissim_\Psi(x_j, x_t) : \forall x_t \in R$ then
 9:             $R \leftarrow R \cup x_j$
10:     $R_Q = R_Q - T$
11:     $B_\mathcal{L} \leftarrow T - R$
12:     Increase $\mathcal{L}$ by one
13: for $\ell = 1 \rightarrow \mathcal{L}$ do
14:     $v_\ell \leftarrow$ Compute $v_\ell$ according to Equation 1
15:     while the number of items in $R$ from $B_\ell < v_\ell * |R|$ do
16:         $R \leftarrow R - x_j$, s.t. $x_j \in R, I_{x_j} < I_{x_k} : \forall x_k \in R$
17:         $R \leftarrow R \cup x_i$, s.t. $x_i \in B_\ell, I_{x_i} \geq I_{x_t} : \forall x_t \in B_\ell$
18:         $B_\ell \leftarrow B_\ell - x_i$
19: Return $R$
```

iteration ℓ is calculated through the following equation:

$$v_\ell = \frac{\sum_{x \in B_\ell} I_x}{\sum_{j=1}^{\mathcal{L}} \sum_{x_c \in B_j} I_{x_c}} \quad (1)$$

Recall from Section 2.1.1, I_x is the intensity value of data item x. The idea here is that at least v_ℓ proportion (i.e., percent) of the item in the final representative result set should be extracted from iteration ℓ, as early iterations would always have a higher aggregated intensity value, and thus, would occupy a bigger portion of the final representative set R. Our empirical results show that by employing v to compensate for the loss of relevance, we can prevent too many relevant results from being dropped.

To actually generate the final representative results according to v_ℓ, PrefDiv-PR needs to first run PrefDiv to obtain the initial representative set R, as well as records the number of iterations $\mathcal{L}$ taken to obtained R. During each iteration of PrefDiv, the items that are initially extracted from R_Q (before applying the diversity constraints) will also be recorded into a separate set B_ℓ. Afterward, PrefDiv-PR will examine the set of items in R that are extracted from each B_ℓ with the corresponding v_ℓ to determine if additional items need to be extracted from B_ℓ and added into R. Note that if such extraction is necessary, depending on the number of items needed to be extracted, the set of items with the highest intensity value in B_ℓ that have not been included in R will be chosen from B_ℓ and placed in R. Finally, once all v are satisfied for each iteration, the set of k items with the highest intensity value in R will be retrieved as the final results.

3.4 Optimize Diversity Constraints for Coverage

As discussed previously in Section 2.1.3, coverage is yet another important property of the representative set. It gives us two major benefits. First, it helps to ensure that the underlying data space (i.e., the original set) has been well represented by the selected representative items. Second, it enables the possibility for the user to retrieve items that are not in the representative result set by performing “zoom-in” operations—each representative item can be seen as the leader of a set of similar items, and by

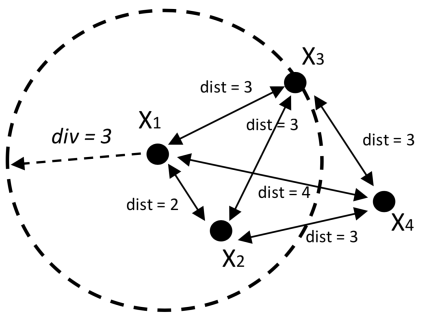


Figure 4: Illustration of the optimal radius, when $k = 2$

"zooming-in" to one of the leaders, similar items around the leader can be revealed.

Since the definition of coverage depends on the similarity between data objects, it is defined by the set of diversity constraints. In order to boost coverage, a set of appropriate diversity constraints must be defined. Below, we will discuss a general approach to determine such diversity constraints through an example.

Example 3.1. Consider a set of initial results R_Q that contains 100 items, each with two dimensions, $k = 30$, a single diversity constraint ψ that considers both dimensions, and the Euclidean distance as the diversity measure. Furthermore, assume that no partial diversity is allowed, meaning the final representative set produced must be a strict DT-k set. Obviously, whether it is possible to produce a DT-k set with 30 items is dependent on the definition of diversity constraints, such that if the diversity constraint consists of a diversity threshold that is beyond the maximum pair-wise distance between any pair of items in the original result set, then only a single item can be included in R, as the rest of the data items would be discarded due to the violation of the diversity constraint. Clearly, returning a result set with a single item when $k = 30$ is not ideal, and thus, the diversity threshold should be adjusted lower. In contrast, a minimum possible diversity threshold (i.e., 0) would lead to an arbitrary set of k items, which gives no guarantee of either diversity or coverage.

Clearly, from the example, an *optimal diversity constraint* should include a diversity threshold that exhibits the following properties: (1) be as large as possible to improve the coverage; and (2) be small enough to allow a strictly diverse representative set (i.e., DT-k set) with k mutually dissimilar items being formed.

With the above observations, we define the optimal diversity constraint as:

Definition 6. *Optimal Diversity Constraint.* For a given set of item R, an integer number k, and a distance function $dist(x_1,x_2)$, an optimal diversity constraint must contain the largest possible distance threshold, denoted as div_{opt}, that exists between a pair of items in R that can be used to generate a DT-k subset R_Q from R, such that $|R_Q| >= k$ and no two item in R_Q are similar according to $dist(x_1,x_2)$ and div_{opt}.

As illustrated in Figure 4, assume we have a set of points $P = \{x_1,...,x_4\}$, such that each point consists of a 2-D coordinate and a diversity constraint ψ that consist of both dimensions and uses Euclidean distance. In such case, if $k = 2$, then $div = 3$ will be the div_{opt} (i.e., optimal diversity threshold) for ψ. If any value less than 3 is chosen to be the div_{opt}, then at least three points will remain after removing all similar points because only p_2 and p_3 are considered to be similar with respect to a diversity threshold of 2. If any value larger than 3 is chosen to be the div_{opt}, then only one point will remain in P after removing all similar points. Thus, 3 is the only option for div_{opt}, as no other distance threshold would be able to produce a result with three items. Unfortunately, finding the optimal diversity threshold for a given distance measure and a set of attributes is NP-hard.

According to Definition 3, two items x_i,x_j are dissimilar iff they fail to satisfy a diversity constant Ψ with a diversity threshold div and distance function $dist(x_1,x_2)$. Since an item, x_i can be included in a DT-k subset R *if and only if* x_i satisfies all diversity constraints with respect to other items in R, the maximum distance d between any pair of items in R increases along with the diversity thresholds inside each diversity constraints. Based on the Definition 6 of the optimal diversity constraint, the problem of finding the optimal diversity thresholds (i.e., div_{opt}) for a given diversity constraint can be mapped to the MaxMin Diversity Problem, which aims to select a representative subset $R \subseteq R_Q$, such that $|R| = k$, and the minimum distance between any pair of items in R is maximized. As the MaxMin Diversity problem has been previously proven to be an NP-hard problem [4], finding the optimal diversity threshold is also an NP-hard problem.

Inspired by the MaxMin Diversity problem, we adopt a greedy heuristic (Algorithm 3), which automatically computes an approximation of the optimal diversity thresholds for a given set of diversity constraints. As illustrated in Algorithm 3, we first find a subset $S \subseteq R_Q$ that maximize the minimum distance between items in R (Lines 5 - 7). Then, we generate the optimal diversity threshold by comparing the pair-wise distance of all items that are in R_Q but not in S (Lines 8 - 11). The optimal diversity threshold is defined as the largest distance between a pair of items in R_Q that is smaller than the minimum distance between any pair of items in R. As proven in [30], the result produced by this greedy heuristic has a $\frac{1}{2}$ approximation of the optimal solution and a quadratic complexity, and no other polynomial algorithm can provide a better guarantee.

ALGORITHM 3: SearchOptimalDiversityThreshold

```
Require:
 1: A set of items $R_Q$, a size $k$, a set of attribute $\Delta$, and a
    distance function $dist(x_1,x_2,\Delta)$
Ensure:
 2: A diversity threshold $div_{opt}$
 3: $S \leftarrow$ an initial item $x \in R_Q$
 4: $div_{opt} \leftarrow \emptyset$
 5: while $|S| < k$ do
 6:     $x^* \leftarrow \arg\max_{x \in (R_Q - S)}(\min(dist(x,x_j,\Delta) : \forall x_j \in S))$
 7:     $S \leftarrow S \cup x^*$
 8: $\Theta \leftarrow$ minimum distance between any pair of items in S
 9: $x^R \leftarrow \arg\max_{x \in (R_Q - S)}(\min(dist(x,x_j,\Delta) : \forall x_j \in$
    $S, s.t. dist(x,x_j,\Delta) < \Theta))$
10: $div_{opt} \leftarrow \min(dist(x^R,x_j,\Delta) : \forall x_j \in R_Q)$
11: Return $div_{opt}$
```

4 EXPERIMENTAL EVALUATION

To study the effectiveness of our PrefDiv and PrefDiv-PR algorithms, we compare them to the two most effective diversified top-k algorithms, namely *Swap* [37] and *MMR* [5], as suggested in [34]. We also compare them to four diversity-focused algorithms, *MaxSum, MaxMin, K-Medoids*, and *DisC Diversity*, to assess how well diversity has been preserved when relevance is taken into account. All the algorithms in our evaluation are discussed in Section 5.

4.1 Experimental Testbed

We implemented all of the algorithms with JDK 8.0 on an Intel machine with Core i7 2.5Ghz CPUs, 16GB RAM, and 512GB SSD.

Algorithms. We implemented MMR and Swap based on their published descriptions [5] and [37], respectively. The MaxMin and MaxSum algorithms used in our experiments are based on Definitions 8 and 9 (in Section 5), respectively, and DisC Diversity is taken directly from the original author [10]. However, DisC Diversity is not top-k bounded algorithms, and the size of the result set that DisC Diversity produces is heavily dependent on the radius. To allow for a comparison, we modified the DisC Diversity to stop when the size of the result set equals k. We also included one well-known clustering algorithm, K-Medoids [26], which aims to group a set of data objects into clusters through some distance measure, so objects within a cluster are close to each other and objects outside of the cluster are unrelated to the objects inside the cluster.

In our experiment, we implemented K-Medoids based on [26]. Since K-Medoids does not capture the relevance in any regard, we improved the performance of K-Medoids in balancing the relevance *vs.* diversity trade-off by choosing the object with the highest intensity value as the final recommendation from each of its k clusters. This improvement significantly enhances the performance of the K-Medoids with respect to relevance, while exhibiting the minimum decrease in diversity.

Most of the diversification techniques involved in our experiments require some parameters, since finding the best parameters for each technique that are optimum under all situations would be too difficult. For the purpose of comparison, in all of our experiments, there is only one diversity constraint ψ for any given set of experiments, which is used by all algorithms that require a diversity constraint during its execution. We fixed the diversity threshold div used in ψ for each set of the experiments, which is computed with the optimal radius by Algorithm 3. All other parameters except ψ and r are fixed for all runs and adjusted according to the suggestion of the original authors, or based on the best overall performance. For MMR, we set $\lambda = 0.3$, for Swap, we set the $UB = 0.1$, and for PrefDiv, we set $A = 0.6$.

Datasets. We ran our tests on two real-world datasets: Cameras [10], and Foursquare. We selected these datasets in order to experiment with two different distance functions, *Hamming* with Cameras and *Euclidean* with Foursquare. The Cameras dataset consists of 579 records and 7 attributes per record. The Foursquare dataset is collected from the major location-based social network, Foursquare. We obtained real-life user preferences, used Foursquare's public venue API, and queried information for 14,011,045 venues. In order to build realistic user profiles for our evaluations, we used a dataset collected by Cheng et al. [7] that includes geo-tagged user-generated content from a variety of social media between September 2010 and January 2011. This dataset includes 11,726,632 check-ins generated by 188,450 users. Accordingly, each reading in our Foursquare dataset has the following tuple format: <ID, latitude, longitude, # check-ins, # unique users>. In our experiments, we consider only data items (i.e., venues) from New York City (NYC), which consists of 10912 items and San Francisco (SF), which consists of 7859 items.

User Preferences. The intensity values (I) for each individual dataset is generated as follows:

For the Cameras dataset, we generated 100 different sets of user preferences, such that in each profile the preference intensity value for each individual camera is generated based on a uniform distribution, and each individual user preference is represented as one unique query.

For the Foursquare dataset, we obtained the *real-life* user preferences based on the hierarchy of the Foursquare dataset, such that every individual venue v in the dataset is associated with a type T_v. For example, an Italian restaurant belongs to the category "Italian restaurant", which can belong to the higher level category "Restaurants", which can itself belong to the category "Food", and so on. In order to build highly personalized and specific profiles, we use the bottom layer of the hierarchy, as well as the specific venues visited. In particular, given the set of check-ins C_u of user u, we build a hierarchical profile $\mathcal{P}$ where at the top level, the preferences of the user are expressed in terms of the (normalized) frequencies of this user's visitations with respect to the types of venues. The second layer of the user profiles further provides the normalized frequencies of venues for the different types of locations visited by u. Since our user profile is sparsely gathered during a short period of time, to resemble a real-world user profile, we merged the 1000 sparse Foursquare user profiles to create one superuser profile. We performed our experiments by randomly selecting 50 query points from each city (100 query points in total). For each query point, we considered all venues located within a 1.5 kilometer radius of the query location.

4.2 Evaluation Metrics

In our experimental evaluation, we evaluate the performance of all models based on three well-known and commonly used metrics: *Normalized Relevance* [35], *Coverage* [10] (Definition 5), and *Execution Time*. Note, there are two other commonly used metrics for evaluating ranking algorithms such as DCG and Spearman rho. However, both metrics focus on measuring the correctness of the order of the results produced by the ranking algorithm. Thus, they are not the ideal evaluation metrics for evaluating the effectiveness of result diversification algorithms. As stated in previous sections, our proposed PrefDiv algorithms are post-processing steps of initial query results, which does not impact the relative order of the original result set. In other words, the produced representative result set of PrefDiv algorithms essentially follows the original order of the initial results. Thus, metrics that focus on the correctness of the ranking order do not fit the context of this evaluation.

Definition 7. *Normalized Relevance.* Let S be a set of items and $S_k^* \subseteq S$ such that $|S_k^*| = k$. The Normalized Relevance of a subset S_k^* is defined as the sum of the intensity value of items in S_k^* over the sum of k items with highest intensity value in S.

$$nRev(S_k^*) = \frac{\sum_{x \in S_k^*} I_x}{\max_{S_k \subseteq S, |S_k| = k} \sum_{x \in S_k} I_x} \quad (2)$$

In order to calculate the coverage for the Foursquare dataset, given that it is difficult to find a fixed radius that would work with any query location, we calculate the coverage with respect to the optimal radius generated by Algorithms 3 for every output size k. In the case of a Camera dataset, where the hamming distance is employed, the coverage is calculated with a fixed diversity threshold/radius $div = 3$ (which is the mid-point of the maximum distance allowed). For a fair comparison, all algorithms are evaluated with respect to the same diversity threshold/radius. Note that Normalized Relevance and Normalized Intensity Value would be used interchangeably in the following sections.

4.3 Experimental Results

Here we report the findings of our experimental evaluation.

4.3.1 Normalized Relevance. As demonstrated in Figures 5, 8, and 11, we can see a clear separation between two groups of algorithms for all datasets, where PrefDiv, PrefDiv-PR, Swap, MMR, and K-Medoids tend to group together, and MaxMin, MaxSum, and DisC Diversity form another group. The reason for this is that the second group does not take relevance into account; hence, it would be unlikely for them to retrieve a representative subset that has a high total intensity value. In contrast, the first group of algorithms takes relevance into account, and, as such, it

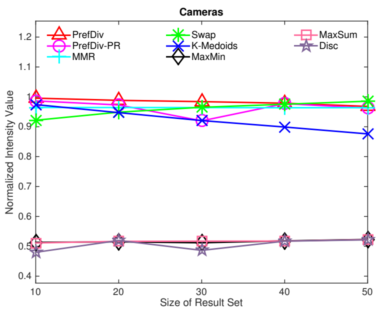


Figure 5: Normalized Intensity Value of Cameras

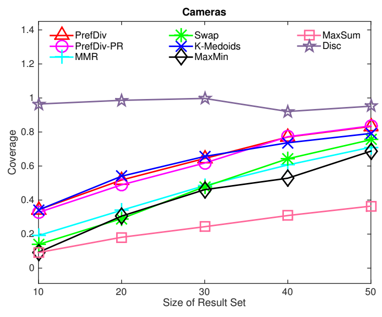


Figure 6: Coverage of Cameras.

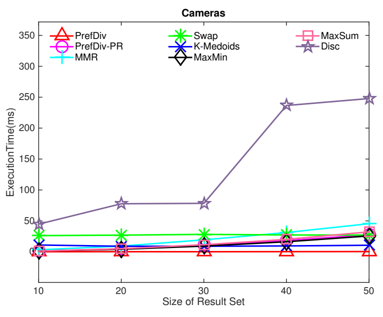


Figure 7: Execution Time of Cameras.

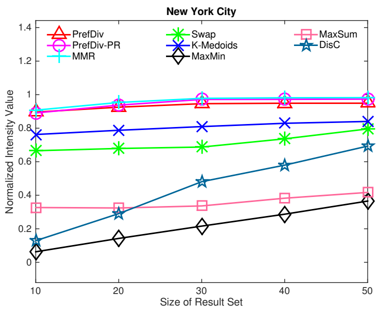


Figure 8: Normalized Intensity Value for the Foursquare, NYC

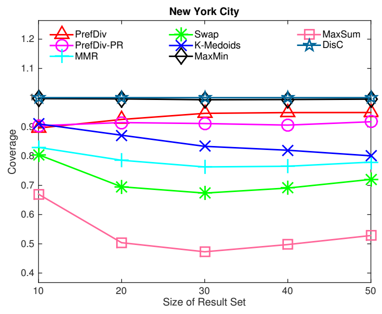


Figure 9: Coverage of Foursquare, NYC

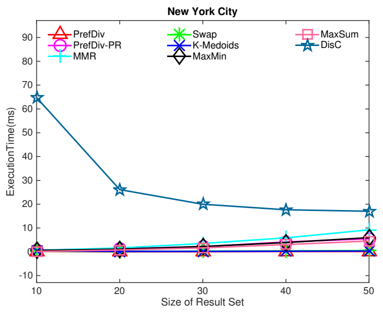


Figure 10: Execution Time for the Foursquare, NYC

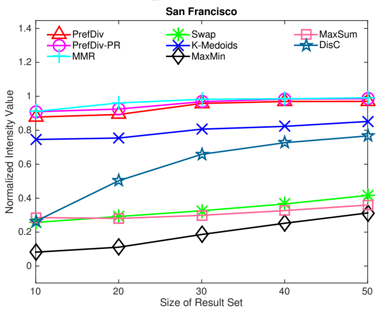


Figure 11: Normalized Intensity Value of Foursquare, SF

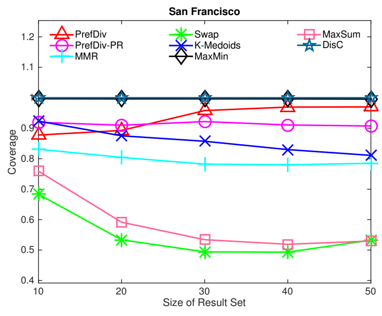


Figure 12: Coverage of Foursquare, SF

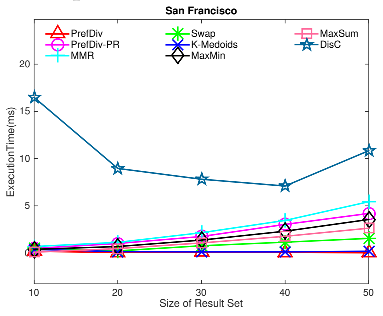


Figure 13: Execution Time of Foursquare, SF

achieves a significantly higher performance in terms of retrieving relevant items.

4.3.2 Coverage. Figures 6, 9, and 12 show that our PrefDiv and PrefDiv-PR exhibit better coverage on average when compared with MMR and Swap by 20% and 42%, respectively. The reason could be because both MMR and Swap optimize dissimilarity as their definition of diversity. In contrast, both PrefDiv and PrefDiv-PR are coverage-aware algorithms, which seek an optimal radius that directly improves the coverage of the representative result set. Therefore, both PrefDiv and PrefDiv-PR can perform much better than Swap and MMR. We also observed that on average PrefDiv is able to outperform MaxSum in terms of coverage by 160%, which could be explained because MaxSum as pure dissimilarity-based algorithm fails to cover the entire space of the dataset. K-Medoids demonstrates good coverages for both datasets. However, both PrefDiv and PrefDiv-PR still exhibit slightly better coverage in general.

The MaxMin algorithm performs well in terms of the Foursquare dataset, although the performance dropped significantly for the cameras dataset. The reason could be because in the Foursquare dataset the average number of venues around each query point is about 90 venues. In contrast, the Cameras dataset consists of 579 tuples. This shows that MaxMin is able to obtain good coverage with a relatively small dataset and Euclidean distance that takes a wide range of values as distance, but fails to cover the space with large datasets and hamming distance that only takes the number of attributes + 1 distinct values as distance. In both datasets, DisC Diversity demonstrated the highest coverage, which is to be expected since DisC Diversity is the only algorithm in the experiment that directly optimizes coverage as the only objective.

Another interesting observation is that, in the Foursquare dataset, except PrefDiv, PrefDiv-PR, and DisC Diversity, other algorithms appear to have a drop in coverage when the value of k increases, although, in general, with the increase in result size the coverage should increase as well. The reason for such behavior is that in the Foursquare dataset, we employed the optimal radius as the criterion for determining the similarity between items. Therefore, with the increase in result size, the optimal radius becomes smaller, thus leading to a decrease of coverage for some algorithms.

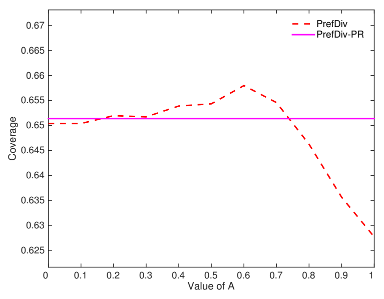


Figure 14: Coverage of different settings of A, with optimal radius and k = 30.

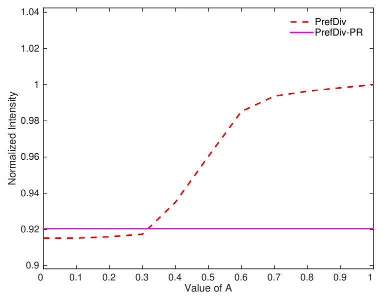


Figure 15: Normalized Relevance of different settings of A, with optimal radius and k = 30.

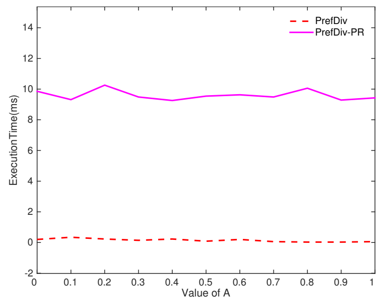


Figure 16: Execution Time of different settings of A, with optimal radius and k = 30.

4.3.3 Execution Time. We have measured the execution time required by all algorithms. As shown in Figures 7, 10, and 13, our PrefDiv and PrefDiv-PR appears to be the overall fastest algorithm when compared to all other alternatives. In general, PrefDiv and PrefDiv-PR perform near identically in terms of runtime, which is expected as the additional computation overhead introduced in PrefDiv-PR is negligible. To illustrate the efficiency of our proposed greedy heuristic for searching the optimal diversity threshold, we have included its runtime in the PrefDiv-PR, so the runtime difference between PrefDiv and PrefDiv-PR reflects the runtime of the search optimal diversity threshold algorithm. With that in mind, PrefDiv-PR is still on average faster than both MMR and Swap by up to 72% and 116%, respectively. Specifically, in the Camera dataset, PrefDiv is able to execute 57 times faster than K-Medoid, 127 times faster than MMR, and 159 times faster than Swap. In the Foursquare dataset, most algorithms tend to be faster when compared with Cameras, because the number of venues near each query point in Foursquare is much smaller than in the Cameras dataset. However, we still observed that, on average, PrefDiv is able to outperform MMR and Swap by 30 and 36 times, respectively. When compared to K-Medoid, which is also very efficient when dealing with this type of dataset, PrefDiv still appears to be 2.7 times faster than K-Medoid on average. As mentioned previously, if the optimal radius for most frequent queries is stored, PrefDiv-PR would not need to calculate the optimal radius for these queries again. Furthermore, for the fairness of the comparison, all of the algorithms run in a single-threaded mode. Since the optimal radius computation method that we adopted is fully parallelizable, it can take advantage of the modern multi-threaded CPU architecture to speed up the computations. In fact, we have observed linear speed up with respect to the number of CPU cores in the system up to 16 cores (the highest we have experimented). One interesting remark here is that, in the Foursquare dataset, the execution time of DisC Diversity drops when k increases from 10 to 20. The reason is that the runtime of DisC Diversity is also affected by the length of the radius, therefore, with smaller output sizes, the optimal radius will become larger, which leads to the relativity longer execution time of DisC.

4.3.4 Parameter A of PrefDiv. As illustrated in Figures 6 to 11, a performance difference between PrefDiv and PrefDiv-PR exists due to the existence of the accuracy parameter A in the PrefDiv algorithm. In this section, we conducted an experiment to study the effect of parameter A in PrefDiv with the Camera dataset and k = 30. As shown in Figures 14, 15, and 16, when A increases from 0 to 1, we observed an improvement of normalized relevance, albeit with a decrease in coverage. This is as expected since, with higher values of A, PrefDiv will select more relevant items, and with lower values of A, more diverse items will be selected that lead to an increase in coverage. However, this would be at the expense of lower relevance. The execution time appears to be stable regardless of the value of A. This is because, for each iteration, PrefDiv only requires a tiny amount of execution time. Therefore, the additional iterations introduced by the low value of *A* would not have a large impact on the overall runtime.

4.3.5 Relevancy vs. Diversity. Lastly, as a summary, we present three scatterplots that capture the trade-off between relevance and diversity. Each point in Figures 17 and 18 are corresponding to the average of over 50 different query locations with one value of k, and each point in Figures 19 are corresponding to the average of over 100 different user profiles with one value of k. As shown in the figures, we have Normalized Intensity Value as the y-axis and Coverage as the x-axis. Algorithms located in the upper left corner of the figure exhibit the best coverage result, while those in the lower right corner have the highest relevance scores. As we can observe, both PrefDiv and PrefDiv-PR are located towards the upper right corner (circled) for all three scatter plots, which indicates that both PrefDiv and PrefDiv-PR exhibit better ability to handle the trade-off between relevance and diversity with respect to both datasets and distance measures. One may notice that in the Camera dataset, the advantage of PrefDiv and PrefDiv-PR with respect to other alternatives is relatively smaller compared to that in the Foursquare dataset. This is because the Cameras dataset uses the Hamming distance as the distance measure, which has a much smaller domain than the Euclidean distance used in the Foursquare dataset, thus weakening the benefit of the optimal diversity threshold. These results also indicate that the relational proportionality introduced in PrefDiv-PR does effectively improve the quality of the result, since PrefDiv-PR is able to outperform (although slightly) the PrefDiv with manually configured relevance parameter *A*.

4.3.6 Additional Observations. Despite the fact that both PrefDiv and PrefDiv-PR run up to 159 times faster than other alternatives, the greedy heuristic (Algorithm 3) that we proposed for finding the optimal diversity threshold/radius runs at a quadratic time complexity, and thus, it is much slower. Although it is not required to run this heuristic before each execution of the PrefDiv/PrefDiv-PR, it certainly helps improve its performance. However, this is not an issue with our PrefDiv/PrefDiv-PR algorithm because other algorithms (e.g., DisC Diversity) also benefit from the optimal diversity threshold as much as PrefDiv/PrefDiv-PR.

Fortunately, this greedy heuristic only needs to run once for each query, and thus, it can simply be cached to boost the runtime of frequent queries significantly.

5 RELATED WORKS

In this section, we discuss works that are closely related to ours from multiple aspects.

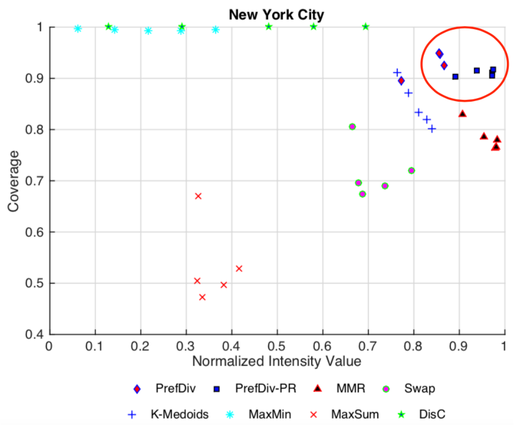


Figure 17: Relevance VS. Diversity (NYC).

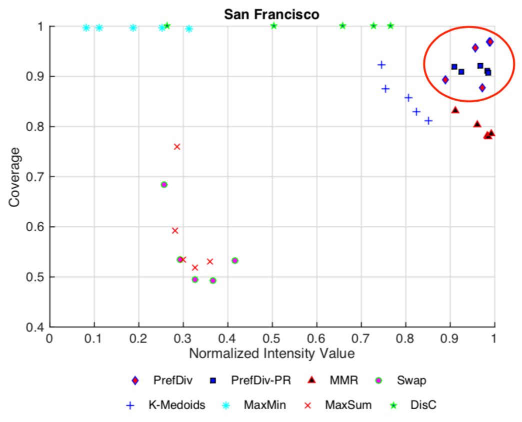


Figure 18: Relevance VS. Diversity (SF).

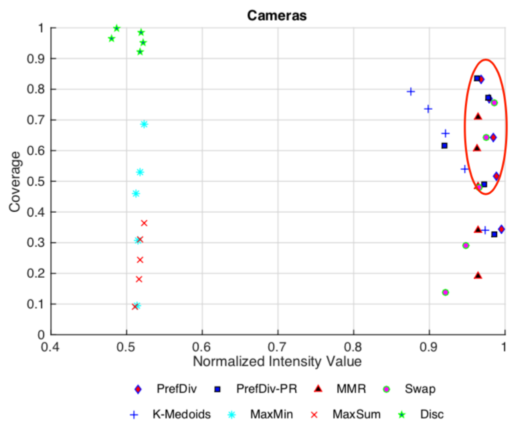


Figure 19: Relevance VS. Diversity (Cameras).

5.1 Relevance Ranking Techniques

Many ranking techniques using preference have been proposed. These are comprehensively surveyed in Stefanidis et al. [33]. As mentioned above, these techniques can be distinguished based on the type of preferences they support for filtering and ordering data. These techniques primarily handle only one type of preference, either *quantitative* or *qualitative*. However, each preference type has its own advantages and disadvantages. Hybrid schemes that support both qualitative and quantitative preferences have been proposed in an attempt to exploit the advantages of both types of preferences while eliminating their disadvantages [17, 22]. In this work, our proposed algorithms can work with any existing relevance ranking model that returns a set of sorted tuples/objects along with their scores/intensity values.

More recently, in [6], the author studied the problem of producing rankings while preserving a given set of fairness constraints. In particular, the proposed algorithm takes as input, a utility function, a collection of sensitive attributes (e.g., gender, race), and a collection of fairness constraints that restrict the number of items with each sensitive attribute that are allowed to appear in the final results. It outputs a ranking that maximizes the relevance with respect to the given utility function while respecting the fairness constraints. As mentioned previously, our proposed PrefDiv algorithms can leverage any existing relevance ranking model. Therefore, in the case where the required sensitive attributes and fairness constraints can be provided by the user, PrefDiv can be used in conjunction with the ranking produced in [6].

5.2 Diversity Techniques

Result diversification has been studied in many different contexts and with various definitions [10], such as similarity, semantic coverage [2], and novelty [8]. In our work, we focus on the similarity definition and use MaxMin and MaxSum, which are two widely used diversification models, as baselines.

The goal of these two diversification models is to select a subset S from the object space R, so that the minimum or the total pairwise distances of objects in S is maximized. Recently, a number of variations of the MaxMin and MaxSum diversification models have also been proposed (e.g., [9, 25]) to address the problem of diversifying continuous data. Formally, MaxMin and MaxSum are defined as follows:

Definition 8. *MaxMin* generates a subset of R with maximum $f = min_{p_i,p_j \in S} dt(p_i,p_j)$ where dt is some distance function $p_i \neq p_j$ for all subsets with the same size.

Definition 9. *MaxSum* generates a subset of R with maximum $f = \Sigma_{o_i,o_j \in S} dt(o_i,o_j)$ where dt is some distance function $o_i \neq o_j$ for all subsets with the same size.

DisC Diversity [10] is the most recently proposed diversity framework and solves the diversification problem from a different perspective. In DisC Diversity, the number of retrieved diverse results is not an input parameter. Instead, users define the desired degree of diversification in terms of a tuning parameter r (radius). DisC Diversity considers two objects o_i and o_j in the query result R to be similar objects if the distance between o_i and o_j is less than or equal to a tuning parameter r (radius). It selects the representative subset $S \in R$ according to the following conditions: (1) For any objects in R, there should be at least one similar object in S; and (2) All objects in S should be dissimilar to each other. These two conditions ensure both the coverage and the dissimilarity property of a diverse result set.

In addition, DisC Diversity also introduces two problems, *Covering* and *CoveredBy* [11]. These can be used to model the issue of generating a representative result set that is both diverse and relevant to a user's individual preference (without using preferences). The Covering problem is used to model the case where users want highly relevant items to cover a large area around them. In order to achieve this goal, a relatively larger radius is assigned to items with larger weights. The CoveredBy problem is used to model a case where a user wants to see more relevant objects. In that case, a smaller radius is assigned to items with larger weights. These two problems together illustrate the possibility of using DisC to handle relevance together with diversity.

The key differences between PrefDiv algorithms and DisC Diversity are: (1) PrefDiv algorithms follow the Top-k paradigm, which provides users with the option to specify the size of the final result set by assigning a value to parameter k, whereas DisC Diversity adjusts the size of the result set by changing its radius parameter r. (2) The PrefDiv algorithms focus on both the *relevance* of the result set with respect to the users' preference and the *diversity* of the result set. DisC Diversity focuses mainly on the most diverse representative subset with two scenarios that only illustrate the possibility of using DisC Diversity to handle such relevance-aware diversity requests; however, they do not mention any specific strategies on how one can dynamically change *r* with respect to Covering or CoveredBy. In addition, our implementation of PrefDiv-PR eliminates the need for identifying *r* (i.e., diversity threshold) manually by automatically finding the most suitable diversity threshold under any given situation.

Another way to generate a diverse, representative set of results is through clustering. One example of this would be *k-Medoids*, which is a well-known clustering algorithm that attempts to minimize the distance between points in a cluster and the center point (medoid element) of that cluster. The k-Medoids algorithm can be classified into two stages: In its first stage, it generates a set of k clusters C = {c1, c2, ..., ck} based on some distance function dt. In the second stage, one element from each cluster is selected to be part of the result set *R*. Several strategies for selecting an element from each cluster could be employed. For instance, one strategy is to choose the center point of each cluster that is expected to deliver high diversity, and another strategy would be to choose the point that has the highest intensity value for each cluster. However, since there is no parameter that can be tuned, either manually or automatically, to balance the trade-off between relevance and diversity, k-Medoids is unable to balance such a trade-off in fine granularity.

5.3 Multi-Criteria Objective Optimization

In the past, diversification and retrieval of relevant results have often been studied together as a multi-objective optimization problem with two objectives, where the first objective is relevance, and the second objective is dissimilarity [38]. The following are some representative techniques that are related to our work.

In [27], the authors considered the optimization of the diversified Top-K problem as finding the optimal solution for the maximum weight independent set problem, which has been proven to be an NP-hard problem. The authors proposed an approach, called *div-astar*, which uses a diversity graph that consists of N nodes, where each node corresponds to one item in the original data. This diversity graph is sorted according to the relevance score, and an a^* algorithm is used to find the optimal solution for diversifying Top-K Results. In addition to the div-astar solution, two enhancements have also been proposed, called *div-dp* and *div-cut*: div-dp takes advantage of dynamic programming to divide the initial graph into disconnected components, and div-cut is a cutpoint-based approach that further decomposes each disconnected component based on loosely connected sub-graphs. PrefDiv algorithms are different from div-astar [27] (Section 5.3), in that the main objective of div-astar is to find the exact solution for the maximum weight independent set; hence, even with all the enhancements and decompositions, each sub-problem is still NP-hard. On the other hand, although PrefDiv algorithms also consider the maximum weight independent set problem as part of the algorithm, they take advantage of greedy approximation with a relaxed constraint, which allows similar items to be included in the result set if the relevance distribution of the original data can be better reflected in the resulting set. Furthermore, such relaxed constraints allow PrefDiv to be more practical for border usage, especially for tasks that require a short response time.

One widely used approach that was targeted directly at optimizing the trade-off between diversity and relevance was introduced by [5]. In this work, the authors proposed the famous twin-objective function called *Maximal Marginal Relevance* (MMR), which combines both relevance and diversity aspects in a single, comprehensive objective function. Formally, MMR defines its objective function as:

$$\arg\max_{D_i \in R \setminus S}[\lambda(Sim_1(D_i, Q) - (1-\lambda)\max_{D_j \in S} Sim_2(D_i, D_j))] \tag{3}$$

Where λ is a scaling factor that specifies the preference between relevance and diversity. When $\lambda = 1$, the MMR function equals a standard relevance ranking function. When $\lambda = 0$, it computes a maximal diversity ranking. Comparing PrefDiv to MMR [5] approach, one can clearly see the difference: there are no comprehensive objective functions being used in the PrefDiv algorithms. Our approach addresses the combined problem of relevance and diversity through a combination of multiple steps, rather than solving it in one single function.

Recently, a new bi-criteria objective optimization approach based on MMR has been proposed [19]. This approach integrates *regret minimization* with traditional MMR to generate a new relevance score that takes into consideration the case of minimizing the disappointment of users when they see k representative tuples rather than the whole database. In this work, the authors proposed two approximation algorithms called *ReDi-Greedy* and *ReDi-SWAP*, which find the set of items consisting of k items having the highest score with respect to their MMR function.

In [32], the author has conducted a study on personalized, keyword-based search over relational databases, which includes the notion of diversity and coverage. Specifically, the author provided good discussions on modeling the relevance, user preferences, diversity, and coverage for keyword-based searches over relational databases by means of Join Tree of Tuples. Join Tree are trees of tuples connected through primary to foreign key dependencies. However, PrefDiv algorithms assume that a utility function F is given in advance to reflect the relevance and user preference, and thus does not focus on modeling the relevance and user preferences. Furthermore, PrefDiv algorithms are general, post-processing techniques for result diversification, and hence, do not restrict themselves to the keyword-based search over relational database settings. As long as proper utility functions and distance measures are given, PrefDiv algorithms can be applied to any data types (e.g., structured, unstructured, semi-structured). Consequently, the definition of coverage in [32] is also different than the definition of coverage in this work. Wheres [32] focuses on covering more user intents based on user profiles, PrefDiv algorithms focus on the proximity between the representative results and original results.

Swap is another recent Top-K diversification technique that is related to ours [37]; Swap starts with K items with the highest relevance scores. Among these K items, Swap picks an item with the lowest contribution to the diversity of the entire set, then swaps this item with the item that has the next highest relevance score. A candidate is successfully swapped with one of the items in the Top-K set if and only if it can contribute more in terms of the overall diversity of the result set. In order to preserve the relevance aspect, Swap introduces an optional pre-defined threshold called UB that specifies how much decrease in relevance can be tolerated. UB can serve as a terminal condition that stops the algorithm when the item with the highest relevance among the remaining set is no longer high enough for the algorithm to perform a swap operation. Our PrefDiv is different from the Swap, as Swap seeks diversity through pairwise distances of items among the result set, filters out items that contribute less to diversity, and ensures relevance by throwing out items that cause the relevance to drop below the pre-defined threshold. In contrast, PrefDiv algorithms seek diversity by eliminating similar items and ensuring relevance by using a relevance-focused greedy algorithm along with proportionality, which can reflect the relevance distribution of the original domain.

5.4 Data Summarization

Some recent works [24, 36] have studied the problem of providing interactive exploration and summarization support for tuples in a given table. The goal of this type of approach is to produce an informative hierarchy that organizes the underlying tuples essentially in k clusters. In order to display tuples as clusters, each cluster is folded into a single, representative tuple, with only the common attribute values among all members of the cluster being displayed. The rest of the attributes are shown as "?", which indicates that there are objects with different values with respect to these attributes inside the cluster. To explore each cluster, the user can gradually expand each "?" symbol contained in the current representative tuple of a cluster. Each time the user expands a "?" symbol, more tuples that contain a different value with respect to the corresponding attributes will be displayed. Clearly, these works are different than ours. We focus on producing a representative subset that is most informative to the user with adjustable size, rather than summaries of subsets of a table.

5.5 Impacts of PrefDiv

The efficiency of PrefDiv and its ability to balance the trade-offs between relevance, diversity, and coverage have already benefited the design of some real-world systems that need to produce highly informative representative subsets, or require interactive efficiency in producing the representative results (e.g., [14–16, 28, 29]) .

One example is a novel mobile recommendation service that provides a set of diverse points-of-interest (POI's) recommendations [14–16], where the interactive efficiency has been weighted equally important as the quality of the produced recommendations.

Another example is in the scientific domain and dimensionality reduction, which PrefDiv has been employ as a novel way to select subsets of highly informative dimensions for high-dimensional gene expression datasets [28, 29]. Those selected dimensions will then be used to enable effective downstream analysis in a variety of medical and bioinformatics researche.

6 CONCLUSIONS

Scalability from a human point of view is a very challenging problem as it consists of finding the perfect balance between the conflicting objectives of relevance and diversity. Traditional top-k result diversification approaches focus on producing a subset of results that balance the trade-off between selecting highly relevant items and items that are dissimilar with respect to each other. In order to achieve the above-mentioned objectives, most algorithms rely on a number of tunable control parameters, making them harder to configure (and be adopted). Coverage is another important factor of diversity, which has been mostly ignored in previous top-k result diversification algorithms.

In this work, we addressed these problems and proposed an efficient online solution called *Preferential Diversity* (PrefDiv). PrefDiv produces a set of high-quality representative items from a large set of initial answers, where each representative item is chosen to optimize both the *relevance* and *diversity* (i.e., dissimilarity, and coverage). We also proposed a number of optimizations that further improve PrefDiv's usability, efficiency, and effectiveness.

We theoretically analyzed and experimentally compared our algorithms to the state-of-the-art, top-k diversification algorithms. Our evaluation showed that our algorithms achieve similar performance in terms of normalized relevance, but outperforms the state-of-the-art algorithms in terms of coverage by a noticeable margin, while achieving a speedup of the runtime up to two orders of magnitude.

REFERENCES

[1] Independent set (graph theory). https://en.wikipedia.org/wiki/Independent_set_(graph_theory), 2018.

[2] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. In *WSDM*, pages 5–14, 2009.

[3] A. Angel and N. Koudas. Efficient diversity-aware search. In *ACM SIGMOD*, pages 781–792, 2011.

[4] J. Carbonell and J. Goldstein. The discrete p-dispersion problem. *EJOR*, 46:48–60, 1990.

[5] J. Carbonell and J. Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *ACM SIGIR*, pages 335–336, 1998.

[6] L. E. Celis, D. Straszak, and N. K. Vishnoi. Ranking with fairness constraints. In *ICALP*, pages 28:1–28:15, 2018.

[7] Z. Cheng, J. Caverlee, K. Lee, and D. Sui. Exploring millions of footprints in location sharing services. In *ICWSM*, 2011.

[8] C. L. Clarke, M. Kolla, G. V. Cormack, O. Vechtomova, A. Ashkan, S. BÃijttche, and I. MacKinnon. Novelty and diversity in information retrieval evaluation. In *ACM SIGIR*, pages 659–666, 2008.

[9] M. Drosou and E. Pitoura. Dynamic diversification of continuous data. In *EDBT*, pages 216–227, 2012.

[10] M. Drosou and E. Pitoura. Result diversification based on dissimilarity and coverage. In *VLDB*, pages 13–24, 2012.

[11] M. Drosou and E. Pitoura. Multiple radii disc diversity: Result diversification based on dissimilarity and coverage. *ACM TODS*, 40(1), 2015.

[12] P. Fraternali, D. Martinenghi, and M. Tagliasacchi. Top-k bounded diversification. In *ACM SIGMOD*, pages 421–432, 2012.

[13] X. Ge, P. K. Chrysanthis, and A. Labrinidis. Preferential diversity. In *ExploreDB*, 2015.

[14] X. Ge, P. K. Chrysanthis, and K. Pelechrinis. Mpg: Not so random exploration of a city. In *IEEE MDM*, 2016.

[15] X. Ge, A. Daphalapurkar, M. Shimpi, D. Kohli, K. Pelechrinis, P. K. Chrysanthis, and D. Zeinalipour-Yazti. Data-driven serendipity navigation in urban places. In *IEEE ICDCS*, pages 2501–2504, June 2017.

[16] X. Ge, S. R. Panati, K. Pelechrinis, P. K. Chrysanthis, and M. A. Sharaf. In search for relevant, diverse and crowd-screen points of interests. In *EDBT*, 2017.

[17] R. Gheorghiu, A. Labrinidis, and P. K. Chrysanthis. A user-friendly framework for database preferences. In *CollaborativeCom*, pages 205–214, 2014.

[18] M. Grohe. Descriptive and parameterized complexity. In *Computer Science Logic, 13th Workshop, number 1683 in LNCS*, pages 14–31, 1999.

[19] Z. Hussain, H. A. Khan, and M. A. Sharaf. Diversifying with few regrets, but too few to mention. In *ExploreDB*, pages 27–32, 2015.

[20] H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi. Big data and its technical challenges. *Commun. ACM*, 57(7):86–94, July 2014.

[21] J. M. Keil, J. S. B. Mitchell, D. Pradhan, and M. Vatshelle. An algorithm for the maximum weight independent set problem on outerstring graphs. In *CCCG*, 2015.

[22] W. Kiessling and G. Kostler. Preference SQL: design, implementation, experiences. In *VLDB*, pages 990–1001, 2002.

[23] A. Labrinidis. The big data - same humans problem. In *Proc. of Conference of Innovative Data Systems Research*, 2015.

[24] A. P. Manas Joglekar, Hector Garcia-Molina. Interactive data exploration with smart drill-down. In *IEEE ICDE*, 2016.

[25] D. Panigrahi, A. D. Sarma, G. Aggarwal, and A. Tomkins. Online selection of diverse results. In *WSDM*, pages 263–272, 2012.

[26] H.-S. Park and C.-H. Jun. A simple and fast algorithm for k-medoids clustering. *Expert Systems with Applications*, 36(2):3336–3341, 2009.

[27] L. Qin, J. X. Yu, and L. Chang. Diversifying top-k results. In *VLDB*, pages 1124–1135, 2012.

[28] V. K. Raghu, X. Ge, A. Balajee, D. J. Shirer, I. Das, P. V. Benos, and P. K. Chrysanthis. A pipeline for integrated theory and data-driven modeling of genomic and clinical data. In *ACM BioKDD*, 2019.

[29] V. K. Raghu, X. Ge, P. K. Chrysanthis, and P. V. Benos. Integrated theory-and data-driven feature selection in gene expression data analysis. In *IEEE ICDE*, pages 1525–1532, 2017.

[30] S. S. Ravi, D. J. Rosenkrantz, and G. K. Tayi. Facility dispersion problems: Heuristics and special cases. In *WADS*, 1991.

[31] R. L. T. Santos, C. Macdonald, and I. Ounis. Search result diversification. In *Foundations and Trends in Inf Retrieval*, volume 9, pages 1–90, 2015.

[32] K. Stefanidis, M. Drosou, and E. Pitoura. Perk: personalized keyword search in relational databases through preferences. In *EDBT*, 2010.

[33] K. Stefanidis, G. Koutrika, and E. Pitoura. A survey on representation, composition and application of preferences in database systems. *ACM TODS*, 36(19), 2011.

[34] D. C. Thang, N. T. Tam, N. Q. V. Hung, and K. Aberer. An evaluation of diversification techniques. *LNCS*, 9262:215–231, 2015.

[35] H. Tong, J. He, Z. Wen, R. Konuru, and C.-Y. Lin. Diversified ranking on large graphs: an optimization viewpoint. In *ACM KDD*, pages 1028–1036, 2011.

[36] Y. Wen, X. Zhu, S. Roy, and J. Yang. Interactive summarization and exploration of top aggregate query answers. In *VLDB*, 2018.

[37] C. Yu, L. Lakshmanan, and S. Amer-Yahia. It takes variety to make a world: Diversification in recommender systems. In *EDBT*, pages 368–378, 2009.

[38] C.-N. Ziegler, J. A. K. Sean M. McNee, and G. Lausen. Improving recommendation lists through topic diversification. In *WWW*, pages 22–32, 2005.

Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins

Andreas Kipf Harald Lang Varun Pandey Raul Alexandru Persa
Christoph Anneser Eleni Tzirita Zacharatou[*] Harish Doraiswamy[◇]
Peter Boncz[★] Thomas Neumann Alfons Kemper
TUM TU Berlin[*] NYU[◇] CWI[★]
{kipf, langh, pandey, raul.persa, anneser, neumann, kemper}@in.tum.de
eleni.tziritazacharatou@tu-berlin.de harishd@nyu.edu boncz@cwi.nl

ABSTRACT

Connected mobility applications rely heavily on geospatial joins that associate point data, such as locations of Uber cars, to static polygonal regions, such as city neighborhoods. These joins typically involve expensive geometric computations, which makes it hard to provide an interactive user experience.

In this paper, we propose an adaptive polygon index that leverages *true hit filtering* to avoid expensive geometric computations in most cases. In particular, our approach closely approximates polygons by combining quadtrees with true hit filtering, and stores these approximations in a query-efficient radix tree. Based on this index, we introduce two geospatial join algorithms: an approximate one that guarantees a user-defined precision, and an exact one that adapts to the expected point distribution. In summary, our technique outperforms existing CPU-based joins by up to two orders of magnitude and is competitive with state-of-the-art GPU implementations.

1 INTRODUCTION

Connected mobility companies need to process vast amounts of location data in near real-time to run their businesses. For example, Uber needs to map locations of cars and passenger requests (points) to predefined zones (polygonal regions) for allocation and dynamic pricing purposes [40]. These polygonal regions are typically largely disjoint (non-overlapping) and mostly static. Points, on the other hand, are often not known a priori. Thus, the problem is how to efficiently find the polygons that contain an incoming point.

Traditionally, such point-polygon joins [19] follow the *filter and refine* approach. In this two-phase evaluation strategy, the filtering phase typically uses an index (e.g., an R-tree) on the minimum bounding rectangles (MBRs) of polygons and probes the index for each point to obtain a list of candidate join pairs. Then, in the refinement phase, expensive point-in-polygon (PIP) tests are performed to discard false matches.

We argue that the time has come to rethink this strategy: First, main memory is not a scarce resource anymore and modern machines offer multiple terabytes of memory. Combined with the city-centric model of geospatial applications (e.g., Uber), we show that it is possible to maintain highly fine-grained indexes for entire cities (e.g., Uber's operating zones) in main memory, dramatically reducing the number of CPU-intensive PIP tests. Second, geospatial positions, nowadays typically obtained by smartphones or wearables, are inherently imprecise [41]. Thus, we argue that it is in many cases admissible to trade off accuracy for performance. Based on these two insights, we transform the traditionally CPU-intensive problem of point-polygon joins into one that is bound by memory access latencies.

In contrast to the classical filter and refine approach, *true hit filtering* [9] identifies actual join pairs already in the filtering phase, and thus partially avoids expensive refinements. This is achieved by using additional approximations (such as inner rectangles [20]) to approximate the interior of polygons, so that when a point falls into an interior approximation, it can be safely deducted that the point is contained in the polygon.

Building on this seminal idea, we present an improved algorithm that combines true hit filtering with quadtrees [23] to *holistically* index an entire set of polygons. This is in contrast to existing implementations of true hit filtering that approximate polygons individually [15, 21] or use non-hierarchical (single-resolution) grids [6, 39, 49]. In our approach, polygons are translated into a *single* set of multi-resolution grid cells that approximates their boundary and interior areas. To support efficient queries, we store one-dimensional identifiers of the cells in a new in-memory radix tree (trie) named *Adaptive Cell Trie* (ACT). We show that ACT is more query-efficient than previous approaches for indexing cell identifiers (e.g., B-trees, like in [21]).

Another distinguishing feature of our approach is that it can entirely avoid the expensive refinement phase by refining cells in the boundary areas until a user-defined precision is guaranteed. Naturally, this comes at the cost of higher memory consumption than traditional filter and refine approaches. However, as stated above, we argue that we can nowadays actually afford this higher memory consumption in exchange for higher performance.

Our approach can also provide accurate results by performing expensive PIP tests for points that are potential hits. To reduce their number, we adapt (train) our index based on historical data points to provide higher precision where it is actually needed. As we show in our experiments, our accurate algorithm performs very few PIP tests. Compared to a filter based on the polygons' MBRs, our index (trained with 1 M historical points) reduces the number of required PIP tests by >97% for a join between NYC taxi pick-up locations and neighborhood polygons. This algorithm can also be used when ACT cannot guarantee the desired precision given a certain memory budget.

In summary, we make the following contributions:

- An algorithm that computes quadtree-based grid approximations for sets of polygons with precision guarantees
- A radix tree data structure (ACT) that is optimized for indexing cell identifiers: for a join of NYC's yellow taxi data with NYC's neighborhoods, we achieve a throughput of >50 M points/s per CPU core under a <4 m precision bound

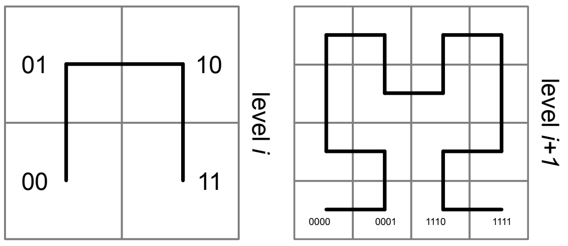


Figure 1: Quadtree-based cell decomposition and Hilbert curve-based enumeration.

- An evaluation of ACT in contrast to more traditional data structures, such as B-trees
- An accurate algorithm that trains the index structure based on historical data points
- An experimental comparison against state-of-the-art GPU-based point-polygon joins

In the remainder of this paper, we first give some background about the building blocks of our approach in Section 2. Section 3 describes our approach and Section 4 presents the evaluation with real-world and synthetic data. Finally, we summarize related work in Section 5 before concluding in Section 6.

2 BACKGROUND

Location Discretization. Our approach relies on a quadtree-based (hierarchical) decomposition of space (the surface of the Earth in this case). This decomposition is static and thus *data independent*. We enumerate the quadtree cells using a space-filling curve (e.g., the Hilbert or the Z curve) to index them in a one-dimensional data structure. Our approach does not depend on a concrete space-filling curve. For our indexing strategy to work, the cell enumeration must only fulfill the property that child cells share a common prefix with their parent cell.

Figure 1 shows the hierarchical decomposition of two cells at levels i and $i+1$ and the corresponding bitwise representations that encode the cells' positions along the Hilbert curve. Each cell consists of four sub cells, which it completely covers. Child cells share a common prefix with their parent cell, allowing us to compute *contains* relationships using efficient bitwise operations. In our implementation, we use the Google S2 library [32] for mapping latitude/longitude coordinates to 64-bit cell identifiers, which we call *cell ids* in the following. A cell id encodes up to 30 levels with two bits per level.

Polygon Approximations. To obtain fine-grained polygonal approximations, we need a method that maps polygons to sets of quadtree cells (possibly at different levels). In particular, our algorithms take as input approximations of the boundary and interior areas of *single* polygons. In our implementation, we use the S2 library to obtain these individual polygon approximations. Figure 2 illustrates a covering (in blue) and an interior covering (in green) of a polygon. A point contained in a covering cell is either within or outside of the polygon while points that match interior cells are known to be within the polygon (true hits). The cell marked with ① is one of the largest covering cells and only minimally intersects the polygon. Any point contained in this cell has at most a distance of $\sqrt{2} * \epsilon$ (with ϵ being the side length of the cell) to the polygon. To allow for an efficient search, S2 stores the cell ids of a covering in a sorted vector. Besides sorting the cell id vector, it allows for the covering to be *normalized*. A normalized covering contains neither conflicting nor duplicate cells. Two cells are conflicting when one cell contains the other. Only when the covering is normalized can cell containment checks be efficiently implemented using a binary search on the sorted vector ($O(\log n)$).

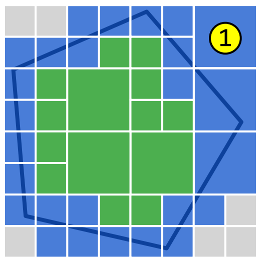


Figure 2: A covering (blue cells) and an interior covering (green cells) of an individual polygon.

While binary search on a sorted vector is a good strategy for querying small collections of cells (e.g., the covering cells of a single polygon), it is not the most efficient way to search larger collections (e.g., coverings of multiple polygons). In this work, we store large cell collections in ACT, a query-efficient radix tree, and evaluate its performance compared to alternative physical representations (including a sorted vector and a B-tree).

PIP Test. A point-in-polygon (PIP) test determines whether a point lies within a polygon. Typically such a test is performed using complex geometric operations, such as the *ray-tracing algorithm* [17], which involves drawing a line from the query point to a point that is known to be outside of the polygon and counting the number of edges that the line crosses. If the line crosses an odd number of edges, the query point lies within the polygon. The runtime complexity of this algorithm is $O(n)$, n being the number of edges. While there are many conceptual optimizations to the PIP test, this operation remains computationally expensive since it processes real numbers (e.g., latitude/longitude coordinates) and thus involves floating point arithmetics.

3 GEOSPATIAL JOIN APPROACH

In this work, we target the problem of mapping points to static, largely disjoint polygons. We show how to accelerate such joins by computing fine-grained cell-based approximations of sets of polygons and maintaining them in a query-efficient in-memory radix tree, which enables efficient cell lookups and significantly reduces (or even eliminates) expensive geometric tests.

In contrast to techniques that first reduce the number of candidate polygons using an index, e.g., an R-tree on the polygons' MBRs, and then refine candidates using geometric operations, our approach leverages true hit filtering [9] and identifies most or even all join pairs in the filter phase. On a high level, our approach first computes cell-based approximations of all polygons, called coverings and interior coverings, and merges them to form a *super covering*. Then, it stores these approximations in a specialized in-memory radix tree (named ACT) which allows for efficient lookups. Finally, ACT is probed for every point to obtain a list of *true* and *candidate* point-polygon pairs. The candidate pairs are either refined by performing geometric computations to obtain an accurate result, or deemed to be part of the join result when small approximation errors can be tolerated.

The following provides more information about our indexing technique and the two geospatial join algorithms that are based on it: the approximate one that completely avoids expensive PIP tests while still guaranteeing a user-defined precision, and the exact one that reduces expensive computations by adapting to the expected point distribution. These algorithms allow us to trade memory consumption with precision (approximate approach) and performance (exact approach). Thus, they both favor

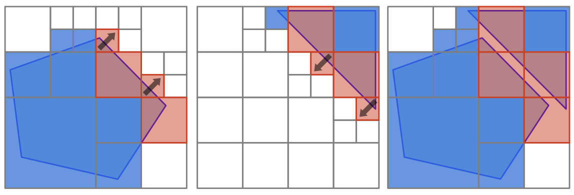

(a) Covering (b) Covering (c) Combined cov.

Figure 3: A combined covering may be less selective than two individual coverings. The arrows indicate that the cells will be expanded.

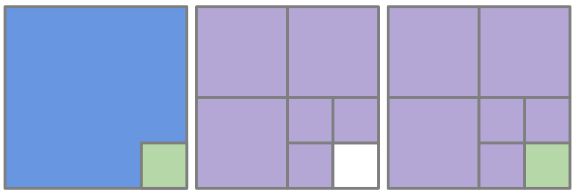

(a) c_1 and c_2 (b) Difference d (c) d and c_2

Figure 4: Precision preserving conflict resolution. c_1 is marked in blue, c_2 in green, and the cells in d in purple. Note that c_1 contains c_2.

modern hardware with large main memory capacities and high memory bandwidths. In summary, the key contribution of our indexing strategy is the novel combination of the super covering that approximates the polygons precisely, and the radix tree data structure that allows these approximations to be queried efficiently. With this design, we revisit the concept of true hit filtering in the context of modern hardware.

3.1 Adaptive Cell Trie (ACT) Indexing

3.1.1 Super Covering Computation. The super covering consists of a set of multi-resolution grid cells. All grid cells are disjoint in the sense that each geographical point is covered by at most one cell, even if two (or more) polygons overlap. A single cell of the super covering can therefore be associated with multiple polygons. The super covering maintains a list of *polygon references* for each individual cell. A polygon reference has two attributes:

polygon id The identifier of the polygon that this cell references.

interior flag Whether the cell is an interior or a boundary cell of the polygon.

The precision of the super covering determines the selectivity of the index. When combining the approximations of the individual polygons, we need to take special care of conflicting cells[1] to not lose precision. However, this is challenging for two reasons. First, conflicts may occur between the cells of a covering of a given polygon and the cells of its interior covering. The interior cells always overlap some (if not all) covering cells. Second, when different polygons overlap or are close to each other, conflicts may occur between the cells of their coverings.

One approach for retaining the precision would be to not resolve such conflicts at all and maintain all conflicting cells. However, this would have the consequence that a query point could match with more than one cell, which would affect lookup performance. In a radix tree, this would mean that we would need to keep searching lower levels once we found a match at a higher level.

Ensuring that cells are non-overlapping results not only in higher lookup performance, but also in a more compact radix tree. The reason is that for a given entry in a tree node, we only need to differentiate between a pointer (to a child node) and a value. With overlapping cells, we would have to store a pointer *and* a value.

There are two obvious solutions for resolving a conflict between two cells c_1 and c_2, where c_1 is an ancestor of c_2 in the quadtree (c_1 contains c_2). One is to replace c_2 with c_1, which leads to a precision loss as shown in Figure 3. Figures 3a and 3b show the coverings of two individual polygons. The red cells have conflicts with cells of the other covering. Figure 3c shows a combined covering, where the originally smaller cells are subsumed by larger cells, causing a precision loss. The other solution is to replace c_1 with a set of smaller cells at the same level (i.e., of the same size) as c_2. While this retains the precision, it can significantly increase the number of cells in the combined covering.

Without compromising on precision, we would like to reduce the number of cells introduced. To solve this problem, instead of storing both conflicting cells c_1 and c_2, we compute their difference d and store c_2 and d. This has the advantage that there will not be any overlap between the indexed cells, and thus an index lookup will return *at most* a single cell. The side effect is that the total number of cells will increase since d consists of at least three cells.

Figure 4 illustrates this precision preserving conflict resolution. Assume that c_1 (blue) and c_2 (green) are cells of two different coverings and that c_1 contains c_2. First, we compute d, which consists of six cells. We then copy all polygon references of c_1 to d and c_2 and omit c_1. Note that the cell count is increased by five. Overall, our approach retains the precision and the type (boundary or interior) of the individual cells as well as the mappings of cells to polygons.

Listing 1 outlines this algorithm. We iterate over all input cells and try to insert them into the super covering. When a cell already exists, this means that it is also part of another covering that has already been processed. When two cells conflict, this means that either the current cell covers the other cell or vice versa. These two cases may happen when polygons overlap or are close to each other, or when we first insert the cells of the covering of a given polygon and then the cells of its interior covering. To address these cases, we apply the precision preserving conflict resolution strategy described above. As mentioned earlier, this strategy increases the total number of cells. However, a more precise index reduces the number of expensive PIP tests and thus increases overall performance.

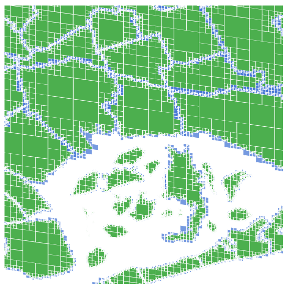


Figure 5: A super covering of neighborhoods in NYC's Jamaica Bay.

Figure 5 shows a super covering of neighborhoods in NYC's Jamaica Bay. Boundary (former covering) and interior cells are again marked in blue and green, respectively. Most of the area shown is covered by either interior cells or by no cells at all. Only

[1]Recall that a conflict between two cells exists if one cell contains the other. We do not consider duplicate cells as conflicting.

```
input:
 a list of coverings coverings // one per polygon
 a list of interior coverings interiors // one per polygon
output:
 // a list of (cell, polygon references)
 the super covering superCovering
procedure:
 for (covering in coverings) {
  for (cell in covering) {
   if (superCovering already contains cell) {
    add references of cell to existing cell
    continue
   }
   if (cell conflicts with existing cell in superCovering) {
    // cell is covered by existing cell or vice versa
    // resolve conflict
    $c_1$ = ascendant cell // may be cell or existing cell
    $c_2$ = descendant cell // may be cell or existing cell
    $d$ = difference of $c_1$ and $c_2$
    add references of $c_1$ to $d$ and $c_2$
    remove $c_1$ from superCovering // only required if the existing cell
      is the ascendant cell
    add $c_2$ and $d$ to superCovering
    continue
   }
   add {cell, {covering.polygonId, interior flag=false}} to superCovering
  }
 }
 // ... same code for interior coverings (with interior flag=true)
```

Listing 1: Build precision preserving super covering.

in the unlikely event that a query point hits a blue (boundary) cell, we may experience false positives (approximate approach) or we will need to enter the refinement phase (exact approach).

3.1.2 Data Structures. To store the super covering and enable efficient queries over it, we use two data structures: (i) a specialized radix tree (ACT) that indexes the cells of the super covering, and (ii) a lookup table that maintains the (variable-length) polygon references. Both data structures are designed for in-memory processing and are optimized for lookup performance.

Adaptive Cell Trie. ACT is a specialization of a textbook radix tree that indexes 64 bit cell ids. We call it adaptive for two reasons: (i) it indexes cells of adaptive sizes (to guarantee user-defined precision), and (ii) it can adapt to the expected point distribution. All adaptation is performed at build time. Once ACT is built, it is a static (immutable) data structure. We leave updates such as adding new polygons to an existing ACT for future work. However, we would like to point out that supporting updates is straightforward: In the build phase, cells of individual polygons are inserted one-by-one into ACT. The same procedure could be used to add new polygons at runtime, with appropriate synchronization between readers and writers. Code for removing polygons would follow the same logic, with the only difference being that we may want to (periodically) reorganize (i.e., compact) the lookup table.

We refer to the cell ids stored in the radix tree as *keys*. Each key denotes the path of a cell in the hierarchical grid. In the following, we first outline why a radix tree, in general, is a good choice for indexing quadtree cells, and we then explain how ACT differs from a general-purpose radix tree.

The main reasons why we choose a radix tree to index a super covering are (i) space efficiency and (ii) support for efficient prefix lookups. Compared to storing cell ids in a list, a radix tree avoids redundantly storing common prefixes, which reduces memory consumption. Prefix lookups, on the other hand, are required to find matching cells: The query point, which is a cell id at the most fine-grained grid level, is used to search for cell ids within the radix tree that share a common prefix (i.e., cover the query point). The runtime complexity of these lookups is in $O(k)$ with k being the key length, as opposed to the $O(\log n)$ of binary search that could be used on a sorted list. In other words, the number of node accesses in a radix tree is bounded by the maximum key length k_{max}, which is 60 when 30 quadtree levels are used (which is the case in our implementation). In practice, a lower k_{max} is often sufficient. For example, $k_{max} = 44$ allows for indexing cells up to level 22, which corresponds to a precision of less than 4 m (i.e., the distance between a point and a polygon in a false match is at most 4 m). A further advantage of the radix tree is that most queries can be answered using the upper levels of the tree: larger cells use fewer bits and are thus indexed closer to the root node. In the likely event that a query point hits a larger cell, we can complete the tree probe sooner.

We now discuss the design choices of ACT. The fanout f of the radix tree controls space consumption and lookup performance. A fanout of four means that we consume two bits at every tree level. With that configuration, our data structure matches the quadtree scheme (each node has four children, cf. Figure 6 for an example). While this would ease the implementation, it would require up to 30 nodes to be accessed per lookup. With a higher fanout, we can reduce this number. To maximize lookup performance, ACT uses a *default* fanout of 256 (= 8 bits). Thus, each level in ACT corresponds to four levels in the quadtree (each quadtree level is encoded with two bits). Let g be the cell level granularity of ACT (with $f = 256$, $g = 4$). While a fanout of 256 may result in sparsely occupied trie nodes, it allows for efficient lookups as it reduces the height of the trie to k_{max}/g. With $f = 256$, the maximum number of node accesses is $\lceil 60/\log_2(256) \rceil = 8$ for 30 quadtree levels.

Now we exploit a property of the hierarchical cells that we index: We *extend* their cell ids (keys) such that the key length matches the granularity of ACT. This process involves replacing a cell that we want to index with all its descendant cells at the next supported granularity level, and replicating the payload of the original cell to the smaller cells. In other words, if a cell does not match the tree granularity, we recursively split it into smaller cells that cover the same area. The following holds for indexed keys (cells):

$$level(cell) \bmod g = 0$$

Each cell c for which this equation does not hold is decomposed into a set of smaller cells C, with $|C| = 4^{g-(level(c) \bmod g)}$. This is possible since points are represented by cells at the most fine-grained grid level and use the maximum key length. Therefore, for a query point, it does not make a difference whether it matches with the originally inserted cell or with one of its descendant cells. This insight greatly simplifies the memory layout of a tree node and saves many CPU instructions: (i) we do not need to store the level with a cell, since all cells indexed in a tree node will have the same level, and (ii) a lookup in a node (an array) becomes a single offset access. Without this artificial key extension, we would need to perform multiple accesses per node to traverse all cell levels indexed in that node.

Figure 6 illustrates ACT indexing three polygons. While the example shows ACT with a fanout of four, by default we actually use a fanout of 256 to reduce the tree height. Every node thus consists of a fixed-sized array of 256 entries of 8 byte pointers. Entries that neither contain a child pointer nor a value point to a sentinel node indicating a false hit (no hit).

Values (i.e., polygon references) can be found in any level of the tree. This is because the indexed keys (64 bit cell ids in our case) typically use only a small fraction of the 64 bits with the remaining bits all set to zero. Larger cells that use fewer bits are indexed higher up in the tree, possibly even in the root node. In our example, polygon *a* is indexed by a cell in the upper level, while polygons *b* and *c* are indexed by cells in the lower level. Instead of storing values in separate nodes (e.g., adjacent to the tree nodes), we use combined pointer/value slots like in [25]. This design consumes less space and avoids an unnecessary indirection. Here, we exploit another property of the cell ids that we index: Cells in the super covering are disjoint, therefore a tree lookup will return at most one result. Due to this property, we never need to store a pointer and a value in an array entry at the same time. Using *pointer tagging*, we differentiate between pointers and values. We therefore refer to both pointers and values as *tagged entries*.

As stated before, each cell is associated with a set of polygon references. Thus, each value stored in the tree has to identify such a set. The canonical design would be to make each cell point to an entry in a lookup table that stores the references. However, at least in the case of largely disjoint polygons, cells mostly reference only one or two polygons. Therefore, to eliminate additional indirections, when there are no more than two polygon references, we store these references directly in the tree. A tagged entry can thus be:

- An 8 byte pointer to a child or the sentinel node (recall that a pointer to the sentinel node indicates a false hit)
- An inlined polygon reference (a 31 bit value)
- Two inlined polygon references (two 31 bit values)
- An offset (a 31 bit value) into a lookup table indicating that there are at least three polygon references

We use the two least significant bits of the 8 byte pointer to differentiate between these four possibilities. For an inlined polygon reference, we differentiate between a true hit and a candidate hit using the least significant bit of the 31 bit value. Thus, we can effectively only store 30 bit polygon ids (i.e., can index up to 2^{30} polygons).

We have experimented with path compression, but have found that storing common prefixes with inner and leaf nodes only barely reduces the number of nodes. Thus, the additional cache miss to access the prefix does not pay off. We therefore only use a common prefix at the root level.

We have also considered introducing adaptive node sizes, as proposed by the adaptive radix tree (ART) [25]. However, experiments have shown that introducing a second (compressed) node type with four children (Node4 in ART) (i) saves only a negligible amount of space for our workload and (ii) has a significant performance impact (due to the additional instructions and branch misses for dispatching between node types [25]). Also, lookups in compressed node types are more expensive.

Lookup Table. When a cell references more than two polygons, the tree contains an offset into a lookup table. Since cells often reference the same set of polygons, we only store *unique* polygon reference lists. The reference lists are split into two parts, a list with true hits and a list with candidate hits. Both lists contain

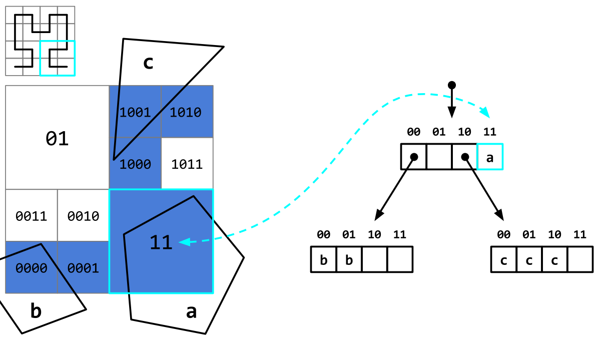


Figure 6: Adaptive Cell Trie indexing three polygons a, b, and c. Here, ACT uses two bits per level. In practice, we use up to eight bits (a fanout of 255) to reduce the tree height. Note that the figure only shows the cell rasterization for the part of the map that corresponds to the radix tree.

```
input:
  root node of ACT rootNode
  the cell id of the query point cellId
output:
  tagged entry taggedEntry
procedure:
  if (common prefix of rootNode does not match)
    return invalid entry
  level = 0
  currNode = rootNode
  bits = getBits(cellId, level++) // extract relevant bits
  // traverse the tree until we either hit the sentinel node or found a
       value
  while (taggedEntry = currNode.getEntry(bits) is a pointer) {
    if (taggedEntry points to the sentinel node)
      return false hit
    currNode = taggedEntry
    bits = getBits(cellId, level++)
  }
```

Listing 2: Probe Adaptive Cell Trie.

polygon ids. The lookup table is encoded as a single 32 bit unsigned integer array. The offsets stored in the tree are simply offsets into that array. Each encoded entry contains the number of true hits followed by the true hits, the number of candidate hits, and the candidate hits.

3.1.3 Index Probing. An ACT lookup returns, at most, one cell mapping to a set of polygon references. Listing 2 shows the probe algorithm. While traversing the radix tree does not involve any key comparison, a comparison is performed to check whether the returned tagged entry contains a payload. For that, we need to differentiate between (i) one polygon reference, (ii) two polygon references, and (iii) an offset. In the first case, we check whether the polygon reference is invalid, which indicates a false hit. Otherwise, we extract the interior flag (the least significant bit of the 31 bit payload) and the polygon id and return the reference. In the second case, we extract and return both references. Only in the third case, we need to access the lookup table to retrieve the polygon references.

3.2 Approximate Join with Precision Bound

The complete point-polygon join algorithm is shown in Listing 3. It is essentially an index nested loop join, using our novel ACT

```
input:
 points points // lat/lng coordinates and cell ids
 polygons polygons // lat/lng coordinates of vertices
 root node rootNode
 lookup table lookupTable
output:
 list of join pairs pairs // point/polygon pairs
procedure:
 for (point in points) {
  taggedEntry = probeAdaptiveCellTrie(rootNode, point.cellId) //
      cf., Listing 2
  if (taggedEntry is invalid)
   continue
  references = getPolygonReferences(lookupTable, taggedEntry) //
      returns a list of polygon references
  for (reference in references) {
   polygonId = reference.polygonId
   polygon = polygons[polygonId]
   if (reference is true hit) {
    add {point, polygon} to pairs
   } else { // candidate hit
#ifdef __APPROX
    // treat candidate hit as true hit
    add {point, polygon} to pairs
#else
    // EXACT: enter refinement phase
    if (polygonCoversPoint(polygon, point)) // PIP test
     add {point, polygon} to pairs
#endif
 }}}
```

Listing 3: The join algorithm.

index that makes the point-cell containment tests very efficient. For a given point, we retrieve the cell that contains it (if such a cell exists) and go over all references of this cell. When approximate results are sufficient, we omit the expensive refinement phase, simply treat all points contained in boundary cells as (approximate) hits, and immediately output the join pairs. In doing so, we introduce false positives. However, the distance of false positives from the polygon is bounded by the diagonal of the largest boundary cell: Any point contained in that cell has at most a distance of $\sqrt{2} * \epsilon$ (with ϵ being the side length of the cell) to the polygon. In order to control this distance, our approximate algorithm exposes a precision bound as a parameter to the user. Based on this bound, we compute the minimum cell level for boundary cells. For example, to guarantee a 4 m precision, the largest boundary cell can at most have a diagonal of 4 m, which corresponds to a minimum cell level of 22 in our implementation (i.e., cell level 21 would be too coarse-grained). We replace all boundary cells in the super covering with their descendant cells at the required level. For each of these descendant cells, we determine whether they intersect, are fully contained in, or do not intersect polygons at all, and update ACT accordingly: We remove the original cell c_o from ACT and insert *only* those descendant cells that intersect or are fully contained in polygons. The new cells may reuse the lookup table entry of c_o or create their own in the event that they only map to a subset of c_o's polygons.

Note that [39] makes use of a similar distance-based precision bound, however, uses a single-resolution grid. When it is not possible to maintain a sufficiently fine-grained index within a certain memory budget, the user can fall back on our accurate approach, in which we train the index with historical data points.

3.3 Accurate Join

When applications require accurate results, or when we cannot build an index that satisfies a user-defined precision without exceeding a memory budget, we use an approach that may enter the expensive refinement phase (cf. Listing 2). To minimize the number of (expensive) PIP tests, we increase the precision of the index by adapting it to the expected point distribution. Since we make use of true hit filtering, a finer-grained index allows us to identify more join partners during the filter phase.

3.3.1 Index Training. To minimize the likelihood of PIP tests, we train the index to adapt to the expected distribution of query points. We train ACT with historical data points (e.g., from a previous year) which has the effect that popular areas that expect more hits are approximated using a more fine-grained grid than less popular areas. This training process replaces *expensive* cells with up to four of their child cells. We define expensive cells as cells that map to polygon reference sets with at least one candidate hit. When we hit such a cell during the join, we need to perform expensive PIP tests.

Specifically, the training works as follows: When a training point hits an expensive cell, for each of its four child cells we check whether they intersect, are fully contained in, or do not intersect the referenced polygons at all, and update ACT accordingly. The cell replacement procedure is the same as for the approximate algorithm (i.e., remove original cell, insert descendant cells, and update lookup table, cf. Section 3.2) with the only difference being that we always replace an expensive cell with its direct children one level below. We do not replace a cell with even smaller cells to be more robust against outliers. In practice, we would stop refining the index once a user-defined memory budget is exhausted. In this work, we focus on training the index in a dedicated training phase. Training the index at runtime would introduce additional concurrency and buffer management issues that we leave for future work. We show the effect of training the index in Section 4.2.

3.4 Implementation Details

Join Predicate. Our current implementation follows the semantics of the ST_Covers join predicate (cf. PostGIS [30]). ST_Covers evaluates whether one geospatial object (e.g., a polygon) covers another (e.g., a point).

Individual Polygon Coverings. We compute the individual polygon coverings using the S2 library. Note that our approach does not depend on S2 and, in fact, works with any other quadtree-based hierarchical grid in which each (implicit) quadtree node [16] corresponds to a geographical area (space partitioning). For our approach to work, each quadtree node needs to be *uniquely identifiable* with a bit sequence that represents the path to the given node starting from the root. Thereby, any (consistent) enumeration scheme (e.g., the Hilbert space-filling curve used by S2 or the Z curve used by Roth [31]) of the four quadrants is valid. To store these encoded node identifiers in a trie, we require the identifiers of child nodes to share a common prefix with their parent node.

Face Nodes. Since our implementation uses S2, which projects points on Earth onto a surrounding cube, we need to maintain up to six radix trees (one for each face). Using the first three bits of the query cell id, we select the appropriate radix tree.

Index Probing. The probe (filter) phase is the performance-critical part of our approach. We therefore parallelize this phase to accelerate lookups in the radix tree. Individual processing threads fetch batches of 16 tuples at a time and synchronize using an atomic counter.

PIP Test. In the refinement phase, we use S2's PIP test, which implements the ray tracing algorithm (cf. [29] for performance numbers).

4 EXPERIMENTAL EVALUATION

In this section, we present a thorough experimental analysis of our point-polygon join algorithms. We use taxi data from NYC, which we join with different polygonal regions of NYC, such as neighborhoods. We also experiment with geo-tagged Twitter data from different cities. Besides these (skewed) real-world point datasets, we experiment with (uniform) synthetic point data. We focus our experiments on the probe phase of the join (probing points against a pre-built polygon index). For completeness, we also report build times.

Our evaluation is structured into three parts: First, we evaluate the performance and space consumption of our approximate algorithm with different data structures, including ACT, a B-tree, and a sorted vector. We demonstrate that for a city like NYC (with its 289 neighborhoods), an approximate index with very high precision (<4 m precision bound) easily fits into the main memory of a single machine and, in the case of ACT, allows for very high probe performance (>50 M points/s per CPU core). We think that this is a good fit with the city-centric model of mobility companies (e.g., Uber, DriveNow [13]). We show that ACT outperforms other physical representations by a large margin, while being more space-efficient in many cases. Second, we evaluate our accurate algorithm and show that it benefits greatly from true hit filtering. We compare it against other filter and refine approaches, including an R-tree on the polygons' MBRs, a geospatial index by Google, and PostgreSQL (PostGIS). We demonstrate that the high precision of our index can be further improved by training it with historical data points. Third, we show that both our algorithms are competitive with state-of-the-art GPU approaches.

Infrastructure. We use a server-class machine that is equipped with two 14-core Intel Xeon E5-2680 v4 CPUs and 256 GiB DDR4 RAM. All CPU-based approaches are implemented in C++ and compiled with GCC version 5.4.0 with `O3` and `march=core-avx2` flags. We conduct the experiments on a single socket to eliminate NUMA effects. For the comparison against the GPU join algorithms, we use these Amazon Web Services (AWS) instances [5]:

c5.4xlarge 16 vCPUs, USD 0.68/hour
g3s.xlarge NVIDIA Tesla M60 GPU, USD 0.75/hour

Datasets and Queries. We use 1.23 B points (pick-up locations) from the NYC yellow taxi dataset (years 2009 to 2016), which is publicly available in CSV format [38]. For each point, we load its lat/lng coordinate and convert it to an `S2Point` [33] (which represents a point on the unit sphere as a 3D vector of doubles) and to an `S2CellId` (an 8 byte value, cf. Section 2) prior to performing any experiments. We maintain one `std::vector` of `S2Points` and another one storing the corresponding cell ids. We join these points against the polygon datasets summarized in Table 1 (top). All three polygon datasets cover approximately the same area. While there are only five boroughs, their polygons are significantly more complex.

In addition, we use geo-tagged tweets collected from Twitter's live public feed over a period of five years. From these, originally over 2.29 B tweets spread across the entire US, we extract four point datasets based on the MBRs of NYC, Boston (BOS), Los Angeles (LA), and San Francisco (SF), consisting of 83.1 M, 13.6 M, 60.6 M, and 9.57 M points, respectively. We join these points against the corresponding neighborhood polygons: NYC (289), BOS (42), LA (160), and SF (117). Since we extract the points using the MBR of the entire polygon dataset and not the individual neighborhood polygons, there are points that do not join with any polygon.

We also generate synthetic point datasets, uniformly distributed within the MBR of the respective polygon dataset.

We focus our experimental evaluation on the probe phase and simply count the number of points per polygon instead of materializing the join result. To avoid any contention in the multi-threaded experiments, we maintain thread-local counters that we aggregate in the last step. Since we are focusing on the case of static polygons, the reported throughput times reflect the time to compute the counts using an *existing* (pre-built) polygon index. We report the time it takes to build the polygon index separately. However, we would like to point out that we did not optimize the build phase.

Polygon Approximations. Our default configuration for computing the individual polygon coverings is as follows: max covering cells = 128, max covering level = 30, max interior cells = 256, and max interior level = 20.

4.1 Approximate Join

We first analyze the performance and space consumption of our approximate algorithm. In all of the following experiments, we first build super coverings (sets of cell/value pairs, cf. Section 3) and then index them with different data structures.

Super Covering Construction. Table 1 shows different metrics of the super coverings for the three polygon datasets with 60 m, 15 m, and 4 m precision. With each cell occupying 64 bits, the largest super covering (census 4 m, 39.8 M cells) amounts to 304 MiB of raw key data and another 304 MiB for the values (64 bit tagged entries, cf. Section 3). Given that most cells reference fewer than three polygons, most polygon references are inlined, which keeps the lookup table small. While the computation of the individual coverings is parallelized over the number of polygons, the construction of the super covering is performed serially.

Data Structures. We essentially need to map cell ids (64 bit integers) to tagged entries (64 bit values). A tagged entry either contains up to two polygon references or an offset into a lookup table. The lookup table is the same among all data structures that we evaluate. The data structure needs to support prefix lookups: given a 64 bit lookup key (the cell id of a query point), find the cell in the super covering (recall that it only contains non-overlapping cells) that shares a common prefix with the lookup key (if such a cell exists). We analyze ACT with three different fanouts: 2, 4, and 8 bits per radix level, which corresponds to 1, 2, and 4 quadtree levels, respectively. Recall that one quadtree level is encoded with two bits. We therefore refer to these three variants as ACT1, ACT2, and ACT4. As competitors we use a B-tree implementation by Google [11] (GBT) and a binary search on a sorted vector implemented with `std::lower_bound` (LB). For GBT, we use a (target) node size of 256 bytes, which turned out to be the most query-efficient configuration. The vector stores pairs of cell ids and tagged entries. We have also experimented with the STX B+-tree [36] but do not include it in this section as its lookup performance is very similar to that of GBT.

The performance of our approximate algorithm is dominated by the costs of the ACT node accesses and the aggregation (count).

Table 1: Metrics of the NYC polygon datasets and of three super coverings with various precisions.

polygons (# polygons / avg. # vertices)	boroughs (5 / 662)			neighborhoods (289 / 29.6)			census (39,184 / 12.5)		
precision [m]	60	15	4	60	15	4	60	15	4
# cells [M]	0.09	1.32	20.9	0.16	0.98	14.0	8.50	8.97	39.8
lookup table [MiB]	0.00	0.00	0.00	0.01	0.01	0.01	1.33	1.33	1.41
build individual coverings [s]	0.11	0.98	16.0	0.07	0.19	1.54	0.96	1.01	3.08
build super covering [s]	0.10	0.94	15.2	0.17	0.81	10.5	11.6	11.8	37.7

Table 2: Metrics of the different data structures (4 m precision).

super cov.	boroughs (20.9 M cells)					neighborhoods (14.0 M cells)					census (39.8 M cells)				
index	ACT1	ACT2	ACT4	GBT	LB	ACT1	ACT2	ACT4	GBT	LB	ACT1	ACT2	ACT4	GBT	LB
size [MiB]	328	198	173	359	319	224	138	143	240	214	624	421	1234	684	608
build [s]	2.11	1.46	1.06	1.39	-	1.36	0.98	0.69	0.85	-	4.00	3.11	2.80	2.85	-

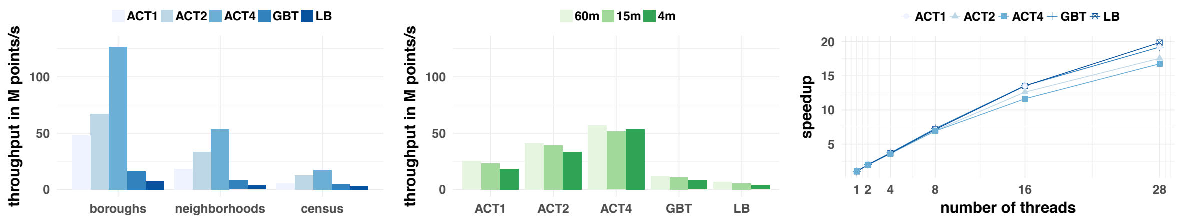


Figure 7: Throughput and scalability of our approximate algorithm (taxi dataset). Left: Single-threaded execution with different data structures (4 m precision). Middle: Single-threaded execution with different precisions and data structures (neighborhood polygons). Right: Multi-threaded execution (neighborhood polygons, 4 m precision).

To better understand the results, we therefore first analyze the space consumption of ACT and compare it with GBT and the sorted vector. Table 2 shows size and build time (single threaded) of the different data structures on the super coverings introduced above (4 m precision only). In many cases, ACT consumes less space than the sorted vector (LB). Due to the high density of the cell ids, ACT is more space-efficient with higher fanouts, except for census where ACT4 consumes the most space: Like for all datasets, ACT4 has fewer (but larger) nodes than ACT1 and ACT2. However, in this case, its nodes are very sparsely populated compared to those of ACT1 and ACT2. The reason is that ACT4's nodes cover too much space for the relatively small census cells. To mitigate the size impact of sparse ACT nodes, one can represent them with a more compact data structure [4]. All 4 m indexes exceed the 35 MiB L3 cache of our evaluation machine. Note that there is no additional build time for LB, since the super covering contains cell id/tagged entry pairs already sorted by cell id.

Single-Threaded Throughput. For this experiment, we compute a super covering with a 4 m precision bound on the three NYC polygon datasets and store it in the different data structures introduced above. We then join the full taxi dataset (all 1.23 B points) against each of these indexes and report the throughput in M points/s (cf. Figure 7 (left)).

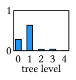
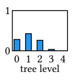
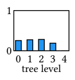
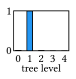
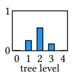
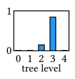
ACT clearly dominates the B-tree and the binary search on the sorted vector, especially in its highest fanout configuration (ACT4). A higher fanout means that we consume more bits of the lookup key per tree level and thus require fewer node accesses (i.e., need to traverse fewer levels) to find a key (an indexed cell). With ACT4 for example, we consume 8 bits per tree level and thus need at most 64/8 = 8 node accesses. Since we reduce the tree height further by storing a common prefix at the root level (cf. Section 3), ACT requires even fewer node accesses (e.g., at most five with 4 m precision).

Table 3: Speedups of lookups in smaller (more coarse-grained) over larger (more fine-grained) polygon datasets for different data structures (b = boroughs, n = neighborhoods, c = census).

	b over n	b over c	n over c
ACT1	**2.63×**	**8.63×**	**3.28×**
ACT2	2.00×	5.33×	2.66×
ACT4	2.36×	7.29×	3.08×
GBT	2.05×	3.51×	1.71×
LB	1.83×	2.63×	1.44×

Another insight is that ACT benefits the most from the larger (coarser-grained) cells in the smaller polygon datasets as shown in Table 3. Going from the most fine-grained census dataset (39,184 polygons) to the most coarse-grained boroughs dataset (5 polygons), GBT's lookup performance improves by 3.51×, while ACT1's increases by 8.63×. The reason for ACT's large gain is that larger cells are indexed higher up in the radix tree and are thus found sooner. GBT, in contrast, does not benefit from these larger cells, which might as well be stored in the leaf nodes of the B-tree. GBT's performance gain comes from the smaller number of cells used for indexing the boroughs dataset and the resulting smaller B-tree (i.e., fewer branch and cache misses per point).

Table 4: Distribution of the tree traversal depth (ACT4 with 4 m precision).

points	boroughs	neighborhoods	census
uniform			
taxi			

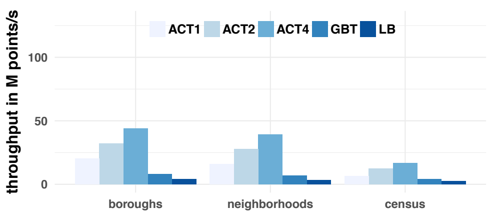


Figure 8: Single-threaded throughput of our approximate algorithm (4 m precision) with uniform point data.

Likewise, the binary search on the sorted vector (LB) is only affected by the number of cells and not their granularity.

Different Precisions. Next, we vary the precision of the indexed super covering. We perform this experiment using the medium size neighborhoods dataset. Figure 7 (middle) shows the throughput numbers for the different data structures. While GBT's and LB's performance decreases by 33.4% and 39.4%, respectively from 60 m to 4 m, ACT4's performance is hardly affected (-5.73%) by the larger number of cells of the more precise super covering. Compared to the 60 m covering, the more precise coverings contain a larger number of small cells (in the boundary areas of the polygons). Query points are unlikely to hit these cells in contrast to the large (more coarse-grained) cells, which are indexed in the upper (cached) ACT nodes (due to their shorter cell ids). ACT1 and ACT2 are more affected by the precision increase (-27.8% and -17.9%, respectively). The reason is that the added small cells have a stronger effect on the depths of these trees. While the average node depth for ACT4 only increases from 2.83 to 2.97 (+4.95%) from 60 m to 4 m respectively, the same metric increases from 10.8 to 14.6 (+35.2%) for ACT1. Although—as already stated above—the new small cells are unlikely to be hit, they still cause a performance hit for lower fanouts.

ACT4's throughput is similar for 15 m and 4 m (-4.15%) because its structure is identical for both precisions. In both cases, it has 70,786 nodes occupying 143 MiB. The only difference is the nodes' structure: Due to the more fine-grained cell approximation, the average node occupancy (measured in terms of occupied slots) of ACT4 at tree level 3 decreases from 88.2% (60 m) to 85.2% (4 m). The occupancies of all other levels are the same. This lower occupancy for 4 m saves some aggregations (for updating the polygon hit counts), causing a slightly higher performance.

In summary, the impact of precision on query performance is less significant for ACT than for the other data structures.

Multi-Threaded Throughput. In this experiment, we study the lookup performance of the different data structures with an increasing number of threads on the neighborhoods dataset with a 4 m precision bound. We use up to 28 threads, which matches the number of hyperthreads of a single NUMA node of our evaluation machine. Figure 7 (right) shows the speedups over single-threaded execution. Up to 8 threads, all index structures scale almost linearly (speedup of around 7× in all cases). This is what we would expect for immutable data structures.

The fact that an oversubscription of cores (hyperthreading) has a positive performance impact suggests that the lookup is bound by memory access latencies (having more threads than physical cores can hide these latencies).

Synthetic Points. To show the general applicability of our approach, we also experiment with synthetic point data. We generate 100 M points uniformly distributed within the MBR of the respective (NYC) polygon dataset. Table 4 shows the probability distribution of the number of search steps during the tree traversal for the synthetic and the taxi point dataset. As expected, the distribution for the uniform data is skewed towards the root. That is because the larger cells (which are more likely to be hit) are indexed closer to the root. The distribution for the taxi data depends on the polygon dataset. For boroughs, most traversals end at tree level 1, while for census, points mostly hit small cells indexed in tree level 3.

Figure 8 shows the single-threaded throughput for the different data structures with the uniform point data. ACT achieves the highest throughput, with ACT4 again being the most query-efficient configuration. The absolute numbers, however, are lower than for the (real-world) taxi data: ACT4's throughput decreases by 65.2%, 26.8%, and 3.11% for boroughs, neighborhoods, and census, respectively.

The reason for this slowdown is simple: The synthetic point data is uniformly distributed, which leads to more branch and cache misses (cf. Table 5 for performance counters on neighborhoods). In contrast, the real-world taxi data is highly clustered with the majority of points located in Manhattan (>90%) and around the airports. For boroughs (not shown in Table 5), ACT4 endures 0.79 and 0.01 branch misses per point for the synthetic and the taxi points, respectively. This is the main cause of the 65.2% performance drop mentioned above.

Twitter Data. Next, we analyze the performance of our approach on the four Twitter datasets and the corresponding neighborhood polygons (cf. Figure 9). The numbers are similar across the different cities, with the highest throughput achieved for BOS with its only 42 neighborhood polygons. Next comes SF followed by LA and NYC, for which the throughput is very close to what we obtained with the taxi data (cf. Figure 7 (left)). In fact, with a 4 m precision, ACT4 achieves a single-threaded throughput of 52.1 M points/s, which is almost the same as the 53.6 M points/s on the taxi data. Similarly to the taxi points, the tweets are clustered, with certain areas having more tweeting activity than others. In contrast, with uniform point data, ACT4 only achieved 39.3 M points/s. This confirms that our approach benefits from the skewed distribution of real-world data. For all four cities, the numbers are (again) hardly affected by the precision.

4.2 Accurate Join

We now evaluate our accurate algorithm, which eliminates false positives in an additional refinement phase. We demonstrate that our index benefits significantly from true hit filtering and that index training with historical data can further improve its effect.

Competitors. We compare against the boost R-tree (1.6.0) [8] on the polygons' MBRs (RT), Google's S2ShapeIndex [34] (SI), and PostgreSQL 9.6.1 (PostGIS 2.3.1) [30] with a GiST index on

Table 5: Performance counters per point (neighborhoods, 4 m precision).

points	uniform					taxi				
index	ACT1	ACT2	ACT4	GBT	LB	ACT1	ACT2	ACT4	GBT	LB
cycles	154	99.8	71.3	415	569	172	93.8	56.4	416	817
instructions	214	121	82.4	486	927	202	121	81.3	393	564
branch misses	1.06	1.04	0.88	5.32	8.38	0.96	0.83	0.48	7.06	10.8
cache misses	0.29	0.23	0.18	0.70	1.89	0.22	0.17	0.15	0.29	0.37

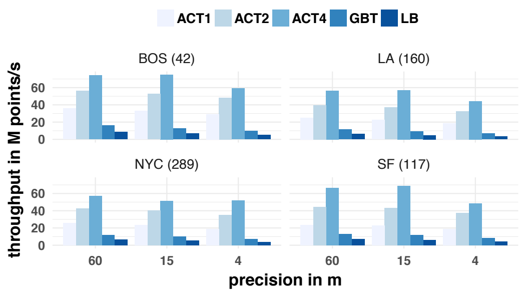


Figure 9: Single-threaded throughput of our approximate algorithm (Twitter datasets, polygon counts in brackets).

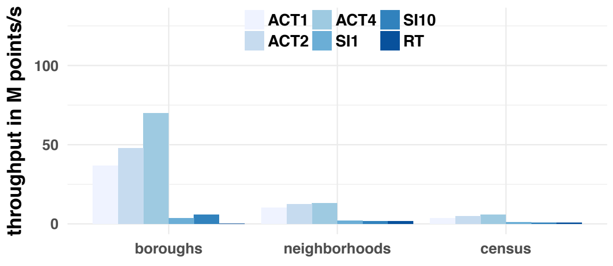


Figure 10: Single-threaded throughput of our accurate algorithm (with different ACT fanouts) compared to S2ShapeIndex (with 1 and 10 edges per cell) and the R-tree.

polygons (PG). Our algorithm and the R-tree both use the same PIP test implementation (cf. Section 3.4). SI also uses that implementation, however, restricts the test to a subset of edges of the polygon in question. This is achieved by using a hierarchical grid approximation of polygons, and internally mapping grid cells (64 bit S2CellIds) to polygon edges using a B-tree. This hierarchical grid approximation is much more coarse-grained than our super covering, given its higher focus on build time than on query performance (compared to our approach). SI allows the maximum number of edges per cell to be configured, essentially controlling the granularity of the employed grid approximation. We evaluate SI with its default configuration of 10 edges (SI10) and 1 edge per cell (SI1). Note that SI1 is the most fine-grained configuration possible. SI also employs true hit filtering (cf. Section 3) to avoid PIP tests, but in a much less effective way than ours (due to its coarser-grained grid). Furthermore, SI does not offer an approximate version. For the R-tree, we use the splitting strategy `rstar` with at most 8 elements per node which performs best in all workloads.

Taxi Data. For this experiment, we compute coarse-grained super coverings that do not guarantee a certain precision, and instead fall back on a refinement phase for candidate hits. Here, the resolution of a super covering is determined by our default configuration for computing individual polygon coverings introduced earlier (cf. Section 4). Thus, these super coverings consist of much fewer cells than those guaranteeing a certain precision. For example, the approximation for the neighborhoods dataset now only consists of 98,687 cells (ACT4 size: 25.9 MiB) compared to the 13.2 M cells (ACT4 size: 143 MiB) needed to guarantee a 4 m precision. For this dataset, SI1, SI10, and RT consume 1.20 MiB, 0.23 MiB, and 27.9 KiB, respectively.

Figure 10 shows the single-threaded throughputs for the accurate join. ACT4 achieves the highest performance for all three datasets. For the medium size neighborhoods dataset, it outperforms SI1 by 6.96×, followed by SI10, which is only 7.41% slower than SI1. For census, ACT4 still outperforms SI1 by 5.79×. RT has the lowest numbers with 0.21, 1.77, and 0.79 M points/s for boroughs, neighborhoods, and census, respectively. The reason for its slow performance for boroughs is as follows: The complexity of each PIP test (ray-tracing algorithm) is linear with the size (number of edges) of the polygon. Since the boroughs are complex polygons with many edges, the PIP tests in the refinement phase are very expensive. Here, our algorithm shines since it can identify most join partners in the filter phase and only enters the refinement phase for 0.1% of the points. As a point of reference, PG achieves 0.39, 1.09, and 0.69 M points/s for boroughs, neighborhoods, and census, respectively (because we use all hyperthreads on our evaluation machine, PG's numbers are not directly comparable and are excluded from the plot). Similar to RT, PG is affected by the complex boroughs polygons.

Index Training. As readers may have noticed, there is a large performance gap between our approximate and our accurate algorithm. For example, ACT4 (accurate) is 75.3% slower than its approximate counterpart (with 4 m precision) on the taxi data/neighborhoods join. The reason is the expensive PIP tests needed to compute an accurate result.

We now show how to narrow this performance gap. The idea is to reduce the likelihood for PIP tests by training the index with historical data points (cf. Section 3.3.1). In other words, we increase the precision of the index by making it more fine-grained in areas where we expect more points. One effect this has is that the size of the area covered by (expensive) boundary cells will decrease. We train the index with taxi points sampled from the year of 2009 and only use the points from 2010 to 2016 for the join. Table 6 shows the performance impact. With 100 K training points, ACT4's performance improves by 1.56× for neighborhoods and increases further to 2.18× with 1 M points (due to a 84.0% reduction in the number of PIP tests). The size of ACT4 only increases from 25.9 MiB (untrained) to 28.0, 34.8, and 44.3 MiB when trained with 100 K, 500 K, and 1 M historical data points, respectively. In absolute terms, ACT4 trained with 1 M points achieves a throughput of 29.1 M points/s for neighborhoods and

Table 6: Speedups of single-threaded lookups when training ACT4 with an increasing number of historical data points (over untrained ACT4).

no. of train. points	boroughs	neighborhoods	census
100 K	1.25×	1.56×	1.16×
500 K	1.40×	2.00×	1.40×
1 M	1.44×	2.18×	1.53×

Table 7: Effect of training the index with 1 M historical data points (STH = solely true hits).

metric	boroughs	neighborhoods	census
STH (%)	99.9 → 99.9	87.2 → 97.7	72.2 → 88.7

thus narrows the performance gap to its approximate counterpart (with 4 m precision) from 75.3% to 45.7% while consuming 68.9% less space. This shows that a trained accurate index is a good alternative to our approximate indexes when main memory is sparse. Table 7 shows the effect of true hit filtering when training the index with 1 M training points. The metric `solely true hits` (STH) indicates the percentage of points that skipped the expensive refinement phase, which is clearly above 70% in all cases (even without training). Training the index significantly improves STH for neighborhoods and census.

4.3 Comparison with GPU Algorithms

Finally, we compare our approximate and accurate (untrained ACT) algorithms against state-of-the-art GPU counterparts [39]. The GPU approaches leverage the graphics rendering pipeline, and in particular the rasterization operation, which converts a polygon into a collection of (equi-sized) pixels. Similar to our approach, the GPU join also comes in two variants: Bounded Raster Join (BRJ), which guarantees a user-defined precision by appropriately scaling the rendering resolution, and Accurate Raster Join (ARJ), which performs PIP tests for points falling on the pixels forming the boundaries of the polygons. To enable a fair comparison, we do not consider any preprocessing times on the polygons (such as triangulation time). Note that the preprocessing time for the GPU join is minimal. In fact, it is designed for computing the join on-the-fly without a priori knowledge of the polygonal regions.

We now compare the throughput of both approaches on two similarly priced AWS machines (cf. Infrastructure in Section 4). Figure 11 shows the results of joining 612 M taxi rides with the NYC polygon datasets. While our approximate algorithm is again hardly affected by the precision (15 m vs. 4 m), BRJ takes a significant performance drop. The reason for BRJ's slowdown is simple: Once the required resolution is higher than what is natively supported by the GPU, it needs to split the scene and perform more rendering passes. This is essentially related to the fact that BRJ relies on a uniform grid. On the contrary, BRJ is barely affected by the polygon datasets, while our approximate algorithm is. The reason is again related to the granularity of the grid: With the more fine-grained census dataset, we need to traverse more tree nodes (as the cells that approximate the polygons are smaller), while the rendering resolution in BRJ depends only on the size of the bounding box of the polygon dataset and the precision. With

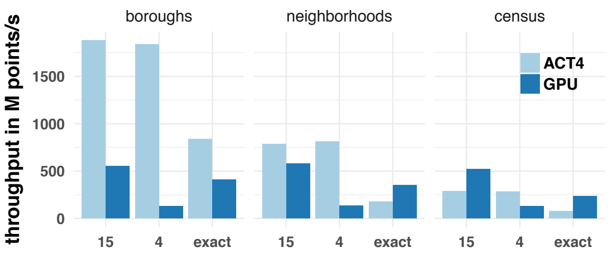


Figure 11: Throughput of ACT4 (16 threads) compared to the two GPU algorithms on AWS (GPU = Bounded Raster Join for 15 m and 4 m and Accurate Raster Join for exact).

exact results, our approach outperforms ARJ for boroughs, while ARJ takes the crown for neighborhoods and census.

5 RELATED WORK

Prior Publications. In [24], we describe a novel approach to reduce control flow divergence on AVX-512 platforms to further increase ACT's lookup performance. Note that [24] is based on an earlier (4-page) version of this work [22].

Spatial Join Techniques. The point-polygon join is one of the core operations in spatial databases, and, a large body of related work on algorithmic techniques [19] is available accordingly.

Naturally, we are not the first to index polygons using raster approximations. Early on, Orenstein [27] proposed decomposing single polygons into multiple cells. Later, Brinkhoff et al. [9] proposed true hit filtering in the form of maximum enclosed rectangles and circles, allowing the refinement phase to be skipped in many cases. Zimbrao et al. [49] followed up on this approach by using raster approximations in the form of uniform grids, thereby improving selectivity. Kothuri et al. [20] recursively divide the MBR of a polygon into four cells until a certain granularity is reached, identify interior cells, and index them in an R-tree to skip refinement checks. The primary goal was to minimize I/O, an important performance factor for disk-based systems. In contrast to these early works on true hit filtering and also to the recent proposal by Tzirita Zacharatou et al. [39], we use a quadtree-based (multi-resolution) grid that can be very coarse-grained in interior and very fine-grained in boundary areas.

Research has, however, also been performed on true hit filtering with quadtree-based rasterizations, including work in Oracle Spatial [21] and Microsoft SQL Server [15]. In both of these works, *individual* polygons are approximated using a set of multi-resolution grid cells. These grid cells are enumerated using one-dimensional cell identifiers and stored in a B-tree. In contrast, we *holistically* approximate and index an entire set of polygons and store these (in our case duplicate-free) cell identifiers in a novel radix tree (ACT), which is more query-efficient than a B-tree. Additionally, these existing approaches neither offer an approximate mode nor allow the accurate index to be trained with historical data points to improve query performance.

To decrease the probability of false matches, [35] improves the precision of MBRs by clipping away empty space that is concentrated around the MBR corners. In contrast to our work, [35] uses the classical filter and refine evaluation strategy.

Related to our approximate algorithm is work by Azevedo et al. [7] that provides precision estimates for approximate polygon-polygon joins using a less space-efficient single-resolution grid. Tzirita Zacharatou et al. [39] propose a similar precision bound to ours but also use a single-resolution grid (cf. Section 4.3 for a comparison).

The PH-tree [47] is another example of a trie data structure that indexes multi-dimensional data. In contrast to ACT, it only indexes points, not higher-level grid cells. Winter et al. [43] propose a query-efficient storage layout for point data that automatically adapts to polygonal queries. Along the same lines, Vorona et al. [42] train a model to approximately answer spatial aggregation queries.

Systems. Several database systems support geospatial joins. PostGIS [30], a geospatial extension to PostgreSQL [1], uses an R-tree implemented on top of GiST [18] for indexing geospatial objects. In recent years, various spatial data management systems based on Hadoop [3, 14] and Spark [26, 37, 44, 46] have emerged. [28] provides a comprehensive analysis of these modern spatial analytics systems by a thorough experimental evaluation. In contrast to our work, most of these systems rely on offline partitioning of the data points.

Modern Hardware. Most work on using modern hardware for geospatial joins focuses on GPU offloading [2, 12, 39, 45, 48] while [10] proposes a GPU-accelerated end-to-end spatial system.

6 CONCLUSIONS

We have presented two point-polygon join algorithms that use a multi-resolution grid indexed in a query-efficient radix tree. We have transformed a traditionally compute-intensive problem into a memory-intensive one. We have shown that it is possible to refine the index up to a user-defined precision and identify all join partners in the filter phase. We have demonstrated that the accurate version of our algorithm can adapt to the expected point distribution. We have also shown that our approach outperforms existing CPU-based joins by up to two orders of magnitude and can compete with dedicated GPU implementations.

ACKNOWLEDGMENTS

This work has been partially supported by the German Federal Ministry of Education and Research (BMBF) grants `01IS12057` (FASTDATA) and `01IS18025A` (BBDC II). It is further part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government. Harish Doraiswamy was supported in part by the Moore-Sloan Data Science Environment at NYU, and NSF awards `CNS-1229185`, `CCF-1533564`, `CNS-1544753`, `CNS-1730396`, and `CNS-1828576`.

REFERENCES

[1] PostgreSQL. http://www.postgresql.org/.
[2] D. Aghajarian and S. K. Prasad. A spatial join algorithm based on a non-uniform grid technique over gpgpu. In *Proc. of SIGSPATIAL*, pages 56:1–56:4, 2017.
[3] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. H. Saltz. Hadoop-GIS: A high performance spatial data warehousing system over MapReduce. *PVLDB*, 6(11):1009–1020, 2013.
[4] C. Anneser, A. Kipf, H. Lang, T. Neumann, and A. Kemper. The case for hybrid succinct data structures. In *Proc. of EDBT*, 2020.
[5] AWS instance types. https://aws.amazon.com/ec2/instance-types/.
[6] L. G. Azevedo, R. H. Güting, R. B. Rodrigues, G. Zimbrão, and J. M. de Souza. Filtering with raster signatures. In *Proc. of ACM-GIS*, pages 187–194, 2006.
[7] L. G. Azevedo, G. Zimbrão, and J. M. de Souza. Approximate query processing in spatial databases using raster signatures. In *Proc. of GEOINFO*, pages 53–72, 2006.
[8] *boost::geometry::index::rtree - 1.60.0.* https://www.boost.org/doc/libs/1_60_0/libs/geometry/doc/html/geometry/reference/spatial_indexes/boost__geometry__index__rtree.html.
[9] T. Brinkhoff, H. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *Proc. of SIGMOD*, pages 197–208, 1994.
[10] H. Chavan, R. Alghamdi, and M. F. Mokbel. Towards a GPU accelerated spatial computing framework. In *ICDE Workshops*, pages 135–142, 2016.
[11] Google C++ B-tree. https://code.google.com/archive/p/cpp-btree/.
[12] H. Doraiswamy, E. Tzirita Zacharatou, F. Miranda, M. Lage, A. Ailamaki, C. T. Silva, and J. Freire. Interactive visual exploration of spatio-temporal urban data sets using urbane. In *Proc. of SIGMOD*, pages 1693–1696, 2018.
[13] DriveNow. https://www.drive-now.com/de/en.
[14] A. Eldawy. SpatialHadoop: Towards flexible and scalable spatial processing using MapReduce. In *SIGMOD, PhD Symposium*, pages 46–50, 2014.
[15] Y. Fang, M. Friedman, G. Nair, M. Rys, and A. Schmid. Spatial indexing in microsoft SQL server 2008. In *Proc. of SIGMOD*, pages 1207–1216, 2008.
[16] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, 1982.
[17] E. Haines. Point in polygon strategies. *Graphics gems IV*, 994:24–26, 1994.
[18] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proc. of VLDB*, pages 562–573, 1995.
[19] E. H. Jacox and H. Samet. Spatial join techniques. *ACM Trans. Database Syst.*, 32(1):7, 2007.
[20] K. V. R. Kanth and S. Ravada. Efficient processing of large spatial queries using interior approximations. In *Proc. of SSTD*, pages 404–424, 2001.
[21] K. V. R. Kanth, S. Ravada, and D. Abugov. Quadtree and r-tree indexes in oracle spatial: a comparison using GIS data. In *Proc. of SIGMOD*, pages 546–557, 2002.
[22] A. Kipf, H. Lang, V. Pandey, R. A. Persa, P. A. Boncz, T. Neumann, and A. Kemper. Approximate geospatial joins with precision guarantees. In *Proc. of ICDE*, pages 1360–1363, 2018.
[23] A. Klinger. Patterns and search statistics. In *Optimizing methods in statistics*, pages 303–337. Elsevier, 1971.
[24] H. Lang, A. Kipf, L. Passing, P. A. Boncz, T. Neumann, and A. Kemper. Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. In *DaMoN*, pages 5:1–5:8, 2018.
[25] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proc. of ICDE*, pages 38–49, 2013.
[26] *Magellan: Geospatial Analytics Using Spark.* https://github.com/harsha2010/magellan.
[27] J. A. Orenstein. Redundancy in spatial databases. In *Proc. of SIGMOD*, pages 295–305, 1989.
[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. Kipf, D. Vorona, T. Mühlbauer, T. Neumann, and A. Kemper. High-performance geospatial analytics in HyPerSpace. In *Proc. of SIGMOD*, pages 2145–2148. ACM, 2016.
[30] PostGIS: Spatial and geographic objects for PostgreSQL. http://postgis.net/.
[31] J. Roth. The extended split index to efficiently store and retrieve spatial data with standard databases. In *Proc. of IADIS*, pages 85–92, 2009.
[32] Google S2 library. http://s2geometry.io/.
[33] S2Geometry Basic Types. http://s2geometry.io/devguide/basic_types.html.
[34] Google S2ShapeIndex. https://s2geometry.io/devguide/s2shapeindex.
[35] D. Sidlauskas, S. Chester, E. Tzirita Zacharatou, and A. Ailamaki. Improving spatial data processing by clipping minimum bounding boxes. In *Proc. of ICDE*, pages 425–436, 2018.
[36] STX B+-tree. http://panthema.net/2007/stx-btree/.
[37] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref. LocationSpark: A distributed in-memory data management system for big spatial data. *PVLDB*, 9(13):1565–1568.
[38] TLC Trip Record Data. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.
[39] 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.
[40] Uber geofence. https://eng.uber.com/go-geofence/.
[41] F. van Diggelen and P. Enge. The world's first gps mooc and worldwide laboratory using smartphones. In *Proc. of ION GNSS+*, pages 361–369, 2015.
[42] D. Vorona, A. Kipf, T. Neumann, and A. Kemper. DeepSPACE: Approximate geospatial query processing with deep learning. In *Proc. of SIGSPATIAL*, pages 500–503, 2019.
[43] C. Winter, A. Kipf, T. Neumann, and A. Kemper. GeoBlocks: A query-driven storage layout for geospatial data. *CoRR*, abs/1908.07753, 2019.
[44] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *Proc. of SIGMOD*, pages 1071–1085, 2016.
[45] S. You, J. Zhang, and L. Gruenwald. Parallel spatial query processing on gpus using r-trees. In *SIGSPATIAL Workshops*, pages 23–31, 2013.
[46] J. Yu, J. Wu, and M. Sarwat. Geospark: a cluster computing framework for processing large-scale spatial data. In *Proc. of SIGSPATIAL*, pages 70:1–70:4, 2015.
[47] T. Zäschke, C. Zimmerli, and M. C. Norrie. The ph-tree: a space-efficient storage structure and multi-dimensional index. In *Proc. of SIGMOD*, pages 397–408, 2014.
[48] J. Zhang and S. You. Speeding up large-scale point-in-polygon test based spatial join on gpus. In *Proc. of SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pages 23–32. ACM, 2012.
[49] G. Zimbrao and J. M. de Souza. A raster approximation for processing of spatial joins. In *Proc. of VLDB*, pages 558–569, 1998.

DISGD: A Distributed Shared-nothing Matrix Factorization for Large Scale Online Recommender Systems

Heidy Hazem[*], Ahmed Awad[**], Ahmed Hassan[*], Sherif Sakr[**]
[*]Nile University, Egypt
[**]University of Tartu, Estonia
h.hazem@nu.edu.egy,ahmed.awad@ut.ee,ahassan@nu.edu.egy,sherif.sakr@ut.ee

ABSTRACT

With the web-scale data volumes and high velocity of generation rates, it has become crucial that the training process for recommender systems be a *continuous* process which is performed on live data, i.e., on *data streams*. In practice, such systems have to address three main requirements including the ability to adapt their trained model with each incoming data element, the ability to handle concept drifts and the ability to scale with the volume of the data. In principle, matrix factorization is one of the popular approaches to train a recommender model. Stochastic Gradient Descent (SGD) has been a successful optimization approach for matrix factorization. Several approaches have been proposed that handle the first and second requirements. For the third requirement, in the realm of data streams, distributed approaches depend on a shared memory architecture. This requires obtaining locks before performing updates.

In general, the success of main-stream big data processing systems is supported by their shared-nothing architecture. In this paper, we propose `DISGD`, a distributed shared-nothing variant of an incremental SGD. The proposal is motivated by an observation that with large volumes of data, the overwrite of updates, lock-free updates, does not affect the result with sparse user-item matrices. Compared to the baseline incremental approach, our evaluation on several datasets shows not only improvement in processing time but also improved recall by 55%.

1 INTRODUCTION

We are living in the era of data abundance whereby good decisions are backed by data-driven approaches. In addition to business-related decisions, we can use data for our personal daily lives. For example, what products to buy, where to have lunch, and best places to spend our vacations are all decisions that we need to make.

Recommender systems [11] have emerged to predict and suggest objects that could be of interest to the user. In general, recommender systems receive input in the form of user-item rating. These ratings are used to update a rating matrix R where the rows represent the users and columns represent items, where usually R is sparse. Collaborative filtering (CF) [5] is a successful technique to guess user preferences based on R. Matrix factorization-based (MF) CF algorithms have shown to be successful. For example, it was able to win the Netflix prize [2]. MF works by decomposing R into two low-dimension vectors of latent factors. Stochastic Gradient Descent (SGD) is used to optimize the weights of these latent factors. In general, SGD is an iterative algorithm that works on a static data set. As data velocity have accelerated, there has become a crucial need to get recommendations with low latency. Therefore, the need to analyze these data and generate new suggestions moved from an offline task on a finite set of data into an online task on a possibly infinite *stream* of data. Thus, a scalable online recommender system has to address three main requirements [3]: 1) The model must be able to produce a result and be updated after each record has been received without passing over all the past data (*latency*). 2)*Concept drifts* [10] ought to be taken care of by adjusting the model with each instance. 3) Online learning from big data must be processed in a distributed streaming environment (*scalability*).

Vinagre et al. [12] have proposed ISGD as an incremental SGD that needs to process each data element once in a streaming fashion. ISGD addresses the first and second requirements above. Yet, it remains a centralized (one worker) solution. Several approaches have introduced parallel (distributed) variants of (I)SGD [1, 4, 6, 7, 13]. However, the common limitation in these approaches is the need to access a shared memory to update the weights among parallel workers. In an online-setting, the overhead to obtain a lock leads to higher latency. An interesting observation by Recht et al. [9] is that with large data, having a lock-free update mechanism, i.e. lost updates, does not affect the overall performance and SGD finally converges. The authors also prove it. Based on this observation, in this paper, we present `DISGD` as a distributed shared-nothing variant of ISGD. By utilizing the shared-nothing architecture, we allow the best *scalability* as each worker is independent. In particular, the main contributions of this paper can be summarized as follows:

- DISGD: A distributed shared-nothing incremental stochastic gradient descent for a distributed online recommender system (Section 2),
- A comparative evaluation with the baseline ISGD on several data sets showing the superiority of our approach not just in the processing speed but also in the improved recall (Section 3).

2 DISGD

ISGD [12] is an incremental matrix factorization algorithm that is based on SGD. ISGD works centrally where training data are streamed element-by-element. For every received element, ISGD updates the model. So this algorithm overcomes the first two challenges we mentioned in Section 1. In this section, we describe our approach towards addressing the third challenge, scalability.

2.1 Background

In order to reach a scalable ISGD, we depend on the observation that usually the ratio of items to users is petite. For instance, Netflix data set has millions of users and only thousands of items. We start from the observation that the rating matrix $R^{n \times m}$ is sparse. So, we decompose the rating matrix into two matrices, the users' matrix $U^{n \times k}$ and the items' matrix $I^{m \times k}$ with low-dimension k, where $k << n$ and $k << m$, latent features that underlie the items' rating by users. So, we can predict the rating of user u to item i in $R^{n \times m}$, by calculating the dot product between their vectors as in Formula 1.

$$\hat{r_{ui}} = U_n . I_m^T = \sum_{k=1}^{k} u_{nk} . i_{mk} \tag{1}$$

The two matrices U and I are initialized with Gaussian random values. Then, iteratively, SGD calculates how different their product is from the rating matrix R and then makes an effort to minimize this difference. ISGD is dealing with positive feedback only so the error can be calculated by $err_{ui} = 1 - \hat{R_{ui}}$ and we are following that in our algorithm. Furthermore, the gradient descent algorithm iterates many times and updates the vectors which are the rows of the matrices U and I with the purpose of finding a local minimum of the difference following the loss function formulated in 2 where λ is the regularization parameter.

$$min_{U.,I.} \sum_{(u,i) \in D} (R_{ui} - U_u . I_i^T)^2 + \lambda(\|U_u\|^2 + \|I_i\|^2) \tag{2}$$

To parallelize ISGD by distributing the workload among n_c processors, the rating matrix R has to be divided into several blocks and the blocks get assigned to different processors. The issue here is that two processors working on different blocks may need to update the same column of U and/or I. The blocks must be distributed in a way that avoids conflicting updates. Our proposal to solve this problem, and thus addressing the scalability challenge, is by utilizing a splitting and replication mechanism of users and items vectors.

2.2 Splitting and Replication Mechanism

Receiving the rating interactions from users formulated as $<u, i, r>$, Algorithm 1 will distribute the received streamed data of tuples by hashing each record where the user vector and item vector reside over the nodes. For each received tuple $< u, i, r>$, ISGD updates the user vector and the item vector according to the two equations below, where η is the gradient step size.

$$U_u = U_u + \eta(err_{ui} . I_i - \lambda U_u) \tag{3}$$

$$I_i = I_i + \eta(err_{ui} . U_u - \lambda I_i) \tag{4}$$

The aim behind our splitting and replication mechanism is to guarantee that the vectors of users and items are divided over the nodes as it would grow larger than the capacity of one node (central solution). It is assumed that the items are known beforehand. Hence, starting by the item matrix, it is divided into n_i splits (partitions) and each split is replicated over n_c/n_i of the nodes -where n_c is number of nodes in the cluster- while each user vector should exist in n_i of the nodes to always guarantee that a tuple $<u, i, r>$ hits one node where its user and item vectors reside. As a requirement, the number of nodes in the cluster n_c should be equal to $n_i^2 + w.n_i$ where $w \in \mathbb{N}_0$. The distribution technique in Algorithm 1 offloads the storage of vectors to around n_i/n_c of the nodes. For example, when $n_i = 2$, the item matrix I is divided into two halves, each half is stored on half of the nodes. The user matrix U is divided over $n_c/2$ of the nodes and each user vector should exist in two nodes, given $n_i = 2$, over the cluster. Hence, any received tuple is always distributed, in such a way that its user and item vectors are always represented in only one node. Thus, the entire rating matrix will not be needed at any point of time for any single processing task.

Algorithm 1: Parallel ISGD algorithm

```
Data: data stream of {< u, i, r >} ∈ D
Input: N, λ, η, n_i, k
Output: Top N recommended list.
1: function DISTRIBUTINGFN(u,i,n_i)
2:     $n_c = n_i^2 + w.n_i$
3:     iList ← Map hashing of item id to its $n_c/n_i$ of nodes
4:     uList ← Map hashing of user id to its $n_i$ nodes
5:     key ← Get the common node from uList and iList
6:     outStream < key, u, i, rate >
7: end function
while receiving {< u,i,r >} ∈ D on each node do
    DISTRIBUTINGFN(u,i,n_i)
    if $u \notin Rows(U)$ then
        $U_u$ ← Vector(size : k);
        $U_u \sim \mathcal{N}(0,0.1)$;
    end
    if $i \notin Rows(I)$ then
        $I_i$ ← Vector(size : k));
        $I_i \sim \mathcal{N}(0,0.1)$;
    end
    $err_{ui} \leftarrow 1 - U_u . I_i^T$;
    $U_u \leftarrow U_u + \eta(err_{ui} . I_i - \lambda U_u)$;
    $I_i \leftarrow I_i + \eta(err_{ui} . U_u - \lambda I_i)$
end
```

Algorithm 1 describes how we scale ISGD by means of splitting and replication of the users and items vector. A new top N recommendations list is generated every time a tuple is received. Based on the distributing mechanism shown in Figure 1. This function accords a key to the tuple for maintaining that the pair of user and item vector exists in one node while the single item vector should be in n_c/n_i nodes and the single-user vector should reside in n_i of nodes. This key is produced by hashing the user and item then mapping the hashing output to a predefined list of n_i nodes and n_c/n_i nodes and get the common node number to be the key whereby this key is used for distributing. This node processes the received data and outputs top N recommendations. This particular node only processes $1/n_i$ of the items matrix I that is received in the hashing process described earlier. This does not mean that this particular user will always get his recommendation from this node based on the same items.

It is a random process, based on which $1/n_i$ of the items stored in which node that tuple hits. Moreover, the repetition of the user vector helps offload the storage, making it possible for the algorithm to recommend items for user from different n_i pools of candidates which boosts our recommendation algorithm by giving a wide view for all the items. Algorithm 1 does not need to synchronize between the n_i same user's vectors or n_c/n_i same item's vectors repeatedly stored in the nodes, as Figure 1 shows. It has been proved by Recht et al. [9] that SGD algorithm running over parallel processors with shared memory can converge when the threads overwrite each other and calculate gradient using the outdated current solution which leads to asynchronous machine learning algorithms. Keeping the vectors asynchronous accomplishes two important things, first, it makes DISGD faster and avoids any synchronization or need for lock management.

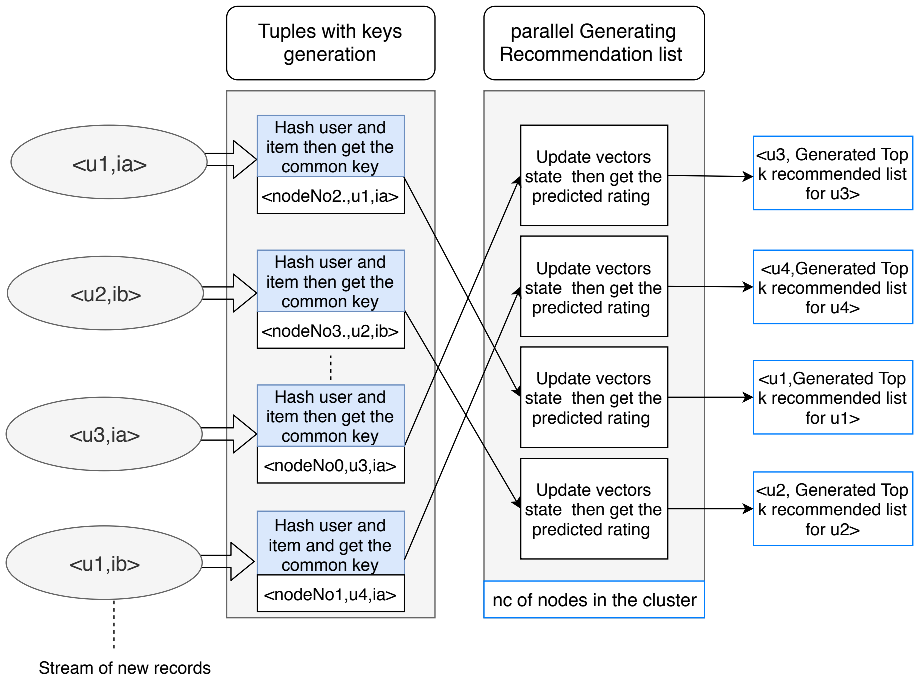

Tuples with keys generation
parallel Generating Recommendation list
<u1,ia>
<u2,ib>
<u3,ia>
<u1,ib>
Hash user and item then get the common key
<nodeNo2.,u1,ia>
<nodeNo3.,u2,ib>
<nodeNo0,u3,ia>
Hash user and item and get the common key
<nodeNo1,u4,ia>
Update vectors state then get the predicted rating
<u3, Generated Top k recommended list for u3>
<u4,Generated Top k recommended list for u4>
<u1,Generated Top k recommended list for u1>
<u2, Generated Top k recommended list for u2>
nc of nodes in the cluster
Stream of new records

Figure 1: Overview of DISGD collaborative filtering

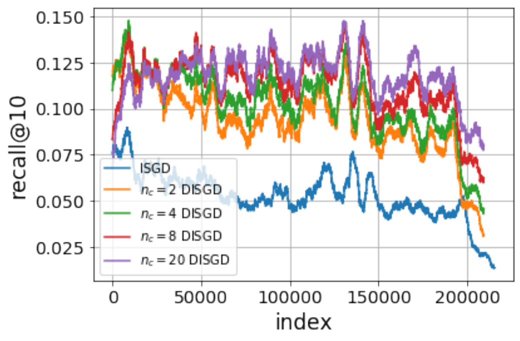
```
Algorithm 2: Prequential online evaluator using recall
    (1) Recommend top-N recommendation list for the user's coming
        interaction if the user is known otherwise move to step 3.
    (2) score top-N recommendation list based on the coming item i
        using recall.
        $Recall@N = \begin{cases} 1, i \in topNrecommendationlist \\ 0, i \notin top-Nrecommendationlist \end{cases}$
    (3) Updated the vectors with the coming instance
```

3 EVALUATION

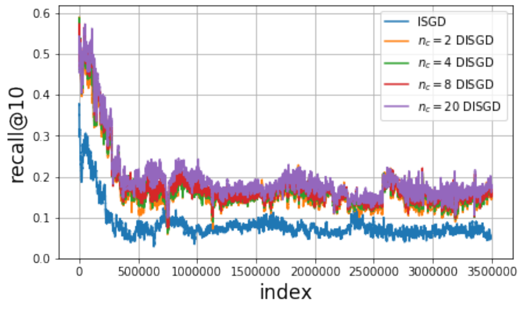
We evaluate DISGD against ISGD as a baseline. The evaluation experiments are done without handling cold start problem as it is not our concern in this paper. We follow prequential evaluation[8] which is suitable and mostly used for streaming algorithms. Prequential evaluation works as follows: for every received instance; it is used first for testing then feed the model with it for training. Specifically, we are following prequential evaluation for streaming recommender systems proposed by Vinagre et al. [12] using the recall evaluation metric which gives indication of how many true positive hits from the user side to the recommendation list by measuring the ratio of relevant items recommended to the total. We compute recall as per Algorithm 2.

The hyperparameter values of equations 3 and 4 used in our experiments are $\lambda = 0.01$, $\mu = 0.05$. We compute the recall with $N = 10$ and set the number of latent features to $k = 10$. DISGD has been implemented on top of Apache Flink version 1.8.1 deployed in a standalone cluster mode with 64 workers. Each worker is a single core running at 2.3 *GHz* with 30 *GB* of main memory. To run the baseline ISGD, we implemented it also as a Flink application and force it to run on a single worker. All the code of our experiments are available for reproducibility [1].

[1]https://github.com/DataSystemsGroupUT/DISGD



(a) Recall@10 for Moveilens 1M

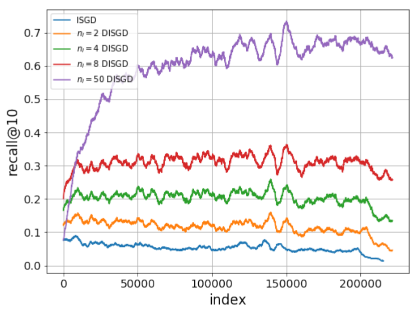


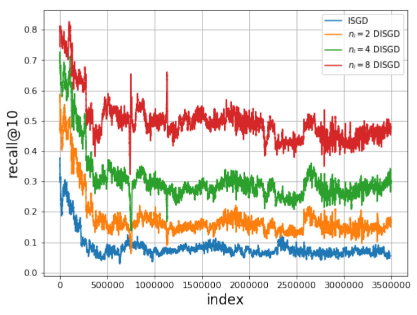
(b) Recall@10 for Netflix

Figure 2: Development of recall@10 testing different n_c. The plotted lines relate to a moving average of the recall@10 got for every recommendation with window size w=5000 with replication factor $n_i = 2$

Data Sets. For our experimntal evaluation, we have used three popular datasets: `Movielens 1M`[2], `Netflix`[3] and `last.FM`[4].

[2]https://grouplens.org/datasets/movielens/1m/
[3]https://www.kaggle.com/netflix-inc/netflix-prize-data
[4]https://www.last.fm/


(a) Moveilens 1M


(b) Netflix

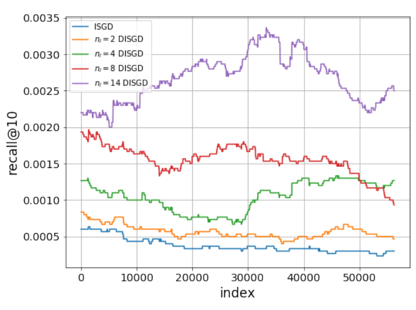

(c) LastFM

Figure 3: Development of recall@10 testing different n_i. The plotted lines relate to a moving average of the recall@10 got for every recommendation with window size w=5000 with different replication factor

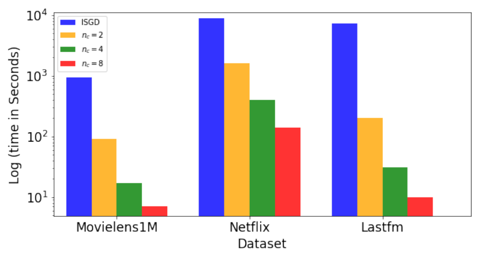


Figure 4: Comparison between the processing time for ISGD and DISGD with different n_i applied on Movielens, Netflix and lastFM datasets.

Movielens and Netflix are notable datasets of rating films and they are sparse. Both the dataset's schema consists of *user*, *item*, *rating* (from 1 to 5) and *time-stamp*. The same preprocessing is done for both datasets. The datasets have been ordered chronologically as indicated by the timestamp for capturing the pattern of how the user interacts with the items consecutively and as our algorithm depends on positive feedback the datasets have been filtered out from any records with a rating under 5. LastFM dataset contains records of listening to music tracks. We extracted the tuples of *<user,trackID>* assuming that the occurrence of the pair as positive feedback and the dataset has been ordered by timestamp.

Experiments and Results. We have run two main experiments. In the first one, we fixed $n_i = 2$; which means that each item vector exists with two versions each on half of the nodes. As per condition in our mechanism, the nodes in the cluster should be n_c=4+2w. We have varied the value of w. The results are reported in Figure 2 showing the results of summing a moving average recall SMA with window size 5000 elements for data sets Movielens 1M and Netflix. We can clearly observe that DISGD achieves significantly a higher recall than the baseline. Obviously, increasing the number of nodes n_c with the same n_i results in higher recall. The recall slightly improves with increasing n_c. The same observation for enhanced recall applies to Netflix.

Regarding the second experiment, DISGD has been tested using different replication factor n_i values with minimum $n_c = n_i^2$. In particular we run our experiments with $n_i \in \{2, 4, 8, 50\}$[5], for `Movielens 1M` dataset and with $n_i \in \{2, 4, 8\}$ for `Netflix`. `LastFM` is tested with $n_i \in \{2, 4, 8, 14\}$[5]. The model has been evaluated using SMA recall at $N = 10$. The results of Figure 3 shows that it is obvious that our approach can scale with different replication factors with enhanced recall in comparison to ISGD.

In general, *processing time* is a major factor in handling streaming data. The x-axis of the graph in Figure 4 shows the three data sets while the log processing time in seconds is on the y-axis. The results show that processing time reduces significantly from ISGD to DISGD and the time decrease dramatically when n_c has risen. It is observed that DISGD is between $6 - 15$ times faster than ISGD with respect to the data sets and the parallelism factor n_c while keeping a significantly higher recall.

[5]The experiment with $n_i = 50$ and $n_i = 14$ have been applied separately on a larger cluster

4 CONCLUSION AND FUTURE WORK

In this paper, we presented `DISGD`, a distributed shared-nothing variant for stochastic gradient descent for streaming data. Our solution allows much lower latency in serving for recommender systems. However, as with other streaming applications, the data distribution change might lead to skewness in the load on workers. Load rebalancing techniques already exist in literature, however, the effect of moving/merging state on the performance of the algorithm is unknown and is an interesting subject for our future work.

REFERENCES

[1] Muqeet Ali, Christopher C Johnson, and Alex K Tang. 2011. *Parallel collaborative filtering for streaming data.* Technical Report.

[2] Robert M Bell and Yehuda Koren. 2007. Lessons from the Netflix prize challenge. *SiGKDD Explorations Newsletter* 9, 2 (2007), 75–79.

[3] András A Benczúr, Levente Kocsis, and Róbert Pálovics. 2018. Online machine learning in big data streams. *arXiv preprint arXiv:1802.05872* (2018).

[4] Badrish Chandramouli, Justin J Levandoski, Ahmed Eldawy, and Mohamed F Mokbel. 2011. StreamRec: a real-time recommender system. In *SIGMOD.*

[5] David Goldberg et al. 1992. Using collaborative filtering to weave an information tapestry. *Commun. ACM* 35, 12 (1992).

[6] Shohei Hido, Seiya Tokui, and Satoshi Oda. 2013. Jubatus: An open source platform for distributed online machine learning. In *NIPS 2013 Workshop on Big Learning.*

[7] Mu Li et al. 2014. Scaling distributed machine learning with the parameter server. In *OSDI.*

[8] Robert Nishihara et al. 2017. Real-Time Machine Learning: The Missing Pieces. In *HotOS 2017.* ACM, 106–110.

[9] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems.* 693–701.

[10] Paul Resnick and Hal R Varian. 1997. Recommender systems. *Commun. ACM* 40, 3 (1997), 56–59.

[11] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2011. Introduction to recommender systems handbook. In *Recommender systems handbook.* Springer, 1–35.

[12] João Vinagre, Alípio Mário Jorge, and João Gama. 2014. Fast incremental matrix factorization for recommendation with positive-only feedback. In *International Conference on User Modeling, Adaptation, and Personalization.*

[13] Kais Zaouali et al. 2018. Distributed Collaborative Filtering for Batch and Stream Processing-Based Recommendations. In *OTM.*

Forming Compatible Teams in Signed Networks

Ioannis Kouvatis [1], Konstantinos Semertzidis [2]
Maria Zerva [1], Evaggelia Pitoura [1], Tsaparas Panayiotis [1]
[1] Department of Computer Science and Engineering, University of Ioannina, Greece
{ikouvatis,mzerva,pitoura,tsap}@cse.uoi.gr
[2] IBM Research, Dublin, Ireland
konstantinos.semertzidis1@ibm.com

ABSTRACT

The problem of team formation in a social network asks for a set of individuals who not only have the required skills to perform a task but who can also communicate effectively with each other. Existing work assumes that all links in a social network are positive, that is, they indicate friendship or collaboration between individuals. However, it is often the case that the network is *signed*, that is, it contains both positive and negative links, corresponding to friend and foe relationships. Building on the concept of structural balance, we provide definitions of compatibility between pairs of users in a signed network, and algorithms for computing it. We then define the team formation problem in signed networks, where we ask for a *compatible* team of individuals that can perform a task with small communication cost. We show that the problem is NP-hard even when there are no communication cost constraints, and we provide heuristic algorithms for solving it. We present experimental results to investigate the properties of the different compatibility definitions, and the effectiveness of our algorithms.

1 INTRODUCTION

Given a task that requires a set of skills, a pool of workers who possess some of the skills and are organized in a social network, team formation refers to finding a subset of the workers that collectively cover the skills and can communicate effectively with each other [9]. The communication cost of the team is measured using the distances between the team members in the network. The idea is that socially well connected users will be more effective in working together.

Since the pioneering work in [9], there has been considerable research activity on the problem [1, 8, 12]. All existing work assumes that the social network contains only positive ties between individuals. That is, the edges denote friendship or successful collaboration between two users. However, very often, we have *signed* networks with both positive and negative ties. A negative edge indicates a contentious relationship and inability of two users to collaborate and thus they should not be in the same team.

In this paper, we study the problem of team formation in signed networks. In addition to the known requirements of the team formation problem, we ask that the team contains users that are all *compatible* with each other. Defining compatibility in a signed network is an interesting problem in itself. Clearly, users connected with a positive edge are compatible, while users connected with a negative edge are incompatible. We infer the compatibility of non-connected pairs of users by using the structure of the graph, and the principle of *structural balance*. Structural balance [4] is based on the premise that "the friend of my friend is my friend", "the enemy of my enemy is my friend", and "the enemy of my friend is my enemy". Using this premise, we determine the compatibility of two users by looking at the paths that connect them. For example, a path of two positive or two negative edges indicates compatibility, while a path of one positive and one negative edge indicates incompatibility. We formalize this idea, and we provide definitions of compatibility of varying strictness. We perform experiments with four real datasets, where we evaluate the different compatibility definitions, and the performance of the algorithms for the team formation problem.

2 PROBLEM DEFINITION

We are given as input a pool of n individuals organized in an *undirected signed* graph $G = (V, E)$. Each node in V corresponds to an individual, and $E = \{(u, v, \ell) : u, v \in V, \ell \in \{+1, -1\}\}$ is a set of edges *labeled* as either positive or negative to indicate that u and v are friends or enemies respectively. We assume that G is connected. We will use a function $sign : E \to \{+1, -1\}$ that returns the label of each edge in E.

We are also given as input a universe S of skills. Each individual u in V possesses a set of skills, $skill(u) \subseteq S$. We define a task as the subset of skills $T \subseteq S$ required for its completion. Given a task, the team formation problem asks for a team X of individuals, $X \subseteq V$, that collectively covers the required skills and whose members can work together *effectively*. The effectiveness of a team is typically quantified by the communication cost, $Cost(X)$, of the team defined as some function of the distances of its members in the graph.

However, when the graph contains both positive and negative edges, we need to take into account that some individuals, although close in the graph, may not be *compatible* with each other. To capture whether two users are compatible, we introduce a relation $Comp \subseteq V \times V$ such that $(u, v) \in Comp$, if and only if, u and v can work together. Two natural requirements for $Comp$ are (1) reflexivity: $(u, u) \in Comp$, and (2) symmetry: if $(u, v) \in Comp$, then $(v, u) \in Comp$. Furthermore, we require that the $Comp$ relation satisfies the following two intuitive properties:

(1) *Positive Edge Compatibility:* For all $(u, v) \in E$, such that $sign(u, v) = +1$, $(u, v) \in Comp$.
(2) *Negative Edge Incompatibility:* For all $(u, v) \in E$, such that $sign(u, v) = -1$, $(u, v) \notin Comp$.

We will provide various definitions of compatibility in Section 3. We now define formally our problem.

Definition 2.1. (Team Formation in Signed Networks (TFSN)) Given a signed graph $G = (V, E)$, a compatibility relation $Comp$, and a task T, find $X \subseteq V$ such that (1) $\bigcup_{u \in X} skill(u) \supseteq T$, (2) for each $u, v \in X$, $(u, v) \in Comp$, and (3) $Cost(X)$ is minimized.

The TFSN problem contains as a special case the original team formation problem which is NP-hard [9], thus TFSN is also NP-hard. Moreover, we have shown that just finding a set of compatible users is NP-hard. Let TFSNC denote the simplified version of the TFSN problem, where we drop the third requirement of minimizing the cost. In particular, we have proven the following theorem.

THEOREM 2.2. *The decision version of* TFSNC *is NP-hard for any compatibility relation that satisfies positive edge compatibility and negative edge incompatibility.*

3 USER COMPATIBILITY

We start with two basic definitions of compatibility.

Definition 3.1. Direct Positive Edge (DPE) compatibility: $Comp_{\text{DPE}} = \{(u,v) \subseteq V \times V : (u,v,+1) \in E\}$.

Definition 3.2. No Negative Edge (NNE) compatibility: $Comp_{\text{NNE}} = \{(u,v) \subseteq V \times V : (u,v,-1) \notin E\}$.

DPE is the strictest form of compatibility, while NNE is the most relaxed one. Specifically, $Comp_{\text{DPE}}$ is the minimal subset of pairs of nodes that satisfies the positive edge compatibility property, while $Comp_{\text{NNE}}$ is the maximal subset of pairs of nodes that satisfies the negative edge incompatibility property.

We will now use the theory of *structural balance* [2, 4, 7] to provide more refined definitions of compatibility. The theory is based on the following socially and psychologically founded premises: (1) the friend of my friend is my friend, (2) the friend of my enemy is my enemy, and (3) the enemy of my enemy is my friend. Let $P = (v_0, \ldots, v_{k+1})$, $(v_i, v_{i+1}) \in E$ denote a path between nodes v_0 and v_{k+1} in a signed graph G. We define the sign of the path as $sign(P) = \prod_{i=0 \ldots k} sign(v_i, v_{i+1})$. We say that path P is positive if $sign(P) = +1$ and negative if $sign(P) = -1$.

CLAIM 1. *A positive path P_{uv} between two nodes u and v indicates compatibility, while a negative one indicates incompatibility.*

The claim follows from the basic principle of structural balance. To see this, let $P_{uv} = (x_0, x_1, ..., x_k, x_{k+1})$, $x_0 = u$, $x_{k+1} = v$ be a path that connects u and v. Let F_u be the set of friends of u and E_u be the set of enemies of u. We start by placing node u in F_u and traverse the path as follows. When we traverse edge (x_i, x_{i+1}), if the edge is positive we place x_{i+1} in the same set as x_i. That is, the friends of my friends are also my friends, and the friends of my enemies are my enemies. If the edge (x_i, x_{i+1}) is negative then we place x_{i+1} in the opposite set of x_i. That is, the enemies of my enemies are my friends, and the enemies of my friends are my enemies. If the path P_{uv} is positive then v will be placed in F_u, while if the path is negative it will be placed in E_u.

We first look at shortest paths. We use SP_{uv} to denote the set of shortest paths between nodes u, v, SP^+_{uv} to denote the positive, and SP^-_{uv} the negative ones.

Definition 3.3. Shortest Path (SP) compatibility relations: – All Shortest Path (SPA) compatibility: $Comp_{\text{SPA}} = \{(u,v) \subseteq V \times V : \forall P_{uv} \in SP_{uv}, sign(P_{uv}) = +1\}$.
– Majority Shortest Path (SPM) compatibility: $Comp_{\text{SPM}} = \{(u,v) \subseteq V \times V : |SP^+_{uv}| \geq |SP^-_{uv}|\}$.
– One Shortest Path (SPO) compatibility: $Comp_{\text{SPO}} = \{(u,v) \subseteq V \times V : \exists P_{uv} \in SP_{uv}, sign(P_{uv}) = +1\}$.

We further relax compatibility by asking for positive paths that are not necessarily the shortest ones. Based on structural balance, certain triangles are more stable. A general signed graph is structurally balanced, if it does not contain any cycle with an odd number of negative edges [7].

Given a path P, let $G_P = (P, E[P])$ be the graph induced by the nodes of P. We say that path P is structurally balanced if the subgraph G_P is structurally balanced. Let BP_{uv} denote the set of all structurally balanced paths between u and v.

Algorithm 1 The SP-compatibility algorithm.

Input: Signed graph G, query node q.
Output: The number of positive and negative shortest paths from q to all other nodes in the graph.

```
1: Initialize $N^+(q) = 1$, $N^-(q) = 0$ $N^+(x) = N^-(x) = 0$, $L(q) = 0$,
   $L(x) = \infty$, empty queue $Q$.
2: $Q$.enqueue($q$)
3: while $Q \neq 0$ do
4:     $u = Q$.dequeue()
5:     for $x$ adjacent to $u$ do
6:         if $L(u) + 1 \leq L(x)$ then
7:             if $x \notin Q$ then
8:                 Q.enqueue($x$)
9:             $L(x) = L(u) + 1$
10:            if $sign(u,x) = +1$ then
11:                $N^+(x)$ += $N^+(u)$; $N^-(x)$ += $N^-(u)$
12:            else if $sign(u,x) = -1$ then
13:                $N^-(x)$ += $N^+(u)$; $N^+(x)$ += $N^-(u)$
14: return ($N^+$, $N^-$, $L$)
```

Definition 3.4. Structurally Balanced Path (SBP) compatibility: $Comp_{\text{SBP}} = \{(u,v) \subseteq V \times V : \exists P_{uv} \in BP_{uv}, sign(P_{uv}) = +1\}$.

The motivation for SBP compatibility is that, in addition to P_{uv} being positive, asking for G_P to be structurally balanced means that the sign of any edge connecting u and v must be positive, otherwise a cycle with an odd number of negative edges will be created. Note that SBP-compatibility does not imply SP-compatibility. Consider the example in Figure 1(a). The (only) shortest path between u and v is (u, x_1, v) which is negative, and thus u, v are not SP-compatible. However, u and v are SBP-compatible, since the path (u, x_2, x_3, x_4, v) is positive and structurally balanced. Note that there is a shorter path (u, x_2, x_1, v) between u and v that is positive, but not structurally balanced, since the shortcut edge (u, x_1) creates the unbalanced triangle (u, x_1, x_2).

It is easy to see that the following holds:

PROPOSITION 3.5. $Comp_{\text{dpe}} \subseteq Comp_{\text{spa}} \subseteq Comp_{\text{spm}} \subseteq Comp_{\text{spo}} \subseteq Comp_{\text{sbp}} \subseteq Comp_{\text{nne}}$.

Algorithms. We now present algorithms for SP and SBP compatibility. Algorithm 1 shows the modified BFS algorithm for counting positive and negative shortest paths. Given the query node q, for each node $x \in V$ in the graph, the algorithm maintains the numbers $N^+(x)$ and $N^-(x)$ of positive and negative shortest paths respectively and the length of the shortest path $L(x)$ from q to x. When reaching node x from node u through a shortest path (line 6), if the edge (u, x) is positive, we increment the number of positive and negative paths of x by $N^+(u)$ and $N^-(u)$ respectively, since all paths retain their sign. If the edge (u, x) is negative, we increment $N^-(x)$ by $N^+(u)$, and $N^+(x)$ by $N^-(u)$, since the sign of the paths change. Each edge is examined only once.

The efficient enumeration of shortest paths is possible due to the prefix property that a shortest path between q and x that goes through node u must use a shortest path from q to u. However, this is not the case for shortest structurally balanced paths. Consider the example in Figure 1(b). The shortest structurally balanced path from u to x_4 is (u, x_3, x_4). However, the shortest structurally balanced path $(u, x_1, x_2, x_4, x_5, v)$ from u to v goes through node x_4 but not through the shortest structurally balanced path from u to x_4, since the path (u, x_3, x_4, x_5, v) is not structurally balanced.

Since the exact algorithm is prohibitively expensive for large graphs, due to the exponential number of paths, we also consider a heuristic alternative for SBP-compatibility that counts only

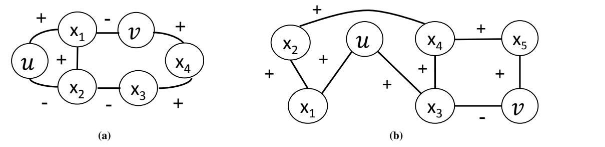


Figure 1: (a) u and v are SBP but not SP compatible. (b) It does not suffice to keep a single path from u to x_4.

Algorithm 2 Team formation algorithm.

```
    Input: Signed graph $G$, task $T$, compatibility relation $Comp$.
    Output: Team $X$.
 1: Initialize $S \leftarrow \emptyset$.   //S: skills covered so far
 2: Initialize $\mathcal{L} \leftarrow \emptyset$.   //$\mathcal{L}$: set of candidate teams
 3: Select skill $s$ from $T$   //skill selection
 4: for each $u$ with skill $s$ do
 5:     $X \leftarrow \{u\}$   // X: candidate team
 6:     $S \leftarrow S \cup (T \cap skills(u))$
 7:     while $S \neq T$ do
 8:         Select skill $s$ from $T - S$   //skill selection
 9:         Select user $v$ with skill $s$   //user selection
10:             s.t. $(v, x) \in Comp$ for all $x$ in $X$
11:         $X \leftarrow X \cup \{v\}$
12:         $S \leftarrow S \cup (T \cap skills(v))$
13:     $\mathcal{L} \leftarrow \mathcal{L} \cup X$
14: return $argmin_{X \in \mathcal{L}}(Cost(X))$
```

paths having the prefix property. We will use SBP to denote the compatibility relation computed by the exact exhaustive algorithm, and $\mathrm{SBP_H}$ the output of the heuristic algorithm.

4 TEAM FORMATION

We now present algorithms for the TFSN problem. Recall that our goal is to find a team of compatible users, that covers all skills, and minimizes the communication cost. The communication cost is defined as the largest distance between any two pairs of users in the team. We define the distance between two users looking at the positive paths connecting them. Specifically, for DPE and SP compatibility, distance is the length of the shortest path, while for SBP the length of the shortest structurally balanced positive path. For NNE compatibility, since there be no positive paths, we define distance as the length of the shortest path ignoring its sign.

Algorithm 2 is a generic algorithm that incrementally builds a solution, each time considering an uncovered skill and adding a compatible user having this skill, until all skills are covered. There are two placeholders in this algorithm. The first is the policy for selecting a skill (lines 3 and 8), and the second the policy for selecting a candidate user (line 9).

We consider two policies for selecting skills: select the rarest skill first (as in [9]), and select the least compatible skill first. We define the compatibility degree $cd(s)$ of skill s based on the compatibility between the users with skill s and the users with all other skills: $cd(s) = \sum_{s_j \in S, s_i \neq s} cd(s, s_j)$, where $cd(s, s_j) = |\{(u_i, u_j) : (u_i, u_j) \in Comp, s \in skills(u_i) \text{ and } s_j \in skills(u_j)\}|$.

Table 1: Dataset Statistics

	Slashdot	*Epinions*	*Wikipedia*
#users	214	28,854	7,066
#edges	304	208,778	100,790
#neg edges	89 (29.2%)	34,941 (16.7%)	21,765 (21.5%)
diameter	9	11	7
#skills	1,024	523	500

We also consider two policies for selecting users: select the user that has the minimum distance, and select the user that is most compatible among the remaining users. The first selection aims at minimizing the cost, while the second at maximizing the chances of finding a group of compatible users. We experimentally evaluate different combinations of these policies in Section 5.

Table 2: Comparison of compatibility relations

	SPA	SPM	SPO	$\mathrm{SBP_H}$	SBP	NNE
			Slashdot			
comp. users	44.72	55.72	72.45	97.85	99.38	99.64
comp. skills	80.57	86.19	92.63	99.11	99.47	99.50
avg distance	4.13	4.37	4.57	4.95	4.97	4.53
			Epinions			
comp. users	29.61	62.98	86.46	99.82	–	99.99
comp. skills	97.25	98.90	99.66	99.99	–	99.99
avg distance	3.48	3.82	3.87	3.97	–	3.83
			Wikipedia			
comp. users	21.98	59.33	87.51	99.56	–	99.91
comp. skills	66.17	87.31	97.32	99.87	–	99.96
avg distance	2.85	3.23	3.30	3.38	–	3.25

Table 3: Comparison with unsigned team formation

	SPA	SPM	SPO	SBP	NNE
Ignore sign	0%	2%	2%	26%	30%
Delete negative	0%	2%	18%	66%	76%

5 EXPERIMENTAL EVALUATION

In this section, we compare the different compatibility relations on real datasets and evaluate the team formation algorithm.

Datasets. Table 1 details our real-world datasets. *Slashdot* contains information about users and their posts on Slashdot. We obtained a network of Slashdot users [11], where users have tagged their relationships as friend or foe. Then we used the categories of users' posts as skills. *Epinions* contains information about users and their reviews about products. The dataset is created by combining a signed network of Epinions users [11] with the RED [1] dataset which contains information about the products and product categories the users have reviewed. We used the unique user ids to match users in the two datasets, and we assigned as skills to users the categories of the products they have reviewed. *Wikipedia* [11] is a signed network of editors. The edge sign corresponds to a positive or negative vote in admin elections. Since there was no skill information, we assigned synthetically generated skills to its users. We generated 500 distinct skills with frequencies following a Zipf distribution as in real data. Each skill is assigned to users in the network uniformly at random.

Compatibility Relations. In Table 2, we report the percentage of compatible pairs of users and skills. The DPE is excluded from our analysis, since this corresponds to finding cliques and team formation is too restrictive. Two skills s_1 and s_2 are compatible if they have compatibility degree $cd(s_1, s_2) > 0$, i.e., there is at least one compatible pair of users (u, v) such that u has s_1 and v has s_2 (including self-compatibility, if the same user has both skills).

As expected, the number of compatible user and skill pairs increases as we relax the notion of compatibility. For SPA, less than half of the pairs of nodes are compatible, and as low as 21.98% for the case of *Wikipedia*. Also, in most cases, for a sizeable fraction of pairs of skills there are no compatible users, indicating that for many skill combinations there can be no compatible team. Another interesting observation is that the fraction of compatible pairs for SBP is comparable with that for NNE. This means that, for all pairs that are not directly connected with a negative edge, there exists at least one positive structurally balanced path that connects them.

[1] https://projet.liris.cnrs.fr/red/

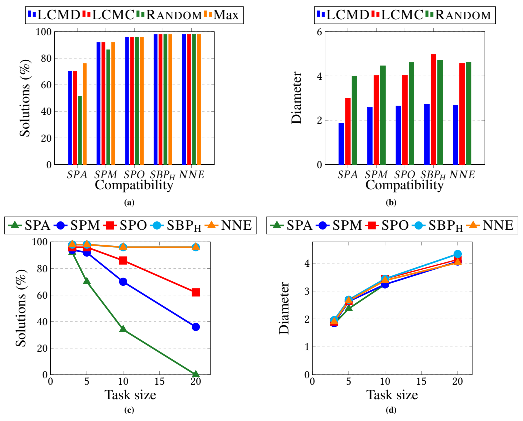


Figure 2: Team formation: comparison of algorithms ((a) and (b)), varying task size (c) and (d).

Distance. In Table 2, we also report the average distance between compatible users. The distance steadily increases as we relax the compatibility definition. The exception is NNE in which we allow negative paths, and thus we are able to discover shorter ones.

Comparison of **SBP** ***and*** **$\mathrm{SBP_H}$.** We also compare the exact (SBP) and heuristic ($\mathrm{SBP_H}$) structurally balanced compatibility for the *Slashdot* dataset, for which we can compute the exact relation. Table 2 shows the difference between SBP and $\mathrm{SBP_H}$ which is only ~2.5%

Team Formation. Due to space limitation, in this set of experiments, we only report results using *Epinions*. Results are similar for the other networks. We generate tasks of different sizes. For a given task of size k, we generated 50 tasks by randomly selecting k skills. First, we compare the four different team formation algorithms obtained by combining the two different skill and user selection policies.We report results for the two algorithms that performed the best which are the algorithms that select the least compatible skill. The LCMD, selects the user with the minimum distance, while the LCMC, the user who is the most compatible with the existing team. We also experiment with a baseline RANDOM that selects a compatible user at random.

In Figure 2(a), we report the percentage of times that each algorithm was able to find a compatible team for $k = 5$. The last bar (MAX) shows the percentage of tasks that contain compatible skills. This is a rough upper bound on the number of compatible teams, since it is based on compatible skills and not the compatibility of users. The two algorithms perform equally well indicating that optimizing for compatibility makes very little difference. Figure 2(b) shows the average cost of the teams produced and indicates that LCMD is the best choice.

In Figures 2(c) and (d), we report results for teams of varying task sizes using LCMD. As expected, more skills means that more people need to co-operate to complete the task, making it harder to form a compatible team, and more likely to include a distant node. The number of solutions drops steeply for more strict compatibility relations, while it remains more or less constant for NNE and $\mathrm{SBP_H}$.

Finally, we compare our approach with previous work on team formation. Since there is no previous work on team formation on signed network, we create two unsigned *Epinions* networks by (1) ignoring the sign of the edges and (2) deleting the negative edges. We run a team formation algorithm [9] on each of these two networks using the same tasks with $k = 5$ skills as in the previous experiments. In Table 3, we report the percentage of the returned teams that satisfy compatibility for the different compatibility relations. As shown, most of the teams returned are incompatible.

6 RELATED WORK

To the best of our knowledge, our work is the first to address team formation in signed networks.

Team Formation. Lappas et al. [9] were the first to formally define the problem of finding a team of experts using the network structure to quantify the quality of the team as a whole. There is considerable amount of work extending their model, (e.g., [1, 8, 12]), but none of these works considers a signed network.

Signed Networks. There is a fair amount of work on signed networks [13]. A problem somehow related to our work is that of link and sign prediction [3, 11]. However, we differentiate, since we are not interested in predicting future links, but rather in evaluating the compatibility between any two individuals in the network. There is also work on detecting communities in signed networks (e.g., see [14]). The notion of the team is somehow related to that of the community, but the objective of team formation is different.

Structural Balance. There is a rich literature in psychology on positive and negative relations among groups of people using structural balance theory, e.g., [2, 4]. Structural balance has been used e.g., for identifying clusters [5] and polarization in networks [10], finding communities [6].

7 CONCLUSIONS

In this paper, we introduced the novel problem of team formation in a signed network. The problem poses the challenge of defining node compatibility in a signed network. To this end, we provided a principled framework by utilizing the theory of structural balance. In the future, we plan to investigate different ways to combine compatibility and communication cost and to exploit compatibility for other tasks, such as link prediction or clustering.

ACKNOWLEDGEMENTS

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: 1873).

REFERENCES

[1] Aris Anagnostopoulos, Luca Becchetti, Carlos Castillo, Aristides Gionis, and Stefano Leonardi. 2012. Online Team Formation in Social Networks. In *WWW*.
[2] Dorwin Cartwright and Frank Harary. 1956. Structural balance: a generalization of Heider's theory. *Psychological review* 63, 5 (1956).
[3] K. Chiang, N. Natarajan, A. Tewari, and Inderjit S. D. 2011. Exploiting Longer Cycles for Link Prediction in Signed Networks. In *CIKM*.
[4] James A Davis. 1963. Structural balance, mechanical solidarity, and interpersonal relations. *Amer. J. Sociology* 68, 4 (1963).
[5] Lúcia Maria de A. Drummond, Rosa M. V. Figueiredo, Yuri Frota, and Mário Levorato. 2013. Efficient Solution of the Correlation Clustering Problem: An Application to Structural Balance. In *OTM*.
[6] Hongzhong Deng, Peter Abell, Ofer Engel, Jun Wu, and Yuejin Tan. 2016. The influence of structural balance and homophily/heterophobia on the adjustment of random complete signed networks. *Social Networks* 44 (2016).
[7] David Easley and Jon Kleinberg. 2010. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press. http://books.google.co.uk/books?id=atfCl2agdi8C
[8] Mehdi Kargar and Aijun An. 2011. Discovering top-k teams of experts with/without a leader in social networks. In *CIKM*.
[9] Theodoros Lappas, Kun Liu, and Evimaria Terzi. 2009. Finding a team of experts in social networks. In *KDD*.
[10] Duan-Shin Lee, Cheng-Shang Chang, and Ying Liu. 2016. Consensus and Polarization of Binary Opinions in Structurally Balanced Networks. *IEEE Trans. Comput. Social Systems* 3, 4 (2016).
[11] Jure Leskovec, Daniel P. Huttenlocher, and Jon M. Kleinberg. 2010. Signed networks in social media. In *CHI*.
[12] C. T. Li and M. K. Shan. 2010. Team Formation for Generalized Tasks in Expertise Social Networks. In *SocialCom/PASSAT*.
[13] Jiliang Tang, Yi Chang, Charu Aggarwal, and Huan Liu. 2016. A Survey of Signed Network Mining in Social Media. *ACM Comput. Surv.* 49, 3 (2016).
[14] Bo Yang, William Cheung, and Jiming Liu. 2007. Community Mining from Signed Social Networks. *IEEE Trans. on Knowl. and Data Eng.* 19, 10 (2007).

Short paper

A Learning Based Approach to Predict Shortest-Path Distances

Jianzhong Qi[1], Wei Wang[2], Rui Zhang[1], Zhuowei Zhao[1*]
[1]The University of Melbourne, Melbourne, Australia
[2]The University of New South Wales, Sydney, Australia
[1]{jianzhong.qi@, rui.zhang@, zhuoweiz1@student.}unimelb.edu.au, [2]weiw@cse.unsw.edu.au

ABSTRACT

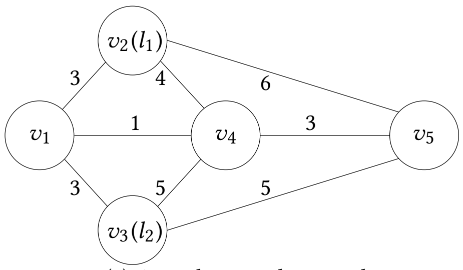
Shortest-path distances on road networks have many applications such as finding nearest *places of interest* (POI) for travel recommendations. To compute a shortest-path distance, traditional approaches traverse the road network to find the shortest path and return the path length. When the distances are needed first (e.g., to rank POIs) while the shortest paths may be computed later (e.g., after a POI is chosen), one may precompute and store the distances, and answer distance queries by simple lookups. This approach, however, falls short in the worst-cast space cost – $O(n^2)$ for n vertices even with various optimizations. To address these limitations, we propose to learn an embedding for every vertex that preserves its distances to the other vertices. We then train a *multi-layer perceptron* (MLP) to predict the distance between two vertices given their embeddings. We thus achieve fast distance predictions without a high space cost. Experimental results on real road networks confirm these advantages. Meanwhile, our approach is up to 97% more accurate than the state-of-the-art approaches for distance predictions.

1 INTRODUCTION

Computing shortest-path distances on road networks with a high efficiency is fundamental for applications such as "finding restaurants within 5 km distance" or "ranking restaurant search results by distance". Real road networks (e.g., Florida road network [8]) may contain millions of vertices, while thousands of users may issue distance queries at the same time (e.g., Google Maps has over a billion active users [1]). Answering distance queries under such settings poses significant challenges in both space and time costs. We aim to address such challenges in this paper.

Problem formulation. We consider a road network graph $G = \langle V, E\rangle$, where V is a set of n vertices (road intersections) and E is a set of m edges (roads). A vertex $v_i \in V$ has a pair of geo-coordinates. An edge $e_{i,j} \in E$ connects two vertices v_i and v_j, and has a *weight* $e_{i,j}.w$, which represents the distance to travel across the edge. Fig. 1a shows an example, where $v_1, v_2, \dots, v_5$ are the vertices, and the numbers on the edges are the weights. For simplicity, in what follows, our discussions assume undirected edges, although our techniques also work for directed edges.

A path $p_{i,j}$ between vertices v_i and v_j consists of a sequence of vertices $v_i \to v_1 \to v_2 \to \dots \to v_x \to v_j$ such that there is an edge between any two adjacent vertices in the sequence. The *length* of $p_{i,j}$, denoted by $|p_{i,j}|$, is the sum of the weights of the edges between adjacent vertices in $p_{i,j}$, i.e., $|p_{i,j}| = e_{i,1}.w + e_{1,2}.w + \dots + e_{x,j}.w$. We are interested in the path $p^*_{i,j}$ between v_i and v_j with the smallest length, i.e., the *shortest path*. Its length is the *(shortest-path) distance* $d(v_i, v_j)$ between v_i and v_j, i.e., $d(v_i, v_j) = |p^*_{i,j}|$. Consider vertices v_1 and v_5 in Fig. 1a. Their



(a) A road network example

	v_1	v_2	v_3	v_4	v_5
v_1	0	3	3	1	4
v_2	3	0	6	4	6
v_3	3	6	0	4	5
v_4	1	4	4	0	3
v_5	4	6	5	3	0

(b) Distance labeling

	$v_2(l_1)$	$v_3(l_2)$
v_1	3	3
v_2	0	6
v_3	6	0
v_4	4	4
v_5	6	5

(c) Landmark labeling

Figure 1: Shortest-path distance problem

distance $d(v_1, v_5) = 4$ is the length of path $v_1 \to v_4 \to v_5$. We aim to predict $d(v_i, v_j)$ given v_i and v_j with a high accuracy and efficiency, which is defined as the *shortest-path distance query*.

Definition 1.1 (Shortest-path distance query). Given two query vertices v_i and v_j in graph G, *a shortest-path distance query* returns the shortest-path distance between v_i and v_j, i.e., $d(v_i, v_j)$.

For simplicity, we use distance to refer to shortest-path distance hereafter as long as the context is clear.

Related work. A simple solution is to use shortest path algorithms (e.g., Dijkstra's algorithm) to compute the shortest paths and then return the path lengths. In applications such as those mentioned above, the distances are needed first (e.g., to rank restaurants by distance) while the shortest paths may be computed later (e.g., after a restaurant is chosen). Studies (e.g., [3, 7, 12]) thus build data structures to enable fast distance queries without online shortest path computations. *Distance labeling* is commonly used. Its basic idea is to precompute a vector of (distance) values for each vertex as its *distance label*. In an extreme case, the distance label of a vertex contains its distances to all other vertices (cf. Fig. 1b). A distance query is answered by a lookup in $O(1)$ time, but this requires $O(n^2)$ storage space for the distance labels. Techniques (e.g., *2-hop labeling* [7] and *highway labeling* [12]) are proposed to reduce the label size. However, for general graphs, the worst-case space costs are still $O(n^2)$ [11].

To avoid the $O(n^2)$ space cost, approximate techniques are proposed [6, 18, 21], among which *landmark labeling* [10, 16, 20] is a representative approach. This approach uses a subset of k ($k \ll n$) vertices as the *landmarks*. Every vertex v_i stores its distances to these landmarks as its distance label, i.e., a k-dimensional vector $\langle d(v_i, l_1), d(v_i, l_2), \dots, d(v_i, l_k)\rangle$, where $l_1, l_2, \dots, l_k \in L$ represent the landmarks and $d(\cdot)$ represents the distance. At query time, the distance labels of the two query vertices v_i and v_j are scanned, where the distances to the same landmark are summed up. The smallest distance sum, i.e., $\min\{d(v_i, l) + d(v_j, l) | l \in L\}$,

[*]The authors are ordered alphabetically.

is returned. In Fig. 1, v_2 and v_3 are chosen as the landmarks (denoted by l_1 and l_2, respectively), and the distance labels are shown in Fig. 1c. The distance between v_1 and v_5 is computed as $\min\{d(v_1,l_1)+d(v_5,l_1), d(v_1,l_2)+d(v_5,l_2)\} = \min\{3+6, 3+5\} = 8$, which is twice as large as the actual distance between v_1 and v_5 (i.e., 4). Even though landmark labeling reduces the space cost to $O(kn)$, it may not return the exact distance. How the landmarks are chosen plays a critical role in the distance accuracy. Since finding the k optimal landmarks is NP-hard [16], heuristics are proposed [16, 19] such as choosing the vertices that are on more shortest paths as the landmarks. Theoretical results (e.g., [6, 15]) are offered to bound the relationship between the label size and the distance accuracy. On undirected graphs, it is shown [21] that any algorithm with an approximation ratio of $\alpha < 2c+1$ ($c \in \mathbb{N}^+$) must use $\Omega(n^{1+\frac{1}{c}})$ space. A structure is proposed [21] using $O(cn^{1+\frac{1}{c}})$ space and $O(cmn^{\frac{1}{c}})$ time to obtain an approximation ratio of $\alpha = 2c-1$ and an $O(c)$ query time. Chechik [6] improves the space cost to $O(n^{1+\frac{1}{c}})$ and the query time to $O(1)$, with an $O(n^2+mn^{\frac{1}{2}})$ prepossessing time. These studies are mainly of theoretical interest. They do not offer empirical results.

Our contributions. To preserve more information in the distance labels and obtain higher distance accuracy, we propose to learn an *embedding* for every vertex as its distance label. Our idea is motivated by recent advances in graph embeddings [4, 5, 9], which show that vertices can be mapped into a latent space where their *structural similarity* (e.g., the number of common neighboring vertices) can be computed. This motivates us to map the vertices into a latent space to compute their *spatial similarity*, i.e., shortest-path distances. We make the following contributions:

(i) We propose a learning based model *vdist2vec* to predict vertex distances. This model learns vertex embeddings while jointly trains a *multi-layer perceptron* (MLP) to predict vertex distances. It has an $O(k)$ distance prediction time and an $O(kn)$ storage space, where k is a small constant denoting the vertex embedding dimensionality. Our model is highly accurate, since the embeddings are guided by distance predictions directly.

(ii) We further propose two models *vdist2vec-L* and *vdist2vec-S* with an improved loss function and an improved model structure to optimize the embeddings for different types of vertices.

(iii) We perform experiments on real road networks. The results show that, comparing with state-of-the-art approaches, our models reduce the distance prediction errors by up to 97%.

2 PROPOSED MODEL

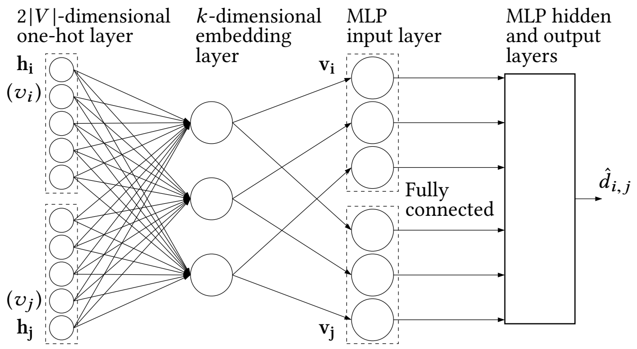


Figure 2: Vdist2vec model structure

Vdist2vec. As Fig. 2 shows, our vdist2vec model takes two vertices v_i and v_j as the input, which are represented as two size-$|V|$ *one-hot vectors* $\mathbf{h_i}$ and $\mathbf{h_j}$. The next layer is an *embedding layer* for representation learning. This layer has k nodes, and its weight matrix is a $|V| \times k$ ($2|V| \times k$ for directed graphs) matrix to be used as the vertex vectors for all vertices, denoted by $\mathbf{V} = [\mathbf{v_1}^T, \mathbf{v_2}^T, ..., \mathbf{v}_{|\mathbf{V}|}{}^T]^T$. Multiplying $\mathbf{h_i}$ ($\mathbf{h_j}$) by $\mathbf{V}$ yields $\mathbf{v_i}$ ($\mathbf{v_j}$):

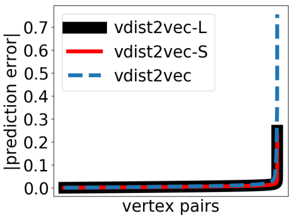
$$\mathbf{v_i} = \mathbf{h_i}\mathbf{V} \tag{1}$$

Vectors $\mathbf{v_i}$ and $\mathbf{v_j}$ are then fed into a distance prediction network (i.e., an MLP) to predict the distance between v_i and v_j. The loss function $\mathcal{L}_d$ is the mean square error on the actual vertex distances $d(v_i, v_j)$ and the predicted distances $\hat{d}_{i,j}$:

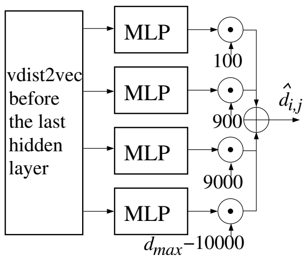
$$\mathcal{L}_d = E_{\mathcal{P}}\left[(d(v_i,v_j) - \hat{d}_{i,j})^2\right] \tag{2}$$

At training, the vertex representation matrix $\mathbf{V}$ is randomly initialized. Vertex pairs are fed into the network in batches to train the MLP. The training loss $\mathcal{L}_d$ will be propagated back to optimize the MLP and the vertex representations in $\mathbf{V}$.

At query time, the query vertex vectors $\mathbf{v_i}$ and $\mathbf{v_j}$ are fetched from $\mathbf{V}$ and fed into the MLP to make a distance prediction.



(a) Error distribution (DG)



(b) vdist2vec-S

Figure 3: Prediction error distribution and vdist2vec-S

Next, we optimize the prediction accuracy further. Our motivation comes from an observation on the distance prediction error distributions. In Fig. 3a, we plot the normalized absolute distance prediction errors of vdist2vec on a real dataset (DG, cf. Section 3). The x-axis represents vertex pairs (sorted by the prediction errors) and the y-axis represents their corresponding prediction errors. We see that a very small portion (e.g., less than 1%) of the vertex pairs have much larger prediction errors (see the spike to the right of the figure) than the other vertex pairs. Such vertex pairs dominate the training error and force the model to focus on them. To optimize the overall prediction accuracy, we need to guide the model to attend more to the other vertex pairs.

Vdist2vec-L. Our first optimization is a novel loss function denoted by $\mathcal{L}_n$ to shrink the larger errors:

$$\mathcal{L}_n = E_{\mathcal{P}}\left[f_\delta(d(v_i,v_j) - \hat{d}_{i,j})\right] \tag{3}$$

$$f_\delta(x) = \begin{cases} \delta|x|, \text{if } |x| \le \delta \\ \frac{1}{2}(x^2+\delta^2), \text{otherwise} \end{cases}$$

Function $f_\delta(\cdot)$ is motivated by the *Huber loss* and is continuously differentiable at $x = \delta$. We set δ as the top 1% largest prediction error after each epoch. If $|x| > \delta$, $\frac{1}{2}(x^2+\delta^2) \le x^2$ which shrinks the error. Replacing $\mathcal{L}_d$ with $\mathcal{L}_n$ results in *vdist2vec-L*.

Vdist2vec-S. Our second optimization is motivated by *ensemble learning*. As Fig. 3b shows, we replace the last hidden layer of vdist2vec with four separate MLPs, each focusing on producing distance predictions in the ranges of (0, 100), (0, 900), (0, 9000), and (0, d_{max}−10000), where d_{max} is the road network diameter. This is done by multiplying ("$\odot$") the sigmoid output of each MLP with 100, 900, 9000, and $d_{max} - 10000$, respectively. The output of the MLPs are summed up to produce the final predictions. We name this model *vdist2vec-S*. Its advantage is in that each MLP can focus on vertex pairs in different distance ranges, making it easier to learn more accurate predictions.

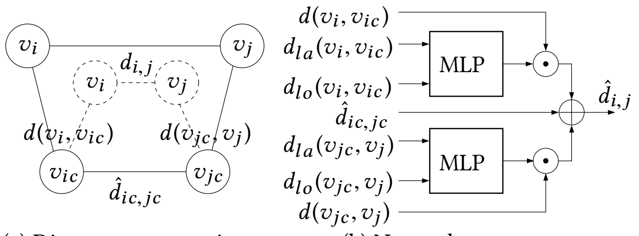
As shown in Fig. 3a, the error distributions of vdist2vec-L and vdist2vec-S are less skewed than that of vdist2vec.

Handling large road networks. Our models compute a $|V| \times k$ embedding matrix. This is cheaper than a $|V|^2$ matrix for all vertex distances. However, we still need to train over $|V|^2$ pairs of vertices. Next, we reduce the number of training pairs.

We cluster (e.g., using *k-means*) the vertices into $|V_c|$ clusters based on their geo-coordinates, where $|V_c|$ is a small constant. The vertex nearest to each cluster center is chosen as a *center vertex.* We train our models over the center vertices. Given query vertices v_i and v_j, their distance $\tilde{d}_{i,j}$ is approximated by their distances to their cluster center vertices v_{ic} and v_{jc} (which are precomputed) plus the predicted distance $\hat{d}_{ic,jc}$ of v_{ic} and v_{jc}:

$$\tilde{d}_{i,j} = \lambda_1 \cdot d(v_i, v_{ic}) + \hat{d}_{ic,jc} + \lambda_2 \cdot d(v_{jc}, v_j) \qquad (4)$$

There are two coefficients λ_1 and λ_2 to weight the contributions of $d(v_i, v_{ic})$ and $d(v_{jc}, v_j)$ based on the relative positions of the vertices (cf. Fig. 4a). To learn λ_1 and λ_2, we build another neural network as shown in Fig. 4b, where $d_{la}(\cdot)$ and $d_{lo}(\cdot)$ return the difference in latitude and longitude between two vertices, respectively. This neural network feeds the coordinate difference between v_i and v_{ic} and the coordinate difference between v_j and v_{jc} into two MLPs to predict λ_1 and λ_2, respectively. The last layer of each MLP uses a *tanh* activation function, which maps λ_1 and λ_2 into the range of $(-1, 1)$. The output of these two MLPs is multiplied ("$\bigodot$") with $d(v_i, v_{ic})$ and $d(v_{jc}, v_j)$, and the products are added ("$\bigoplus$") with $\hat{d}_{ic,jc}$ to produce $\tilde{d}_{i,j}$, which implements Equation 4. For training, we use loss function $\mathcal{L}_d$ but with only a sampled subset of non-center vertices (e.g., $|V_c||V|$ pairs), since the input space (i.e., coordinate difference) is now much smaller.



(a) Distance computation (b) Network structure

Figure 4: Distance prediction for large road networks

Handling updates. Our models can be rebuilt in 13 hours for road networks with over a million vertices (Section 3). This allows us to handle a low update frequency by periodic rebuilds. Our models can also provide distance predictions upon vertex or edge updates without rebuilding, although the accuracy may drop. We leave more robust update handling for future study.

Cost analysis. We consider an MLP to have an O(1) space cost for its parameters and an O(1) time cost for inference. Such costs depend mainly on the model size rather than the input size. Also, the inference can be done by GPUs efficiently. Then, our models can be trained in $O(|V|^2)$ time ($O(|V_c||V|)$ for large road networks). Our models take $O(k|V|)$ space for the embeddings. They take $O(k)$ time to read and feed the query vertex embeddings into the MLP for distance prediction in $O(1)$ time.

3 EXPERIMENTS

We run experiments on a Linux PC with an Intel(R) Xeon(R) E5-2630 V3 CPU (2.40GHZ), a GeForce GTX TITAN X GPU, and 32GB memory. All models are implemented with Python 2.7.12. The neural networks are implemented with Tensorflow 1.13.1.

Datasets. We use six road network datasets as summarized in Table 1, where $\overline{dgr}$ denotes the average degree and d_{max} denotes the diameter. All datasets are undirected except for MB.

Table 1: Datasets

Dataset	$\|V\|$	$\|E\|$	$\overline{dgr}$	d_{max}
Dongguan, China (**DG**) [14]	8K	11K	2.76	96km
Florida, USA (**FL**) [8]	1.07M	1.35M	2.36	1,200km
Melbourne, Australia (**MB**) [2]	3.6K	4.1K	1.14	6km
New York City, USA (**NY**) [8]	264K	366K	2.80	160km
Shanghai, China (**SH**) [14]	74K	100k	2.70	127km
Surat, India (**SU**) [14]	2.5K	3.6K	2.88	50km

Baselines. We compare with five baselines: **landmark-bt** [19]: it uses the top-k vertices passed by the largest numbers of shortest paths between the vertex pairs as the landmarks; **landmark-km**: it uses the k vertices that are the closest to the vertex k-means centroids (computed in Euclidean space) as the landmarks; **ado** [18]: it recursively partitions the vertices into subsets of *well separated vertices* and stores the distance between subsets to approximate the distance between vertices (we tune its approximation parameter ϵ such that it has a similar space cost to ours); **geodnn** [13]: it trains an MLP to predict the distance of two vertices given their geo-coordinates (we use its recommended settings); **node2vec** [17]: it uses node2vec [9] to learn vertex embeddings and trains an MLP to predict vertex distances given the learned embeddings (we use its recommended settings).

Hyperparameters. For our models, the MLP distance prediction component has two hidden layers of 100 and 20 nodes, respectively. We use ReLU as the activation function for the hidden layers and sigmoid for the output layer. We set the batch size to be $|V|$ (we find that a large batch size helps the training efficiency without impacting the prediction accuracy). We initialize the MLP parameters using the truncated normal distribution with 0 as the mean and 0.01 as the standard deviation. The training data is randomly shuffled. We train our model in 20 epochs with early stopping using *AdamOptimizer* and a learning rate of 0.01. Each ensemble MLP of vdist2vec-S has a layer of 20 nodes.

In all approaches except node2vec, we use $k = 2\%|V|$ for DG, MB, and SU, $k = 0.05\%|V|$ for SH, and $k = 0.005\%|V|$ for FL and NY. For node2vec, we use $k = 128$ as suggested by [17].

Evaluation metrics. We predict the distance between every two vertices in each dataset and measure the *mean absolute error* (**MAE**, in meters), *mean relative error* (**MRE**), *precomputation/training time* (**PT**), and *average distance prediction (query) time* (**QT**). The ground truth distances are precomputed using the *contraction hierarchy* algorithm.

Overall results. Table 2 shows the prediction errors. Our models outperform the baseline models across all six datasets and reduce the MAE and MRE by up to 97% and 99% (5 vs. 192 and 0.006 vs. 0.488 for vdist2vec-S and landmark-bt on MB). On NY, landmark-km has a slightly lower MAE than ours, while our MRE is still lower (by more than 50%).

The advantage of our models comes from their capability to learn the vertex distances and preserve them in the embeddings. Landmark-bt and landmark-km rely on the landmarks and may not preserve the distance for all vertices. Ado is designed to control the relative error. It yields the lowest MRE among the baselines on most datasets, but its MAE may be large. Geodnn uses Euclidean distance to approximate shortest-path distance. It suffers on large road networks (e.g., FL and NY) with rivers and large detours. Node2vec focuses on the neighborhood of the vertices. It also suffers on large road networks such as NY (it cannot train on FL in 48 hours which is marked as "OT").

In Table 2, we also show the space required to store the learned embeddings and model parameters on the FL dataset. Geodnn has the smallest space requirement, as it only stores the MLP

Table 2: Mean Absolute and Relative Errors

		DG		MB		SU		FL			NY		SH	
		MAE	MRE	MAE	MRE	MAE	MRE	MAE	MRE	Size	MAE	MRE	MAE	MRE
baseline	landmark-bt	2,234	0.442	192	0.488	468	0.281	OT	OT	OT	24,851	0.167	6,144	0.554
	landmark-km	74	0.028	15	0.040	142	0.090	58,869	0.113	428 MB	**13,431**	0.105	1,314	0.159
	ado	2,108	0.057	75	0.072	642	0.074	175,571	0.052	431 MB	31,737	0.064	4,539	0.147
	geodnn	1,566	0.092	95	0.097	442	0.108	363,661	0.317	**17 MB**	207,694	0.862	14,842	0.990
	node2vec	2,329	0.199	118	0.161	658	0.175	OT	OT	OT	217,400	0.703	19,465	1.276
proposed	vdist2vec	135	0.015	12	0.014	83	0.027	34,757	0.027	30 MB	16,805	**0.052**	1,290	**0.068**
	vdist2vec-L	75	0.015	6	0.014	50	0.024	34,860	0.027	30 MB	16,793	**0.052**	1,290	**0.068**
	vdist2vec-S	**71**	**0.011**	**5**	**0.006**	**49**	**0.014**	**34,329**	**0.026**	33 MB	16,649	**0.052**	**1,287**	**0.068**

Table 3: Preprocessing and Query Times

		DG		MB		SU		FL		NY		SH	
		PT	QT	PT	QT	PT	QT	PT	QT	PT	QT	PT	QT
baseline	landmark-bt	0.1h	5.832μs	62.7s	4.579μs	32.6s	4.463μs	OT	OT	39.6h	11.712μs	14.7h	8.423μs
	landmark-km	**2.2s**	6.024μs	**0.7s**	4.439μs	**0.4s**	4.322μs	**145.1s**	63.718μs	**66.3s**	15.343μs	**9.5s**	8.930μs
	ado	1.0h	1.080ms	0.2h	0.490ms	195.7s	0.356ms	1.5h	0.110ms	0.8h	0.148ms	138s	0.053ms
	geodnn	0.9h	**0.366μs**	0.2h	**0.396μs**	0.2h	**0.375μs**	0.1h	**0.444μs**	0.1h	**0.458μs**	0.1h	**0.432μs**
	node2vec	2.2h	0.829μs	0.9h	0.820μs	0.5h	0.809μs	OT	OT	26.3h	0.751μs	2.8h	0.781μs
proposed	vdist2vec	2.3h	1.039μs	0.9h	0.644μs	0.4h	0.589μs	12.5h	3.981μs	6.1h	1.215μs	0.2h	0.797μs
	vdist2vec-L	2.3h	1.039μs	0.9h	0.644μs	0.5h	0.589μs	12.5h	3.981μs	6.1h	1.215μs	0.2h	0.797μs
	vdist2vec-S	2.8h	1.366μs	1.1h	1.005μs	0.6h	0.921 μs	12.5h	3.981μs	6.1h	1.215μs	0.2h	0.797μs

parameters. Our models store MLP parameters and center vertex embeddings, which require more space than geodnn but less than landmark-km and ado. Note that, if our models learn vertex embeddings for a full graph, we expect a slightly higher space requirement than landmark-km and ado.

Table 3 shows the preprocesing (training) time PT and distance prediction (query) time QT. In terms of PT, the landmark approaches are much faster on the small road networks DG, MB, and SU. Their precomputation is simpler than the training of the learning based models. On large road networks FL, NY, and SH, our models use the proposed clustering based strategy ($|V_c| = 0.4\%|V|$ and 100,000 random vertex pairs to learn λ_1 and λ_2), which reduces the training time significantly. Ado and node2vec need to run on the full road networks. Their PT grows with the road network size. For geodnn, we randomly sample 100,000 vertex pairs for training on the large road networks. It does not learn vertex embeddings and hence has a lower PT.

In terms of QT, the learning based approaches are highly efficient because their distance prediction is a simple forward propagation, which can be done by GPUs efficiently. Geodnn is the fastest, as its input layer only has four nodes (i.e., two geo-coordinates). The other learning based approaches including ours have very similar MLP structures and input sizes which are larger than that of geodnn. Thus, their QT are similar and are larger than that of geodnn. Ado has the largest QT because it needs to first locate the subsets containing the query vertices.

Among our models, vdist2vec-S yields the smallest distance prediction errors, as it can cope with distances in varying ranges. This advantage comes with a larger PT. In contrast, vdist2vec-L has almost the same PT as vdist2vec but achieves smaller distance prediction errors due to its optimized loss function.

4 CONCLUSIONS

We proposed a representation learning based approach for the shortest-path distance problem. Our approach learns vertex embeddings that preserve the distances between vertices, which only take an $O(kn)$ storage space. At query time, the vertex embeddings are fed into an MLP to predict the distance, which takes a constant time. Experimental results show that our approach is highly efficient. It reduces the distance prediction errors by up to 97% comparing with the state-of-the-art. For future work, we plan to extend our techniques to more types of (and larger) graphs such as social networks. We also plan to study real-time updates for learning based distance prediction models.

ACKNOWLEDGMENTS

This work is partially supported by Australian Research Council (ARC) Discovery Projects DP180102050 and DP180103332.

REFERENCES

[1] 2017. Google announces over 2 billion monthly active devices on Android. https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users.

[2] 2017. Planet OSM. https://planet.osm.org.

[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*.

[4] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2015. GraRep: Learning graph representations with global structural information. In *CIKM*.

[5] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2016. Deep neural networks for learning graph representations. In *AAAI*.

[6] Shiri Chechik. 2015. Approximate distance oracles with improved bounds. In *STOC*.

[7] 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.

[8] Camil Demetrescu, Andrew Goldberg, and David Johnson. 2006. 9th DIMACS implementation challenge–Shortest Paths. *American Math. Society* (2006).

[9] Aditya Grover and Jure Leskovec. 2016. Node2vec: Scalable feature learning for networks. In *KDD*.

[10] Andrey Gubichev, Srikanta Bedathur, Stephan Seufert, and Gerhard Weikum. 2010. Fast and accurate estimation of shortest paths in large graphs. In *CIKM*.

[11] Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong, and Yanyan Xu. 2014. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *PVLDB* 7, 12 (2014), 1203–1214.

[12] Ruoming Jin, Ning Ruan, Yang Xiang, and Victor Lee. 2012. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *SIGMOD*.

[13] Ishan Jindal, Xuewen Chen, Matthew Nokleby, Jieping Ye, et al. 2017. A unified neural network approach for estimating travel time and distance for a taxi trip. *arXiv preprint arXiv:1710.04350* (2017).

[14] Alireza Karduni, Amirhassan Kermanshah, and Sybil Derrible. 2016. A protocol to convert spatial polyline data to network formats and applications to world urban road networks. *Scientific Data* 3 (2016), 160046.

[15] David Peleg. 2000. Proximity-preserving labeling schemes. *Journal of Graph Theory* 33, 3 (2000), 167–176.

[16] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. 2009. Fast shortest path distance estimation in large networks. In *CIKM*.

[17] Fatemeh Salehi Rizi, Joerg Schloetterer, and Michael Granitzer. 2018. Shortest path distance approximation using deep learning techniques. In *ASONAM*.

[18] Jagan Sankaranarayanan and Hanan Samet. 2009. Distance oracles for spatial networks. In *ICDE*.

[19] Frank W. Takes and Walter A. Kosters. 2014. Adaptive landmark selection strategies for fast shortest path computation in large real-world graphs. In *WI-IAT*.

[20] Liying Tang and Mark Crovella. 2003. Virtual landmarks for the internet. In *SIGCOMM*.

[21] Mikkel Thorup and Uri Zwick. 2005. Approximate distance oracles. *J. ACM* 52, 1 (2005), 1–24.

Optimizing Data Movement with Near-Memory Acceleration of In-memory DBMS

Donghun Lee[†], Minseon Ahn[†], Jungmin Kim[†], Kangwoo Choi[†], Oliver Rebholz[‡]

Andrew Chang[§], Jongmin Gim[§], Jaemin Jung[§], Vincent Pham[§], Krishna Malladi[§], Yang Seok Ki[§]

[†] SAP Labs Korea [‡] SAP SE
{dong.hun.lee, minseon.ahn, jungmin.kim, kangwoo.choi, oliver.rebholz}@sap.com
[§] Samsung Semiconductor Inc.
{andrew.c1, gim.jongmin, j.jaemin, tung1.pham, k.tej, yangseok.ki}@samsung.com

ABSTRACT

Despite the increase of memory capacity and CPU computing power, memory performance remains the bottleneck of in-memory DBMS due to ever-increasing data volumes and application demands. Since the scale of data workload has outpaced traditional CPU caches and memory bandwidth, it is essential to optimize data movement from memory to computing units. In this work, we present a near-memory Database Accelerator (DBA) framework that offloads data-intensive database operations via or to a near-memory computation engine. DBA's system architecture includes a DBA software module/driver and memory module with DBA engine. We build a Proof-of-Concept (PoC) of DBA using FPGAs with attached DIMMs, and then conduct an experimental evaluation.

1 INTRODUCTION

Low cost and high capacity of DRAM accelerated the market of in-memory database management systems (IMDBMS). The latest IMDBMS architecture capable of running both Online Transactional Processing (OLTP) and Online Analytical Processing (OLAP) applications in a single system removes the data redundancy and provides higher performance and efficiency with lower total cost ownership (TCO) [9]. However, with ever-increasing data volumes and application demands, memory performance becomes the main performance bottleneck of IMDBMSs. Our study with OLTP/OLAP applications shows that performance can be bound by expensive data-intensive operations like table scan and aggregation of OLAP workloads. These data-intensive operations have very little data reuse for further computation but consume more than 50% of CPU resources and almost all memory bandwidth in many cases. The other mission critical workloads suffer from cache conflicts (or cache thrashing) and memory bandwidth bottleneck. Therefore, it is essential to optimize data movement from memory to computing units.

The best way to optimize this data movement in IMDBMS would be to process these data-intensive operations within memory devices. Instead of transferring the whole data to computing units, forwarding the filtered results to the next processing step could minimize the overhead. Near-storage computing [3, 4] tries to accelerate the data-intensive operations by minimizing the data transfer overhead from storage to processing nodes or CPU. However, this research does not deliver byte addressability and much lower latency necessary for IMDBMS. Previous work to accelerate database operations using FPGA [7, 8, 10] and GPGPU [5, 6] shows an order of magnitude performance gain in compute-intensive operations. However, these approaches show a smaller gain in data-intensive operations because of the data movement overhead [1]. Even Hybrid CPU-FPGA approaches [7, 10] require data movement from host memory to accelerator computing units which has a high memory bandwidth overhead.

Processing-In-Memory (PIM) approaches like UPMEM [2] advance the concept of near-memory computing but are still in early stage. Furthermore, the data needs to be reformatted to utilize the processing units, thus the existing data structure cannot be reused directly.

In this paper, we propose near-memory database accelerator (DBA) to offload data-intensive operations of IMDBMS to memory devices. By placing simple arithmetic units near DRAM within memory devices like DIMMs, we 1) save CPU cycles for data-intensive operations, 2) avoid cache thrashing among threads, and 3) reduce the host memory bottleneck. We implement our proof-of-concept (PoC) system using FPGAs with attached DIMMs. Its DBA kernel is designed to perform parallel comparisons in a SIMD manner fully utilizing internal memory bandwidth. Our evaluation shows that near-memory DBA has more than 2 times performance improvement in OLTP workloads when offloading the data-intensive operations. Finally, we discuss the obstacles to embody the approach in real memory devices.

2 BACKGROUND

2.1 Motivational example

Figure 1 shows the performance degradation of OLTP workload by the interference from the scan workloads on OLAP data in the server having 4 sockets and 72 physical cores. The two workloads managed by two separate processes access the different sets of data but compete with each other for limited hardware resources like CPU, cache and memory bandwidth. As the number of scan threads increases, the CPU resources allocated for OLTP workloads are reduced, thus the throughput of OLTP workloads decreases.

It is quite common to apply SIMD instructions to data-intensive operations like scan within a DBMS [11, 12], as SIMD performs the same operation on multiple data points simultaneously exploiting data level parallelism. We observe that the OLTP workloads show a larger performance degradation, when the scan operation is implemented with SIMD commands like AVX2 or AVX 512 because of much higher memory bandwidth usage. As

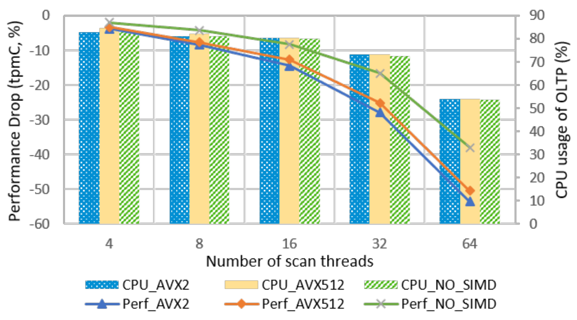


Figure 1: OLTP throughput by number of concurrent scans

Table 1: Memory bandwidth usage (%) by scan workloads

#Threads	AVX2	AVX512	No_SIMD
4	11.3	6.0	0.8
8	21.8	12.3	1.3
16	41.3	23.0	2.8
32	73.8	46.3	5.5
64	94.8	85.3	11.3

shown in Table 1, 64 scan threads consume almost all memory bandwidth of 4 sockets with SIMD, only 12% of memory bandwidth is consumed without SIMD. Interestingly, there is no difference in CPU usage between SIMD and NO-SIMD, but the OLTP throughput shows a larger performance degradation with SIMD. In Figure 1, with 64 scan threads, the CPU usage by OLTP decreased by 30% but the OLTP throughput decreases by about 40% without SIMD and more than 50% with SIMD. This supports our claim that the larger memory bandwidth usage by data-intensive workloads degrades OLTP performance more.

2.2 Scan operation in In-memory DBMS

Recent IMDBMSs are designed to support both OLTP and OLAP workloads and keep the data in the columnar storage for fast read accesses of the tables, storing the majority of data of each column in the read optimized main storage, and maintaining the separate delta storage for optimized writes [9]. The delta storage is periodically merged to the main storage [8]. To reduce the memory footprint (or TCO), the main storage uses dictionary encoding where the distinct values are stored in the dictionary and the individual values are replaced with the corresponding value IDs of the dictionary separately with the bit-packed compression [9]. A scan in IMDBMS reads this value ID array with filter conditions. In this work, the two common scan operations - Range search (having from/to filter conditions) and Inlist search (having a list of filtered values) are offloaded to DBA as they are simple and common data-intensive operations that often consume relatively high CPU usage (5-10% by itself). They include the decompression of value ID (integer) array and return row IDs satisfying the predicates. Offloading only low-level data access operators in the query plans reduces the effort to integrate them with the existing query optimizer.

3 DBA ARCHITECTURE

This section discusses the architecture and design of our proposed near-memory Database Accelerator (DBA). Figure 2 describes the system architecture of DBA. The objective of this work is 1) to demonstrate the offloading feasibility within current ecosystem, 2) to provide a framework to measure the stand-alone DBA engine performance, and more importantly, 3) to study the system impact of our proposal. To remove the unnecessary data movement, database operations are performed by DBA in the device memory where the source data is stored. After DBA completes the operations, the result output is written back to the device memory in FPGA. The host has access to the device memory via memory mapped I/O (MMIO). This eliminates speed and coherency limitations of the PCIe interface from this study and yet leverages the current driver software stack with the OS and the application.

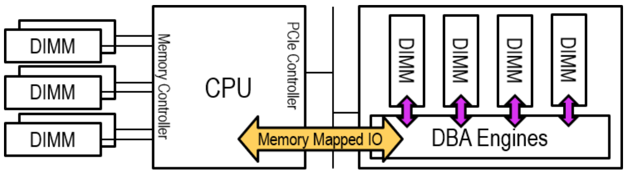


Figure 2: DBA System Architecture

Figure 3 describes the DBA FPGA micro-architecture with functional partitions of host interface, multiple DBA kernels and memory subsystem. The host interface exposes DBA control parameters to the driver that manages offloading from the application API call to the hardware accelerator. Each DBA kernel consists of data prefetcher reading the data, SIMD engine comparing the data with the predicate, and result handler writing the results. The memory subsystem provides the infrastructure to access device memories on the FPGA.

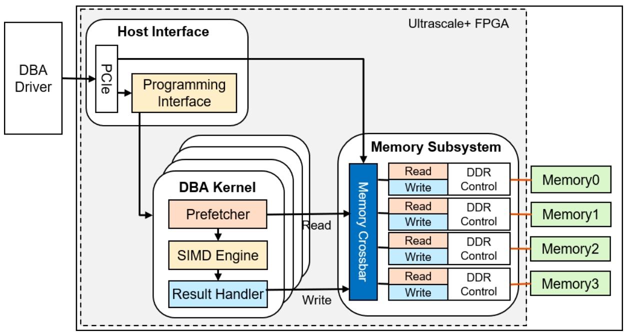
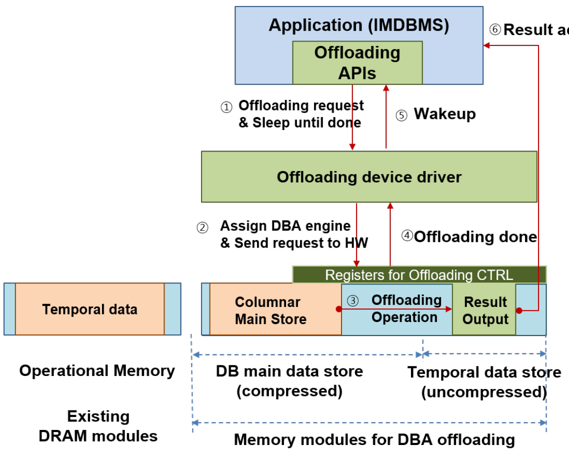


Figure 3: DBA FPGA Micro-Architecture

Internally, DBA kernels read 64B bit-compressed data at a time from the memory. A programmable extractor logic splits the data into multiple values. They are fed into an array of simple processing units and each unit performs a simple comparison independently. The number of parallel units in the array is determined by the number of values in the 64B data so that DBA kernels can keep up with the input data rate. Compared to fixed-length instruction-based processors, each DBA kernel takes the full advantage of parallelism in the data stream due to the flexibility of hardware design. The results are packed into 64B and written back to the device memories. Thus, the data flow is highly optimized for available memory bandwidth.

Figure 4 describes DBA software architecture. Unlike GPU/FPGA accelerators, the DBA engine is located within memory devices. Hence, it allows zero data copy with performance and energy gains. The DBA device driver assigns one DBA engine to a thread of the IMDBMS application per request. Once offloaded, a thread that requested an offloading yields to free CPU for processing. When offloading is done, DBA driver wakes up the requester to resume.



Application (IMDBMS)
Offloading APIs
⑥Result access
① Offloading request & Sleep until done
⑤ Wakeup
Offloading device driver
② Assign DBA engine & Send request to HW
④Offloading done
Registers for Offloading CTRL
Temporal data
Columnar Main Store
③ Offloading Operation
Result Output
Operational Memory
DB main data store (compressed)
Temporal data store (uncompressed)
Existing DRAM modules
Memory modules for DBA offloading

Figure 4: DBA Software Architecture

Normally, applications use a virtual address (VA) to access memory in the host system while the DBA engines access memory with a device physical address (DPA). This implies that the DBA driver is responsible to translate all VA parameters of an offloading request into DPA. The DBA driver first obtains the corresponding system physical address (SPA) by referring to a page table. Then, converting DPA to SPA is trivial because the system BIOS has stored the start SPA of device memory in the Base Address Registers (BAR) of the PCI device at boot time.

4 EVALUATION

4.1 Experimental setup

The system consists of the embedded TPCC benchmark in an IMDBMS and a separate micro-benchmark program to generate scan workloads in a single server as shown in Figure 5.

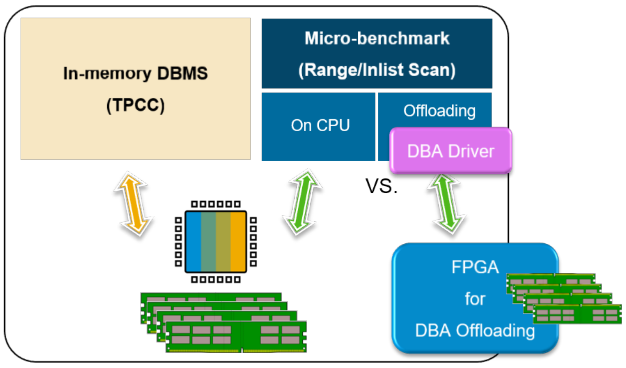


Figure 5: Conceptual diagram

In our experiments, we use the TPCC benchmark for OLTP workload. Its generator is embedded within the IMDBMS engine to remove the communication and session management overhead because the total throughput is usually bound by the session layer, not IMDBMS engine. We want to maximize the throughput (i.e. resource consumption) of the TPCC workload.

The micro-benchmark performs the scan workloads 1) on CPU, or 2) via FPGA. Its data is randomly generated and bit-compressed. The separate data for scans avoids the internal overhead of IMDBMS like locking by two different workloads and enables us to focus on the performance effect by hardware resources. In our experiment, scans read 2 billion bit-compressed integer values and return the row IDs satisfying the filter conditions. When it runs on CPU, the same number of scan threads are bound to each socket to prevent workload skew among the sockets on the 4-socket server (Intel Xeon Gold 6140@2.30GHz, 18 cores and 6 * 64 GB memories per socket). For DBA offloading, we attach one Ultrascale+ FPGA@250MHz per socket and populate 6 scan engines with 4 * 64 GB DDR4 DIMMs @1866MHz per FPGA. The scan data is copied to the memory in each FPGA to emulate that DBA offloading runs within memory devices where the data resides. We compare the performance variation of TPCC workloads and measure the latency and throughput scalability of scan workloads in both options (on CPU vs. on FPGA), while the number of scan threads increases.

4.2 Evaluation results

This section summarizes our DBA PoC evaluation results compared with a state-of-art 4-socket Skylake system having 72 physical cores.

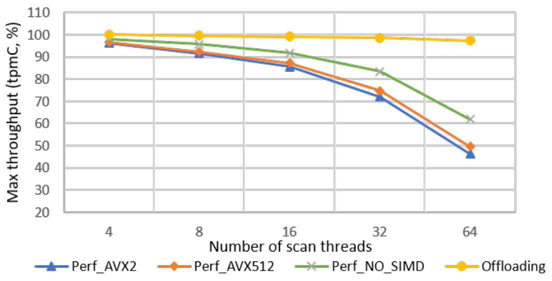


Figure 6: OLTP throughput gain by DBA

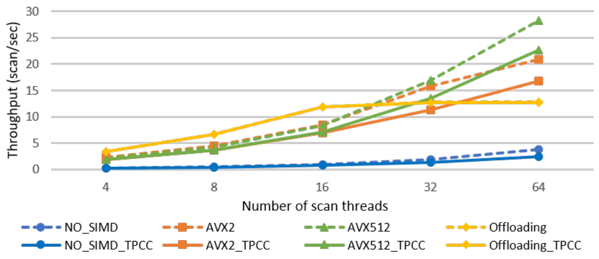


Figure 7: Scan throughputs with/without TPCC

Figure 6 demonstrates the system performance gain of IMDBMS. While TPCC workload runs in the server, the scan micro-benchmark runs on either CPU or DBA with a different number of threads. As a result, DBA offloading shows less performance slowdown as the number of scan threads increases. Therefore, DBA offloading shows 115% better tpmC (transactions per minute) in TPCC workloads when all 64 scan threads are offloaded than when 64 threads use AVX2 on CPU. The results confirm that DBA offloading can alleviate CPU conflict, cache thrashing and memory bandwidth conflict by data-intensive operations.

Table 2: Average latency (sec/scan) of a single scan

Latency of single scan (sec/scan)	On CPU			Offloading
	NO_SIMD	AVX2	AVX512	
	4.16	0.44	0.47	0.29

DBA offloading shows the better performance in scan operation itself, when scans run without OLTP workloads. DBA offloading shows 1.5x better latency (sec/scan) than AVX2 and 14.3x better than NO-SIMD as shown in Table 2.

As for the throughput (scans/sec) scalability, DBA offloading shows quite promising performance as shown in Figure 7. The solid lines represent the throughputs of scans when TPCC workloads are executed concurrently. The dotted lines mean the scan throughputs without TPCC workloads. DBA Offloading shows similar performance regardless of the presence of TPCC workloads, while scan with SIMD/NO-SIMD shows significant performance drop because of the interference from TPCC workloads. DBA offloading outperforms the scan with SIMD/NO-SIMD up to 16 threads and shows similar performance to SIMD scan with 32 threads. With the current implementation, each DBA FPGA has 6 scan engines and 4 DIMM slots. The results show the throughput by DBA offloading is saturated with 16 threads (4 threads per FPGA) because of the limited memory bandwidth of 4 memory channels and resources within the FPGA. Each CPU has 6 DDR channels with 128GB/sec bandwidth while each FPGA has 4 with 60GB/sec. When DBA has the same number of threads, it performs better than CPU running with SIMD. In a SoC (System on Chip) implementation where DBA offloading is embedded in real memory devices, these limitations will be relieved, and the overall performance will be improved further by higher clock frequency or more DBA engines.

In our work, we have the similar performance gain with both range and inlist scans, and similar results regardless of bit-cases used in bit-packed compression [9]. Due to the limited space in this paper, we show only the results of the range scan.

5 DISCUSSION

This research was done using FPGAs with attached DIMMs. The host system accesses the device memory through PCIe MMIO by mapping the device memory in the same address space of the host memory. Even with the slow performance of MMIO in PCIe, the offloading performance is not affected, because our offloading implementation only accesses the local device memory on FPGA once offloading operation starts.

DBA offloading can be implemented on the diverse memory form-factors with their own pros and cons. DIMM-based memory is quite common and very fast, but the memory controller will naturally interleave the data among memory channels. Therefore, even a single value can be crossed on two DIMMs and the DBA driver should handle the data interleaving while processing offloaded operations.

Recently proposed interfaces like CXL(Compute Express Link), Gen-Z and OpenCAPI will enable a new memory pool hosting the columnar main storage in IMDBMS. Although these interfaces introduce a bit higher latency than DIMM, the memory devices are not part of host memory controller pool where data are typically interleaved at 64B granularity. This allows DBA to assume a contiguous data layout in its attached memory and operates without considering data interleaving across memory channels. One more hurdle of DBA offloading would be non-contiguity in the physical address space of the contiguous data in the virtual address space. DBA offloading will provide so-called 'scatter and gather' feature by building a page translation table.

In Clouds, the micro-services of IMDBMS can be spread out among several nodes according to its role. The front-end computing nodes to process the transactions may be easily scaled out, but the storage node cannot be done simply having the same issues on our claim. We believe DBA offloading can contribute to resolving them in Clouds as well.

6 CONCLUSION

We showed that the OLTP-like mission critical workloads can interfere with data-intensive operations like massive scans. We proposed a near-memory database accelerator (DBA) to optimize the data movement and showed that performing the expensive scan operations in the memory devices can alleviate CPU load, cache conflict, host memory bandwidth bottleneck. To confirm its feasibility, we implemented the offloading system using FPGAs with attached DIMMs. Its results show more than 2x performance gain in OLTP workload when offloading the data-intensive operations, and higher or similar throughput scalability with better latency in offloaded scan workloads.

Aggregation is another data-intensive operation in IMDBMS consuming about 20-50% of CPU usage depending on the workloads. While it reads large amounts of data, most of it is rarely referenced again. DBA offloading on aggregation is being investigated as the next target operation.

ACKNOWLEDGMENTS

The authors would like to thank Ismail Oukid for his valuable feedback and contribution in developing our implementation.

REFERENCES

[1] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, and etc. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. 316–331.
[2] Febrice Devaux. 2019. True Processing in Memory with DRAM accelerator. *Hot Chips 31* (2019).
[3] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, and etc. 2016. Biscuit: A Framework for Near-data Processing of Big Data Workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. 153–165.
[4] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, and etc. 2016. YourSQL: A High-performance Database System Leveraging In-storage Computing. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 924–935.
[5] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2017. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *Proc. VLDB Endow.* 10, 7 (March 2017), 733–744.
[6] Tomas Karnagel, Renè Müller, and Guy M. Lohman. 2015. Optimizing GPU-accelerated Group-By and Aggregation.. In *ADMS@VLDB*. 13–24.
[7] Nusrat Jahan Lisa, Annett Ungethüm, Dirk Habich, Wolfgang Lehner, Tuan D. A. Nguyen, and Akash Kumar. 2018. Column Scan Acceleration in Hybrid CPU-FPGA Systems. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2018, Rio de Janeiro, Brazil, August 27, 2018*. 22–33.
[8] J. McGlone, P. Palazzari, and J. B. Leclere. 2018. Accelerating Key In-memory Database Functionality with FPGA Technology. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–8.
[9] Hasso Plattner. 2014. The Impact of Columnar In-memory Databases on Enterprise Systems: Implications of Eliminating Transaction-maintained Aggregates. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1722–1729.
[10] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 403–415.
[11] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Färber. 2013. Vectorizing Database Column Scans with Complex Predicates.
[12] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, and etc. 2009. SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 385–394.

Short paper

ODSA: Open Database Storage Access

James Wagner
DePaul University
jwagne32@depaul.edu

Alexander Rasin
DePaul University
arasin@cdm.depaul.edu

Dai Hai Ton That
DePaul University
dtonthat@depaul.edu

Tanu Malik
DePaul University
tanu@depaul.edu

Jonathan Grier
Grier Forensics
jdgrier@grierforensics.com

ABSTRACT

Applications in several areas, such as privacy, security, and integrity validation, require direct access to database management system (DBMS) storage. However, relational DBMSes are designed for physical data independence, and thus limit internal storage exposure. Consequently, applications either cannot be enabled or access storage with ad-hoc solutions, such as querying the ROWID (thereby exposing physical record location within DBMS storage but not OS storage) or using DBMS "page repair" tools that read and write DBMS data pages directly. These ad-hoc methods are difficult to program, maintain, and port across various DBMSes.

In this paper, we present a specification of programmable access to relational DBMS storage. Open Database Storage Access (`ODSA`) is a simple, DBMS-agnostic, easy-to-program storage interface for DBMSes. We formulate novel operations using `ODSA`, such as comparing page-level metadata. We present three compelling use cases that are enabled by `ODSA` and demonstrate how to implement them with `ODSA`.

1 INTRODUCTION

Relational DBMSes adhere to the principle of physical data independence: DBMSes expose a logical schema of the data while hiding its physical representation. A logical schema consists only of a set of relations (i.e., the data). On the other hand, a physical view consists of several objects, such as pages, records, directory headers, etc. Hiding physical representation is a fundamental design of relational DBMSes: DBMSes transparently control physical data layout and manage auxiliary objects for efficient query execution. However, data independence inhibits several security and performance applications requiring low-level storage access. A small example is provided here, while Section 2 presents more detailed use cases.

Example 1. Consider a bank or a hospital that manages sensitive customer data with a commercial DBMS. For audit purposes, they must sanitize deleted customer data to ensure that it *cannot* be recovered and stolen. Very few DBMSes support explicit sanitization of deleted data (e.g., `secure delete` in SQLite exists but provides no guarantees or feedback to the user)[1]. To programmatically verify the destruction of deleted data, a DBA must inspect *all* storage ever used by a DBMS where such data may reside. This includes DBMS auxiliary objects such as indexes, unallocated fragments in DBMS storage, as well as any DBMS storage released to the OS.

[1]DBMS encryption is similar in not providing any feedback. Furthermore, encrypted values should still be destroyed on deletion.

Comprehensive DBMS storage-level access is an inherent challenge due to DBMS storage management. DBMSes control *allocated* storage objects such as a) physical byte representation of relations, b) metadata to annotate physical storage of relation data, and c) auxiliary objects associated with relations (e.g., indexes, materialized views). Users can manipulate allocated objects exposed by SQL. However, as illustrated in Example 1, the DBA may also need access to *unallocated* storage objects not tracked by a DBMS such as deleted data that lingers in DBMS-controlled files, and DBMS-formatted pages released back to the OS and no longer under DBMS control (e.g., files deleted by the DBMS or OS paging files). These objects are certainly part of the physical view and required for any storage access, but currently not exposed by any DBMS. Vendors such as Oracle incorporate the `DBMS_REPAIR` package [3], enabling users to manually fix or skip corrupt pages, but such tools only access DBMS-controlled storage.

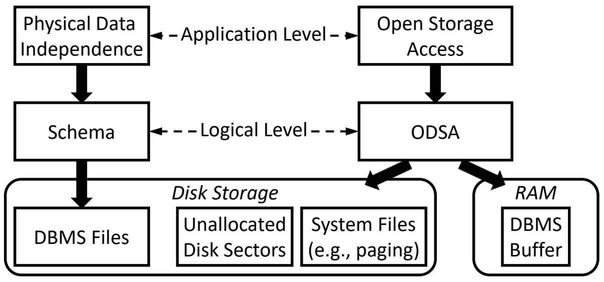


Figure 1: `ODSA` storage access.

In order to enable such security and performance applications, we present Open Data Storage Access (ODSA), an API that provides comprehensive access to all DBMS metadata and data in both (unallocated and allocated) persistent and volatile storage. `ODSA` does not instrument any RDBMS software; it interprets underlying data using database carving methods [8], which we use to expose physical level details. Carving itself is insufficient because the carved data consists of disk-level details making it difficult to program DBMS storage. `ODSA` abstracts low-level disk-level details with a hierarchical view of DBMS storage that is familiar to most DBAs. In particular it organizes them into pages, records, and values, which are resolved to internal, physical addresses. `ODSA` also guarantees the same hierarchy applies to multiple DBMS storage engines, ensuring portability of programmed applications. Figure 1 shows the storage access enabled by `ODSA`.

The rest of the paper is organized as follows. Section 2 presents three representative uses cases that require storage-level access. Section 3 provides an overview of how applications previously had limited access to internal DBMS storage. Section 4 describes the hierarchy exposed by `ODSA` and how it provides a comprehensive view of storage. Section 5 demonstrates implementation and use of `ODSA`. Finally, Section 6 discusses future work for `ODSA`.

2 USE CASES

This section presents three representative use cases that require direct access to different abstractions of storage.

2.1 Intrusion Detection

A bank is investigating mysterious changes to customer data. Unbeknownst to the bank, a disgruntled sysadmin modified the DBMS file bytes at the file system level. This activity bypassed all DBMS access control and logging, and still effectively altered account balances. The sysadmin also disabled file system journaling with `tune2fs` to further hide their activity. The bank cannot determine the cause for inconsistencies with the logs alone. Forensic analysis [7, 9] that detects such malicious activity requires comprehensive storage access to compare volatile storage with allocated and unallocated persistent storage.

2.2 Performance Reproducibility

Alice, an author, wants to share her computation and data based experiments with Bob so he can repeat and verify Alice's work. Out of privacy and access constraints, Alice builds a container consisting of necessary and sufficient data for Bob to reproduce. If the shared data is much smaller than original DBMS file, Bob cannot reproduce any performance-based experiment as the data layout of the smaller data will significantly differ from the original layout. To achieve a consistent ratio between Alice's experiment and Bob's verification, data layout specification at the record and page level must itself be ported. Currently, data layouts as part of a shared DBMS file in a container cannot be communicated [4].

2.3 Evaluating Data Retention

Continuing with Example 1 (Section 1), the bank validates their compliance with data sanitization regulations (e.g,. EU General Data Protection Regulation or GDPR [5]). After deleting data, the bank independently validates data destruction to ensure compliance. No data sanitization validation guidelines for DBMSes exist beyond a complete file overwrite [2]. This guideline is too coarse, especially for DBMS files containing a few deleted records.

Alternatively, consider a compliance officer that has programmatic access to DBMS storage via `ODSA` for validation. The officer can easily access all unallocated storage, and determine the location of deleted data that was not destroyed (e.g., DBMS index or table file, OS paging file).

3 RELATED WORK

We describe built-in tools and interfaces supported by popular DBMSes, which provide physical storage information at different granularities, but no comprehensive views of storage. The ROWID pseudo-column represents a record's physical location within DBMS storage (not disk), and is one of the simplest examples of storage-based metadata available to users most DBMSes. Commercial DBMSes typically provide utilities to inspect and fix page-level corruption. Examples include Oracle's `DBMS_REPAIR`, Oracle's `BBED` (a page editing tool available from Oracle 7 to 10g), and SQL Server's `DBCC CHECKDB`. However, even for accessible metadata such as ROWID, built-in tools do not help interpret its meaning; a DBA must manually make such interpretations. Moreover, no DBMS offers access to unallocated storage. Finally, existing tools only consider persistent storage. `ODSA` offers a universal meaning of DBMS storage (including IBM DB2, Microsoft SQL Server, Oracle, MySQL, PostgreSQL, SQLite, Firebird, and Apache Derby) with support for both persistent and volatile storage.

The term *carving* refers to interpreting data at the byte-level, e.g., reconstructing deleted files without the file system. Wagner et al. previously extended carving to interpret DBMS storage with `DBCarver` [8, 10, 11], retrieving both allocated and unallocated data and metadata without relying on the DBMS. `DBCarver` reads individual files or disk/RAM snapshots and extracts data, including user data and system metadata; it then writes data to a DB3F [12] formatted file. This paper uses `DBCarver` to demonstrate the physical information a DBMS can provide.

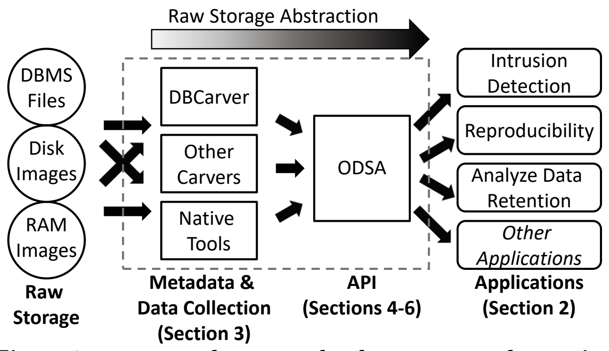


Figure 2: `ODSA` completes raw database storage abstraction in an end-to-end process for storage access.

4 OPEN DATABASE STORAGE ACCESS

Figure 2 shows how `ODSA` relies on carving to access raw storage. `ODSA` abstracts two details from raw storage.

First, it interprets each sequence of raw bytes and classifies it into a physical storage element: **Root**, **DBMS Object**, **Page**, **Record**, or **Value**. Thus, given a collection of interpreted raw storage elements, `ODSA` provides a hierarchical access to these elements by linking them. We provide a brief description of the hierarchy. The root level represents the entry point from all other data to be reached. DBMSes manage their own storage, and a disk partition consisting of both Oracle and PostgreSQL pages, will result in two DBMS roots. The DBMS object level calls return metadata, data, and statistics describing a DBMS object, such as a list of pages or column data types. Pages are uniquely identified by a byte offset in raw storage, rather than the PageID. We also do not rely on the page row directory pointers because deletion may zero out a record's entry.

Second, the `ODSA` hierarchy hides DBMS heterogeneity by accessing physical elements (e.g., pages, records) with physical byte offsets, rather than DBMS-specific pointers.

```
#4.A. Root
class Root:
    def __init__(self, db3f):
        #Initialize
    def get_object_ids(self):
        #Return a list of object ids
    #Calls to Other Instance and Namespace Data
#4.B. Object
class DBMS_Object(Root):
    def __init__(self, parent, object_id):
        #Initialize
    def get_page_offsets(self):
        #Return a list of page offsets
    def get_object_type(self):
        #Return the object type string
    def get_object_schema(self):
        #Return a list of column datatypes
#4.C. Page
class Page(Object):
    def __init__(self, parent, page_offset):
        #Initialize
    def get_record_offsets(self):
        #Return a list of record offsets
    def get_page_id(self):
        #Return a string for page id
    def get_page_type(self):
        #Return a string for page node type
    def get_checksum(self):
        #Return a string for the checksum
    def get_row_directory(self):
        #Return a list of row pointers
#4.D. Record
class Record(Page):
    def __init__(self, parent, record_offset):
        #Initialize
    def get_value_offsets(self):
        #Return a list of value positions
    def get_record_allocation(self):
        #Return Boolean allocation status
    def get_record_row_id(self):
        #Return a string for the row id
    def get_record_pointer(self):
        #Return a string for row pointer
#4.E. Value
class Value(Record):
    def __init__(self, parent, value_offset):
        #Initialize
    def get_value(self):
        #Return string for a data value
```

Figure 3: A sample set of ODSA calls.

Computing a DBMS pointer varies between vendors. For example, Oracle incorporates FileID into index pointer while PostgreSQL does not; index pointers in MySQL differs from both Oracle and PostgreSQL because MySQL relies on index organized tables. Even if all vendors used similar pointer encodings, abstraction is needed in terms of pages since duplicate pages may exist across a storage medium (outside of DBMS-controlled storage, such as paging files). Given Page$_A$ and its physical copy Page$'_A$, ODSA enables application developers to connect an index pointer referencing Page$_A$ along with Page$'_A$.

Implementation. There are multiple ways to implement the hierarchy. The ODSA hierarchy is currently implemented as a pure object hierarchy (Figure 3) and as a relational schema (Figure 4). The pure object hierarchy is stored as a JSON file in the DB3F format [12]. The relational schema is a starting representation – it supports basic applications and is normalized to 3NF requirements. A relational schema is realized since application developers

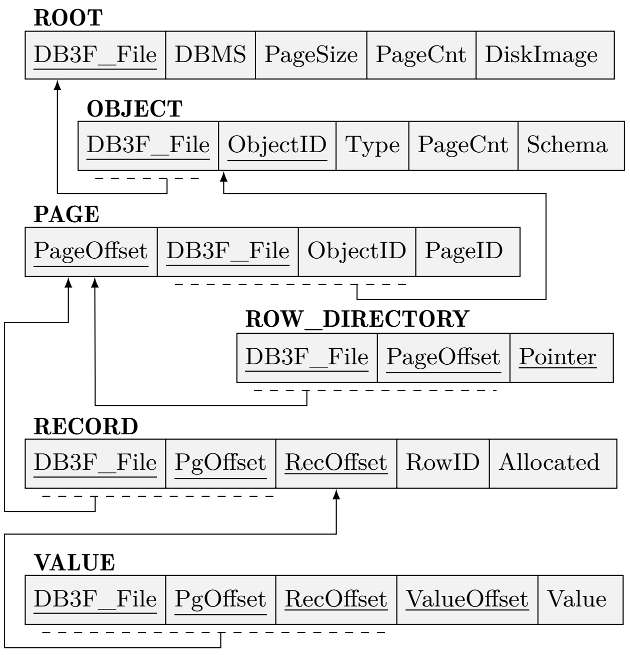


Figure 4: The relational schema used to store ODSA data.

may prefer to access a DBMS storage with SQL rather than calling the ODSA directly. However, as we show in Section 5 the SQL implementation requires several joins and is quite counter-intuitive, despite it being DBMS physical storage.

5 USING ODSA

For use cases in Section 2, two fundamental physical storage access operations are finding unallocated records and matching index pointers to records. ODSA calls enable these operations and show how these operations are achieved in Python and SQL, respectively. The two implementations are shown to contrast programmatic verbosity and maintainability. We focus on ODSA access and do not consider implementation performance.

Example 2: Find Unallocated Records. Use cases 2.1 and 2.3 require a DBA to search and retrieve unallocated records. To retrieve unallocated records, the user must know the carved DBMS file name and the table name (*Customer* table in this example) from which unallocated records are considered. Figure 5 finds and prints all unallocated (e.g., deleted) records from the *Customer* table. All ODSA calls are highlighted.

The implementation in Figure 5 uses ODSA calls to search for unallocated records: Line 3 retrieves page offsets, which uniquely identify pages. Line 5 then iterates through the pages, Line 6 loads each page, and Line 7 retrieves the record offsets for that page. Finally, Line 7 iterates through records using their identifying offsets within a page. Line 11 retrieves the record allocation status to identify and print unallocated records. The same search and retrieval requires an 8-way join in SQL due to the data hierarchy:

```
SELECT PageOffset, RecordOffset, ValueOffset, Value
FROM Object NATURAL JOIN Page
NATURAL JOIN Record NATURAL JOIN Value
WHERE Object.DB_File = 'MyDatabase1.json'
AND Object.ObjectID = 'Customer'
AND Record.Allocated = FALSE;
```

```
DBRoot = odsa.Root('MyDatabase1.json')
CustomerTable = odsa.Object(DBRoot, 'Customer')
PageOffsets = CustomerTable.get_page_offsets()

for PageOffset in PageOffsets:
  CurrPage = odsa.Page(CustomerTable, PageOffset)
  RecordOffsets = CurrPage.get_record_offsets()

  for RecOffset in RecordOffsets:
    CurrRecord = odsa.Record(CurrPage, RecOffset)
    allocated = CurrRecord.get_record_allocation()
    #print unallocated (e.g., deleted) record
    if not allocated:
      print CurrRecord
```

Figure 5: Using ODSA to find deleted records.

```
1  def findIndexEntries(record, Index):
2    RecordPtr = record.get_record_pointer()
3    IndPageOffsets = Index.get_page_offsets()
4
5    for IndPageOffset in IndPageOffsets:
6      IndPage = odsa.Page(Index, IndPageOffset)
7      IndROffsets = IndPage.get_record_offsets()
8
9      for IndROffset in IndROffsets:
10       IndEntry = odsa.Record(IndPage, IndROffset)
11       # IndEntry is a pair (Value, Pointer)
12       IndexPointer = odsa.Value(IndEntry, 1)
13       if IndexPointer == RecordPtr:
14         print IndEntry
```

Figure 6: Using ODSA to find all index entries for one record

Example 3: Match a Record to an Index Pointer(s). To match a record to pointers in a DBMS object such as an index, the user provides as input specific instances of the record and index objects. In Figure 6, Line 5 iterates through all index pages to determine if the input record matches any of the index records. Recall, in an index, records are value-pointer pairs. The code in Figure 6 determines offsets of all index pages (Line 7), and for each index page (Line 9) iterates over all index records in that page. Lines 10 fetches the index entry and Line 12 loads the pointer (offset 1 in value-pair) of the current index entry. Finally, for any index pointer match to the record pointer (Line 13), the index entry is printed.

In this example a brute-force iteration over *all* index pages is necessary, i.e., the program cannot break at the first occurrence of a match in Line 13. In practice, DBMS indexes often contain records of entries that were deleted or updated. For example, consider the record *(42, Jane, 555-1234)* in the *Customer* table where *name* column is indexed. In addition to the expected *(Jane, {PAGEID: 12, ROWID: 37})* entry in the index, the index may also contain *(Jehanne, {PAGEID: 12, ROWID: 37})* if the customer changed their name from Jehanne to Jane (old index entries will only be purged after the index is rebuilt). Moreover, the index might also contain a *(Bob, {PAGEID: 12, ROWID: 37})* entry if another customer named Bob previously deleted their account, free-listing the space for Jane's record at the same location.

As demonstrated in Figure 6, the Python-specific implementation retrieves all records. On the contrary, matching a record to an index in SQL requires a dynamic SQL (shown below) in which after the customary 8-way join to find record values, parameters of each record value must be supplied to match the values. Moreover, this query assumes that there is only one indexed column which is transparently accounted for in the abstraction of the DBMS Object class.

```
SELECT V1.Value
FROM Page NATRUAL JOIN Record
NATURAL JOIN Value V1 NATURAL JOIN Value V2
AND Page.ObjectID = ? --Index name placeholder
AND V1.ValueOffset = 0 --Indexed value at offset 0
AND V2.ValueOffset = 1 --Pointer is at offset 1
AND V2.Value = ( SELECT Record.Pointer FROM Record
     WHERE (DB_File, PageOffset, RecordOffset) =
           (?, ?, ?) /*Record ID placeholders*/);
```

6 CONCLUSION

ODSA was designed based on the principles and challenges described in [1, 6]. In particular, it was designed to be simple and easy-to-use by integrating the terminology used across DBMS documentation. Classes were named based on general concepts giving them an intuitive meaning while abstracting DBMS-specific implementation details. ODSA adheres to single-responsibility principle in that calls focus on single pieces of data and metadata. ODSA supports both 3^{rd} party carving and built-in DBMS mechanisms should vendors choose to expose storage. As a result, ODSA complements physical data independence and enables simple yet powerful implementations of a variety of applications that require access to storage. Additional requirements such as versioning and backward compatibility are future work.

ACKNOWLEDGMENTS

This work was partially funded by the US National Science Foundation Grants CNF-1656268 and CNS-1846418.

REFERENCES

[1] Joshua Bloch. 2006. How to design a good API and why it matters. In *ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 506–507.

[2] Intl. Data Sanitization Consortium. 2019. Data Sanitization Terminology. https://www.datasanitization.org/ data-sanitization-terminology/.

[3] Oracle. 2019. Database Administrator's Guide: Repairing Corrupted Data. https://docs.oracle.com/database/121/ADMIN/ repair.htm#ADMIN022

[4] Quan Pham, Tanu Malik, Boris Glavic, and Ian Foster. 2015. LDV: Light-weight database virtualization. In *IEEE International Conference on Data Engineering*. IEEE, 1179–1190.

[5] General Data Protection Regulation. 2016. Regulation (EU) 2016/679. *Official Journal of the European Union (OJ)* 59, 1-88 (2016), 294.

[6] Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009), 27–34.

[7] James Wagner et al. 2017. Carving database storage to detect and trace security breaches. *Digital Investigation* 22 (2017), S127–S136.

[8] James Wagner et al. 2017. Database forensic analysis with DBCarver. In *Conference on Innovative Data Systems Research*.

[9] James Wagner et al. 2018. Detecting database file tampering through page carving. In *21st International Conference on Extending Database Technology*.

[10] James Wagner, Alexander Rasin, and Jonathan Grier. 2015. Database forensic analysis through internal structure carving. *Digital Investigation* 14 (2015), S106–S115.

[11] James Wagner, Alexander Rasin, and Jonathan Grier. 2016. Database image content explorer: Carving data that does not officially exist. *Digital Investigation* 18 (2016), S97–S107.

[12] James Wagner, Alexander Rasin, Karen Heart, Rebecca Jacob, and Jonathan Grier. 2019. DB3F & DF-Toolkit: The Database Forensic File Format and the Database Forensic Toolkit. *Digital Investigation* 29 (2019), S42–S50.

An Integrated Graph Neural Network for Supervised Non-obvious Relationship Detection in Knowledge Graphs

Phillipp Müller*, Xiao Qin*, Balaji Ganesan†, Nasrullah Sheikh* and Berthold Reinwald*

IBM Research – Almaden*, IBM Research – India†
{phillipp.mueller, xiao.qin, nasrullah.sheikh}@ibm.com*, bganesa1@in.ibm.com†, reinwald@us.ibm.com*

ABSTRACT

Non-obvious relationship detection (NORD) in a knowledge graph is the problem of finding hidden relationships between the entities by exploiting their attributes and connections to each other. Existing solutions either only focus on entity attributes or on certain aspects of the graph structural information but ultimately do not provide sufficient modeling power for NORD. In this paper, we propose KGMatcher– an integrated graph neural network-based system for NORD. KGMatcher characterizes each entity by extracting features from its *attributes*, *local neighborhood*, and *global position* information essential for NORD. It supports arbitrary attribute types by providing a flexible interface to dedicated attribute embedding layers. The neighborhood features are extracted by adopting aggregation-based graph layers, and the position information is obtained from sampling-based position aware graph layers. KGMatcher is trained end-to-end in the form of a Siamese network for producing a symmetric scoring function with the goal of maximizing the effectiveness of NORD. Our experimental evaluation with a real-world data set demonstrates KGMatcher's 6% to 35% improvement in AUC and 3% to 15% improvement in F_1 over the state-of-the-art.

1 INTRODUCTION

Enterprises are equipped with modern computing power, and excel at storing entities of interest and their relationships generated from daily transactions or operations. Making sense of such linked data has gained increasing importance due to its potential of enabling new services. NORD aims at finding relationships between entities in a knowledge graph where the relationships are not explicitly defined in the data.

One of the first NORD systems [2] was designed to detect credit card fraud and later on gained fame for identifying fake identities in casino businesses. The problem of deciding if two entities share a non-obvious relationship such as "fake identity pair" is challenging. First, the attribute information is an important ingredient to characterize entities. However, the attributes are usually expressed in heterogeneous data structures. Extracting useful features and constructing a unified representation from the attributes through manual feature engineering is tedious and ineffective. Second, two related entities may not share similar attribute properties at all. For example, to trick the registration system, the fake identity is often disguised with totally different demographic and contact information. Instead of predicting solely based on the attribute information, a system should also take the surrounding context into consideration. The "neighbors" of the entities of interest in the graph may provide useful information for the detector to tap into the truth of a non-obvious relationship. However, modeling such complex context is a non-trivial task. Third, the entities that share non-obvious relationship usually have a high degree of separation in the graph. It is crucial for a system to be capable of capturing the global position information of the entities in the context of the entire graph so that the distanced separation can be identified later on.

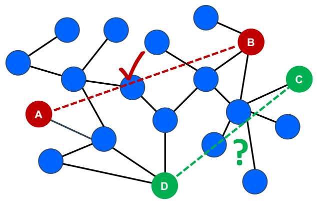


Figure 1: A supervised approach to NORD.

Existing solutions rely on handcrafted features to characterize the entities and domain expert defined rules for the detection. With the rapid growth of the graph size and the increasing complexity of the non-obvious relationships, it is tedious and almost impossible to manually maintain such solutions to achieve high effectiveness. Recently, machine learning approaches designed for similar tasks such as entity resolution [5, 9] and graph link prediction [6] are proposed. However, proven by our experimental results, their approaches are only good at tackling certain aspects of the NORD problem and demonstrate limitations in terms of their overall effectiveness. In this work, we propose KGMatcher– an integrated graph neural network-based system for NORD to address the challenges and overcome the limitations of the existing solutions.

KGMatcher Approach. We approach the problem in a supervised machine learning setting as depicted in Figure 1. That is, in addition to the knowledge graph which contains the entities, their attributes and connections, a set of ground-truth labels indicating the existence of the non-obvious relationships between entities is also available. Our goal is to design a machine learning model that can learn from these pairs and discover new pairs.

In its core, KGMatcher is a neural model that automatically extracts important features from the knowledge graph. The features encode the information regarding their entity *attributes*, *neighborhood* and *position* essential for predicting a non-obvious relationship between two entities. KGMatcher consists of three types of neural layers which are connected and can be trained end-to-end. Attribute features are extracted by the *attribute embedding* layers which support heterogeneous attribute types. The dense representations of the entities generated from these layers as well as the edge information are then fed into two stacks of graph layers. The first type of graph layers called *neighborhood* [1] layer focuses on extracting near-by neighborhood information by aggregating their attribute embeddings to the entity of interest. The second type of graph layers named *position* [8] layer focuses on obtaining global position information of the entities by referring them to a set of sampled anchors. The outputs of the two graph layers are then put together through concatenation to form the final feature vectors of the entities. Finally, KGMatcher's feature extraction network is used in the form of a Siamese network for predicting the existence of a non-obvious relationship between two input entities.

Contributions. It is worthwhile to highlight the following contributions of this work:

(1) We propose a supervised graph neural network-based solution for non-obvious relationship detection called KGMatcher.
(2) We design KGMatcher by adapting and integrating neural architectures for extracting essential NORD features, namely entity *attributes*, *neighborhood* and *position* features.
(3) We demonstrate the effectiveness of KGMatcher using a public dataset. KGMatcher achieves improvement in AUC from 6% to 35% and in F_1 from 3% to 15% over the state-of-the-art.

2 PRELIMINARY

Data Model[1]. Let $G = (V, E)$ denote a knowledge graph where $V = \{v_1, \cdots, v_n\}$ is a set of nodes with each node representing an entity and $E = \{e_1, \cdots, e_m\}$ is a set of edges with each edge $e_k = (v_i, v_j)$ indicating a connection between two entities v_i and v_j. In this work, we assume that the knowledge graph is an undirected graph, i.e., for every $e_k = (v_i, v_j)$, $(v_i, v_j) \equiv (v_j, v_i)$. Let $A = \{a_1, \cdots, a_k\}$ define a set of attributes associated with each entity v_i. An attribute, for example, can be a *date*, an *address*, a *comment*, etc. which means that attributes can be represented in various formats such as numerical, categorical, or text data type. Each entity $v_i = \{x^i_{a_1}, \cdots, x^i_{a_k}\}$ follows the same schema with the attribute types defined by A where $x^i_{a_k}$ denotes the kth attribute value of v_i. In other words, we assume that the entities in the knowledge graph are of the same type, i.e. all the entities share the same attribute types.

Problem Definition. Given two entities v_i and v_j in a knowledge graph G defined above, the goal is to design an algorithm f_G for G that can accurately predict whether or not v_i and v_j share a non-obvious relationship.

In this study, we approach the problem in a supervised learning setting. In addition to the knowledge graph $G = (V, E)$, a ground truth label set $L_G = \{y_1, \cdots, y_r\}$ is also available. A label $y_k = (v_i, v_j)$ where $v_i, v_j \in V$ indicates whether or not there exists a non-obvious relationship between v_i and v_j. Our goal is to design a machine learning model which is able to learn a function f_G from G and L_G that predicts the relationship between two entities presented in G. Following the common practice, the label set is partitioned into train, validation, and test set.

Graph Neural Networks. Graph Neural Networks (GNNs) learn a vector representation of a node from its associated attributes and the graph structure. Modern GNNs [6] adopt a neighborhood aggregation strategy where the representation of a node is learned in an iterative manner by aggregating representations of its neighbors. After k iterations (layers) of aggregation, a node's representation encodes the structural information within its k-hop network neighborhood. Formally, the k-th layer of a GNN is:

$$a^{(k)}_{v_i} = Aggregate^{(k)}\left(\left\{h^{(k-1)}_u | u \in Neighbor(v_i)\right\}\right), \quad (1)$$

$$h^{(k)}_{v_i} = Combine^{(k)}\left(h^{(k-1)}_{v_i}, a^{(k)}_{v_i}\right), \quad (2)$$

[1]The knowledge graph model we adopted in this work can be also seen as a form of the attributed or property graph model referred in the literature.

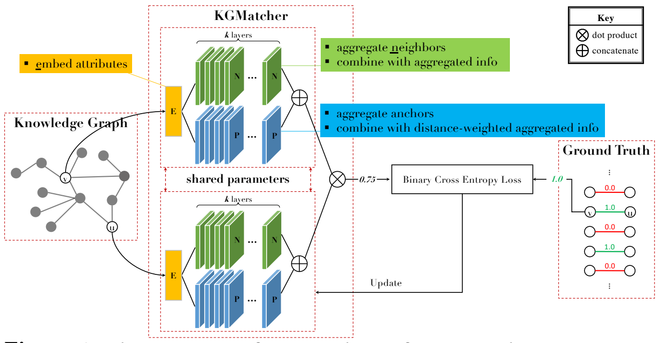


Figure 2: The overview of KGMatcher inference and training.

where $h^{(k)}_{v_i}$ is the vector representation of the node v_i at the k-th iteration. The node's attributes are usually used to initialize $h^{(0)}_{v_i} = Embed(\{x^i_{a_1}, \cdots, x^i_{a_k}\})$ where $Embed(\cdot)$ is an embedding function that obtains a vector representation of the attributes from their raw forms. The exact computation of $Aggregate^{(k)}(\cdot)$ and $Combine^{(k)}(\cdot)$ in GNNs defines their modeling approach. In Section 3.1, we will describe the approaches we introduce to our proposed model and how different models are integrated and trained end-to-end.

3 KGMatcher APPROACH

Overview. We first give an overview of the KGMatcher approach depicted in Figure 2. Each design choice will be discussed in the following sections. The KGMatcher at its core learns a function $f_G(\cdot, \cdot)$ for a knowledge graph G that takes two entities as inputs and produces a score indicating the likelihood of the two entities sharing a non-obvious relationship. KGMatcher characterizes an entity by considering its attributes, its k-hop neighbors' attributes and its position in the knowledge graph. These characterizations are extracted by three types of layers in KGMatcher, namely, **Attribute**, **Neighborhood** and **Position** layer. These layers are connected and trained end-to-end with the goal of producing embeddings of entities which maximize the accuracy of the non-obvious relationship predictions. The function f_G learned by KGMatcher produces a symmetric measure for the input entities which means that $f_G(u, v) \equiv f_G(v, u)$ where $u, v \in V$. The symmetric property is guaranteed by the use of *Siamese network*[4]. The network consists of two identical graph embedding networks which share the same parameters (weight matrices). When measuring two entities, each network takes one of the two inputs and produces the respective embedding. The final score is then computed based on a distance measure between two embeddings.

3.1 Entity Embedding in Knowledge Graph

We introduce the key components in the graph embedding network for entity feature extraction.

Attribute Embedding Layer (E). The attribute information associated with each entity in the knowledge graph provide the central ingredients of the entity. It can be a mixture of structured, semi-structured and/or unstructured data.

By leveraging the existing deep learning based embedding methods, KGMatcher is able to convert arbitrary attributes into a vector representation. First, depending upon the specific type of attribute, one can choose a neural architecture to produce the embedding in an unsupervised or supervised manner. To process "text" type attributes for example, the vector representation can be generated from a pre-trained language model such as XLNet[7]. This unsupervised strategy ensures the generality of the embedding since the pre-trained model is usually obtained from a large general domain corpus. To be able to generate the embeddings that also maximize the accuracy of the entity matching. KGMatcher allows these neural networks to be easily connected to the rest of the KGMatcher's layers and be adjusted through backpropagation from the feedback on the relationship prediction. The connected embedding networks can be either initialized by the pre-trained parameters (weights) and fine tuned onward or randomly initialized and trained from scratch. Second, KGMatcher finally generates the entity attribute embedding by concatenating each of the embeddings of the whole attribute set and feed it to the next layers.

In our experiment, the attributes are of three types, namely numerical, geo-location (in text string) and categorical type. The numerical types are processed by a normalization layer. The geo-location strings are converted into latitude and longitude numbers. For categorical types, we convert them into one-hot-encodings and embed them using multilayer perceptron neural networks.

Neighbors Aggregation Layer (N). An entity in a knowledge graph is usually not isolated by nature. The connectivity among the entities often indicates additional information which may not be explicitly described by their attributes.

To exploit the natural connections among the entities to capture such "surroundings" signal, we adopt a graph neural architecture as part of the KGMatcher embedding network. The GNNs for this purpose broadly follow a recursive neighborhood aggregation scheme. Each node aggregates embedding vectors of its immediate neighbors to compute its new embedding vector. After k iterations of aggregation, a node is represented by its transformed embedding vector, which captures the surrounding information within the node's k-hop distance. Instead of focusing on embedding nodes from a single fixed graph which is assumed by many prior works, we adopt a spectrum of GNNs – *inductive* GNNs, where only the *aggregate* (Equation 5) and *combine* (Equation 6) for a node are learned. The complexity of such GNNs is usually independent to the size of the graph. They are capable of incorporating unseen nodes and easy to scale.

In particular, we implement *GraphSage*[1] layers as our Neighborhood layers. The *Aggregate* function is formulated as:

$$a_{v_i}^{(k)} = \max\left(\left\{\mathrm{ReLU}\left(W_a^N \cdot h_u^{(k-1)}\right) | u \in Neighbor(v_i)\right\}\right), \quad (3)$$

where W_a^N is a learnable matrix and *max* represents an element-wise max-pooling. The *Combine* function is formulated as a concatenation followed by a linear mapping:

$$h_{v_i}^{(k)} = W_c^N \cdot \left[h_{v_i}^{(k-1)}, a_{v_i}^{(k)}\right], \quad (4)$$

where W_c^N is a learnable matrix. The embedding vector of a node is initialized by its attribute embedding vector obtained from the Attribute layer(s).

Position Measuring Layer (P). The GNNs for neighborhood modeling are good at capturing *local* context. One of the limitations of such modeling approach is their lack of emphasis on the position/location of the embedded node within the broader context of the entire graph. To be able to model the high degree of separation between entities, KGMatcher is equipped with P-GNN layers [8] as Position layers to capture the *global* position information for each entity. The intuition of the P-GNN approach is that the absolute position of a node can be defined by its relative positions to a set of reference nodes in the graph.

Specifically, instead of aggregating information from the immediate neighbors, P-GNN aggregates information for an embedded node from a set of "anchor" nodes. Each P-GNN layer samples a set of anchor nodes as references from the graph and then computes the shortest distances from every embedded node to these sampled anchors to encode a distance metric. Each embedding dimension of a node corresponds to the combined of node embedding and the aggregated information from a specific anchor(s) weighted by the distance metric. The weight is inversely proportional to the distance. KGMatcher stacks multiple Position layers to achieve higher expression power. The aggregation for the lth dimension for v_i at time k is formulated as:

$$a_{v_i^l}^{(k)} = \mathrm{mean}\left(\left\{\mathrm{ReLU}\left(W_a^P \cdot (s(v_i,u) \times h_u^{(k-1)}\right) | u \in Anchor^l(v_i)\right\}\right), \quad (5)$$

and the combined embedding of v_i at time k is:

$$h_{v_i}^{(k)} = W_t^P \cdot \left[\cdots, \overbrace{W_c^P \cdot \left[h_{v_i}^{(k-1)}, a_{v_i^l}^{(k)}\right]}^{l\text{th dimension}}, \cdots\right], \quad (6)$$

where $s(v_i, u)$ computes the weight of an anchor node u to the embedded node v_i, $Anchor^l(v_i)$ returns a set of anchors dedicated for computing the embedding value on the lth dimension of the embedded nodes, *mean* represents an element wise mean-pooling and W_a^P, W_t^P and W_c^P are learnable matrices. The weight computed by $s(v_i, u)$ [8] is inversely proportional to the shortest distance between v_i and u in the graph.

Interaction between N and P Layer. To form a single representation that encodes both neighborhood and position information, a few *merging* design choices are made. One can combine the output embeddings immediately after each N and P layer through concatenation or other element-wise operations. The merged node embedding is then fed separately into the next N and P layer. In our work, to clearly separate the contribution of neighborhood and position signal, we allow the information propagation of the two kinds progress in parallel and only merge the two at the very end through concatenation (depicted in Figure 2).

3.2 KGMatcher Inference and Training

Suppose a pair of entities u and v are labeled with a label $y \in L_G$ using a function $d_y(u, v)$. The goal of KGMatcher is to predict such label y for unseen entity pairs. Specifically, KGMatcher solves the problem via learning an embedding function Φ parameterized by θ, where the objective is to maximize the likelihood of the conditional probability $p(y|\Phi_\theta(u), \Phi_\theta(v))$. Formally, we have the learning objective as:

$$\begin{aligned}\min_\theta \; & \mathbb{E}_{(u,v)\sim L_{G_{train}}} \\ & \mathcal{L}(d_z(\Phi_\theta(u), \Phi_\theta(v)) - d_y(u, v)),\end{aligned} \quad (7)$$

where $d_z(\cdot, \cdot)$ is a function that predicts the label based on two entity embeddings. Since the relationship we defined is undirected, d_z should then be a symmetric function. A Siamese neural network, depicted in Figure 2 uses the same weights while working in tandem on two different input vectors to compute comparable output vectors which aligns with the required symmetric property. Therefore, we train KGMatcher's entity embedding layers in the form of Siamese network. In our implementation, d_z computes dot product of the two input vectors.

4 EXPERIMENTAL EVALUATION

Setup & Evaluation Method. We evaluate our proposed system with a publicly available dataset by comparing its performance against other baselines. All methods are implemented in PyTorch and trained on a CentOS server with Intel(R) Xeon(R) Gold 6138 @ 2.00GHz CPUs and NVIDIA Tesla P100 GPUs.

Table 1: Test results of different models. ↑ indicates that the higher the score the better the performance. (·) after each score (average±standard deviation) indicates the ranking of the method (vertical comparison) w.r.t the specific evaluation metric. ⋆ indicates that the baseline is implemented by ourselves. The cut-off thresholds of these reported methods are 0.210, 0.445, 0.620, 0.580, 0.440, 0.375 and 0.530 which produce the maximum F_1 score for each respective method.

Method	AUC ↑	F_1 ↑	Precision ↑	Recall ↑
DeepMatcher [5] ⋆	0.6658±0.0351 (6)	0.6755±0.0108 (6)	0.5255±0.0131 (6)	0.9455±0.0000 (2)
GIN [6]	0.7367±0.0329 (3)	0.7400±0.0311 (3)	**0.7036**±0.0415 (1)	0.7818±0.0000 (6)
GCN [3]	0.7080±0.0333 (5)	0.7210±0.0192 (5)	0.6837±0.0346 (2)	0.7636 ±0.0000 (7)
GraphSAGE [1]	0.7633±0.0320 (2)	0.7442±0.0200 (2)	0.6598±0.0311 (4)	0.8545±0.0000 (5)
PGNN [8]	0.7090±0.0337 (4)	0.7255±0.0199 (4)	0.6108±0.0261 (5)	0.8942±0.0195 (4)
PGNN (w/o attributes) [8]	0.5988±0.0424 (7)	0.6668±0.0023 (7)	0.5003±0.0019 (7)	**0.9994**±0.0040 (1)
KGMatcher ⋆	**0.8079**±0.0343 (1)	**0.7660**±0.0239 (1)	0.6676±0.0331 (3)	0.8998±0.0222 (3)

Since our task is binary classification – predict the existence of the relationship between two input entities, we measure the performance of all methods using *receiver operating characteristic* (ROC), *area under the curve* (AUC), *precision*, *recall* and F_1 typical metrics for the evaluation of a binary classification task. We report the average measurements and the standard deviations of all methods on the test set of 100 repetitions.

Dataset. We use the UDBMS *Person*[2] dataset for our evaluation. The original dataset contains 502,529 unique person entities. Each entity has up to 48 attributes. We use 8 attributes – *relation*, *relative*, *spouse*, *child*, *parent*, *partner*, *predecessor*, *successor*, *opponent* and *rival* to build the edges and select another 25 popular attributes in terms of their presenting rate as entity attributes. We use *subject* information to annotate the non-obvious relationships, i.e. two entities sharing the same *subject* have a non-obvious relationship. We further trim down the dataset by only selecting the entities with reasonable amount of attributes and reasonable level of connectivity. Finally, we have 1,294 person entities, 3,480 edges and 316 relationships. The relationship pairs are split into train, validation and test set as the positive samples. The negative samples are uniformly sampled by fixing one of the entities in the positive sample pairs.

Baseline. We compare KGMatcher against 5 other baselines:

• **DeepMatcher** [5] is a supervised deep learning solution which aims to identity pair of data instances that are referring to the same entity based on their attributes.

• **Graph Convolutional Network** (GCN) [3] is a semi-supervised spectral-based graph neural network. It models the graph topology as well as the node attributes.

• **GraphSAGE** [1] is a spatial-based graph neural network which models the graph topology through neighbors aggregation. The aggregated information is based on the node attributes.

• **Graph Isomorphism Network** (GIN) [6] is a graph neural network that generalizes *Weisfeiler-Lehman* test for maximum discriminative power. It also models the node attributes.

• **Position-aware Graph Neural Network** (PGNN) [8] is a generalized spatial-based graph neural network which aims to identify the node position in the context of the the entire graph through sampled anchors.

Effectiveness. Given two entities on the knowledge graph, all methods produce a score [0-1] which can be interpreted as the level of confidence of predicting a non-obvious relationship. As a data analyst, one has to specify a cut-off threshold for the model to give a firm (binary) answer. To simulate this scenario, we measure the effectiveness using *precision*, *recall* and F_1 where the cut-off thresholds must be provided. Since maximizing either *precision* or *recall* can easily done by varying the threshold, we report these measurements by selecting the thresholds that maximized each method's F_1 score. As shown on Table 1, KGMatcher outperformed all baselines in F_1. Although, GIN and PGNN (w/o attributes) perform well either on *precision* or *recall*, their measurements on the other side (*recall* or *precision*) are poor. The unbalanced performance may not be acceptable to many applications. In particular, the model that only considers attributes (DeepMatcher) or global position information (PGNN w/o attributes) performs worse than the ones that model both attributes and some topology of the graph.

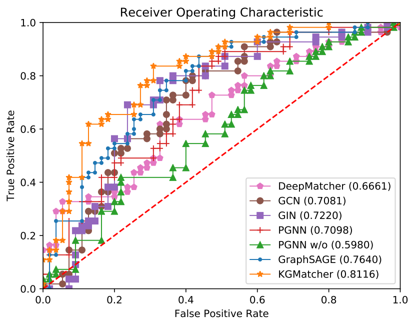


Figure 3: ROC plot of all baselines. (·) after each method in the legend box indicates its corresponding AUC value.

To further evaluate the overall performance of all the methods across different thresholds, we plot the ROC in Figure. 3 and report the AUC value in Table. 1. Our proposed method KGMatcher is significantly better than all the other baselines.

In summary, our experimental evaluation demonstrates KGMatcher's 6% to 35% improvement in AUC and 3% to 15% improvement in F_1 over the other baselines.

REFERENCES

[1] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the NeurIPS 2017, 4-9 December 2017, Long Beach, CA, USA.* 1024–1034.

[2] Jeff Jonas. 2006. Identity resolution: 23 years of practical experience and observations at scale. In *Proceedings of the SIGMOD 2006, June 26-29, 2006, Chicago, Illinois, USA.* ACM, 718–718.

[3] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the ICLR 2017, April 24-26, 2017, Toulon, France.*

[4] Evgeny Krivosheev, Mattia Atzeni, Katsiaryna Mirylenka, Paolo Scotton, and Fabio Casati. 2020. Siamese Graph Neural Networks for Data Integration. arXiv:cs.DB/2001.06543

[5] 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 *Proceedings of the SIGMOD 2018, June 10-15, 2018, Houston, TX, USA.* 19–34.

[6] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *Proceedings of the ICLR 2019, May 6-9, 2019, New Orleans, LA, USA.*

[7] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *Proceedings of the NeurIPS 2019, December 8-14, 2019, Vancouver, Canada.* 5754–5764.

[8] Jiaxuan You, Rex Ying, and Jure Leskovec. 2019. Position-aware Graph Neural Networks. In *Proceedings of the ICML 2019, June 9-15, 2019, Long Beach, California, USA.* 7134–7143.

[9] Wen Zhang, Kai Shu, Huan Liu, and Yalin Wang. 2019. Graph Neural Networks for User Identity Linkage. *CoRR* abs/1903.02174 (2019).

[2]http://udbms.cs.helsinki.fi/?datasets/person_dataset

Outlier detection in multivariate functional data based on a geometric aggregation

Clément Lejeune[*]
IRIT UMR 5505 CNRS,
Airbus Commercial Aircraft
Toulouse, France
clement.lejeune@irit.fr

Josiane Mothe
IRIT UMR 5505 CNRS, INSPE,
Université de Toulouse
orcid:0000-0001-9273-2193, France
josiane.mothe@irit.fr

Olivier Teste
IRIT UMR 5505 CNRS
Toulouse, France
olivier.teste@irit.fr

ABSTRACT

The increasing ubiquity of multivariate functional data (MFD) requires methods that can properly detect outliers within such data, where a sample corresponds to $p > 1$ parameters observed with respect to (w.r.t) a continuous variable (*e.g.* time). We improve the outlier detection in MFD by adopting a geometric view on the data space while combining the new data representation with state-of-the-art outlier detection algorithms. The geometric representation of MFD as paths in the p-dimensional Euclidean space enables to implicitly take into account the correlation w.r.t the continuous variable between the parameters. We experimentally show that our method is robust to various rates of outliers in the training set when fitting the outlier detection model and can detect outliers which are not detected by standard algorithms.

1 INTRODUCTION

1.1 Functional data context and taxonomy

In many fields (*e.g.* engineering, biology or medicine), detecting atypical behaviors of complex systems enables to better anticipate and understand both undesired and rare situations (*e.g.* engine failure, heart disease). Most of the time, detecting atypical behaviors requires the analysis of p system parameters ($p \geq 1$) measured by high sampling-rate sensors. The raw sensor measurements result in noisy data dependent on a continuous variable (*e.g.* time, wavelength) being discretized by the sampling process of the p sensors. Such data are referred as *univariate* or *multivariate functional* data depending on whether one ($p = 1$) or several parameters ($p > 1$) are analyzed, respectively.

Thus the observation of the parameters along the continuous variable is seen as the realization of an underlying (unknown) function that values in $\mathbb{R}^p$. We emphasize that in the functional data framework, a data set sample is represented as a function rather than a high-dimensional vector of different dimension containing the raw measurements. Dimension refers to the number of measurements which can be different from a sample to another). We refer to [13] for a comprehensive introduction to functional data analysis.

Here we adopt the following notations : the dependent continuous variable is denoted by $t \in \mathcal{T} \subset \mathbb{R}$ where $\mathcal{T}$ is a closed interval of $\mathbb{R}$, the data samples are sub-scripted by $i \in \{1...n\}$, univariate functional data (UFD) samples are denoted by lower case letter $x_i(t) \in \mathbb{R}$ and multivariate functional data (MFD) samples are denoted by capital letter $X_i(t) = (x_{i1}(t), ..., x_{ik}(t), .., x_{ip}(t)) \in \mathbb{R}^p$. Thus a MFD is made up of p UFD which are potentially correlated.

[*]The author can also be contacted at: clement.lejeune@airbus.com

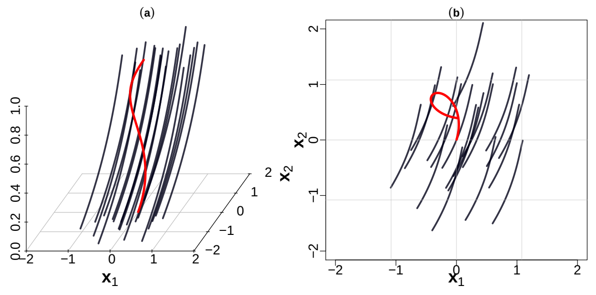


Figure 1: Example of 21 MFD ($p = 2$) with one shape persistent outlier in red. (a) (t, x_{i1}, x_{i2}) representation. (b) (x_{i1}, x_{i2}) representation *i.e.* projection along t-axis.

Detecting atypical behaviors is referred as *outlier* or *anomaly detection*. An outlier is defined as a sample which is rare and very different from the rest of the data set based on some measure [1]. A taxonomy of functional outliers into two classes has been proposed by Hubert *et al.* [8]. First, an *isolated outlier* is defined as a sample which exhibits an extreme behavior for very few points t. For instance a narrow vertical peak in the curve depicted by a parameter x_{ik} w.r.t t is named a *magnitude isolated* outlyingness and a high horizontal translation in the curve is referred as *shift isolated* outlyingness. Second, a *persistent outlier* is a sample which never exhibits extreme behavior but deviates from inliers for many points t, an example of shape persistent outliers is given in Fig.1. Persistent outliers can be divided into other sub-classes, see [8] for detailed examples. Note that an outlier can be of mixed type, *i.e.* a sample entailing several outlier classes. For an instance, one parameter has a shape persistent outlyingness and another one has an isolated outlyingness.

In this paper we focus on a geometric representation for outlier detection in MFD and highlight the situation of outliers of mixed type. The MFD case is more challenging than the UFD one since the potential correlation between the p parameters (*i.e.* how x_{ik} and $x_{ik'}$ are correlated w.r.t t) has to be taken into account additionally to the individual variations of the single parameters w.r.t to t [8]. Indeed contrary to outliers in UFD, where the outlyingness of a sample only consists of an atypical variation w.r.t t of a single parameter, in MFD the outlyingness of sample might be hidden in an atypical variation of the *relationship* between some parameters [8, 11] as well as an atypical variation of one of the p parameters. Note that the representation of MFD we propose can also be used for other tasks than outlier detection (*e.g.* classification) as well as other geometric representations of 2D and 3D shapes which can be applied for the $p = 2$ and $p = 3$ (respectively) MFD cases [16].

1.2 Related work

The outlier detection in MFD is recent and has been addressed by statistical depth functions [18] originally proposed to provide an outward-center ranking score, also named a *depth score* (*e.g.* in the interval $[0, 1]$), of multivariate data which are basically sample points in $\mathbb{R}^d$. In this general multivariate data context, where each sample is regarded as a point in a d-dimensional point cloud, the first ranked samples are the most central ones within such point cloud and are seen as most representative, whereas the last ranked samples are the least central ones and thus they are likely to be outliers. Such ranking is ensured through the monotonicity property of the depth function (see [18] for theoretical understanding of a depth function). Hence, the depth score can be viewed as an outlyingness score which reflects the degree of outlyingness at the sample level.

Some statistical depth functions have been extended first to handle UFD [3] and then MFD [2, 8]. The UFD extension consists in computing a depth function on $x_i(t), \forall i$ at each t and then to compute the integral over $t \in \mathcal{T}$ of the resulting depth scores [3, 6] which in turn provides an average sample depth score for all i. Note that this extension is an aggregation of the depth function applied in a univariate manner since, for a given t, $\{x_i(t)\}_{i\leq n}$ is a point cloud in $\mathbb{R}$. Since $\{X_i(t)\}_{i\leq n}$ forms a point cloud in $\mathbb{R}^p$, the MFD extension relies on the application of a depth function in a multivariate manner and integrates the depth scores as in the UFD case [2]. Such an extension suffers from important issues :

(1) First, it is not sensitive enough to persistent outliers because their point-wise depth scores (*i.e.* for each t) do not differ from those of inliers. One can augment the MFD samples by adding some derivatives functions of the parameters as supplementary (unobserved) parameters but it increases both computations and the complexity of the data analysis.
(2) Second, even if the point-wise depth scores of an isolated outlier are different from those of an inlier, its sample depth score will be mixed with inliers because the integration of the point-wise depth scores acts as an average.
(3) Furthermore, since the capacity of the depth function to capture different types of outlier is fundamental, outliers caused by abnormal correlation between the parameters (*i.e.* outliers of mixed type) are hard to detect. Such an abnormal correlation can result in outliers of mixed type.

To address the first issue (1), and especially to detect shape persistent outliers, several depth functions have been proposed. Khunt and Rehage (*FUNTA*) [9] proposed a depth function based on the intersection angles between a curve sample depicted by x_{ik} and $\{x_{jk}\}_{j\leq n, j\neq i}$ and then average these angles over both their number and the parameters. Such method is not able to detect outliers caused by an abnormal correlation between the parameters and also isolated outliers because their depth function is only focused on shape persistent outliers.

To address the second issue (2) the integral can be replaced by the infimum as the aggregation of the point-wise depth scores, which avoids the masking of outliers having few different point-wise depth scores.

To address the third issue (3), Dai and Genton [4] proposed the *directional outlyingness* (*Dir.out*), a point-wise depth function based on the direction of $X_i(t)$ in $\mathbb{R}^p$ toward the projection-depth [17] of $\{X_i(t)\}_{i\leq n}$. To compute the sample depth score, the point-wise depth scores are aggregated through an integral over $\mathcal{T}$ which is further decomposed into an average component and a variance-like component. Such sample depth score decomposition enables to detect multiple outliers and also to identify their class by analyzing how the two depth components are distributed according the other sample depth scores (*e.g.* samples with high variance-like component value are likely persistent shape outliers and samples with high average component value are likely isolated outliers). However, to detect persistent shape outliers, the direction of $X_i(t)$ is not a sufficient feature and further geometrical representation has to be considered.

1.3 Contribution

In this paper, we propose a different framework than the statistical depth to remedy these issues by treating MFD as trajectories in $\mathbb{R}^p$ from which we extract geometrical features such as the curvature. Such geometrical features are computed by interpretable (from a geometric standpoint) aggregation functions, named *mapping functions* in the sequel, which combine some derived functions (*e.g.* derivatives, integral) from the MFD. We then apply a state-of-the-art algorithm on the geometrical MFD representation to achieve the outlier detection. Considering MFD in a geometric manner enables to implicitly capture the correlation between the p parameters w.r.t t and thus to detect different classes of outliers as well as mixed types. Moreover, such combination results in a more robust outlier detection method *e.g.* when there are more than 5% of outliers in the training set. Thus, we both take benefit from outlier detection algorithm for multivariate data as well as the geometry of the curve (*i.e.* the geometry of X_i in $\mathbb{R}^p$ and the geometry of each parameter x_{ik} w.r.t t).

2 FUNCTIONAL DATA REPRESENTATION

The first step in functional data analysis is to approximate the unknown function, $X_i : \mathcal{T} \to \mathbb{R}^p$, underlying the noisy measurement samples $X_i(t_1), ..., X_i(t_{m_i})$ where m_i is the number of measurements for each parameter of the sample i, by an approximation function $\tilde{X}_i$ defined as X_i. Note that no assumption is made on the distribution of the measurement points $\{t_1...t_{m_i}\} = t_{i\bullet}$, thus the functional data representation can deal with sparse measurements as well as uniform ones.

The functional approximation step aims at removing the noise and thus enables to achieve accurate evaluations of some derived functions that we need for the mapping function computation. This section introduces how $\tilde{X}_i = (\tilde{x}_{i1}, ..., \tilde{x}_{ik}, ..., \tilde{x}_{ip})$ is specified as well as it is inferred from the data.

2.1 Functions as a basis expansion

First, we specify the functional form of the approximation function as a finite linear combination of basis functions, where each basis function depends on $t \in \mathcal{T}$. Suppose we want to approximate x_{ik}. Intuitively, it aims to represent $\tilde{x}_{ik}$ with a small number of "specific functions", each one being able to capture some local features of X_i in hopes to recover it with a small approximation error. Hence, the following form is given for $\tilde{x}_{ik}$ [13],

$$\forall t \in \mathcal{T}, \tilde{x}_{ik}(t) = \sum_{l=1}^{L_{ik}} \alpha_{ikl}\phi_l(t) = \boldsymbol{\alpha}_{ik}^\top \boldsymbol{\phi}(t) \tag{1}$$

where $\boldsymbol{\phi}(t) = \{\phi_l(t)\}_{1\leq l\leq L_{ik}}$ is a vector of orthonormal basis functions at t for some $L_{ik} \in \mathbb{N}^*$ (referred as the basis size) with fewer basis functions than sampled observation points ($L_{ik} \ll m_i$), and $\boldsymbol{\alpha}_{ik}^\top = \{\alpha_{ikl}\}_{1\leq l\leq L_{ik}}$ is the coefficient vector which element α_{ikl} is the importance of the l-th basis function. The choice of the basis of functions is data dependent.

Here we consider that x_{ik} are smooth and so we choose the B-spline basis of functions which are basically piece-wise polynomial functions. If the data were periodic data, one could choose the Fourier basis. We refer to [13] for a discussion on the choice of basis functions. Note that from the functional approximation Eq.1, one can easily compute some derivatives or integral based functional data since by linearity,

$$D^q \tilde{x}_{ik} = \sum_{l=1}^{L_{ik}} \alpha_{ikl} D^q \phi_l(t) \tag{2}$$

where $D^q = \frac{\mathrm{d}^q}{\mathrm{d}t^q}$ is the $q-$th derivative operator, provided that the basis functions ϕ_l are differentiable at the q-th order.

2.2 Inference

Assuming the data were sampled with a white noise ϵ_{ij}, *i.e.* $x_{ik}(t_{ij}) = \tilde{x}_{ik}(t_{ij}) + \epsilon_{ij}$ where ϵ_{ij} is independent from $\tilde{x}_{ik}(t_{ij})$, we can compute the coefficient vector $\boldsymbol{\alpha}_{ik}$ by minimizing the following penalized least-squares criteria:

$$J_{\lambda_k}(\boldsymbol{\alpha}_{ik}) = \|x_{ik}(t_{i\bullet}) - \boldsymbol{\Phi}_{ik}\boldsymbol{\alpha}_{ik}\|^2 + \lambda_k \boldsymbol{\alpha}_{ik}^\top \mathbf{R}_{ik} \boldsymbol{\alpha}_{ik} \tag{3}$$

where $\|\cdot\|$ stands for the l_2-norm, $\boldsymbol{\Phi}_{ik} = \{\phi_l(t_{ij})\}_{1\le j\le m_i, 1\le l\le L_{ik}}$ is the $m_i \times L_{ik}$ matrix containing all the L_{ik} basis functions evaluated at the measurement points $t_{i\bullet}$ and $\mathbf{R}_{ik} = \{\int_{\mathcal{T}} D^q \boldsymbol{\phi}_j(t) D^q \boldsymbol{\phi}_m(t)\mathrm{d}t)\}_{1\le j,m\le L_{ik}}$ is a $L_{ik} \times L_{ik}$ positive semi-definite matrix containing the inner products of the q-th derivative of the basis functions which enforces the approximation function to have a small q-th derivative *i.e.* to vary smoothly; $\lambda_k > 0$ is a hyper-parameter controlling the weight of the penalty and can be set to 0 for no penalization. In practice it is common to choose $q = 1$ or $q = 2$ (i.e to penalize the velocity or the acceleration) and to compute λ_k by cross-validation.

Equaling the gradient of $\boldsymbol{J}_{\lambda_k}$ to $\mathbf{0}$ w.r.t $\boldsymbol{\alpha}_{ik}$ leads to the minimizer in Eq.4 [13] which is a special case of the ridge regression solution:

$$\boldsymbol{\alpha}^*_{ik,\lambda} = (\boldsymbol{\Phi}_{ik}^\top \boldsymbol{\Phi}_{ik} + \lambda_k \mathbf{R}_{ik})^{-1} \boldsymbol{\Phi}_{ik}^\top x_{ik}(t_{i\bullet}) \tag{4}$$

The estimated coefficient vector $\boldsymbol{\alpha}^*_{ik,\lambda}$ can then be plugged in Eq.1 to evaluate $\tilde{x}_{ik}$ over an arbitrary discretization of $\mathcal{T}$.

3 MAPPING FUNCTION

We propose to regard MFD as paths in $\mathbb{R}^p$ to highlight some underlying shape outlyingness features corresponding to a change in the relationship between the parameters. We feature this change with a mapping function that we define as a geometric aggregation of the p parameters. We refer to [15] for an introduction to shape analysis from functional data.

In this section, we present the *curvature* as an example of mapping function. The curvature is a measure of how much bended a curve is, more formally how the curve locally deviates from the tangent line, see Fig.2. It is defined as:

$$\kappa(t) = \frac{\|D^1(\frac{D^1X(t)}{\|D^1X(t)\|})\|}{\|D^1X(t)\|} \tag{5}$$

where $\|\cdot\|$ denotes the Euclidean norm in $\mathbb{R}^p$. One can interpret κ in Eq.(5) as follows: $\frac{D^1X(t)}{\|D^1X(t)\|}$ gives the direction vector (i.e the normalized tangent vector), therefore $D^1 \frac{D^1X(t)}{\|D^1X(t)\|}$ gives the change direction vector and the denominator $\|D^1X(t)\|$ aims to relate the change of direction w.r.t the tangent vector, *i.e.* how the direction vector varies w.r.t a tangent line. Consequently, the

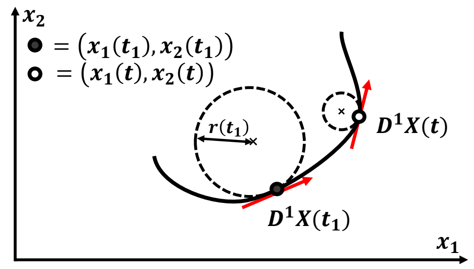


Figure 2: Curvature mapping κ. The curvature measures how large the radius of the tangent circle is. Here, in the neighbourhood of the curve at t_1 (dark-grey dot), the *tangent vector* $D^1X(t_1)$ keeps the same direction, hence the tangent circle has a large radius ($r(t_1) = \frac{1}{\kappa(t_1)}$) resulting in a small curvature. In the neighbourhood of the curve at t (white dot), the tangent vector $D^1X(t)$ quickly changes direction, hence the tangent circle has a lower radius *i.e.* a higher curvature than at t_1.

curvature mapping can highlight functional outliers which curve exhibits a different bended shape than the other samples.

Thus if a curve abnormally changes direction (*i.e.* it deviates from a tangent line) w.r.t most of the data set, then the curvature mapping can highlight outliers. As a result, if the curve X_i depicts a line (*i.e.* the parameters are linearly correlated w.r.t t), then the curvature is constant w.r.t t since the directions do not vary in $\mathbb{R}^p$. Clearly, this is a geometric characterization of MFD.

From the reconstructed samples $\{\tilde{X}_i\}_{i\le n}$, transformed to UFD by the mapping function, we detect the outliers with state-of-the-art algorithms initially proposed to deal with multivariate data (not functional). Here we use Isolation-Forest (*iFor*) [10] and One-class SVM (*OCSVM*) [14] which are both unsupervised.

4 NUMERICAL EXPERIMENTS

We conducted an experimental study on real data. We compare our approach with state-of-the-art depth-based methods, *FUNTA* and *Dir.out* [4, 9] (Sec.1.2) which take the MFD as input.

4.1 Experimental procedure

We experiment our method on a well-known real data set of electrocardiogram (ECG) time series [7] also used in outlier detection in [4]. Such data set correspond to time series of electrical activity and can reveal abnormal heartbeat. The time series are UFD (with number of measurements $m_i = 85, \forall i$) and in order to show the applicability of our approach in the MFD case, we augment the original UFD data to MFD ($p = 2$, bivariate) by adding the square of the initial time series. We did not add some derivatives-based functions as supplementary parameters since it is already considered by our mapping function (see Eq.5).

We evaluate our approach through multiple random splittings. We randomly split the data into a training and a test set. We generate the training set by setting the ratio of outliers (referred as the contamination level c) to 5, 10, 15, 20 and 25%. For each value of c, we repeat the random splitting 50 times, we fit *iFor* and *OCSVM* on the training set and compute the average and standard deviation Area Under (AUC) the Receive Operating Curve (ROC) on the corresponding test set. We present the results in Fig.3 and discuss them in Sec.4.3.

For each sample and each variable $x_{ik} \forall i, k$ we use a B-spline basis of functions (piece-wise polynomial functions, [13]) to achieve the functional approximations and we select the basis sizes L_{ik}

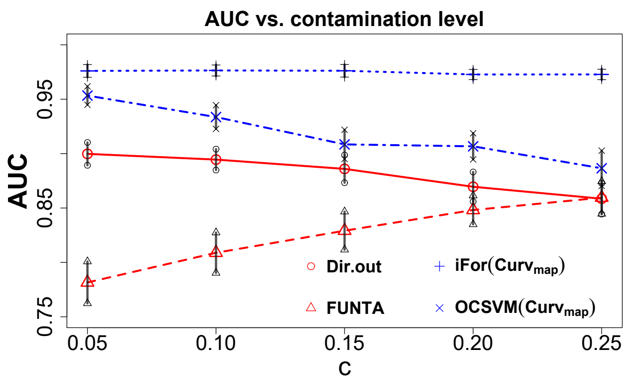


Figure 3: AUC (vertical axis) and standard deviation (vertical segments' length equals one standard deviation) from ECG data set - average results over 50 repetitions considering five contamination levels c (horizontal axis).

trough a leave-one-out cross validation procedure. We evaluate each $\tilde{X}_i$ on the same regular grid of $\mathcal{T}$ with length $m_i = 85$. We compute the mapping function by combining the first and second derivatives, according to Eq.2 and Eq.5, and we apply $iFor$ and $OCSVM$ on the resulting UFD.

4.2 Outlier detection step

We use $iFor$ and $OCSVM$ as outlier detection algorithms on the UFD that our mapping function κ returns (Eq.5). $iFor$ and $OCSVM$ are unsupervised and, like the depth-based methods, output a normalized outlyingness score for each sample. In practice, in outlier detection one has not necessarily access to labeled samples, *i.e.* information depicting whether a sample is an outlier or not, but if he has, the labels can be combined with their corresponding outlyingness scores to learn an outlyingness threshold that can best discriminate outliers from inliers. Such a threshold can be learned from the ROC as well as an imbalanced classification algorithm [5, 12] in a one dimensional manner from the scores. Here, we do not learn any threshold and only consider the label information for empirical demonstration purpose, *i.e.* by computing the AUC on the test set.

4.3 Discussion of the results

From the results in Fig.3, we see that we outperform the two depth-based methods for all the contamination levels in average and perform equally in terms of standard deviation. Since $FUNTA$ is only able to detect persistent shape outliers and $Dir.out$ is expected to detect isolated as well as persistent outliers[1], we can deduce that the abnormal class (*i.e.*outliers) in the ECG data set not only contains persistent shape outliers but also isolated ones or outliers of mixed type which are well discriminated by the curvature mapping function. Thus the curvature mapping enables to detect mixed type outliers.

Moreover, we note that as c increases both $iFor(Curv_{map})$ and $OCSVM(Curv_{map})$ still outperform the baselines. Hence, we show that our combination of outlier detection algorithm with MFD mapped to a geometrical representation is more robust to the presence of outliers in the training set than the baselines. We note that OCSVM degrades as c increases. It is due to the ν hyper-parameter (we tune it on the training set with a 5-fold cross validation) corresponding to an estimate of contamination level in the training set. We observed that such hyper-parameter is hard to tune as c increases and thus could decrease the performance w.r.t c.

[1]Justification can be found in the experiments in [4] which were conducted on several synthetic data sets where each one contains a unique type of outlier.

5 CONCLUSION AND FUTURE WORK

We propose an approach to detect outliers in MFD. It consists in computing a geometrical representation of MFD followed by an outlier detection algorithm. We compare our approach with recent depth-based methods which handle MFD as input.

Through one example of mapping function, we show that the geometrical representation of MFD is well suited to detect outliers of mixed type. However, it is hard to interpret what such mixed type outliers are made up: given a detected outlier, ideally one would like to access to the amount of the different outlyingness classes *e.g.* the amount of shape persistence and shift isolated outlyingness. As future work, a mean to achieve such an interpretability is first to detect some specific outliers with depth functions, second to train outlier detection algorithms (combined with a mapping function) on training sets containing each one a unique class of outlier previously detected and then to average all the models trained to form an ensemble one. As a result, one could know which model(s) in the ensemble most contribute to the outlyingness and deduce the outlyingness composition.

ACKNOWLEDGMENTS

This work is supported by ANRT within the CIFRE framework (grant N°2017-1391).

REFERENCES

[1] Charu C Aggarwal and Philip S Yu. 2001. Outlier detection for high dimensional data. In *ACM Sigmod Record*, Vol. 30. ACM, 37–46.

[2] Gerda Claeskens, M Hubert, Leen Slaets, and K Vakili. 2014. Multivariate Functional Halfspace Depth. *J. Amer. Statist. Assoc.* 109, 505 (2014), 411–423.

[3] Antonio Cuevas and Manuel Febrero. 2007. Robust estimation and classification for functional data via projection-based depth notions. *Computational Statistics* 22, 3 (2007), 481–496.

[4] Wenlin Dai and Marc G. Genton. 2019. Directional outlyingness for multivariate functional data. *Computational Statistics and Data Analysis* 131 (2019).

[5] Shounak Datta and Swagatam Das. 2015. Near-Bayesian support vector machines for imbalanced data classification with equal or unequal misclassification costs. *Neural Networks* 70 (2015), 39–52.

[6] Ricardo Fraiman and Graciela Muniz. 2001. Trimmed means for functional data. *Test* 10, 2 (2001), 419–440.

[7] Ary L Goldberger, Luis AN Amaral, Leon Glass, Jeffrey M Hausdorff, Plamen Ch Ivanov, Roger G Mark, Joseph E Mietus, George B Moody, Chung-Kang Peng, and H Eugene Stanley. 2000. PhysioBank, PhysioToolkit, and PhysioNet: components of a new research resource for complex physiologic signals. *Circulation* 101, 23 (2000), e215–e220.

[8] Mia Hubert, Peter J. Rousseeuw, and Pieter Segaert. 2015. Multivariate functional outlier detection. *Statistical Methods and Applications* 24, 2 (2015).

[9] Sonja Kuhnt and André Rehage. 2016. An angle-based multivariate functional pseudo-depth for shape outlier detection. *Journal of Multivariate Analysis.* 146 (2016), 325–340.

[10] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation Forest. In *ICDM.*

[11] Sara López-pintado, Yin Sun, Juan K Lin, and Marc G. Genton. 2014. Simplicial band depth for multivariate functional data. *Advances in Data Analysis and Classification* 8, 3 (2014), 321–338.

[12] Art B Owen. 2007. Infinitely imbalanced logistic regression. *Journal of Machine Learning Research.* 8 (2007), 761–773.

[13] James O. Ramsay and B.W. Silverman. 2006. *Functional Data Analysis.* Springer.

[14] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. 2001. Estimating the support of a high-dimensional distribution. *Neural computation* 13, 7 (2001), 1443–1471.

[15] Anuj Srivastava and Eric P Klassen. 2016. *Functional and Shape Data Analysis.* Springer.

[16] Weiyi Xie, Oksana Chkrebtii, and Sebastian Kurtek. 2019. Visualization and Outlier Detection for Multivariate Elastic Curve Data. *IEEE Transactions on Visualization and Computer Graphics* (2019).

[17] Yijun Zuo et al. 2003. Projection-based depth functions and associated medians. *The Annals of Statistics* 31, 5 (2003), 1460–1490.

[18] Yijun Zuo and Robert Serfling. 2000. General Notions of Statistical Depth Function. *The Annals of Statistics.* 28, 2 (2000), 461–482.

REMI: Mining Intuitive Referring Expressions on Knowledge Bases

Luis Galárraga
Inria
luis.galarraga@inria.fr

Julien Delaunay
University of Rennes I
juliendelaunay35000@gmail.com

Jean-Louis Dessalles
Télécom ParisTech
dessalles@telecom-paristech.fr

ABSTRACT

A *referring expression* (RE) is a description that identifies a set of instances unambiguously. Mining REs from data finds applications in natural language generation, algorithmic journalism, and data maintenance. Since there may exist multiple REs for a given set of entities, it is common to focus on the most concise and informative (i.e., intuitive) ones. We present REMI, a method to mine intuitive REs on large knowledge bases. Our experimental evaluation shows that REMI finds REs deemed intuitive by users. Moreover we show that REMI is several orders of magnitude faster than an approach based on inductive logic programming.

1 INTRODUCTION

A *referring expression* (RE) is a description that identifies a set of entities unambiguously. For instance, the expression "x is the capital of France" is an RE for Paris, because no other city holds this title. The automatic construction of REs is a central task in natural language generation (NLG). The goal of NLG is to describe concepts in an accurate and compact manner from structured data such as a knowledge base (KB). REs also find applications in automatic data summarization, algorithmic journalism, virtual smart assistants, and KB maintenance, e.g., in query generation. Quality criteria for REs is context-dependent. For instance, NLG and data summarization aim at *intuitive*, i.e., *short* and *informative* descriptions. In this vibe, it may be more intuitive to describe Paris as "the city of the Eiffel Tower" than as "the resting place of Victor Hugo". Indeed, the world-wide prominence of the Eiffel Tower makes the first RE more informative to an average user.

Some approaches can mine intuitive REs from semantic data [1, 4–6]. Conceived at the dawn of the Semantic Web, these methods are not suitable for current KBs for three main reasons. Firstly, they cannot handle current KBs because they were designed to mine REs on scenes[1] for the sake of NLG. Scenes have much fewer predicates and instances than today's KBs. Secondly, most existing approaches are limited to conjunctive expressions on the attributes of the entities, e.g., $is(x, City) \wedge country(x, France)$. However, our experience with today's KBs suggests that this language bias does not encompass all possible intuitive expressions. For instance, to describe Johann J. Müller, we could resort to the fact that he was the supervisor of the supervisor of Albert Einstein, i.e., $supervisor(x, y) \wedge supervisor(y, Einstein)$, which goes beyond the traditional language bias due to the existentially quantified variable y. Thirdly, state-of-the-art RE miners define intuitiveness for REs in terms of number of atoms. In that spirit, the single-atom REs $capitalOf(x, France)$ and $restingPlaceOf(x, V. Hugo)$ are equally concise and desirable as descriptions for Paris, even though the latter may not be informative to users outside France.

[1]The exhaustive description of a place and its objects

The approach in [6] overcomes this limitation to some extent, by allowing users to provide a ranking of preference for the attributes used in the description. Nevertheless, providing such a ranking can be tedious for KBs with thousands of predicates.

We tackle the aforementioned limitations with a solution to mine intuitive REs on large KBs. How to use such REs is beyond the scope of this work, however we provide hints about potential use cases. In summary, our contributions are:

- A scheme based on information theory to quantify the intuitiveness of entity descriptions extracted from a KB.
- REMI, an algorithm to mine intuitive REs on large KBs. REMI extends the state-of-the-art language bias for REs and allows for expressions such as $mayor(x, y) \wedge party(y, Socialist)$. This design choice increases the chances of finding intuitive REs for a set of target entities.
- A user study to assess the intuitiveness of REMI's descriptions.

2 PRELIMINARIES

2.1 RDF Knowledge Bases

This work focuses on mining REs on RDF knowledge bases (KBs). A KB $\mathcal{K}$ is a set of assertions in the form of facts $p(s, o)$ with predicate $p \in \mathcal{P}$, subject $s \in \mathcal{I} \cup \mathcal{B}$, and object $o \in \mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$. In this formulation, $\mathcal{I}$ is a set of entities such as London, $\mathcal{P}$ is a set of predicates, e.g., *cityIn*, $\mathcal{L}$ is a set of literal values such as strings or numbers, and $\mathcal{B}$ is a set of blank nodes, i.e., anonymous entities. An example of an RDF triple is $cityIn(London, UK)$. KBs often include assertions that state the class of an entity, e.g., $is(UK, Country)$.

2.2 Referring Expressions

2.2.1 Atoms. An *atom* $p(X, Y)$ is an expression such that p is a predicate and X, Y are either variables or constants. We say an atom has matches in a KB $\mathcal{K}$ if there exists a function $\sigma \subset \mathcal{V} \times (\mathcal{I} \cup \mathcal{L} \cup \mathcal{B})$ from the variables $\mathcal{V}$ of the atom to constants in the KB such that $\mu_\sigma(p(X, Y)) \in \mathcal{K}$. The operator μ_σ returns a new atom such that the constants in the input atom are untouched, and variables are replaced by their corresponding mappings according to σ. We call $\mu_\sigma(p(X, Y))$ a bound atom and σ a matching assignment. We extend the notion of matching assignment to conjunctions of atoms, i.e., σ is a matching assignment for $\bigwedge_{1 \leq i \leq n} p_i(X_i, Y_i)$ iff $\mu_\sigma(p_i(X_i, Y_i)) \in \mathcal{K}$ for $1 \leq i \leq n$.

2.2.2 Expressions & Language Bias. Atoms are traditionally the building blocks of referring expressions. We say that two atoms are connected if they share at least one variable argument. Most approaches for RE mining define REs as conjunctions of connected atoms with bound objects. We call this language bias, *the state-of-the-art language bias*. We extend this language by allowing atoms with additional existentially quantified variables. For this purpose, we propose subgraph expressions as the new building blocks for REs. A *subgraph expression* $\rho = p_1(x, Y_1) \wedge \bigwedge_{1 < i \leq n} p_i(X_i, Y_i)$, rooted at variable x, is a conjunction of connected atoms such that for $i > 1$, atoms are

1 atom	$p_0(x, I_0)$
Path	$p_0(x, y) \wedge p_1(y, I_1)$
Path + star	$p_0(x, y) \wedge p_1(y, I_1) \wedge p_2(y, I_2)$
2 closed atoms	$p_0(x, y) \wedge p_1(x, y)$
3 closed atoms	$p_0(x, y) \wedge p_1(x, y) \wedge p_2(x, y)$

Table 1: REMI's subgraph expressions.

transitively connected to $p_1(x, Y_1)$ via at least another variable besides x. Examples are: (i) $cityIn(x, France)$, and (ii) $cityIn(x, y) \wedge officialLang(y, z) \wedge langFamily(z, Romance)$. An *expression* $e = \bigwedge_{1 \leq j \leq m} \rho_j$ is a conjunction of subgraph expressions rooted at the same variable x such that they have only x—the root variable—as common variable. Finally, we say e is a *referring expression* (RE) for a set of target entities $T \subseteq \mathcal{I}$ in a KB $\mathcal{K}$ iff:

(1) $\forall t \in T : \exists \sigma : (x \mapsto t) \in \sigma$, i.e., for every target entity t, there exists a matching assignment σ in $\mathcal{K}$ that binds the root variable x to t.
(2) $\nexists \sigma', t' : (x \mapsto t') \in \sigma' \wedge t' \notin T$, in other words, no matching assignment binds the root variable to entities outside the set T of target entities.

For example, consider a complete and accurate KB $\mathcal{K}$ as well as the following conjunction of two subgraph expressions:

$$e = in(x, S.\ America) \wedge officialLang(x, y) \wedge langFamily(y, Germanic)$$

We say that e is an RE for $T = \{Guyana, Suriname\}$ in $\mathcal{K}$ because matching assignments can only bind x to these two countries.

While we do not limit the number of subgraph expressions in REs, we do not allow more than one variable and three atoms in individual subgraph expressions, leading to the expressions in Table 1. This design decision aims at keeping both the search space and the complexity of the REs under control. Indeed, expressions with multiple non-root variables make comprehension and translation to natural language more effortful.

3 REMI

Given an RDF KB $\mathcal{K}$ and a set of target entities T, REMI returns an intuitive RE—a conjunction of subgraph expressions—that describes unambiguously the input entities T in $\mathcal{K}$. Intuitive REs are concise and resort to concepts that users are likely to understand. We first show how to quantify intuitiveness in Section 3.1. We then elaborate on REMI's algorithm in Section 3.2.

3.1 Quantifying intuitiveness

There may be multiple ways to describe a set of entities uniquely. For example, $capitalOf(x, France)$ and $birthPlaceOf(x, Voltaire)$ are both REs for Paris. Our goal is therefore to quantify the *intuitiveness* of such expressions without human intervention. We say that an RE e is more intuitive than an RE e', if $C(e) < C(e')$, where C denotes the Kolmogorov complexity. The Kolmogorov complexity $C(e)$ of a string e (e.g., an expression) is a measure of the absolute amount of information conveyed by e and is defined as the length in bits of e's shortest effective binary description [8]. If e_b denotes such binary description and M is the program that can *decode* e_b into e, $C(e) = l(e_b) + l(M)$ where $l(\cdot)$ denotes length in bits. Due to C's intractability, applications can only *approximate* it via suboptimal encodings and programs $(\hat{e_b}, \hat{M})$, hence $C(e) \approx \hat{C}(e) = l(\hat{e_b}) + l(\hat{M})$ with $C(e) \leq \hat{C}(e)$.

Our proposed encoding builds upon the observation that intuitive expressions resort to prominent concepts. For example, it is natural and informative to describe Paris as the capital of France, because the concept of capital is well understood and France is a very salient entity. In contrast, it would be more complex to describe Paris in terms of less prominent concepts, let us say, its twin cities. In this spirit, we devise a code for concepts as follows: The code for a predicate p (entity I) is the binary representation of its position k in a *ranking by prominence*. This way, prominent concepts can be rewarded with shorter codes. We can now define the estimated Kolmogorov complexity $\hat{C}$ of a single-atom subgraph expression $p(x, I)$ as:

$$\hat{C}(p(x, I)) = l(k(p)) + l(k(I \mid p))$$

In the formula, $l(\cdot) = log_2(\cdot) + 1$, $k(p)$ is p's position in the ranking of predicates of the KB, and $k(I \mid p)$ is I's conditional rank given p, i.e., I's rank among all objects of p. The latter term follows from the chain rule of the Kolmogorov complexity. For instance, if p is the predicate *city mayor*, the chain rule models the fact that once the concept of mayor has been conveyed, the context becomes narrower and the user needs to rank fewer concepts, in this example, only city mayors. The chain rule also applies to subgraph expressions with multiple atoms. For instance, the complexity of $\rho = mayor(x, y) \wedge party(y, Socialist)$ is:

$$\hat{C}(\rho) = l(k(mayor)) + l(k(party(y, z) \mid mayor(x, y))) + \\ l(k(Socialist \mid mayor(x, y) \wedge party(y, z)))$$

The second term in the sum amounts to the code length of the rank of predicate *party* among those predicates that allow for subject-to-object joins with *mayor* in the KB. Likewise, the complexity of the Socialist party in the third term depends on the ranking of parties with mayors among their members, i.e., the bindings for z in $mayor(x, y) \wedge party(y, z)$. If a city can be unambiguously described as $mayor(x, I)$ for a non-prominent mayor I, we may achieve a shorter code length if we replace I by a variable y, an additional predicate, and a well-known party.

In line with other works that quantify prominence for concepts in KBs [7], we rank concepts by frequency (*fr*), and Wikipedia's page rank (*pr*). We denote the resulting complexity measures using these prominence metrics by $\hat{C}_{fr}$ and $\hat{C}_{pr}$ respectively.

Finally, we can estimate the Kolgomorov complexity of an RE $e = \bigwedge_{1 \leq i \leq m} \rho_i$ as the sum of the complexities of its individual subgraph expressions, i.e., $\hat{C}(e) = \sum_{1 \leq i \leq m} \hat{C}(\rho_i)$.

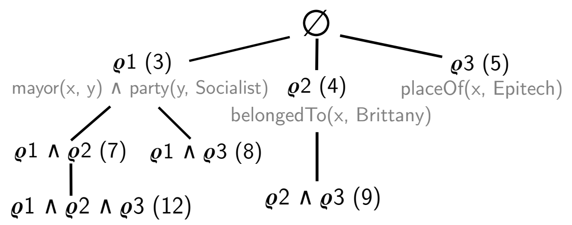


Figure 1: Search space example.

3.2 Algorithm

REMI implements a depth-first search (DFS) on conjunctions of the subgraph expressions common to **all** the target entities. Let us assume the KB knows only three common subgraph expressions ρ_1, ρ_2, and ρ_3 for the entities *Rennes* and *Nantes*, such that $\hat{C}(\rho_1) \leq \hat{C}(\rho_2) \leq \hat{C}(\rho_3)$ as illustrated in Figure 1. Each node in the tree is an expression, i.e., a conjunction of subgraph expressions and its complexity $\hat{C}$ is in parentheses. When visiting a node, DFS must test whether the corresponding expression is an RE, i.e., whether the expression describes exclusively the target entities. If the test fails, the strategy should move to the node's first child. If the test succeeds, DFS must verify whether the expression is less complex than the least complex RE seen so far. If it is the case, this RE should be remembered, and DFS can prune the search space by backtracking. To see why, imagine that $\rho_1 \wedge \rho_2$ in Figure 1 is an RE. In this case, all REs prefixed with this expression (the node's descendants) will also be REs. However, all these REs are more complex. This means that we can stop descending in the tree and prune the node $\rho_1 \wedge \rho_2 \wedge \rho_3$ in Figure 1. We call this step a *pruning by depth*. We can do further pruning if we leverage the order of the entities. In our example, if $\rho_1 \wedge \rho_2$ is an RE, any expression prefixed with $\rho_1 \wedge \rho_i$ for $i > 2$ must be more complex and can be therefore skipped. We call this a *side pruning*. All these ideas are formalized by Algorithm 1 that takes as input a KB $\mathcal{K}$ as well as the entities to describe, and returns an RE of minimal complexity according to $\hat{C}$. For each of the target entities, line 1 calculates (in a BFS fashion) its matching subgraph expressions, and takes those common to all the target entities. The expressions are then sorted by increasing complexity in a priority queue (line 2), which is processed as follows: At each iteration, the least complex subgraph expression ρ is dequeued (line 5) and sent to the subroutine *DFS-REMI* (line 6) with the rest of the queue. This subroutine explores the subtree rooted at ρ and returns the most intuitive RE e' prefixed with ρ. If e' is less complex than the best solution found so far (line 7), we remember it[2]. If *DFS-REMI* returns an empty expression, we can conclude that there is no RE for the target entities T (line 8). To see why, recall that DFS will, in the worst case, combine ρ with all remaining expressions ρ' that are more complex. If none of such combinations is an RE, there is no solution for T in $\mathcal{K}$.

```
Algorithm 1: REMI
  Input: a KB: $\mathcal{K}$, the target entities: $T$
  Output: an RE of minimal complexity: $e$
1 $G := \bigcap_{t \in T}$ subgraphs-expressions($t$)
2 create priority queue from $G$ in ascending order by $\hat{C}$
3 $e := \top$
4 while $|G| > 0$ do
5     $\rho := G.dequeue()$
6     $e' :=$DFS-REMI($\rho, G, T, \mathcal{K}$)
7     if $\hat{C}(e') < \hat{C}(e)$ then $e := e'$
8     if $e = \top$ then return $\top$
9 return $e$
```

We implemented Algorithm 1 in Java 8, including a parallel version called P-REMI (detailed in our technical report [3]).

4 EXPERIMENTAL EVALUATION

We evaluated REMI along two dimensions: output quality, and runtime. The evaluation was conducted on two popular KBs, namely DBpedia and Wikidata[3]. Our technical report [3] offers details about the experimental datasets, as well as a more extensive qualitative evaluation of REMI.

[2]We define $\hat{C}(\top) = \infty$

[3]http://dbpedia.org, http://wikidata.org

metric	#participants	p@1	p@2	p@3
$\hat{C}_{fr}$	44	0.38±0.42	0.66±0.18	0.88±0.09
$\hat{C}_{pr}$	48	0.43±0.42	0.53±0.25	0.72±0.16

Table 2: Average precision@k and standard deviation for $\hat{C}$'s ranking of subgraph expressions in DBpedia

4.1 Qualitative Evaluation

We carried out three user studies in order to evaluate REMI's descriptions on instances of the classes person, settlement, album, film, and organization. The cohort consisted mainly of computer science students, researchers, and university staff. It also included some of their friends and family members.

4.1.1 Evaluation of $\hat{C}$. Subgraph expressions are the building blocks of REs, thus intuitive REs should make use of concise and informative pieces. We measure to which extent the function $\hat{C}$ captures intuitiveness by asking the participants to rank a set of 5 subgraph expressions by simplicity and comparing this ranking with the ranking provided by $\hat{C}$. The expressions come from the common subgraph expressions ranked by Alg. 1 (line 2) using $\hat{C}$, and include the top 3 as well as a baseline defined by (i) the worst ranked, and (ii) a random subgraph expression. We manually translated the subgraph expressions to natural language statements in the shortest possible way using the textual descriptions (predicate *rdfs:label*) of the concepts when available. We show the results of our findings on 24 sets of entities in Table 2 for our two variants of $\hat{C}$. We observe that precision@1 is low. This happens because people usually deem the predicate type the simplest whereas REMI often ranks it second or third (16 times for $\hat{C}_{fr}$). This shows the need of special treatment for the *type* predicate as suggested by [6]. Nevertheless, the high values for the other metrics show a positive correlation between the preferences of the users and the function $\hat{C}$. In 88% of the cases, the three simplest subgraph expressions according to $\hat{C}$ are among the three simplest ones according to users.

4.1.2 Evaluation of REMI's output. A second study requested users to rank by simplicity the answer of REMI and a baseline consisting of 2 to 4 additional REs (solutions encountered during search space traversal). The entities were hand-picked to guarantee the existence of at least two REs sufficiently different from each other. Based on our previous findings, we used *fr* as notion of prominence. We report an average MAP (mean average precision) of 0.64±0.17 for this task on 20 sets of entities with 51 answers each, if we assume REMI's solution as the only relevant answer. We recall that a MAP of 1 denotes full agreement between REMI and the users, while a MAP of 0.5 means that REMI's solution is always among the user's top 2 answers.

4.1.3 User's perceived quality. In order to measure the perceived quality of the reported REs, we requested 86 participants to grade the *interestingness* of 35 Wikidata REs in a scale from 1 to 5, where 5 means the user deems the description interesting based on her personal judgment. Our results exhibit an average score of 2.65±0.71, with 11 descriptions scoring at least 3. During the exchanges with the participants, some of them made explicit their preference for short but at the same time informative REs. The latter dimension is related to the notions of pertinence of concepts and narrative interest. For instance, when asked to select between the REs $country(x, N.\ Zealand) \wedge actor(x, C.Lee)$ and $country(x, N.\ Zealand) \wedge actor(x, y) \wedge religion(x, Buddhism)$ for two movies, 95% of the users preferred the first one. Both REs

Language	DBpedia					Wikidata				
	#solutions	amie+	remi	p-remi	speed-up	#sol.	amie+	remi	p-remi	speed-up
Standard	63	97.4k^{8}	10.3k^{1}	576	13.5kx, 2.44x	44	115.5k^{15}	1.06k	76.2	142kx, 4.7x
REMI's	65	508.2k^{68}	66.5k^{8}	28.9k	5218x, 21.4x	44	608.3k^{60}	21.7k	33.8k	6476x, 7.1x

Table 3: REMI's runtime (in seconds) on DBpedia and Wikidata. Speed-ups are provided for P-REMI w.r.t. AMIE and REMI.

had more or less the same length when translated to natural language, but the second one conveys less information and resorts to a domain-unrelated entity (i.e., Buddhism). These observations suggest that prominence captures the notion of simplicity, but it does not always accurately model the dimension of informativeness. While these examples might discourage the use of existential variables in descriptions, we remark that users also liked REs such as $in(x, Brittany) \wedge mayor(x, y) \wedge party(y, Socialist)$ (DBpedia) for Rennes and Nantes, or $actor(x, y) \wedge leader(y, Pisa)$ for the Italian movie "Altri templi" (Wikidata), as they deemed the first one quite pertinent, and the second one narratively interesting. Other interesting REs from DBpedia include "she died of aplastic anemia" for Marie Curie, and "they were both places of the Inca Civil War" for Ecuador and Peru. Finally, we highlight the impact of noise and incompleteness in the quality of the solutions. For instance, REMI cannot describe France as the country with capital Paris, because Paris is also the capital of the former Kingdom of France in DBpedia.

4.2 Runtime Evaluation

4.2.1 Opponent. RE mining can be conceptually formulated as a rule mining task. Hence, we compare the runtime of REMI and a state-of-the-art rule miner designed for large KBs, namely AMIE+ [2]. Given thresholds on support and confidence, AMIE+ mines Horn rules of the form $p(X, Y) \Leftarrow \bigwedge_{1 \leq i \leq n} p_i(X_i, Y_i)$, such as $speaks(x, English) \Leftarrow livesIn(x, UK)$, on RDF KBs . The *support* of a rule is the number of facts correctly predicted by the rule. If we normalize this measure by the total number of predictions made by the rule, we obtain its *confidence*. RE mining for a target entity set T is equivalent to rule mining with AMIE+, if we instruct the system to find rules of the form $\psi(x, True) \Leftarrow \bigwedge_{1 \leq i \leq n} p_i(X_i, Y_i)$, where ψ is a surrogate predicate with facts $\psi(t, True)$ for all $t \in T$. In this case, the right-hand side of the rule becomes our RE. We set thresholds of $|T|$ and 1.0 for support and confidence respectively. This is because an RE should predict the exact set of target entities, neither subsets nor supersets. AMIE+ does not define a complexity score for rules and outputs all REs for the target entities, thus we use $\hat{C}_{fr}$ to rank AMIE's output and return the least complex RE.

4.2.2 Results. We compared the runtimes of REMI and AMIE+ on a server with 48 cores (Intel Xeon E2650 v4), 192GB of RAM[4], and 1.2T of disk space (10K SAS). We tested the systems on 100 sets of DBpedia and Wikidata entities taken from the same classes used in the qualitative evaluation. Small sets of entities are challenging in our setting, so we picked random sets of 1, 2, and 3 entities of the same class in proportions of 50%, 30%, and 20%. We mined REs for those sets of entities according to (i) the standard language bias of conjunctions of bounded atoms, and (ii) REMI's language of conjunctions of subgraph expressions. We show the total runtime among all sets for AMIE+ and REMI in Table 3. The values in red account for the number of timeouts (for a limit of 2 hours), thus cells with red superscripts define runtime lower bounds. We observe that AMIE+ already timed out 23 times with the state-of-the-art language. In particular, AMIE+ is optimized for rules without constant arguments in atoms, such as $livesIn(x, y) \Leftarrow citizenOf(x, y)$, thus its performance is heavily affected when bound variables are allowed in atoms. In contrast REMI and P-REMI are on average 3 and 4 orders of magnitude (up to 142k times) faster than AMIE+ in this language. In the worst case REMI was confronted with a space of 62 subgraph expressions for the state-of-the-art language bias. For REMI's language bias, however, this number increased to 25.2k, which is challenging even for REMI (8 timeouts in total). Despite this boost in complexity, multithreading makes it manageable: P-REMI can be *at least* 4.7x on average faster than REMI for the extended language bias and *at least* 21x faster for the state-of-the-art language, allowing for real-time RE mining. Finally, we observe that the extended language bias slightly increases the chances of finding a solution (column #solutions in Table 3) in DBpedia. This phenomenon is more common among sets with more than one entity.

[4]AMIE assumes the entire KB fits to main memory

5 CONCLUSION AND FUTURE WORK

In this work we have presented REMI, a method to mine intuitive referring expressions on large RDF KBs. REMI builds upon the observation that users prefer prominent entities in descriptions and leverages this fact to quantify the intuitiveness of descriptions in bits. Our results show that (1) real-time RE generation is possible in large KBs and (2) a KB-based frequency ranking can provide intuitive descriptions despite the noise in KBs. This latter factor impedes the fully automatic generation of intuitive REs for NLG purposes, however our descriptions are applicable to scenarios such as computer-aided journalism and query generation. As future work we aim to investigate if external sources—such as search engines or external localized corpora—can yield even more intuitive REs that model users' background more accurately. We also envision to relax the unambiguity constraint to mine REs with exceptions. We provide the source code of REMI as well as the experimental data at https://gitlab.inria.fr/lgalarra/remi.

REFERENCES

[1] Robert Dale. 1992. *Generating Referring Expressions: Constructing Descriptions in a Domain of Objects and Processes.* A Bradford Book, MIT.
[2] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. 2015. Fast Rule Mining in Ontological Knowledge Bases with AMIE+. *The VLDB Journal* 24, 6 (2015), 707–730.
[3] Luis Galárraga, Julien Delaunay, and Jean-Louis Dessalles. 2019. REMI: Mining Intuitive Referring Expressions on Knowledge Bases. arXiv:cs.AI/1911.01157
[4] Helmut Horacek. 2003. A Best-first Search Algorithm for Generating Referring Expressions. In *Conference on European Chapter of the Association for Computational Linguistics (EACL).*
[5] Emiel Krahmer, Sebastiaan van Erk, and André Verleg. 2003. Graph-based Generation of Referring Expressions. *Computational Linguistics* 29, 1 (2003), 53–72.
[6] Ehud Reiter and Robert Dale. 1992. A Fast Algorithm for the Generation of Referring Expressions. In *Conference on Computational Linguistics (COLING).*
[7] Andreas Thalhammer, Nelia Lasierra, and Achim Rettinger. 2016. LinkSUM: Using Link Analysis to Summarize Entity Data. In *International Conference on Web Engineering (ICWE).*
[8] Arnold Zellner, Hugo A. Keuzenkamp, Michael McAleer, et al. 2001. *Simplicity, Inference and Modelling: Keeping it Sophisticatedly Simple.* Cambridge Univ. Press.

The Case for Hybrid Succinct Data Structures

Christoph Anneser Andreas Kipf Harald Lang Thomas Neumann Alfons Kemper
Technical University of Munich
{anneser,kipf,harald.lang,neumann,kemper}@in.tum.de

ABSTRACT

One of the biggest challenges in data management is to retain the high performance of in-memory processing with the ever increasing data volumes. Recent years have shown that the amount of collected data is increasing at a faster pace than DRAM capacities. Many state-of-the-art index data structures are optimized for performance rather than for low space consumption and quickly exceed the limited main-memory capacities when indexing larger data sets. *Succinct data structures* on the other hand allow for space efficient indexing, but compromise performance while still being orders of magnitude faster than disk-based data structures.

In this work, we propose a novel framework that combines state-of-the-art indexes with succinct data structures to form new *hybrid succinct data structures*. These hybrids enable fine-grained trade-offs between space and performance. Frequently accessed parts of an index, e.g. the upper levels of a tree, are thereby maintained in performance-optimized structures whereas less frequently accessed parts have a space-optimized representation. Our evaluation shows that our approach can significantly reduce the amount of used space by up to 50% (resp. 90% for our compressed version) while retaining 93% (resp. 87%) of the performance.

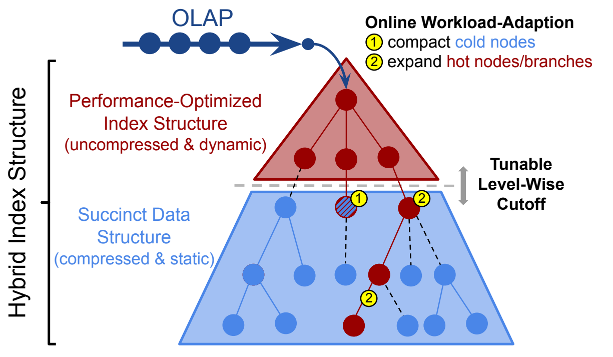

OLAP
Online Workload-Adaption
1 compact cold nodes
2 expand hot nodes/branches
Hybrid Index Structure
Performance-Optimized Index Structure (uncompressed & dynamic)
Succinct Data Structure (compressed & static)
Tunable Level-Wise Cutoff
1
2
2

Figure 1: We propose a novel framework that combines state-of-the-art indexes with succinct data structures. It allows indexing even larger data sets entirely in main memory by taking advantage of space-efficient succinct indexes. Furthermore, our hybrid index allows online adaptation to the actual OLAP workload by storing hot nodes in performance-optimized structures.

1 INTRODUCTION

Back in 2006, Jim Gray stated that memory is the new disk and disk is the new tape [7]. This also applies to modern database systems that store the entire data in random access memory (RAM) to allow near real-time analyses for trading companies and finance services. They need to efficiently process large datasets to react within a few milliseconds to new developments or updates.

To achieve the required performance for near real-time data processing, index structures such as B-trees, hash tables, tries, amongst others are used to efficiently find specific elements. In modern database systems, these index structures are also stored in main memory and are most often highly optimized in terms of performance and underlying hardware rather than space efficiency. However, while the main memory capacities tend to double every three years and its costs decrease by a factor of 10 every five years, the data collected by sensors, smartphones, social media platforms, and IoT-devices *increases at an even higher rate* resulting in data overflows [12]. This development requires in-memory data structures to optimize both performance *and* space.

In this paper, we focus on hierarchical indexes comprising nodes and relationships between them, such as trees, tries, and graphs, and distinguish two different types of data structures:

(1) **Pointer Based Data Structures** (PBDS): These state-of-the-art data structures model relationships between elements *explicitly* using `machine addresses`. Traversing to another node directly translates to a pointer resolution.
(2) **Succinct Data Structures** (SDS): These data structures encode relationships in the data *implicitly* using bitmaps and still allow accessing nodes in constant time [13]. While SDS tend to be smaller than PBDS, they are often slower since traversing requires more complex operations [6].

So far, only a few SDS-based approaches, such as succinct range filters [13] and tree-encoded bitmaps [9], have been successfully applied to database systems, as modern PBDS tend to offer much better performance. However, the explicit maintenance of relationships in PBDS also consumes more space. Using 64-bit addresses may introduce significant overheads, since pointers theoretically allow for differentiating between 2^{64} (more than 18 trillion) items, which is not required for most applications. When indexing larger datasets, SDS become more interesting for the case when PBDS do not fit into main memory anymore [6] and would require staging parts of the data structure to disk.

We propose a new, lightweight framework that takes advantage of both types of data structures and allows combining any hierachical PBDS and SDS to a new hybrid index that consumes less space than the PBDS and offers higher performance than the SDS. It also allows online workload adaptation for use cases, in which some elements tend to be more important than others and get accessed more frequently.

The rest of this paper is organized as follows: In Section 2, we present the foundations of succinct index structures as well as the Fast Succinct Trie [13] which is used in our evaluation. We also give a short overview of point-polygon joins, since our evaluation is based on this use case. In Section 3, we present a detailed overview of our approach. We evaluate our framework for the real-world use case of geospatial point-polygon joins in Section 4 and conclude with our next steps and future work in Section 5.

2 BACKGROUND

In this section, we present the foundations of succinct data structures and the Fast Succinct Trie, which is a succinct trie data structure that almost achieves the performance of uncompressed PBDSs [13]. We also discuss a state-of-the-art index for point-polygon joins where we applied and evaluated our framework.

Succinct Data Structures. A data structure is called *succinct* if its space is close to the information-theoretic optimum while most basic operations are still executable in constant time. In the literature, *close* is defined in different ways – we refer to a data structure as succinct if it uses $O(\text{opt})$ bits, with opt being the minimal number of required bits to represent the data and its relationships.

As stated in Section 1, instead of explicitly modeling relationships between elements using machine addresses, this information is encoded *implicitly* in bitmaps. Consider the trie data structure in Figure 2 where each level encodes *two bits* and keys *are not a prefix* of other keys. For a succinct encoding, we store two bitmaps *labels* and *hasChild* that encode for each *node label* (branch) 00_2, 01_2, 10_2, and 11_2 whether it exists and if there is a following child node. E.g., when labels$[4n+2]$ is set, it indicates that the n-th node contains the label 10_2. Then, all encoded nodes are concatenated in *breadth-first ordering* resulting in the above mentioned bitmaps (cf. labels and hasChild in Figure 2). We define $\texttt{rank}(x)$ to count the number of set bits up and including position x, and $\texttt{select}(x)$ to return the index of the x-th set bit:

$$\texttt{rank}(x) = \sum_{i=0}^{i \leq x} \text{hasChild}[i] \tag{1}$$

$$\texttt{select}(x) = i, \text{ with } \texttt{rank}(i) = x \tag{2}$$

Based on rank and select, more complex operations for a node n starting at position p can be defined. E.g., we can calculate the position for the child node at label/branch x (3), and we can find the position of n's parent node (4) (assuming *fixed-sized* nodes comprising w bits) [13]:

$$\texttt{child}(x,p) = \texttt{rank}(x+p) \times w, \text{with hasChild}[x+p] = 1 \tag{3}$$

$$\texttt{parent}(p) = \texttt{select}(\lfloor p/w \rfloor) \tag{4}$$

As one can see in Figure 2, *six bytes* are sufficient to store the trie structure and its relationships (excluding the values). However, while succinct data structures optimize space, they tend to introduce higher latencies than PBDS since resolving a neighboring element involves multiple rank and select statements.

In 2018, Zhang et al. proposed a new data structure called Fast Succinct Trie (FST) [13]. It almost achieves the performance of state-of-the-art trie data structures such as the Adaptive Radix Tree (ART) [10] but needs less space and allows for efficient value compression. It internally uses a hybrid encoding scheme where upper levels are represented similarly to the trie in Figure 2, and lower levels store the data in a more compressed way by only representing branches that actually exist. In Section 3, we apply our framework to this data structure.

Efficient Point-Polygon Joins. We applied and tested our framework for the use case of point-polygon joins according to the approach introduced by Kipf et al. in 2020 [8]. *Dynamic points* get joined with a *static set of polygons* using prefix lookups on *one-dimensional* 64-bit keys. First, a given space (e.g. the minimum bounding rectangle of the polygons, cf. *green cell* in Figure 3) is recursively decomposed into smaller sub-cells. Then, the cells are

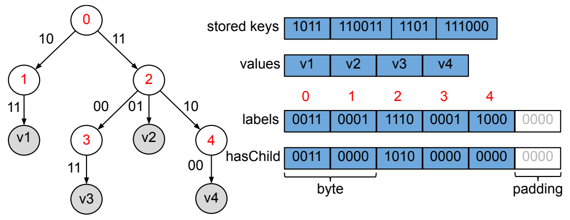


Figure 2: Succinct encoding of a trie with maximum fan-out of four. The relationships between nodes are implicitly encoded in the *labels* and *hasChild* bitmaps in breadth-first order.

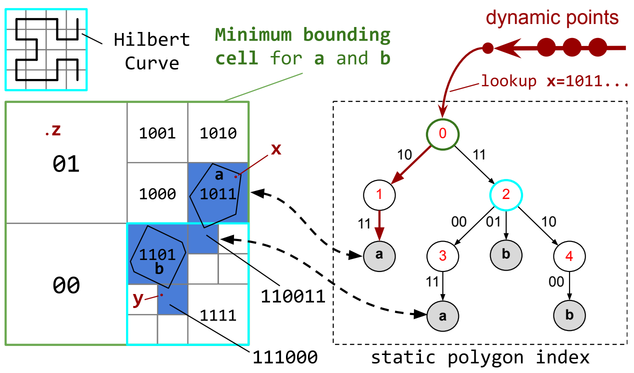


Figure 3: Left: space decomposition into quadtree cells that cover polygons a and b. Right: trie data structure indexing the blue cells level-wise and the exemplary lookup of point x.

enumerated (discretized) by a space-filling curve (e.g. the Hilbert curve) and can be identified by *one-dimensional* keys. Each decomposition divides a given cell into four smaller sub-cells for which reason we store two bits per cell level that uniquely identify one of the four sub-cells. Up to 31 levels can be addressed by a single 64-bit integer where the *least significant set bit* determines the encoded level. Applied to the Earth's surface, we can address every single square centimeter within 64 bits [2].

As a next step, we compute for each polygon p a so-called *covering*, which comprises a set of cells that cover p (cf. blue cells in Figure 3). A specialized pointer-based radix trie with fan-out four, called Adaptive Cell Trie (ACT), indexes the cells of *all combined coverings* (refer to [8] for more details). When joining an incoming point, we first transform the point to the smallest cell level (in this case 31) and then use ACT to find matching polygons using *level-wise prefix checks.* If a cell c_1=0110 is prefix of another cell c_2= $\underline{0110}1101$, then c_2 is fully contained in c_1. E.g., the binary key for point z starts with 01 and we can already detect at ACT's root node *rn* that no polygon encloses the point since the label 01 is not present in *rn*. For point x (1011...), we find the enclosing polygon **a** at level two (cf. exemplary lookup in Figure 3). Point y is an example for a *false positive* match since it is not enclosed by any polygon, but querying ACT indicates that y could be contained by polygon **b**. By this means, this point-polygon join guarantees a precision which corresponds to the diagonal of the largest cell that is not completely enclosed by a polygon.

3 HYBRID SUCCINCT DATA STRUCTURES

We propose a new, lightweight framework that allows combining any hierarchical PBDS and SDS *level-wise* to significantly reduce the memory footprint, while retaining the performance at a large extent. *Online workload adaptation* supports *branch-wise* PBDS refinements for use cases with skewed workloads. Our framework comprises four steps to build a new hybrid index structure for a given dataset:

(1) Build the static, read-only SDS.
(2) Build the PBDS for the upper n levels.
(3) Connect the PBDS nodes at level n to the corresponding child nodes in the SDS (e.g. by using pointer tagging).
(4) Extend the PBDS interface by two operations to allow *branch-wise* online workload adaptation:
    (a) `expand(node)`: Frequently accessed nodes are encoded in the faster PBDS when a specified *threshold* is exceeded.
    (b) `compact(node)`: Colder nodes (under the *threshold*) are compacted, removed from PBDS and indexed in SDS only.

In our approach, we exploit the fact that real-world workloads tend to be *skewed* and therefore, we periodically evaluate the actual queries *at runtime* to determine frequently accessed nodes. For a node n whose accesses exceed a predefined threshold t, we call `expand(n)` to add n to the faster PBDS. In the case that the accesses to n precede t, we simply remove n from the PBDS.

Since *all queries* start at the upper levels of a hierarchical data structure, we encode the frequently queried upper levels l using the performance-optimized PBDS and connect it *level-wise* to the SDS. Furthermore, the upper levels represent only a small fraction of the overall data structure and encoding them as PBDS may not have a noticeable impact on the total size. In contrast to branch-wise refinements, level-wise cutoffs do not introduce additional branch misses since we do *not have to differentiate* between PBDS and SDS references *before* reaching level l.

We deliberately accept that common parts are stored redundantly as it allows the PBDS to become a *dynamic meta index structure* for the static SDS where *lightweight refinements* do affect only the dynamic PBDS. Despite the introduced redundancy, our framework focuses on *minimizing the memory overhead* while our secondary goal is keeping the *read overhead as small as possible*. In accordance with the RUM conjecture by Thanassoulise et al. [5], "there is always a price to pay for every optimization", as our approach does *not* handle updates efficiently so far.

To the best of our knowledge, this is the first approach that applies *tuneable level-wise* cutoffs combined with *branch-wise* workload adaptations to hybrid succinct index structures. In contrast to the proposed framework by Zhang et al. in 2016 [11], we *completely avoid searching redundant parts* of the key space by *directly* pointing from PBDS into SDS using pointer tagging. While Zhang's approach is optimized for OLTP workloads where recently inserted tuples are assumed to be accessed more frequently and therefore are kept in the dynamic structure, we do *not rely* on this assumption but rather adapt to the *actual workload at runtime* using fine-grained branch-wise refinements. Additionally, we connect PBDS and SDS *level-wise*, whereas Zhang et al. completely separate both indexes *tuple-wise*.

Prefix Lookups Based on ACT and FST. In the following, we explain our framework using the Adaptive Cell Trie [8] as PBDS and the Fast Succinct Trie [13] as SDS and apply them to the use case of point-polygon joins as introduced in Section 2.

Given: We start with a set of polygons and its coverings that were calculated according to the aforementioned approach in Section 2. The coverings contain *unique* cell keys where each cell key is represented by a `uint64_t` and for each *key*, the referenced polygon(s) are stored in an `uint64_t`-payload (*value*).

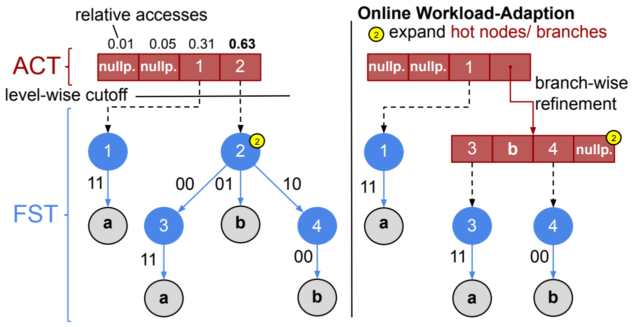


Figure 4: Initial hybrid trie with a threshold of 0.4 and ACT-encoded root node (left). Illustration of hybrid trie after expanding FST node 2 and indexing it in ACT (right).

Step 1: First, we adapted the FST to store *two bits per level* (instead of a *byte*) so that one trie level stores exactly one cell level (cf. Figure 3). The trie indexes the 64-bit cell keys *up to* the cell level while the remaining, unused levels are ignored.

In addition to our framework, we compressed the values using *run-length encoding*. In this use case, neighboring cells are likely to cover the same polygon(s) and thereby share the same polygon reference (value). These values occur directly one after the other in the values vector and offer a high compression potential.

Step 2: As PBDS, we use the Adaptive Cell Trie [8] where one trie level also stores exactly one cell level. Each ACT node comprises an array of four 64-bit *tagged pointers* resulting in an overall size of 256 bits per node. The left trie depicted in Figure 4 stores the cell keys {1011, 110011, 1101, 111000} and shows an ACT encoded root node while the remaining levels are encoded in the FST.

Step 3: We connect ACT and FST by inlining the required information in the ACT pointers. Since pointers do not use the entire 64 bits for memory addressing, we can use the two least significant bits to differentiate whether (i) the pointer stores a memory address for an ACT child node, (ii) a referenced polygon id (value) or (iii) an offset into the FST. In the case a label does not exist in a node, we just store a `nullpointer`.

Step 4: We extended the ACT interface by two functions `expand` and `compact` and stored four access counters for each node (cf. ACT-encoded root node in the left part of Figure 4). During runtime, we periodically determine the number of accesses and after a pre-defined number of lookups, we add those nodes whose access counters exceed a given threshold t to the *candidate set*. Then, we start expanding the candidate nodes and add them to ACT until the candidate list is empty or a given *memory bound* would be exceeded. E.g., node 2 in Figure 4 gets expanded since its relative access counter 0.63 exceeds the threshold of 0.4.

4 EVALUATION

We applied our framework to the use case of point-polygon joins (cf. Section 2). First, we will evaluate the performance of our hybrid trie and compare it against other data structures. Then, we will discuss the impact of workload adaptation on the overall performance and index structure size. Additionally to our approach, we will show a hybrid trie that compresses the values.

We use a real-world data set containing 289 neighborhoods (polygons) in New York City and join them with 1.23 billion publicly available taxi pickup locations (points) for the years 2009 to 2016 [4]. We calculate the cell coverings for the polygons as described in Section 2. Then, we use different index structures

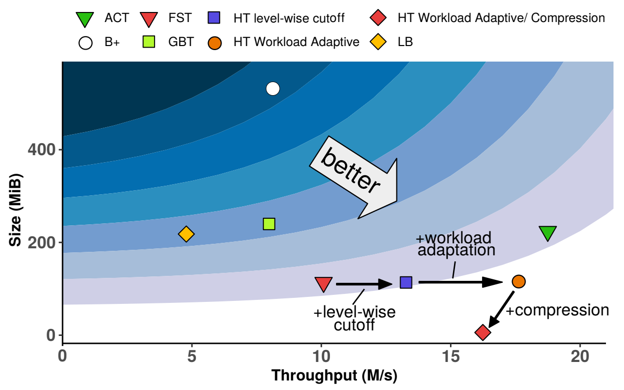


Figure 5: Evaluation of selected index structures considering space and performance. The 'waves' have been added for illustration purposes. Indexes in brighter areas have a better space/performance ratio.

that store the combined covering which contains approximately 14 million cells.

We conduct the experiments on a 14-core Intel Xeon E5-2680 v4 CPUs equipped with 256 GB DDR4 RAM and we compile with GCC 5.4.0 and optimization level 03. Besides ACT and FST, we compare our Hybrid Trie (HT) to the following data structures: the Google B-tree (GBT) [1], the STX B+ Tree (B+) [3], and the `std::lower_bound` algorithm (LB).

Comparing to Other Data Structures. In Figure 5, we show different data structures and their space consumption in MiB for indexing 14 M `uint64_t`-keys on the y-axis. Then, we query 1.23 B keys and denote the performance in M/s on the x-axis.

While LB achieves the lowest throughput (4.78 M/s), the *use-case optimized* ACT allows querying more than 18.75 M entries per second. The most space is used by the B+ tree (535 MiB) and the most space-efficient index structure is the workload-adaptive hybrid trie with enabled *run-length encoding* for the payloads. While GBT uses internal node compression techniques and B+ uses approximately twice the space, they achieve comparable performance (7.98 and 8.10 M/s).

As expected, the performance of the hybrid trie with *level-wise cutoff* (13.27 M/s) is located in between FST (10.09 M/s) and ACT (18.75 M/s), while the required space (113.71 MiB) is increased by a *negligible amount* of 0.007% compared to FST (113.70 MiB, ACT uses 223.65 MiB).

Analyzing Workload Adaptation. In Table 1, we depict the evaluation results for the *workload adaptive* hybrid tries with different level-wise cutoffs and thresholds t. A cutoff level cl means that the upper cl levels are encoded as ACT and the remaining levels are encoded as FST. A node n whose relative accesses *exceed* t gets expanded and added to ACT, whereas n gets removed from ACT if its relative accesses *precede* t. These updates can be performed periodically after a specified amount of time. Entries *without* a given threshold (-) refer to a *non workload-adaptive* hybrid trie. The ACT-encoded part has a *negligible influence* on the total HT size (1.01% for $cl = 11$) and with increasing cutoff levels, the performance impact of the workload adaptation decreases.

For the presented use case, online workload-adaptation works well since taxi pickup locations are skewed (e.g. there are many pickups at the airport and the main train station). The last column of Table 1 shows the percentage of queries that can be answered directly by the refined ACT without entering the FST.

Table 1: Space and performance metrics for the workload-adaptive hybrid tries. With increasing cutoff level and decreasing threshold, the performance increases significantly while the size overhead remains negligible.

Cutoff Level	Refinement Threshold [%]	Size Overhead To FST [%]	TP [M/s] (Perf. of ACT)	ACT-only Lookups [%]
1	-	0.001	10.32 (55%)	0.00
1	1	0.004	14.30 (76%)	31.25
1	0.5	0.008	14.58 (78%)	44.13
6	-	0.007	12.83 (68%)	0.41
6	1	0.009	13.42 (72%)	31.42
6	0.5	0.013	14.65 (78%)	44.30
11	-	1.020	17.28 (92%)	78.13
11	1	1.020	17.27 (92%)	78.13
11	0.5	1.020	17.28 (92%)	78.13

Further Compression Techniques. As discussed in Section 2, most succinct data structures store the payloads in a separate data structure which allows for further compression. We applied *run-length encoding* to the payloads which results in a compression ratio of 19,74 (114856 MiB / 5819 MiB) while the performance of 16.24 M lookups per second is still comparable to ACT. The hybrid trie uses only 2.6% of ACT's space while it retains 86.6% of its performance. By this means, the hybrid trie is *smaller by two orders of magnitude* while it achieves comparable performance to ACT. Note that our approach is not limited to run-length encoding but can of course be combined with any compression technique (e.g. dictionary encoding).

5 CONCLUSIONS AND FUTURE WORK

While our framework achieves good results for the presented use case, it is still an *ongoing work in progress.* As a next step, we will apply the framework to other use cases such as *prefix lookups for strings* and other index structures. We also plan to implement a duplicate-free hybrid trie and apply different compression techniques to the values.

REFERENCES

[1] Google C++ B-tree. https://code.google.com/archive/p/cpp-btree/.
[2] S2 Geometry. https://s2geometry.io/.
[3] STX B+-tree. http://panthema.net/2007/stx-btree/.
[4] *TLC Trip Record Data.* http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.
[5] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing access methods: The RUM conjecture. In *Proc. of EDBT*, pages 461–466, 2016.
[6] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *SEA*, pages 326–337. Springer, 2014.
[7] J. Gray. Tape is dead, disk is tape, flash is disk, ram locality is king. http://research.microsoft.com/en-us/um/people/gray/talks/Flash_is_Good.ppt.
[8] A. Kipf, H. Lang, V. Pandey, R. A. Persa, C. Anneser, E. T. Zacharatou, H. Doraiswamy, P. Boncz, T. Neumann, and A. Kemper. Adaptive main-memory indexing for high-performance point-polygon joins. In *Proc. of EDBT*, 2020.
[9] H. Lang, A. Beischl, V. Leis, P. Boncz, T. Neumann, and A. Kemper. Tree-Encoded Bitmaps. In *Proc. of SIGMOD*. ACM, 2020.
[10] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proc. of ICDE*, volume 13, pages 38–49, 2013.
[11] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *Proc. of SIGMOD*, pages 1567–1581. ACM, 2016.
[12] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *TKDE*, 27(7):1920–1948, 2015.
[13] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF: Practical range query filtering with fast succinct tries. In *Proc. of SIGMOD*, pages 323–336. ACM, 2018.

FairPrep: Promoting Data to a First-Class Citizen in Studies on Fairness-Enhancing Interventions

Sebastian Schelter, Yuxuan He, Jatin Khilnani, Julia Stoyanovich[*]
New York University
[sebastian.schelter,yh2857,jatin.khilnani,stoyanovich]@nyu.edu

ABSTRACT

The importance of incorporating ethics and legal compliance into machine-assisted decision-making is broadly recognized. Further, several lines of recent work have argued that critical opportunities for improving data quality and representativeness, controlling for bias, and allowing humans to oversee and impact computational processes are missed if we do not consider the lifecycle stages upstream from model training and deployment. Yet, very little has been done to date to provide system-level support to data scientists who wish to develop responsible machine learning methods. We aim to fill this gap and present FairPrep, a design and evaluation framework for fairness-enhancing interventions, which helps data scientists follow best practices in ML experimentation. We identify shortcomings in existing empirical studies for analyzing fairness-enhancing interventions and show how FairPrep can be used to measure their impact. Our results suggest that the high variability of the outcomes of fairness-enhancing interventions observed in previous studies is often an artifact of a lack of hyperparameter tuning, and that the choice of a data cleaning method can impact the effectiveness of fairness-enhancing interventions.

1 INTRODUCTION

While the importance of incorporating responsibility — ethics and legal compliance — into machine-assisted decision-making is broadly recognized, much of current research in fairness, accountability, and transparency focuses on the last mile of data analysis — on model training and deployment. Several lines of recent work argue that critical opportunities for improving data quality and representativeness, controlling for bias, and allowing humans to oversee and influence the process are missed if we do not consider earlier lifecyle stages [5, 9, 10, 15]. Yet, very little has been done to date to provide system-level support for data scientists who wish to develop and evaluate responsible machine learning methods. In this paper we aim to fill this gap.

We build on the efforts of Friedler et al. [4] and Bellamy et al. [1], and develop a generalizable framework for evaluating fairness-enhancing interventions called FairPrep. FairPrep implements a modular data lifecycle, enables the re-use of existing implementations of fairness metrics and interventions, and the integration of custom feature transformations and data cleaning operations from real world use cases. Our framework currently focuses on data cleaning (including different methods for data imputation), and model selection and validation (including hyperparameter tuning), and can be extended to accommodate earlier lifecycle stages, such as data integration and curation.

[*]This work was supported in part by NSF Grants No. 1926250 and 1934464, and by the Moore-Sloan Data Science Environment at New York University.

FairPrep by example. Consider Ann, a data scientist at an online retail company who wishes to develop a classifier for deciding which payment options to offer to customers. Based on her experience, Ann decides to include customer self-reported demographic data together with their purchase histories. Following her company's best practices, Ann will start by splitting her dataset into training, validation, and test sets. Ann will then use pandas, scikit-learn, and the accompanying data transformers to explore the data and implement data preprocessing, model selection, tuning, and validation. She will *identify missing values*, and fill these in using a default interpolation method in scikit-learn, replacing missing values with the most frequent value for that feature. Finally, following the accepted best practices at her company, Ann implements model selection and tuning. She identifies several classifiers appropriate for her task, and then *tunes hyperparameters* of each classifier using k-fold cross-validation. As a result of this step, Ann selects a classifier that shows acceptable accuracy, while also exhibiting sufficiently low variance.

No fairness issues were explicitly surfaced in Ann's workflow up to this point. This changes when Ann considers the accuracy of the classifier more closely, and observes a disparity: the accuracy is lower for middle-aged women, and for female customers who did not specify their age as part of their self-reported demographic profile. Ann goes back to data analysis and observes that the value of the attribute age is missing far more frequently for female users than for male users. Further, she compares age distributions by gender, and notices differences starting from the mid-thirties. Ann hypothesizes that age is an important classification feature, revisits the data cleaning step, and selects a state-of-the-art *data imputation method* such as datawig [2] to fill in age (and other missing values) in customer demographics.

Having adjusted data preprocessing to reduce error rate disparities, Ann is now faced with several related challenges:

- How should the data processing pipeline be extended to *incorporate additional fairness-specific evaluation metrics?*
- How can the *effects of fairness-enhancing interventions be quantified and judiciously validated?* These interventions may range from an improved data cleaning method that helps reduce variance in the outcomes for a demographic group, to a fairness-aware classifier, and they may be incorporated at different pipeline stages.
- How does one continue to follow best practices for ML evaluation when incorporating fairness considerations into these pipelines? For example, how does Ann ensure an appropriate level of isolation of the test set, and how does she tune hyperparameters in light of additional objectives?

To address these challenges, Ann will turn to existing development and evaluation frameworks: that by Friedler et al. [4] and IBM's AIF360 [1]. While these frameworks are certainly a good starting point, they will unfortunately fall short of meeting Ann's needs because they (1) are designed around a small number of academic datasets and use cases, (2) lack the flexibility to integrate additional data preprocessing steps that are a crucial part

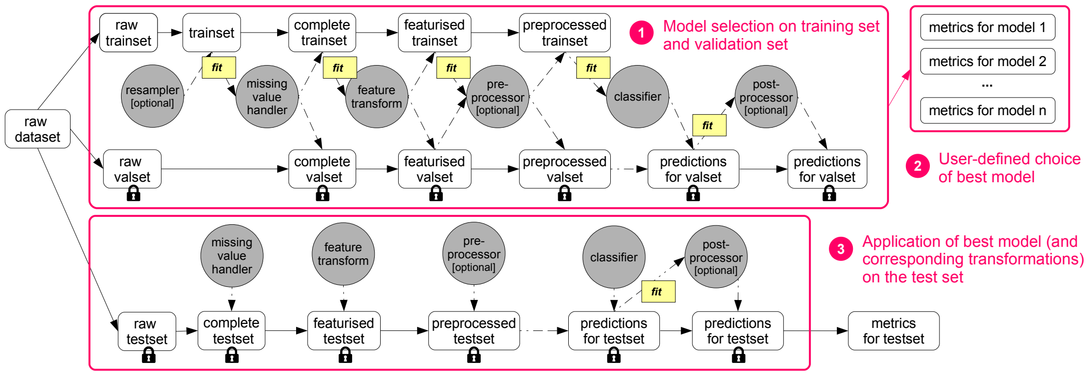


Figure 1: Data life cycle in FairPrep, designed to enforce isolation of test data, and to allow for customization through user-provided implementations of different components. An evaluation run consists of three different phases: (1) Learn different models, and their corresponding data transformations, on the training set; (2) Compute performance / accuracy-related metrics of the model on the validation set, and allow the user to select the 'best' model according to their setup; (3) Compute predictions and metrics for the user-selected best model on the held-out test set.

of existing machine learning pipelines, and (3) are not designed to enforce best practices.

This paper makes the following contributions:

- We discuss shortcomings and violations of sound experimentation practices in existing empirical studies and software for analyzing fairness-enhancing interventions (Section 2).
- We propose FairPrep, a design and evaluation framework that promotes data to a first-class citizen in fairness-related studies (Section 3).
- We demonstrate how FairPrep can be applied to illustrate the impact of violations of best practices of ML experimentation, and how it enables the inclusion of incomplete data into studies, which is not supported by existing frameworks (Section 4).

In what follows, we briefly describe these contributions, see our technical report for additional information [14]. FairPrep is open-sourced at https://github.com/DataResponsibly/FairPrep.

2 SHORTCOMINGS OF PREVIOUS WORK

We inspect the code base of an existing study [4] and of an evaluation framework [1] for fairness-enhancing interventions, and identify a set of shortcomings and violations of best practices that potentially invalidate some the findings of these studies.

Insufficient isolation of held-out test data. A major requirement for the evaluation of ML algorithms is to simulate real world scenarios as closely as possible. In the real world, we train our model (and select its hyperparameters) on observed data from the past. This model is later used to make predictions for unseen target data for which the ground truth is unknown. To emulate this real-world deployment scenario, we evaluate the trained model on a test set that was randomly sampled from observed historical data. It is crucial that this test set be completely isolated from the process of model selection, which, consequently, is only allowed to use the training data (the remaining, disjunct observed historical data). Unfortunately, we encountered violations of the test set isolation requirement in the existing benchmarking framework by Friedler at al. [4], bringing into question the reliability of reported study results. Further, we found that the architecture of the IBM AIF360 toolkit [1] does not support data isolation best practices for feature transformation.

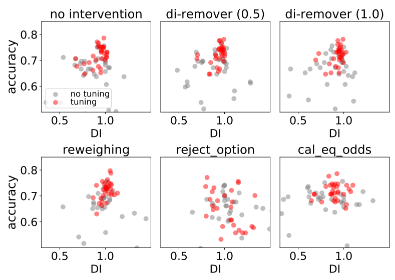
Hyperparameter selection on the test set. Grid search for hyperparameters[1] of fairness-enhancing models and interventions in [4] computes metrics for all hyperparameter candidates on the test set, and returns the candidate that gave the best performance. This strongly violates the isolation requirement. Instead, an evaluation procedure should maintain an additional validation set to select hyperparameters, and only evaluate prediction quality of the resulting single best hyperparameter candidate on the test set, to measure how well the model generalizes on unseen data.

Lack of hyperparameter tuning for baseline algorithms. We additionally found that the study by Friedler et al. [4] did not tune the hyperparameters of the baseline algorithms[2] for which pre-processing and post-processing interventions are applied, even though they tuned the hyperparameters of the fairness interventions. This is problematic because there is no guarantee that the baseline algorithm will converge to a good solution with the default parameters. Friedler et al. [4] found a high variability of the fairness and accuracy outcomes with respect to different train/test splits, which could be an artifact of the described lack of hyperparameter optimisation.

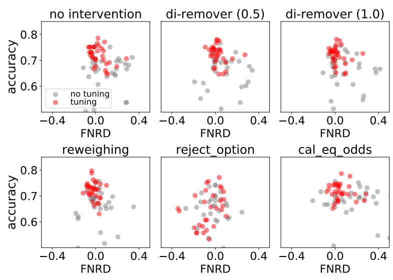
Lack of feature scaling. We observed that both existing frameworks [1, 4] do not normalise the numeric features of the input data, but keep them on their original scale. While some ML models such as decision trees are insensitive to feature scaling, many other algorithm components, such has the L1 and L2 regularizers of linear models, implicitly rely on standardized features.

Removal of records with missing values. Another point of critique is that the study of Friedler et al. [4] ignored records with missing values (by removing them before running experiments), which means that the study's findings do not necessarily generalize to data with quality issues. Thereby, existing frameworks are unable to investigate the effects of fairness enhancing interventions on records with missing values, which could be especially important for cases where a protected group has a higher likelihood of encountering missing values in their data [8].

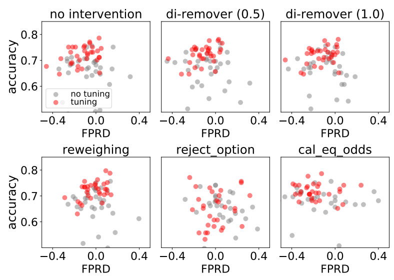
[1]https://github.com/algofairness/fairness-comparison/blob/4e7341929ba9cc98743773169cd3284f4b0cf4bc/fairness/algorithms/ParamGridSearch.py#L41
[2]https://github.com/algofairness/fairness-comparison/tree/35fb53f7cc7954668eeee28eac5fb20faf89b3d8/fairness/algorithms/baseline



(a) Accuracy / disparate impact.



(b) Accuracy / false negative rate difference.



(c) Accuracy / false positive rate difference.

Figure 2: Impact of hyperparameter tuning on the accuracy and fairness metrics of logistic regression models (in combination with various preprocessing and postprocessing interventions) on the `germancredit` dataset. Hyperparameter tuning (red dots) often results in higher accuracy and reduced variance of the fairness outcome compared to no tuning (gray dots).

3 FRAMEWORK DESIGN

The identified shortcomings motivate us to propose FairPrep, an evaluation and experimentation framework.

Design principles. We implement FairPrep on top of scikit-learn [11] and AIF360 [1], and design it based on two principles: (i) *Data isolation* — to avoid target leakage, user code should only interact with the training set, and never access the held-out test set. User code can train models or fit feature transformers on the training data, which will be applied by the framework to the test set later on. The framework should furthermore especially take care of data with quality problems. For example, it should allow experimenters to quantify the effects of their code on records with missing values by computing metrics and statistics separately for these records. (ii) *Explicit modeling of the data lifecycle* — the evaluation framework defines an explicit, standardized data lifecycle that applies a sequence of data transformations and model training in a particular, predefined order. Users influence and define the lifecycle by configuring and implementing particular components. At the same time, the framework should support users in applying best practices from ML experimentation.

Data lifecycle. Figure 1 illustrates the data lifecycle during the execution of a run of FairPrep: ① *Model selection on the training set and validation set*: we train different models on the training data, where we apply the following consecutive steps: (i) resampling of training data (e.g., bootstrapping or balancing, optional); (ii) treatment of records with missing values (either removal or imputation); (iii) feature transformation (e.g., scaling of numeric values, one-hot encoding of categorical values); (iv) potential application of a preprocessing intervention; (v) model training using grid search; (vi) computation of predictions on the train and validation set; (vii) potential application of postprocessing intervention to predictions from train and validation set. ② *User-defined choice of the best model*: Users can choose between the explored models based on accuracy-related and fairness-related metrics computed on the validation set, trading these off as appropriate in their context. ③ *Application of the best model on the test set*: The user-selected best model (and its corresponding data transformations) are finally applied to the test set, and the resulting accuracy of fairness are reported by FairPrep.

4 EXPERIMENTAL EVALUATION

We now demonstrate how FairPrep can be used to showcase one of the shortcomings from Section 2, and how it enables experimentation on incomplete data. For all experiments, data is randomly split into 70% training, 10% validation, and 20% test.

Impact of hyperparameter tuning on the variability of accuracy and fairness. In the first experiment, we aim to investigate the effect of the lack of hyperparameter tuning of baseline models during experimentation (as discussed in Section 2).

For consistency with Friedler et al. [4], we use the `germancredit` dataset[3], which contains 20 demographic and financial attributes of 1000 individuals, including the sensitive attribute sex. The task is to predict each individual's credit risk. We use two baseline models (logistic regression and decision tree) in two different variants each: (i) without hyperparameter tuning, where we just use the default hyperparameters of the baseline model; (ii) with hyperparameter tuning, where we apply grid search (over 3 regularizers and 4 learning rates for logistic regression; over 2 split criteria, 3 depth params, 4 min samples per leaf params, 3 min samples per split params for the decision tree) and five-fold cross validation on the training data. We apply three different fairness-enhancing interventions that preprocess the data: 'disparate impact remover' ('di-remover' in the plots) [3] with repair levels 0.5 and 1.0, and 'reweighing' [6]. Additionally, we experiment with two fairness-enhancing interventions that post-process the predictions: 'reject option classification' [7] and 'calibrated equal odds' [12]. We use 16 different random seeds and execute 1,344 runs in total. We report metrics computed from predictions on the held-out test set.

Results. We plot the results of this experiment in Figure 2, where we show the resulting accuracy and several fairness related measures[4] between the privileged and unprivileged groups, including disparate impact (DI), the difference in false negative rates (FNRD), and the difference in false positive rates (FPRD). The red dots denote the outcome when we apply hyperparameter tuning to the baseline model, while the gray dots denote the outcome using the default model parameters, without tuning. We observe a large number of cases where the tuned variant results in both a higher accuracy *and* a lower variance in the fairness outcome. Examples are (i) accuracy and disparate impact for the 'di-remover' and 'reweighing' interventions in Figure 2(a), (ii) accuracy and false negative rate difference for 'di-remover' in Figure 2(b); and (iii) accuracy and false positive rate difference for 'di-remover' in Figure 2(c). We obtained similar results for the decision tree model and omit the corresponding plots due to lack of space.

[3]https://archive.ics.uci.edu/ml/support/Statlog+(German+Credit+Data)

[4]We plot these measures regardless of whether the intervention optimizes for them.

Our results indicate that the high variability of the fairness and accuracy outcomes with respect to different train/test splits observed by Friedler et al. [4] may be an artifact of the lack of hyperparameter tuning of the baseline models in these studies.

Enabling the inclusion of incomplete data. Next, we showcase how FairPrep can be used to quantify the effect of including records with missing values into an experimental study. These records are commonly filtered out in other studies and toolkits, as discussed in Section 2.

We use the `adult` dataset[5] for this experiment, with a total of 32,561 instances and 14 attributes, including the sensitive attributes `race` and `sex`, and 2,399 instances with missing values. The task is to predict whether an individual earns more or less than $50, 000 per year. Fairness evaluation is conducted between the privileged group of white individuals (85% of records) and the underprivileged group of non-white individuals (15% of records).

Of the 14 attributes, three have missing values — `workclass`, `occupation`, and `native-country`. Based on our analysis, missing values do not occur at random, as the records with missing values exhibit very different statistics than the complete records. For example, the positive class label (high income) is associated with 24% of the complete records, but with only 14% of the records with missing values. Additionally, married individuals are in the vast majority in the complete records, while the most frequent `marital-status` among the incomplete records is *never-married*. Furthermore, the records with missing values from the privileged group are very different from the records with missing values from the underprivileged group. For example, the attribute `native-country` is missing four times more frequently for non-white individuals than for white individuals. Among the incomplete privileged records, 15% are associated with a high income, the second largest age group consists of 60 to 70 year-olds, and the majority of the individuals is married. For the incomplete records from the underprivileged group, however, only 10.6% have a high income, there are very few individuals over 60, and the majority of the individuals is unmarried.

We use logistic regression as the baseline learner, with hyperparameter tuning analogous to previous experiments. As before, we apply two fairness enhancing interventions that preprocess the data: 'disparate impact remover' [3] and 'reweighing' [7]. We use three strategies to treat missing values: (*i*) complete case analysis, removing incomplete records; (*ii*) retain all records and impute missing values with 'mode imputation'[6] (replace a missing value with the most frequent value for that feature); (*iii*) retain all records and apply model-based imputation with datawig [2]. We execute 530 runs and report metrics computed on the held-out test set.

Results. We investigate the classification accuracy for complete and incomplete records, under imputation with mode and datawig. First, we observe that records with imputed values achieve high accuracy. This is a significant result, since these records could not have been classified at all before imputation! Interestingly, we observe higher accuracy for records with missing values compared to the complete records. Based on our understanding of the data, we attribute this to the higher fraction of (easier to classify) negative examples among the incomplete records. Further, we do not observe a significant difference in accuracy between mode imputation and datawig. We attribute this to the skewed distribution of the attributes to impute — a favorable setting for mode imputation. Because datawig does no worse than mode, and is expected to perform better in general [2], we only present results for datawig-based imputation in the final experiment.

We compute the accuracy and disparate impact of complete case analysis (e.g., the removal of incomplete records) versus the inclusion of incomplete records with datawig imputation. We observe a minimally higher accuracy in the case of including incomplete records, but in general find no significant positive or negative impact on disparate impact. Taken together, the results paint an encouraging picture: Imputation allows us to classify records with missing values, and do so accurately, and it does not degrade performance, either in terms of accuracy or in terms of fairness, for the complete records.

5 CONCLUSION

We identified shortcomings in existing empirical studies on fairness-enhancing interventions. Subsequently, we presented the design of our evaluation framework FairPrep. This framework empowers data scientists to conduct experiments on fairness-enhancing interventions with low effort, and at the same time enforces machine learning best practices. We demonstrated how FairPrep can be used to measure the impact of a lack of hyperparameter tuning, and how it enables the inclusion of incomplete data. We aim to extend FairPrep by integrating additional fairness-enhancing interventions [13], datasets, preprocessing techniques, and feature transformations. Additionally, we intend to extend its scope to scenarios beyond binary classification, and introduce *human-in-the-loop* elements by providing visualisations and allowing end-users to control experiments with low effort.

This paper is supplemented by a technical report [14]. FairPrep is open-sourced at https://github.com/DataResponsibly/FairPrep.

REFERENCES

[1] Rachel Bellamy et al. 2019. AI fairness 360: An extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias. In *FAT*ML*.

[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*.

[3] Michael Feldman, Sorelle A. Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and Removing Disparate Impact. In *KDD*.

[4] 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 *FAT*ML*.

[5] Kenneth Holstein, Jennifer Wortman Vaughan, Hal Daumé III, Miroslav Dudík, and Hanna M. Wallach. 2019. Improving Fairness in Machine Learning Systems: What Do Industry Practitioners Need?. In *CHI*.

[6] Faisal Kamiran and Toon Calders. 2012. Data preprocessing techniques for classification without discrimination. *Knowledge and Information Systems* 33, 1 (2012), 1–33.

[7] Faisal Kamiran, Asim Karim, and Xiangliang Zhang. 2012. Decision theory for discrimination-aware classification. In *ICDM*.

[8] Joost Kappelhof. 2017. *Total Survey Error in Practice.* Chapter Survey Research and the Quality of Survey Data Among Ethnic Minorities.

[9] Keith Kirkpatrick. 2017. It's Not the Algorithm, It's the Data. *CACM* 60, 2, 21–23.

[10] David Lehr and Paul Ohm. 2017. Playing with the Data: What Legal Scholars Should Learn about Machine Learning. *UC Davis Law Review* 51, 2 (2017), 653–717.

[11] Fabian Pedregosa et al. 2011. Scikit-learn: Machine learning in Python. *JMLR* 12, 2825–2830.

[12] Geoff Pleiss, Manish Raghavan, Felix Wu, Jon Kleinberg, and Kilian Q Weinberger. 2017. On fairness and calibration. In *NeurIPS*.

[13] Babak Salimi, Luke Rodriguez, Bill Howe, and Dan Suciu. 2019. Interventional fairness: Causal database repair for algorithmic fairness. In *SIGMOD*.

[14] Sebastian Schelter, Yuxuan He, Jatin Khilnani, and Julia Stoyanovich. 2019. FairPrep: Promoting Data to a First-Class Citizen in Studies on Fairness-Enhancing Interventions. *arXiv:1911.12587* (2019).

[15] Julia Stoyanovich, Bill Howe, Serge Abiteboul, Gerome Miklau, Arnaud Sahuguet, and Gerhard Weikum. 2017. Fides: Towards a Platform for Responsible Data Science. In *SSDBM*.

[5] https://archive.ics.uci.edu/ml/datasets/adult

[6] https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer

Short paper

Partially Materializable Delta Trees for Efficient Data Wrangling of Semi-Structured Contents

Nico Schäfer
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
nschaefer@cs.uni-kl.de

Sebastian Michel
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
michel@cs.uni-kl.de

ABSTRACT

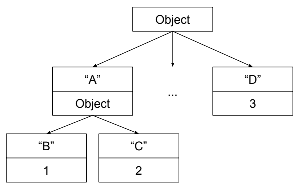
In this paper, we propose delta trees to boost efficiency and reduce storage requirements of iterative data exploration and data wrangling tasks over massive, semi-structured datasets. During such tasks, data is filtered, projected, joined, and converted in multiple successive or independent steps, driven by data scientists or higher-level applications. While the original datasets can often not be disposed, delta trees are necessary to represent only the changes to the original data, instead of creating largely redundant copies. With delta trees, we are able to reduce storage requirements and query execution time for various data manipulation operations, while maintaining acceptable query times for others. We report on a first experimental study over a dataset of Twitter tweets, showing that the expected vast savings of storage consumption can be enjoyed with negligible computational overhead compared to a full data duplication.

1 INTRODUCTION

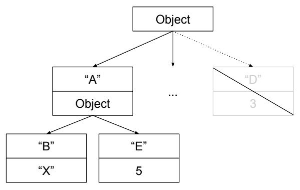
In recent years, the interest in semi-structured file formats steadily increased. Arguably, one of the most visible data formats is JSON, which eliminates the need to force data into relations and is designed to be human-readable. It has been adopted by various platforms and systems, for instance, for data exchange through APIs, to store system access logs, or configuration files. Common operations in semi-structured document processing are adding, removing, and moving values. Consider for instance the case of data scientists working on a large sample of Twitter tweets. While some first extract textual content and geo-coordinates of Canadian tweets written in French they later observe that also the author of the tweet is required, others need to convert the original schema (attribute names) to match their existing data visualization libraries. To execute these operations, systems have to either edit the original document, or perform the operation on a copy. The first option may not always be possible, if the base documents are to be used in future processing steps or by multiple data scientists working in parallel. The second option is undesirable, because it introduces a huge overhead regarding performance and memory usage, for copying the documents. In this paper, we propose a novel way to store changes made to original documents, leading to a vastly reduced memory footprint and an even improved querying performance for some query types, while having a modest performance overhead for others.

1.1 Sketch of the Approach and Related Work

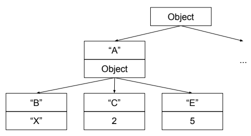
The concept of late materialization [1] is used to push back materializations of (intermediate) results until they are needed. For this to work, the system has to store the queries, or similar information, and the base data. The result is then calculated on demand as soon as it is required.



(a) Base document tree



(b) Delta tree



(c) Combined document

Figure 1: Example of base document with a delta tree

Our approach also aims to reduce the amount of unnecessary transformations of the base data. But instead of storing the operations that lead to the result, we calculate the difference (or delta) of the transformations, compared to the base data and only store this new information, with a reference to the base document. Almeida *et al.* already proposed a map as Conflict-free Replicated Data Type (CRDT) [2], which can be synchronized by sending delta changes of the modification action to replicated instances. This data type can also be nested and JSON documents can be translated to and from nested maps. But the computational overhead required for conflict-free synchronization is too large for our use case. Generally, the concept of extracting or storing deltas from semi-structured documents has been investigated before [3, 6, 7]. It has also been applied in the context of relational database systems to compact historical data [4, 5]. However, in contrast, we store the changes made by incremental data modification operations, in order to improve the memory consumption.

2 DELTA TREES

Figure 1a visualizes the tree representation of one semi-structured document. It consists of (nested) objects containing attributes, with atomic data. Now, a query may transform this base document by deleting the "D" attribute of the root object, changing the atomic data within the "B" attribute, and adding an additional member to the "A" object. Instead of storing the complete transformed document, we only store the changed parts of the tree, as shown in Figure 1b. In this example, the changed "B" member and additional "E" attribute is stored, together with the required parent structure (in this case the root node and "A" object).

Additionally, we keep track of all paths within the tree that have been changed. A path is an ordered list of tokens, identifying all nodes that have to be traversed to arrive at a desired destination. In this example, the changed paths would be "/A/B", "/A/E", and "/D". This information is sufficient to reconstruct the complete transformation result as shown in Figure 1c.

The benefits of this approach may not be immediately evident from the given example. But in real world data sets, documents having hundreds of attributes, distributed over many nested objects, are very common. If a transformation query only modifies a couple of nodes, storing the complete transformation result clearly is unnecessary—and modifying the original data is often prohibited.

2.1 Reconstructing Combined Documents

Given a base tree B and a delta tree D, with the overwritten paths P, we may need to construct the complete result document tree R to return to the user. This is achieved by simultaneously iterating through both trees in a depth-first manner. Starting at the root of the trees, for every unique path p in both trees, there are now five possibilities:

1. $\exists n_B \in B$ and $\exists n_D \in D$ belonging to p
   - (a) $p \in P \Rightarrow$ the path is overwritten in D, hence, we only continue traversing n_D for this sub-tree.
   - (b) $p \notin P \Rightarrow$ the path is shared between B and D, thereby, we continue traversing both n_B and n_D.
2. $\exists n_B \in B$ and $\nexists n_D \in D$ belonging to p
   - (a) $p \in P \Rightarrow$ the path was removed by the transformation and we do not continue traversing this sub-tree.
   - (b) $p \notin P \Rightarrow$ the node is not materialized, but still valid, thus, we continue traversing n_B.
3. $\nexists n_B \in B$ and $\exists n_D \in D$ belonging to p, it must thereby be added by D and we only traverse n_D.

The result is constructed by traversing B and D as above and adding all visited nodes to R.

2.2 Delta Hierarchies

There can be multiple delta trees, which reference the same base tree, in which case the memory savings introduced by delta trees are magnified. Additionally, delta trees may be based on other delta trees. Given $(B, D_1, ..., D_i, ..., D_n)$, where B is the base document, and D_i are delta trees, each based on the previous tree. This is called a *delta hierarchy*. The result document R_1, of B and D_1 may be constructed as shown in Section 2.1. To construct the result document R_i, the same algorithm would then be executed with R_{i-1} as base tree and D_i as delta tree. In case of larger hierarchies, this naïve execution is suboptimal. Instead, we perform the reconstruction algorithm for all trees at the same time, by traversing the whole delta hierarchy simultaneously. For each unique path p in the delta hierarchy, starting at the root: If p exists only in one tree, we follow only this subtree. If p exists in multiple trees, we use the value of the uppermost delta tree, in case of atomic values. In case of objects, whose child attributes are distributed over multiple tree, we continue traversing all affected trees.

3 ARCHITECTURE AND ALGORITHMS

We implemented our approach in our in-house JSON exploration system JODA, written in C++. This system uses the RapidJSON (http://rapidjson.org/) parser, which uses a DOM-tree in-memory representation to store the parsed JSON documents—the extension to XML and YAML is straightforward.

Documents provided to the system are organized in so-called collections, for instance, a collection of Twitter tweets and a collection of blog posts. All documents within a collection are organized into a number of containers. Each container is a self-contained unit, which includes all required information to perform a query upon. After creation, these containers are immutable to make the query execution free of any synchronization overhead.

Queries are simple PIG-style sequences of commands. To begin, a collection can be chosen and data can be imported into the system by the `LOAD` step. This data is then passed to the `CHOOSE` command, which may filter the data depending on a given predicate. The filtered documents may then be transformed in the `AS` step, by using an arbitrary amount of transformation instructions, in the form of (`<destination>:<source>`) tuples, where the destination is a path in the new document, and the source can be any supported function or a path in the base document. The transformed documents are then passed to the `AGG` command for aggregation and may finally stored in a collection or exported into a file with the `STORE` expression.

3.1 Construction

The `AS` instruction also supports the special `*` operator, which copies the whole source document. This operator may be combined with additional transformations to change the source document. If the system detects this combination, delta trees may be used to perform this transformation.

In this case, the system will first create the support structure, by instantiating an empty delta tree with a pointer to the base document, which may also be a delta tree. Each additional transformation is then executed and the result materialized in the newly created delta tree. Additionally, the `<destination>` pointers form the explicitly overwritten paths and are stored with the tree, as described in Section 2. These paths will be the same for all delta trees that are created by a given query. We can therefore store them once in the—previosuly introduced—container class, as it includes all data that may be shared by its documents. Now, a delta hierarchy is given by following the base document pointers from any given delta tree to the bottom.

3.2 Reconstruction

To reconstruct the result document we created a visitor class, which implements the idea described in Section 2.1–2.2. This visitor class simultaneously traverses the delta hierarchy as described. This basic visitor class is then extended to fulfill different needs, like materializing the final result documents or accessing specific subtrees for query evaluation.

Our system preserves the order of members within a JSON object, which is not required by the standard itself. Thereby we have to adapt the traversing algorithm described previously as follows:

Algorithm 1 reconstructs a result document, given a delta hierarchy. Our implementation handles the base document as just another delta tree D_0 which overwrites the whole tree (root path ' '). The algorithm visits a node n, which at the beginning is the root node of D_n. If the given node is shared, i.e., it is an object or array which is distributed over multiple delta trees, then we first search the base document. The base document is the document which has overwritten this node last. Starting from the base, we collect all members of this shared node in all delta trees. The algorithm is then repeated for each of these members.

If the node is not shared, then we retrieve the upper most instance of the node in the delta hierarchy and visit this node.

```
Data: n = root($D_n$); p = ' '; $D_0, ..., D_n$; $P_0, ..., P_n$
if IsShared(n,$P_1, ..., P_n$) then
    Base = GetBaseDeltaTree(n);
    for i = Base to $D_n$ do
        members += GetMembers($D_i$,$P_i$);
    end
    for member in members do
        Recurse(member,p+'/'+member.id,$D_0$,...);
    end
else
    $n_D$ = GetBaseNode(n);
    Visit $n_D$;
end
```

Algorithm 1: Reconstruction algorithm

The visit function belongs to the given visitor, which may, as explained previously, perform different actions for the given node.

3.3 Estimating Memory Usage

The main advantage of delta trees is the reduced memory requirement. We define the memory cost of a tree, as the sum of all costs of its nodes. The cost of a node is given by the byte size of its in-memory representation. Evidently, each delta tree is smaller or equal in size as the result tree, given by combining the base with the delta tree, as all of its nodes are contained in the result, plus potential additional nodes from the base document. Thus, the memory cost of delta trees should always be smaller or equal to materializing the whole result.

In reality, this is often not the case. For instance, the RapidJSON library, that we use to create JSON documents, creates each new object and array with 16 placeholder children. In many cases, this is a sensible decision, as reallocating memory for more children is an expensive operation and objects and arrays often have more than one child. For delta trees that mostly consist of a few nodes, this decision proved to be a disadvantage. Each placeholder child will be added to the cost, which for some queries and documents may be more than materializing the result.

We thereby added a sample step to our system before deciding which execution method, delta trees or complete materialization, to choose. The transformation is performed for $\leq$ 1% of documents with both execution methods. Then the memory requirement of these documents is calculated and the method with the lowest requirement is chosen. This decision is performed on per-container basis in our system, which mostly contain similar documents.

3.4 Accessing values

Queries in our system may use a wide range of functions to evaluate values. The parameters of functions may be nested functions or retrieved directly from the document. Hence, functions have to be able to access all values contained in delta trees. For example, it is not trivial to evaluate the member count of an object, that is distributed over multiple trees. If a function accesses a value, specified by a given path, the system will check top-down for the given delta tree hierarchy if this specific path is shared. If not, the value can be directly passed to the function, without even traversing the whole delta hierarchy. If it is shared, we have to evaluate this function by traversing the delta tree hierarchy as explained in Section 3.2.

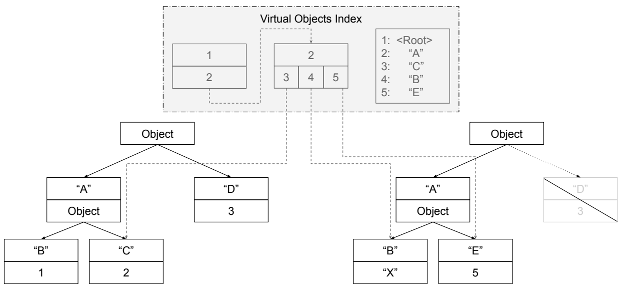


Figure 2: Virtual Object Index

The main cost of accessing values by path is traversing the tree. For each token in the path, the child represented by this token has to be found. For arrays, this is simple, as the token is the index of the child. Objects on the other hand, store their children in order of occurrence in the source document. Here a linear search with string comparisons has to be performed to find the specified child. This operation is the dominating cost factor for finding a value belonging to a path. For delta hierarchies this cost may be amplified, if many trees have to be searched. But if the value is overwritten in one of the higher trees in a delta hierarchy, this cost can be less than for one materialized result document.

4 PARTIAL MATERIALIZATION AND OBJECT INDICES

It may happen, that multiple queries repeatedly require the same value, which may be an object or array distributed over multiple delta trees. In these cases it can be beneficial to materialize the given object or array at the cost of increased memory usage. This is achieved by traversing only this sub-tree in the delta hierarchy, as described previously, and copying the whole sub-tree into the highest delta tree. We call this *partial materialization* of the result. By strategically materializing parts of the result we can still massively reduce the memory footprint while preventing the performance drawbacks of simultaneously traversing multiple trees.

To prevent the materialization of objects, we created an *virtual object index*. This index tries to combine the powers of delta trees with the read performance of materialization. A virtual object, is a list of tuples, containing an attribute id and a pointer to a value or nested virtual object, as shown in Figure 2. The attribute id is a numerical value, retrieved by mapping a string attribute name to a numerical value using a hash map. This has two advantages. (1) Having a numerical value reduces the cost of comparisons needed for the linear search of children. (2) The string dictionary is stored in the container and shared by many documents, thereby reducing the required memory of this index. We create these virtual objects, as soon as an object, that is distributed over multiple trees in the delta hierarchy, is traversed for the first time. During traversal, we map the attribute names of the children to the attribute id and add it to the virtual object, together with the pointer of the actually traversed value. Each value may reside in a different tree within the delta hierarchy. The traversed object is then replaced by the virtual object in the highest delta tree of the hierarchy. Future accesses of the object can then use the created index without traversing multiple trees.

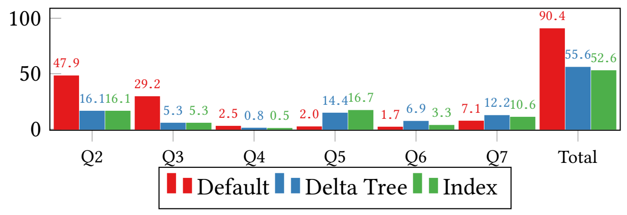


Figure 3: Runtime [s] of different execution methods

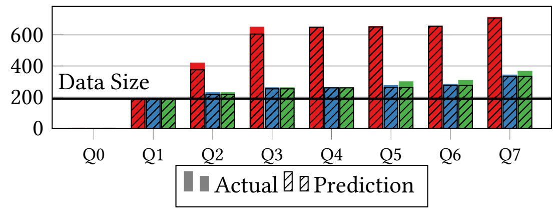


Figure 4: Memory consumption [GB] of different execution methods

5 EVALUATION

In this section, we will evaluate the performance and memory footprint of our delta tree implementation. The data set used is a 109 GB selection of the Twitter JSON stream[1]. It consists of 29,634,708 JSON documents, where each document has between 7 and 348 attributes, containing every JSON type. The documents are split into two major groups. Around 23.5 million (79.33%) documents are normal tweets, while around 6.1 (20.67%) million documents are deletion instructions. The tweets have a varying number of attributes, depending on their status, e.g., retweets and favorites, while the deletion documents consist of seven attributes.

The tests are executed on a machine with 4 Xeon E7-4830 CPUs, each having 12 cores—and 24 threads—with 2.1 GHz. Furthermore, 33 RAM-Kits, each having 32 GB of memory at 2400 MHz, are included, providing the server with around 1 TB of RAM. The data is stored on one HGST Ultrastar 7K4000 HDD, with 7200 RPM. Ubuntu 16.04.3 LTS is used as the underlying operation system.

The first query in Listing 1 loads the Twitter data set. Then a collection is created which adds one member to the user object of the previous data set. Derived from this collection, another attribute is added to the user object. In the following query, only the data added in Q2 is used in an aggregation. Then the member count of the user object is queried in the next two queries. The last query copies the shared user object into a new collection.

```
Q1: LOAD t1 FROM FILES "/data/twitter";
Q2: LOAD t1 CHOOSE EXISTS('/user')
    AS *,('/user/v1':1) STORE t2;
Q3: LOAD t2 AS *,('/user/v2':2) STORE t3;
Q4: LOAD t3 AGG ('':SUM('/user/v1')) STORE a;
Q5: LOAD t3 AS ('':MEMCOUNT('/user')) STORE c1;
Q6: LOAD t3 AS ('':MEMCOUNT('/user')+1) STORE c2;
Q7: LOAD t3 AS ('':'/user') STORE user;
```

Listing 1: Queries iteratively changing an object and reading it

We compare our introduced approaches against the default execution method, which copies and modifies the full JSON documents. The delta tree approach is based on our implementation within the system, as explained in Section 3. The index approach uses the same implementation, but with enabled virtual object indexing, as described in Section 4. The query time plot in Figure 3 is omitting the first data import query, as it is unaffected by the execution method and requires the same time for all of them.

As we can see, for queries Q2 and Q3, the delta tree approaches have a strong advantage over the default execution method. While the default execution copies the whole Twitter data set, the delta tree approaches only require one reference to the base document and the `/user/v<x>` values, with the supporting tree structure. This results in an increase of maximum 37.24GB and 31.89GB to the previous query for Q2 and Q3 respectively, as can be seen in Figure 4. The default approach on the other hand increases its memory consumption by 228GB and 230GB. As copy operations are the dominating cost for these queries, the execution times of the delta tree methods is also significantly lower.

In Q4 the value written in Q2 is read. This is fast and memory unintensive for all approaches, but the delta tree approaches are faster, as the value can be read very fast in one of the delta trees without traversal. In Q5 and Q6 the user object, which is distributed between three delta trees is used. For the default execution method this is fast and requires nearly no memory. For the normal delta tree method, this operation is slow, as the whole delta hierarchy has to be traversed. The index introduces additional overhead, as it creates the virtual object indices. This results in vastly improved query times, but increased memory consumption in Q6, which brings it closer to the default implementation, while the delta tree execution is much slower. In Q7 the modified user object is materialized to a new document. This is relatively fast for the default execution method, but once again slow for the delta tree. The indexed method can use the virtual objects to improve its query time. All in all are the delta tree implementations faster and very similar for this query set. But if it would contain additional read queries this situation could quickly change.

6 CONCLUSION

In this paper, we introduced the concept of delta trees, for materializing only the differences of a transformation, to reduce the memory footprint of exploration systems. We explained the basic idea and specific implementation details, based on our in-house JSON exploration tool JODA. Additionally, we introduced improvements to the systems to mitigate the performance bottlenecks introduced by the approach. As we have seen, delta trees enable systems to perform the same set of queries, with a fraction of the required memory.

REFERENCES

[1] Daniel J Abadi, Daniel S Myers, David J DeWitt, and Samuel R Madden. 2007. Materialization strategies in a column-oriented DBMS. ICDE 2007, 466–475.

[2] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta state replicated data types. *J. Parallel and Distrib. Comput.* 111 (2018), 162–173.

[3] Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change detection in hierarchically structured information. *Sigmod Record*, Vol. 25, 493–504.

[4] Sándor Héman, Marcin Zukowski, Niels J Nes, Lefteris Sidirourgos, and Peter Boncz. 2010. Positional update handling in column stores. *SIGMOD 2010*, 543–554.

[5] David B Lomet and Feifei Li. 2009. Improving transaction-time DBMS performance and functionality. *ICDE 2009*, 581–591.

[6] Amélie Marian, Serge Abiteboul, Gregory Cobena, and Laurent Mignet. 2001. Change-centric management of versions in an XML warehouse. *VLDB 2001*, Vol. 1. 581–590.

[7] Hong Su, Diane Kramer, Li Chen, Kajal Claypool, and Elke A Rundensteiner. 2001. XEM: Managing the evolution of XML documents. *11th International Workshop on Research Issues in Data Engineering (RIDE 2001)*, 103–110.

[1]https://developer.twitter.com/en/docs/labs/sampled-stream

Towards Fine-Grained Data Access Control Through Active Peer Probing

Yael Amsterdamer
Bar-Ilan University
first.last@biu.ac.il

Osnat Drien
Bar-Ilan University
first.last@live.biu.ac.il

ABSTRACT

Data is routinely being shared online by peers, for instance in business transactions, social activities and others. This data, in turn, is often transferred, processed and combined through complex querying and analytics. This raises questions such as the following: who owns the derived data? With whom and for what purpose may it be published? If consent is required for its dissemination, whose consent should be obtained?

The related topics of data sharing, privacy and access control have been extensively studied, but uniquely our focus here is not on data management with known policies but rather on the active probing of peers to ask for their consent. Active probing has the potential to allow finer-grained access control, where it is unreasonable to expect data owners to publish their full policies, defined for all possible sharing scenarios. They may not even have a clear view of their own policies, before asked whether they are willing to share data with a specific third party.

This short paper informally introduces and motivates this new problem. It further identifies interesting connections to two distinct areas: data provenance, which captures the way output data are derived from inputs, and Boolean evaluation, which focuses on effective strategies to probe hidden Boolean values for evaluating a formula. As we shall demonstrate, the composition of these two areas in the context of this problem yields intriguing avenues for further research.

1 INTRODUCTION

When peers share data – on social networks, for event planning or in business collaborations – access control is often a concern. Data may be re-shared and used in analysis that combines input from multiple sources, thereby making it difficult to correctly decide access permissions. For example, the data of Alice's recruitment agency may consist of personal data of job seekers, confidential data on companies and internal information on collaborators. Now assume Alice wants to re-share parts of this data with a collaborating agency. In common data sharing platforms, peers that have originally contributed data to Alice either have no control over the re-sharing of their data, or give a broad consent to re-sharing within some group (e.g., for non-commercial purposes), or disallow re-sharing altogether [1]. However, a finer-grained approach may allow re-sharing more data without compromising the preferences of data owners. For instance, Bob, a collaborating agent, may not have agreed that his data is shared with every third party; but if Alice actively asks for his permission to share data with Carol, a specific mutual collaborator, he may agree. Next, Alice may compute statistics based on combining and analyzing the data provided from many sources. May she share the derived data (analysis results) with a third party? Who should be probed for permissions in this case?

The related topics of data sharing, privacy and access control have been extensively studied (see a brief discussion below), but uniquely our focus here is not on data management with known policies but rather on the *active probing of peers* to ask for their consent, to achieve fine-grained access control. The peer answers may either be given manually (by one of Bob's employees), or (semi)-automatically (by Bob's servers). In either case, probing requires resources and reveals parts of Bob's proprietary policy, and therefore should be minimized. As we shall demonstrate, this active setting, coupled with reasoning performed over the data, leads to novel computational problems. Informally, we introduce the following high-level problem:

We are given a database whose tuples have been contributed by multiple peers, and a query (in some language) over the database. Access control policies with respect to the input *tuples are (completely or partially) unknown, but we may* probe *data owners to ask about them. Our goal is to decide whether* the query output *(or a subset thereof) may be shared with a third party, while minimizing the number of probes (or otherwise optimizing with respect to a related target).*

To continue our above example, assume that an agency owned by Carol seeks information on companies in Pennsylvania in which positions were successfully found for graduates of environmental studies. The agency of Alice, a collaborator of Carol, may have such data at hand, and the information needs of Carol may be captured by an SPJU query on this data – but may the query results be shared? For instance, an output value "PennSolar-Experts Inc." may be the result of joining several tuples involving data on (1) the PennSolarExperts company; (2) three environmental studies graduates who found positions in this company, and (3) the agency owned by Bob who collaborated with Alice in finding positions for some of the candidates (and then projecting on the company name). In this case, to share the query result with Carol, Alice may need the consent of the company, at least one of the assigned workers, and Bob. Importantly, Alice may not know in advance whether the company, workers or Bob agree that computation results based on their data are shared with Carol; they could have shared their data with Alice without giving her permissions to share it with a third party. In this case, Alice would need to actively probe the peers, but what is her best "strategy" for probing (i.e. who should she probe and in what order)?

In this short paper we introduce and motivate this new problem in a high-level manner, through an example. We further outline a promising approach for a solution, based on two seemingly disjoint areas of research: Data Provenance and Boolean Evaluation. We believe that the problem, and our line of attack, are worthy of in-depth investigation.

Related Work. We conclude this Introduction with an overview of related work. The theory and practice of Access control have been extensively studied in different contexts, including social

Companies

cid	name
11	PennSolarExperts Ltd.

Vacancies

vid	cid	position	amount
111	11	analyst	3
112	11	supervisor	1

JobSeekers

sid	name	education	agency
1	David	Env. studies	Bob
2	Ellen	Env. studies	Bob
3	Frank	Env. studies	Alice
4	Georgia	Env. studies	Bob

Assignment

sid	vid	status	agency
1	111	hired	Bob
2	112	rejected	Alice
2	111	hired	Bob
3	111	rejected	Alice
4	112	hired	Alice

Table 1: Example database of Alice's recruitment agency.

networks (e.g., [1–7]), distributed systems (e.g., [8, 9]) cloud services (e.g., [10–12]), Web applications (e.g., [13, 14]) , databases (e.g., [15–17]), and many other areas. With respect to these works, our novelty is in focusing on a setting where access policies may be unknown or undetermined in advance, which requires active probing of involved peers to obtain permissions. Fine-grained access control policies may be too large or complex to be specified by a client, if, e.g., the permission for every peer and action must be specified. To assist clients in specifying policies, previous work has considered (semi-)automatic computation of access control policies. This includes the computation of policies based on example permissions [18]; evaluating the credibility of peers [5]; mining or interactively defining user roles [19, 20]; using semantically-rich languages to compactly capture real-life factors on policy definition [21]; and using game-theoretic considerations in defining policies with respect to risk minimization [22]. These works are complementary to ours, in the sense that we probe peers as a "black box": peer answers may be obtained manually or predefined using methods such as above.

2 MODEL, VIA AN EXAMPLE

We next outline a preliminary model for the problem, illustrated informally via an example.

We are given a Relational Database[1] where each tuple is annotated with a label which we refer to as *concept*, taken from a set of concepts C. Each concept "belongs" to a single owner out of a set of peers $\mathcal{P}$. We will refer to such annotated database as a *shared database*.

Example 2.1. Table 1 outlines a simple DB for the recruitment agency example described above, consisting of details of companies and job vacancies in these companies, with the type of position and the number of open positions; job seekers with their name, education and agency to which they have applied; and the assignment of seekers to vacancies, including the status of the assignment and the agency responsible for matching. The annotations here can be set to reflect the row (table+key) and the owner of the data. In this case, we assume for simplicity that company and vacancy data is owned by the company, and job seeker and vacancy data is owned by the relevant agency as the seekers' representatives. The annotation for the first row in Companies could for instance be `Companies11PennSolar` and the first row of Assignment could be `Assignment1-111Bob`.

The premise is that there is a hidden truth value to whether or not we are allowed to share each concept with a specific third party (or publish it in public, etc.); this truth value is known only to the concept owner.

[1]For brevity, we demonstrate the problem and our approach in a relational setting and for tuple-level access control; it applies similarly, with some extensions, to semi-structured data and value-level access control.

Example 2.2. Recall the database in Table 1, and assume that Alice wishes to share the **JobSeekers** table with Carol. In this case, Alice's agency is the owner of the third tuple, and thus can check whether she can share the data - e.g., if Frank agreed in his contract with the agency to share data with third parties. The yes/no answer would translate to a valuation of true/false respectively to the Boolean variable captured by the annotation `JobSeekers3Alice`. The other tuples correspond to job seekers recruited by Bob's agency, hence we assume that when asked by Alice, Bob's agency can answer yes/no to the sharing request, again translated to a Boolean valuation.

Some concepts may be associated with a semantic interpretation, in which case the hidden truth values are constrained. For instance, in the database from Figure 1, we can assume that access permission to a row in **Vacancies** implies access permission to the relevant company's row in **Companies**. To capture such constraints, we use taxonomies.

Next, instead of sharing the data as-is, we consider a *query* executed on the database to perform some analytics and the sharing of its results. We consider "query" as a broad term here, and variants of the problem will focus on different query languages (e.g., relational SPJU, Datalog, etc.).

```
SELECT DISTINCT c.name
FROM   Companies c,
       JobSeekers j,
       Vacancies v,
       Assignments a
WHERE  c.cid = v.cid AND
       v.vid = a.vid AND
       a.status = `hired' AND
       a.sid = s.sid AND
       s.education = `Env. studies'
```

Figure 1: Query over the example database

Example 2.3. Recall our running example and now assume that Alice wishes to share with Carol the names of companies where environmental studies graduates have successfully found jobs. To this end, she runs the query in Figure 1 on the database in Table 1. In this simplified example the answer is the single company in the database - "PennSolarExperts Ltd.", where David, Ellen and Georgia have been hired.

The question is then: given an (annotated) database and a query, are we allowed to share the result?

Example 2.4. Sharing the single result value returned by the example query, "PennSolarExperts Ltd.", requires the company's consent, as the owner of the relevant tuple. Furthermore, sharing the result may reveal information about other tuples participating in the derivation. For instance, if there are few environmental studies graduates who work in PennSolarExperts, sharing the result with Carol would reveal personal information about them, including that at least one of them was recruited by Alice's agency or her collaborators (and in turn, that this person belongs to the type of job applications at which Alice and her collaborators specialize, e.g., interns, part-time positions, etc.). Beyond our example, peers can ask queries, e.g. Boolean, whose result does not contain any tuple cell, and yet may reveal the existence of other tuples. We shall therefore consider to *which* tuples used in the derivation permissions are needed. In our case, intuitively, it is sufficient to have permissions to the data involved in *one* relevant job assignment, since the existence of additional assignments does not change the result. To share e.g., Georgia's assignment details, Alice needs permission to share all the tuples jointly involved in it in addition to the company's tuple – tuple 4 in **JobSeekers**, vacancy 112 and the relevant assignment. To obtain these permissions, Alice needs to probe Bob Bob, the owner of the relevant **JobSeekers** tuple, and PennSolarExperts for the **Companies** and **Vacancies** tuples. The owner of the assignment itself is Alice's agency, which means she has the information of whether this tuple can be shared or not. If only one of the four aforementioned tuples cannot be shared, Georgia's assignment cannot be shared.

Since access control policies with respect to the individual concepts are "hidden", namely known only to owners, the tool that we have for deciding whether or not a result may be shared is to pose questions or *probes* to the owners of relevant data items. The goal is then to optimally select probes in order to discover whether sharing is permitted.

Example 2.5. In our running example, 9 tuples contributed in some way (to be formalized below through the notion of provenance) to the result of the example query. If the only tuple owned by Alice (the assignment of job seeker 4) can be shared, we are left with 8 tuples. If we ask PennSolarExperts whether their company's details can be shared and get a negative answer, we know that the data cannot be shared and there is no need to ask further questions. As another example, recall that we assumed that vacancies data can only be shared if the company's data can be shared. Then assume we get PennSolarSystem's permission to share data about vacancy 112 (and hence also the company details) and Bob's permissions to share Georgia's details – we obtain that the query result can be shared having used only 2 probes.

Naturally, the number of questions that will be asked in practice depends on the answers received, which are unknown in advance. The goal is to design a strategy for choosing which questions to pose and in what order, where multiple variants of the problem could be of interest. These variants may be based on axes such as the query language expressiveness (e.g. Conjunctive Queries, Datalog, etc.); the optimization goal (e.g. minimizing the number of questions or maximizing the number of shareable results for a given "budget" of questions); optimizing for the worst or expected case (with respect to the peer answers); selecting probes in advance or incrementally; and restricting the per-peer probes or optimizing the overall number of probes.

3 TOWARDS A SOLUTION

Having informally introduced the problem, we next outline a preliminary approach for a solution, combining multiple areas of previous work.

Provenance. We are interested in whether or not we may share *derived* data, whereas (hidden) access control policies are defined with respect to the original, *atomic* data items. The propagation of meta-data from atomic data items to the query results related to them has been studied under the prism of *provenance* [15, 16, 23] (and previously, c-tables [24]). In our case, we may use provenance to compute expressions capturing the access control of derived data, in terms of the concepts annotating the input.

Example 3.1. Recall the query in our running example from Figure 1. Using c-tables [24] (or alternatively Boolean provenance [15, 23]), we can compute a Boolean expression reflecting the dependencies of access control credentials to the output on permissions to view relevant input tuples.

$$\begin{aligned}&\mathtt{Companies11PennSolar}\wedge\\&(\mathtt{Vacancies111PennSolar}\wedge((\mathtt{Assignment1\text{-}111Bob}\wedge\mathtt{JobSeekers1Bob})\\&\qquad\qquad\vee(\mathtt{Assignment2\text{-}111Bob}\wedge\mathtt{JobSeekers2Bob}))\\&\vee(\mathtt{Vacancies112PennSolar}\wedge\mathtt{Assignment4\text{-}112Alice}\wedge\mathtt{JobSeekers4Bob}))\end{aligned}$$

The formula uses the access control concepts of the relevant tuples as variables. Indeed, it matches the intuition of Example 2.5 on how probe answers may affect the final decision: if PennSolarExperts refuse sharing their company details with Carol, `Companies11PennSolar` will be evaluated to false, and the truth value of the entire formula will be false. Alternatively, if we know that `Vacancies112PennSolar`, `Companies11PennSolar`, `Assignment4-112Alice` and `JobSeekers4Bob` evaluate to true, the entire expression evaluates to true.

(Boolean) provenance constructions such as the one exemplified above have been developed for different query languages and formalisms, and the shape of the resulting Boolean expression depends on the formalism for which provenance is tracked, which in turn may affect the probe selection process. For instance, if we restrict attention to Union of Conjunctive Queries, then provenance of each output tuple may be represented in Disjunctive Normal Form of polynomial size with respect to the input database size [24]; negation is needed for queries with relational difference [25]; for Datalog, a polynomial size representation is possible in the worst case only if we resort to Boolean circuits [26]; etc.

(Incremental) Boolean Evaluation. Given Boolean provenance formulas over data items, had we known whether they are authorized for publication, we could simply assign true/false to the corresponding variables in the formulas and decide whether the derived data could be shared. However, policies of peers may be unknown or undetermined, therefore we probe peers to obtain them. We now consider the optimization problem of selecting the best variables to observe next.

To illustrate, we next outline our preliminary solution for the following setting:

- Relational SPJUD queries (select-project-join-union-difference),
- Minimizing the number of questions
- Optimizing the expected case
- Selecting questions incrementally
- Considering either the number of questions overall or per peer
- Assuming an equal cost for all the questions/peers and given answer prior probabilities.

In this case, as explained above, output tuples would be annotated by Boolean expressions (computed via [24]). We then explore results on Boolean evaluation for such expression, and may leverage them to show that the problem is NP-hard, via [27, 28]. In contrast, previous work has studied optimal solutions for restricted cases, heuristics and approximations (see, e.g., [27, 29] for a survey). In particular, in [30] we have adapted and extended an approximate solution by [31], as we next briefly outline.

Denote the set of output tuple annotations by $\mathbf{E}$. For each Boolean formula $e_i \in \mathbf{E}$ we define two *utility* functions $g_0^i, g_1^i : \{1, 0, *\}^{|C|} \to \mathbb{R}^+$, where C is the set of variables in the Boolean expressions and each entry represents either a value assignment to a variable or no assignment ($*$). $g_0^i(\vec{c})$ and $g_1^i(\vec{c})$ are respectively the number of terms (conjunctions) set to 0 in the DNF form of e_i, and the number of clauses (disjunctions) set to 1 in the CNF form of e_i by the partial assignment $\vec{c}$. Denote by m_i and l_i the number of terms and clauses in e_i respectively. The utility function $g^i(\vec{c}) = m_i l_i - (m_i - g_0^i(\vec{c}))(l_i - g_1^i(\vec{c}))$ reaches its maximum, $m_i l_i$, when $g_0^i(\vec{c}) = m_i$ or $g_1^i(\vec{c}) = l_i$, i.e., e_i is evaluated.

To minimize the *overall* number of questions a greedy algorithm is then used. The algorithm repeatedly selects the unknown variable c_j whose probe maximizes the expected value (over the possible answers) of g^i. Some properties of g^i (monotonicity, submodularity) guarantee that the expected number of questions is within a factor of $\ln(m_i l_i) + 1$ from the optimum.

Next, we need to simultaneously evaluate all the formulas in **E**, which may be done by defining $g(\vec{c}) = \sum_{e_i \in E} g^i(\vec{c})$. g is a function that reaches its maximum, $\sum_{e_i \in \mathbf{E}} m_i l_i$, when all the expressions are evaluated. By similar analysis to that of g^i, we get that the solution is a $\ln(\sum_{e_i \in \mathbf{E}} m_i l_i) + 1$ approximation of the optimum.

The above adaptation is an example of a rather direct combination of results from the provenance and Boolean evaluation literature; our setting suggest further novel algorithmic developments. For instance, in [30] we have extended the solution to compute batches of probes rather than single probes and to account for constraints imposed by a taxonomy over the items (i.e., implication constraints between access rights), showing that we still achieve the same approximation bound. We also study an incremental setting where the client is allowed to terminate the process at any point and share partial results via "safe views" – views of the results that are known to be safe for sharing.

Another example for a variant that is not accounted for by previous results is one that targets the minimization of the probes *per peer*, in this case, minimizing. For that, we keep track of the number of probes per peer $p \in \mathcal{P}$ (denoted by probes(p)). At each step, let C^* be the set of variables still unknown. When selecting the next probe, we consider only variables in $\{c \in C^* \mid \text{probes}(\text{owns}(c)) = \min_{p' \in \mathcal{P}^*} \text{probes}(p') \land \mathrm{E}[g(\vec{c})] > 0\}$, i.e., that are owned by least-probed peers and whose utility is non-zero.

Our preliminary results are encouraging and suggest that the combination of provenance constructions and Boolean evaluation methods is a promising direction towards studying our problem complexity and designing efficient algorithms. There are still many challenges to be overcome in this space, both theoretical and practical: for instance, can we achieve better guarantees by restricting the query language? What guarantees can we obtain for more expressive languages? For instance, what if we have aggregates, or recursion and then Boolean circuits rather than formulas? What if we have a "budget" for the number of probes to be used? How can we estimate the probe answer probabilities, in light of constraints defined by the taxonomy, peer trust, and accumulated probe answers? The interplay between the choice of query language/provenance model, optimization goal and constraints leads to many intriguing computational questions which will be central to research on this problem.

4 CONCLUSION

This paper has advocated the study of access control management in a setting where peers are actively probed to ask for their permissions. Our main insight is that the problem may be addressed by computing Boolean provenance for query results, and treating the Boolean expression as input to active Boolean evaluation algorithms. We believe that this high-level approach paves the way to exciting research possibilities at the intersection of these two seemingly unrelated areas.

ACKNOWLEDGEMENTS

This work was funded in part by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Ministers Office, and by the Israel Science Foundation (grant No. 1157/16).

REFERENCES

[1] L. Yu, S. M. Motipalli, D. Lee, P. Liu, H. Xu, Q. Liu, J. Tan, and B. Luo, "My friend leaks my privacy: Modeling and analyzing privacy in social networks," in *SACMAT*, 2018.

[2] A. K. Abdulla and S. Bakiras, "HITC: data privacy in online social networks with fine-grained access control," in *SACMAT*, 2019.

[3] Y. Cheng, J. Park, and R. S. Sandhu, "An access control model for online social networks using user-to-user relationships," *IEEE Trans. Dependable Sec. Comput.*, vol. 13, no. 4, 2016.

[4] M. Cramer, J. Pang, and Y. Zhang, "A logical approach to restricting access in online social networks," in *SACMAT*, 2015.

[5] E. Gudes and N. Voloch, "An information-flow control model for online social networks based on user-attribute credibility and connection-strength factors," in *CSCML*, 2018.

[6] P. Ilia, B. Carminati, E. Ferrari, P. Fragopoulou, and S. Ioannidis, "SAMPAC: socially-aware collaborative multi-party access control," in *CODASPY*, 2017.

[7] N. C. Rathore, P. Shaw, and S. Tripathy, "Collaborative access control mechanism for online social networks," in *ICDCIT*, 2016, pp. 142–147.

[8] S. Abiteboul, P. Bourhis, and V. Vianu, "A formal study of collaborative access control in distributed datalog," in *ICDT*, 2016.

[9] V. Z. Moffitt, J. Stoyanovich, S. Abiteboul, and G. Miklau, "Collaborative access control in webdamlog," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, 2015.

[10] A. Bates, B. Mood, M. Valafar, and K. R. B. Butler, "Towards secure provenance-based access control in cloud environments," in *(CODASPY)*, 2013.

[11] P. Derbeko, S. Dolev, E. Gudes, and S. Sharma, "Security and privacy aspects in mapreduce on clouds: A survey," *Computer Science Review*, vol. 20, pp. 1–28, 2016. [Online]. Available: https://doi.org/10.1016/j.cosrev.2016.05.001

[12] M. G. Solomon, V. S. Sunderam, and L. Xiong, "Towards secure cloud database with fine-grained access control," in *DBSec*, 2014.

[13] H. Qunoo and M. Ryan, "Modelling dynamic access control policies for web-based collaborative systems," in *DBSec*, 2010.

[14] G. S. Tuncay, S. Demetriou, and C. A. Gunter, "Draco: A system for uniform and fine-grained access control for web code on android," in *CCS*, 2016.

[15] J. N. Foster, T. J. Green, and V. Tannen, "Annotated XML: queries and provenance," in *PODS*, 2008.

[16] G. Karvounarakis, Z. G. Ives, and V. Tannen, "Querying data provenance," in *SIGMOD*, 2010.

[17] A. Roichman and E. Gudes, "Fine-grained access control to web databases," in *SACMAT*, 2007.

[18] G. P. Cheek and M. Shehab, "Policy-by-example for online social networks," in *SACMAT*, 2012.

[19] N. Gal-Oz, Y. Gonen, and E. Gudes, "Mining meaningful and rare roles from web application usage patterns," *Computers & Security*, vol. 82, 2019.

[20] T. Wang, M. Srivatsa, and L. Liu, "Fine-grained access control of personal data," in *SACMAT*, 2012.

[21] A. Masoumzadeh and J. B. D. Joshi, "OSNAC: an ontology-based access control model for social networking systems," in *PASSAT*, 2010.

[22] H. Hu, G. Ahn, Z. Zhao, and D. Yang, "Game theoretic analysis of multiparty access control in online social networks," in *SACMAT*, 2014.

[23] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *PODS*, 2007.

[24] T. Imielinski and W. L. Jr., "Incomplete information in relational databases," *J. ACM*, vol. 31, no. 4, 1984.

[25] Y. Amsterdamer, D. Deutch, and V. Tannen, "On the limitations of provenance for queries with difference," in *TaPP'11*, 2011.

[26] D. Deutch, T. Milo, S. Roy, and V. Tannen, "Circuits for datalog provenance," in *ICDT*, 2014.

[27] S. R. Allen, L. Hellerstein, D. Kletenik, and T. Ünlüyurt, "Evaluation of monotone DNF formulas," *Algorithmica*, vol. 77, no. 3, 2017.

[28] L. A. Cox, Q. Yuping, and W. Kuehner, "Heuristic least-cost computation of discrete classification functions with uncertain argument values," *Annals of Operations research*, vol. 21, no. 1, 1989.

[29] T. Ünlüyurt, "Sequential testing of complex systems: a review," *Discrete Applied Mathematics*, vol. 142, no. 1-3, 2004.

[30] Y. Amsterdamer and O. Drein, "PePPer: Fine-grained personal access control via peer probing (demo)," in *ICDE*, 2019.

[31] A. Deshpande, L. Hellerstein, and D. Kletenik, "Approximation algorithms for stochastic Boolean function evaluation and stochastic submodular set cover," in *SODA*, 2014.

Short paper

The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries

Angjela Davitkova
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
davitkova@cs.uni-kl.de

Evica Milchevski
Commerce Connector GmbH
Stuttgart, Germany
evica@commerce-connector.com

Sebastian Michel
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
michel@cs.uni-kl.de

ABSTRACT

We present the ML-Index, a memory-efficient Multidimensional Learned (ML) structure for processing point, KNN and range queries. Using data-dependent reference points, the ML-Index partitions the data and transforms it into one-dimensional values relative to the distance to their closest reference point. Once scaled, the ML-Index utilizes a learned model to efficiently approximate the order of the scaled values. We propose a novel offset scaling method, which provides a function which is more easily learnable compared to the existing scaling method of the iDistance approach. We validate the feasibility and show the supremacy of our approach through a thorough experimental performance comparison using two real-world data sets.

1 INTRODUCTION

Processing queries on multidimensional data is a classical and thoroughly investigated problem. A plethora of index structures provides mechanisms for compactly storing and querying multidimensional datasets, applied in countless application scenarios. A recent idea proposed by Kraska et al. [4], suggests improvement and replacement of traditional index structures with machine learning and deep-learning models. They in particular successfully replace the B-Tree with a recursive learned model that maps a key to an estimated position of a record within a sorted array. With the use of the proposed learned models, they are able to utilize the patterns in the data distribution, resulting in an improvement of both the memory consumption and the execution time over the traditional index.

To provide a generalization of the B-Tree for multiple dimensions, the data needs to be sorted in an order which can be easily learned by supervised learning models. The ordering needs to be done in such a manner that the correctness guarantees are fulfilled when answering range and KNN queries. In practice, techniques, such as the Morton and the Peano-Hilbert order, can be exploited for sorting multidimensional data. However, directly mapping the multidimensional data points within the aforementioned orders cannot be easily learned by deep learning models. Kraska et al. [3] propose an approach for learning an order based on successively sorting and partitioning points along several dimensions into equally-sized partitions. However, choosing only a subset of dimensions may lead to performance degradation when the number of dimensions increases and the deduction of the partition neighbors may not be a time-efficient task.

Therefore, we create a novel Multidimensional Learned (ML) index which generalizes the idea of the famous iDistance scaling method [2] and uses the scaled ordering in combination with a two-layer learned index, to answer multidimensional queries. Unlike existing indexes, it captures the data distribution in two manners, by efficiently partitioning and scaling the data with respect to distribution-aware reference points and by learning the distribution of the sorted scaled values. Harnessing the power of the deep-learning models, the ML-Index is the first complete learned index, able to answer point, range and KNN queries efficiently while having a low memory consumption.

2 RELATED WORK

Until recently, limited research was present in improving data indices by interweaving them with machine learning. One of the first distribution aware indices [1] focuses on the combination of R-Trees with a self-organizing map. Another exact KNN algorithm [9] employs k-means clustering and triangle inequality pruning for efficient query execution.

Recently, Kraska et al. [4] draw the focus on a novel idea of substituting indices with deep learning models. Lead by the assumption that each index is a model, they draw a parallel between the indices and their respective analogue in the machine learning world. For instance, B-Tree Index and Hash-Index can be seen as models that map a key to a position within a sorted and unsorted array accordingly and can be easily replaced with neural network models. A learned database system called SageDB [3], extends the concepts to multidimensional data, by successively partitioning points along a sequence of dimensions into equal-sized cells and ordering them by the cell that they occupy. Although the order produces a layout noted as learnable, the complexity of directly learning the projection of n-dimensional points to an order position increases with the increase of dimensionality. Even though limiting the number of dimensions for partitioning avoids the added complexity, it results in slower execution of range queries including the missing dimensions.

Providing an inexpensive representation of multidimensional points which can be meaningfully sorted has been already widely explored, such as presorting the data by their Z-order [6], Hilbert order [5], or the respective distance to reference points [2]. A learned Z-order Model [8] focuses on combining the Z-order scaling with a staged learned model, for efficiently answering spatial queries. Although applicable for smaller dimensions, both the Z-order model and the UB-Tree are limited when dealing with a larger number of dimensions, which will be analyzed upon the experiments and the direct comparison.

3 THE ML-INDEX

The ML-Index is a compound of two main components, as illustrated in Figure 1. Its creation is carried out in two stages, guided and generalized by the idea of previously existent iDistance index [2]. The upper part consists of a set of reference points, responsible for scaling the multidimensional data to one-dimensional values, which can be easily sorted. The lower part is a Learned Model used for learning the distribution of the scaled values and a sorted array used for searching and storing the data.

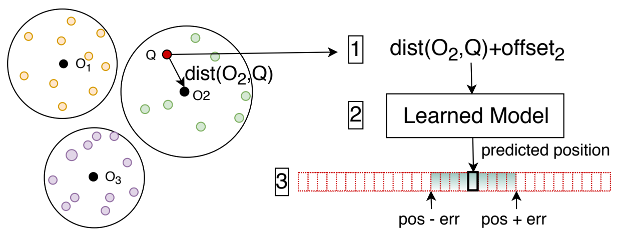


Figure 1: The ML-Index Approach

3.1 Scaling Methods

Consider a set of data points $d_l \in D$, where $d_l = (d_0, d_1, ..., d_n)$ is in an n dimensional metric space. The first stage, responsible for creating the upper part, maps the data points from a n into a one-dimensional data space. The scaling method aims to group points that are similar to each other and project them in one-dimension, such that the similarity is preserved by the proximity of the one-dimensional values. To efficiently do the scaling, m reference points O_i are chosen, each representing an identification or a centroid of the data in partition P_i. The partition P_i is formed from the points whose closest reference point is O_i. This means that the minimal distance of a point d_l to the reference points determines the appropriate partition.

Irrespective of the way to find reference points, we elaborate on the usage of different scaling methods and we discuss their advantages and disadvantages. The straightforward case is scaling d_l with respect to a single reference point, by mapping it to a one-dimensional value $key = dist(O_i, d_l)$. Although applicable for smaller datasets, the method creates a considerably large amount of false positives, produced by mapping multidimensional points that are far from each other to the same value.

The second scaling method is the iDistance method described by Jagadish et al. [2]. This method maps a data point d_l into a one-dimensional value key, based on $key = i * c + dist(O_i, d_l)$, where i is the index of the closest reference point O_i. The constant c serves to partition the points into predefined ranges and based on its value it stretches the ranges differently. Using the constant, the points belonging to a partition P_i will be mapped to a range $[i * c, (i+1) * c]$. Although it provides well-scaled values and drastically reduces the number of false positives, it is highly dependent on the parameter c. A smaller value for c may create an overlap between the partitions, causing multidimensional points from different partitions to be mapped to the same scaled value. Hence, upon search, the number of unnecessarily examined points will be increased. On the other hand, having a larger c directly affects the creation of the second stage of the ML-Index which we thoroughly discuss in Subsection 3.2.

Finding the right value for the constant c, which will cause no overlap and have perfectly ordered partitions, with no gaps between them, is impossible. This comes as no surprise since reference points correspond to partitions of different sizes. Prompted by the previous, we propose a novel scaling approach that provides a perfect ordering between the ranges of different partitions and overcomes the problem of overlapping. The new method termed *offset method*, given a point d_l and its closest reference point O_i, calculates the scaling value as $key = offset_i + dist(O_i, d_l)$, where $offset_i$ is different for every partition P_i. Given an arbitrary ordering of the reference points $O_1, O_2, ..., O_m$ and their adequate partitions $P_1, P_2, ..., P_m$, the offset is calculated as the sum of the radii of their previous partitions:

$$offset_i = \sum_{j<i} r(j)$$

where r is the maximal distance from O_j to the points in partition P_j. This method assures no overlap and it reduces the gap between the partitions, which is crucial regarding the performance of the second stage of the ML-Index. Additionally, it omits the problem of tuning the parameter c, for a suitable range creation. The downside in comparison to iDistance is an unnoticeable memory increase, caused by storing the offsets. Following the scaling of the original data, each data point d_l is associated with a value key. Multiple points can have the same scaled value. Before the execution of the second stage of the ML-Index, we sort the points according to the value key. The resulting order is used as a starting point for creating a learned model, which efficiently predicts the position of a given key within a sorted array.

3.2 Learning the Order

The second stage of the ML-Index represents a *Recursive Learned Index*, similar to the one described by Kraska et al. [4]. The index, mimics the behavior of a traditional B-Tree, by mapping a given lookup key within a sorted array with the guarantee that the key is within proximity of the predicted position $[pos - err, pos + err]$. As observed [4], a model, which performs this task, effectively approximates the cumulative distribution function (CDF), modelled as $p = F(key) * N$, where N is the number of keys and $F(key)$ is the estimated CDF that estimates the likelihood to predict a key smaller or equal than the lookup key. The learned index is built in a top-down manner where a model in each stage provides a prediction used to choose the model in the next stage, or the position of the key when the final stage is reached. For model $f(x)$ from models M_l at stage l, with input x, the loss is calculated:

$$L_l = \sum_{x,y} (f_l^{\lfloor M_l f_{l-1}(x)/N \rfloor}(x) - y)^2 \quad [4]$$

Although Kraska et al. [4] suggest using multiple stages of the learned index, we use only two. The incentive behind this decision is that if the second stage index produces a larger than expected error, the increase in the number of models in the second stage is sufficient to reduce the error. Additionally, unlike their proposed learned index, the second stage of learned models in the ML-Index is constructed solely by using a regression, with the purpose of balancing the construction time and the performance of the learned index. By using a simpler model, the number of multiplications and additions upon search is reduced, leading to lower search time. The final prediction of the learned model is a predicted position within the sorted array where the key, in our case the scaled value is stored. Because a position is predicted with a certain error, one must also search within the error bounds around the prediction for the correct position.

As mentioned, the different scaling methods impact the second stage of the ML-Index. Since the naive scaling method is infeasible, we elaborate only on the impact of the iDistance and the offset scaling method. Both methods result in different functions that need to be estimated by the learned index. When considering the iDistance we distinguish two scenarios, one with overlap between the partitions and one without. When c is smaller than the maximum radius of all the partitions, then an overlap between the ranges of the partitions is inevitable. However, sorting the data with a smaller c creates a function which can be easily learned. This is a result of the overlapping that leads to a larger density of the scaled values which results in a fairly continuous function. The second case is having a larger c, that avoids overlaps, but creates large gaps within the function. Therefore, the neural network will learn the data present in the different ranges. However, upon search time it is possible to search for a point that is not present within these ranges, but it is located within the gaps of the function. Vast research has been proposed for filling the missing values in a function to be learned, however, this leads to a pre-processing step which can be easily avoided by using the offset method. The offset method circumvents the gaps created by the different maximal sizes of the partitions and provides a better "learnable" function in which the missing values may only appear due to sparsity in the clusters and not the scaling method.

4 QUERY PROCESSING

Point Query: The point query identifies the existence of a multidimensional point within the index and it is executed in three steps as shown in Figure 1. The first step includes searching for the closest reference point O_i to the query q and calculating the scaled value $key = offset_i + dist(ClosestO_i, q)$, where $offset_i$ is calculated as $i * c$ for the iDistance scaling and appropriately for the offset method. Once calculated the key is used to predict the position of the point q, within the sorted array of points. The predicted value is the position pos of the key within the sorted array, and it is used in its exponential search with bounds $[pos - err, pos + err]$. The complexity of the first step, for m reference points and d dimensions, is $O(m * d)$. The learned model complexity depends on the architecture of the neural network. A neural network with a single hidden layer with width h and input size N will have $O(hN)$ multiplications and additions.

KNN Query: Given a query q and a parameter k, the KNN query finds the closest k points S_k, to the query such that $\forall d_i \in S_k,\ \forall d_j \in D \setminus S_k,\ dist(q, d_j) \geq dist(q, d_i)$. Since the iDistance was initially created for executing KNN queries, we adapt the algorithm for the ML-Index. The algorithm creates several one-dimensional range queries, whose selectivity expands until the result is complete. The major modification is locating the start of the range. For this purpose, the Algorithm 1 is used, that exploits the learned model to predict the position of a given key within a sorted array. However, since a key which is not present within the array can be provided, we need to search for the key with the smallest difference to the initial key. Nonetheless, searching for the closest key to the initial key is not only dependent on the bounds provided by the error of the learned model but also the bounds of the ranges occupied by the reference points. Therefore, we modify the binary search to also include the $offset_i$ and $offset_{i+1}$, which further reduces the search space. The method *closest* returns the position of the value closest to the key in the range $[offset_i, offset_{i+1}]$. To describe the need for the second bounds, we consider having two reference points and their respective ranges within the array $[1, 5][6, 10]$. Let's further take the assumption that the first range has the keys $[1, 3, 5]$ and the second $[7.5, 8, 10]$. Upon search, we want to find the closest key to $key = 6$ for the reference point O_2. If we do not consider that the reference point O_2 has a lower bound 6 we would retrieve the key 5 as closest which does not belong to the region 2 and thus, in reality, may not be even close to the query point.

Range Query: Widely applicable for smaller dimensions, the range query $q = q_1, q_2, ..., q_n$, where $q_j = [bound_{min}, bound_{max}]$, defines bounds of a dimension j, retrives the data points $d_i \in D$

Algorithm 1 Predict Closest Position

```
Require: scaled value $key$ (if outside of range, set to $offs_i$ or $offs_{i+1}$),
    range for $O_i$ $[offs_i, offs_{i+1}]$
 1: $mid = predict(key),\ s = mid - err,\ t = mid + err$
 2: while $s \leq t$ do
 3:     $mid = \lceil (s + t)/2 \rceil$
 4:     if $data_{mid} == key$ then
 5:         return $mid$
 6:     $withinRange = True$
 7:     if $data_{mid}$ outside $[offs_i, offs_{i+1}]$ then
 8:         $withinRange = False$, set $s$ or $t$ to $mid$
 9:     if $withinRange$ or $|s - t| \leq 1$ then
10:         if $key < data_{mid}$ then
11:             if $key \geq data_{mid-1}$ then
12:                 return $closest(data, key, mid - 1, mid, offs_i)$
13:             $t = mid - 1$
14:         else
15:             if $key \leq data_{mid+1}$ then
16:                 return $closest(data, key, mid, mid+1, offs_{i+1})$
17:             $s = mid + 1$
```

where $\forall j \in n,\ d_{ij} \geq q_{j0}$ and $d_{ij} \leq q_{j1}$. For the execution of the range query within the ML-Index, we adapt the Data-Based Method for range approximating suggested by Schuh et al. [7]. The algorithm iterates over the reference points and for each reference point calculates the closest and the furthest point from the given range q. For the computed furthest and closest point, the algorithm issues a range query of the form $[dist(O_i, point_{closest}) + offset_i,\ dist(O_i, point_{furthest}) + offset_i]$. Since the closest and furthest points can also have keys which are not present within the sorted array, the method described in Algorithm 1 is used.

5 EXPERIMENTS

For evaluation, all indices are in main memory and implemented in Java. The learned model is implemented with TensorFlow and it is extracted to omit the overhead. Its width is set according to the dataset. The reference points selection is done using the KMeans algorithm, due to better results for the real-world data. GMeans was not considered, since the branching factor of the M-Tree is set to k, for a fair comparison. We compare the following structures: the ML-Index, the iDistance index [2] with keys computed by our offset method, the M-Tree, and the index based on learning a Z-order, ZM-Index [8]. Two real-world datasets were used, Color Histogram (dim: 32, points: 68040, size: 19.5 MB) and Forest Cover Type (dim: 10, points: 581012, size: 68.7 MB).

Scaling Methods Comparison: Figure 2a shows the absolute error of learning the order produced by the different scaling methods, by varying the width of a single layer neural network. For the iDistance, c is once set to produce 10% and once 0% range overlap between the different reference points. The data is directly learned in several epochs, without preprocessing. The error produced by the offset scaling method is much lower from the error by the iDistance method when no overlap occurs. Differently, due to the density of the values, the approach with an overlap performs slightly better. However, an overlap between the ranges will result in slower query execution, therefore, the offset method performs the best when considering both aspects.

Memory: Figure 2b (Note the log scale 2) shows the memory consumption of the M-Tree, the iDistance and the ML-Index for both datasets. As observed, the ML-Index has a drastic reduction in memory and has only small storage required for the learned model and the offset distances. Increasing the number of clusters

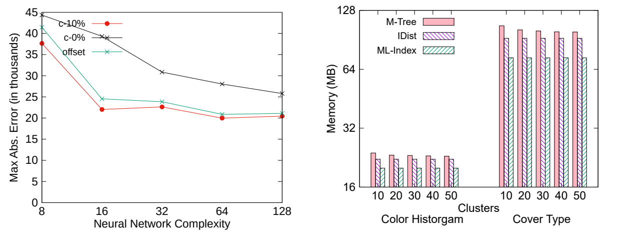


Figure 2: Scaling methods comparison (left) and memory consumption when varying datasets and clusters (right)

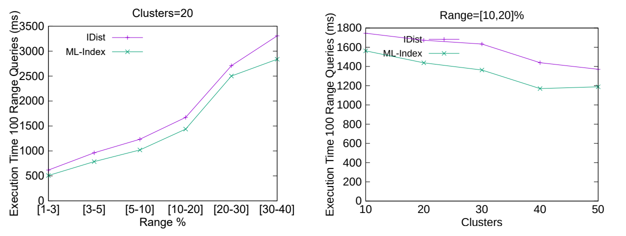


Figure 3: Range search when varying range selectivity (left) and clusters (right) for Forest Cover Type dataset

results in an unnoticeably small increase in memory due to the number of offset values.

Query Execution: Since the M-Tree is drastically slower, we show only the comparison between iDistance and ML-Index. Figure 3 shows the execution time of 100 range queries by varying the query selectivity, representing the fraction of points within the query range from the total number of points, and the number of clusters used for choosing the reference points. Figure 4 shows the execution time of 1000 KNN queries by varying k and the number of clusters. The ML-Index is always faster than the iDistance, which is a result of the superiority of the search with the learned model over the search with B+Tree. This is especially visible for larger ranges and k values where the result may belong to more clusters, requiring multiple predictions from the learned model, causing a more visible difference in the execution time. Upon an increase of clusters, the difference between the search time of the ML-Index and iDistance is smaller. Assuming we search for the closest point, we need to first find the closest cluster among m clusters, which is performed the same for the iDistance and ML-Index. Hence, when m is large, the execution will be impacted more by the cluster iteration than the prediction.

Comparison with learned ZM-Index: For comparison with ZM-Index, we extract 2, 4, and 6 dimensions from the Forest Cover dataset, we scale the values to reduce the number of bits and the large gaps between successive Z-values, which produce a slower execution time. We always use our learned model, since having the number of neurons mentioned in [8], results in incomparably large execution time. ZM-Index outperforms ML-Index when searching for a two-dimensional point, as seen in Figure 5a. This is intuitive since a search through multiple clusters requires more time than a bit shifting operation, which is not the case for larger dimensions and a smaller number of clusters. When considering the range query comparison in Figure 5b, the ZM-Index performs far worse. Upon searching for a next Z-value which is within the range, we exchange every bit accordingly, resulting in a longer

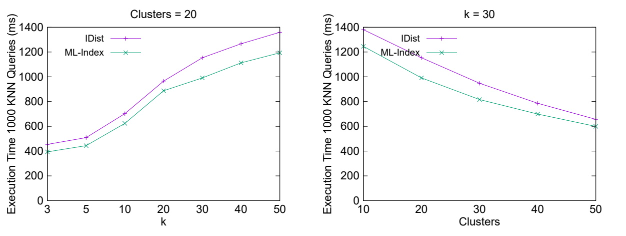


Figure 4: KNN search when varying k (left) and clusters (right) for Forest Cover Type dataset

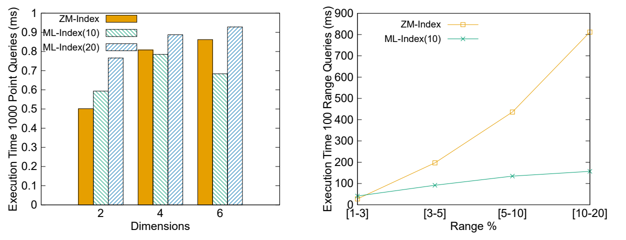


Figure 5: Point search (left) and range search (right) comparison with ZM-Index for Forest Cover Type dataset

execution when dealing with both large numbers and dimensions. More importantly, in each step, the next Z-value within range is calculated, which may correspond to a value not present within the dataset and thus it will lead to unnecessary access.

6 CONCLUSION

We addressed the problem of replacing multidimensional indices with a learned, distribution-aware ML-Index. The ML-Index entails two core tasks: representing the dataset by one-dimensional values based on reference points chosen with respect to the data distribution, and a learned model capable of accurately learning the order of the values. Experimental results demonstrated the feasibility of the approach and its superior performance compared to state-of-the-art competitors. Future work includes rendering the index resilient to updates by observing degenerated performance and triggering retraining only when necessary.

REFERENCES

[1] G. Phanendra Babu. 1997. Self-organizing neural networks for spatial data. *Pattern Recognition Letters* 18, 2 (1997), 133–142.
[2] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. 2005. iDistance: An adaptive B$^+$-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.* 30, 2 (2005), 364–397.
[3] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *CIDR*. www.cidrdb.org.
[4] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2017. The Case for Learned Index Structures. *CoRR* abs/1712.01208 (2017).
[5] Jonathan K. Lawder and Peter J. H. King. 2001. Querying Multi-dimensional Data Indexed Using the Hilbert Space-filling Curve. *SIGMOD Record* 30, 1 (2001), 19–24.
[6] Volker Markl. 1999. MISTRAL: Processing Relational Queries using a Multidimensional Access Technique. In *Ausgezeichnete Informatikdissertationen*. Teubner, 158–168.
[7] Michael A. Schuh, Tim Wylie, Chang Liu, and Rafal A. Angryk. 2014. Approximating High-Dimensional Range Queries with kNN Indexing Techniques. In *COCOON (Lecture Notes in Computer Science)*, Vol. 8591. Springer, 369–380.
[8] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *MDM*. IEEE, 569–574.
[9] Xueyi Wang. 2011. A fast exact k-nearest neighbors algorithm for high dimensional search using k-means clustering and triangle inequality. In *IJCNN*. IEEE, 1293–1299.

RETRO: Relation Retrofitting For In-Database Machine Learning on Textual Data

Michael Günther
Database Systems Group,
Technische Universität Dresden
Dresden, Germany
Michael.Guenther@tu-dresden.de

Maik Thiele
Database Systems Group,
Technische Universität Dresden
Dresden, Germany
Maik.Thiele@tu-dresden.de

Wolfgang Lehner
Database Systems Group,
Technische Universität Dresden
Dresden, Germany
Wolfgang.Lehner@tu-dresden.de

ABSTRACT

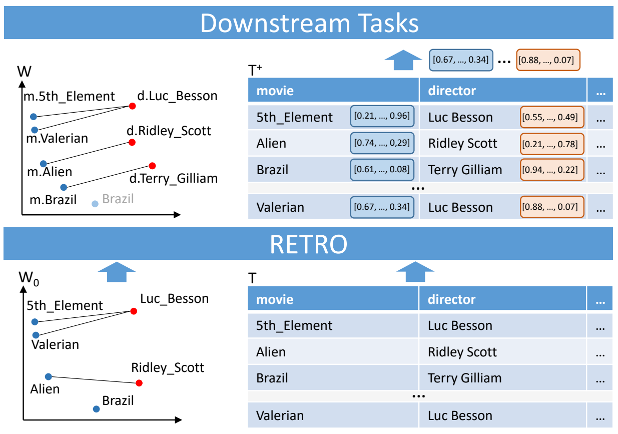
There are massive amounts of textual data residing in databases, valuable for many machine learning (ML) tasks. Since ML techniques depend on numerical input representations, word embeddings are increasingly utilized to convert symbolic representations such as text into meaningful numbers. However, a naïve one-to-one mapping of each word in a database to a word embedding vector is not sufficient since it would miss to incorporate rich context information given by the database schema, e.g. which words appear in the same column or are related to each other. Additionally, many text values in databases are very specific and would not have any counterpart within the word embedding. In this paper, we therefore, propose RETRO (RElational reTROfitting), a novel approach to learn numerical representations of text values in databases, capturing the information encoded by general-purpose word embeddings and the database-specific information encoded by the tabular relations. We formulate *relation retrofitting* as a learning problem and present an efficient algorithm solving it. We investigate the impact of various hyperparameters on the learning problem. Our evaluation shows that embedding generated for database text values using RETRO are ready-to-use for many ML tasks and even outperform state-of-the-art techniques.

1 INTRODUCTION

Due too their appealing properties, word embeddings techniques such as Word2Vec [7], GloVe [8] or fastText [3] have become conventional wisdom in many research fields such as machine learning, NLP or information retrieval. Typically, these embeddings are used to convert text values in a meaningful numerical representations presenting the input for ML tasks. However, a naïve application of a word embedding model is not sufficient to represent the meaning of text values in a database which is often more specific than the general semantic encoded in the raw text embedding.

Thus, we argue to incorporate information given by the disposition of the text values in the database schema into the embedding, e.g. which words appear in the same column or are related. Therefore, we develop a relational retrofitting approach called RETRO which is able to automatically derive high-quality numerical representations of textual data residing in databases without any manual effort.

Relational Retrofitting. Figure 1 provides a small example sketching the main principles of the relational retrofitting process. RETRO expects a database and a word embedding as input, e.g. a movie table T that should be retrofitted into a pre-trained



T^+

movie		director		...
5th_Element	[0.21, ..., 0.96]	Luc Besson	[0.55, ..., 0.49]	...
Alien	[0.74, ..., 0,29]	Ridley Scott	[0.21, ..., 0.78]	...
Brazil	[0.61, ..., 0.08]	Terry Gilliam	[0.94, ..., 0.22]	...
...				
Valerian	[0.67, ..., 0.34]	Luc Besson	[0.88, ..., 0.07]	...

T

movie	director	...
5th_Element	Luc Besson	...
Alien	Ridley Scott	...
Brazil	Terry Gilliam	...
...		
Valerian	Luc Besson	...

Figure 1: Relational Retrofitting: basis word embedding W^0 and relation T (left), retrofitted word embedding W and augmented relation T^+

word embedding W_0. To provide vector representations for textual information in databases one could simply reuse the vectors of pre-trained embeddings, e.g. map each term from T to a term-vector pair in W_0. However, often there will be a semantic mismatch between word embeddings and textual information in databases:
1) Given the movie table, it is known that all entities within the movie column must be movies, although some of the titles, such as “Brazil" or “Alien", may be interpreted differently by the word embedding model.
2) T provides a specific amount of relation types like the movie-director, whereas in the word embedding representation W_0 large amounts of implicit relations are modeled, e.g. if the director of a movie is also the producer or one of the actors this might be represented in the word embedding although not explicitly visible.
3) Terms in T which occurring infrequent in the general domain can not be modeled accurately by word embedding models. For instance Word2Vec has a limited vocabulary according to a frequency threshold. Many terms appearing in a database will therefore have no counter-part within the embedding.
We present RETRO[1], a novel relational retrofitting approach addressing all these challenges. RETRO augments all terms in database tables by dense vector representations encoding the semantics given by the relation T and the word embedding W_0. In the context of our movie example, RETRO would generate a new embedding for “Terry Gilliam" which should be close to other directors and their respective vectors. Furthermore, RETRO would create vectors for all other textual values in the movie table that encode the semantic given of the pre-trained word embeddings and the database. On the one hand, this ensures that vectors

[1] https://github.com/guenthermi/postgres-retrofit

appearing in the same column, such as movies or directors, are close to each other. On the other hand, this ensures that the difference vectors between movie-director pairs are similar. These vectors are ready-to-use for a wide range of ML, retrieval and data cleaning tasks such as classification, regression, null value imputation, entity resolution and many more.

Outline. In Section 2, we give an overview of the problem and a briefly introduce the original retrofitting problem. We then present our novel *relation retrofitting* and formulate the underlying learning problem in Section 3. In Section 4, we show the feasibility of Retro in automatically creating vector representations by defining different classification task and conclude in Section 5.

2 RETROFITTING AND PROBLEM SCOPE

We aim at leveraging powerful word embedding models to generate good vector representations for text values residing within relational databases. We therefore extend the notion of retrofitting which was initially proposed by Faruqui et al. [5]. Retrofitting is performed as a post-processing step and allows to inject additional information into word embeddings. The approach of Faruqui et al. took a matrix $W^0 = (\boldsymbol{v_1'}, \dots, \boldsymbol{v_n'})$ of word embeddings and a graph $G = (Q, E_F)$ representing a lexicon as input. The retrofitting problem was formulated as a dual objective optimization function: The embeddings of the matrix W^0 are adapted to $W = (\boldsymbol{v_1}, \dots, \boldsymbol{v_n})$ by placing similar words connected in the graph G closely together, while at the same time the neighborhood of the words from the original matrix W^0 should be preserved. Hereby, $Q = \{q_1, \dots, q_n\}$ is a set of nodes where each node q_i corresponds to a word vector $\boldsymbol{v_i} \in W$ and $E_F \subset \{(i,j)|i,j \in \{1,\dots,n\}\}$ is a set of edges. The graph is undirected, thus $(i,j) \in E_F \Leftrightarrow (j,i) \in E_F$. The authors specified the retrofitting problem as a minimization problem of the following loss function:

$$\Psi_F(W) = \sum_{i=1}^{n} \left[\alpha_i ||\boldsymbol{v_i} - \boldsymbol{v_i'}||^2 + \sum_{j:(i,j)\in E_F} \beta_i ||\boldsymbol{v_i} - \boldsymbol{v_j}||^2 \right] \quad (1)$$

The constants α_i and β_i are hyperparameters. $\Psi_F(W)$ is convex for positive values of α_i and β_i. Thus, the optimization problem can be solved by an algorithm, which iterates over every node in Q and updates the respective vector in W.

However, while retrofitting is typically used to improve the vector quality of general-purpose word embeddings by using lexical knowledge graphs, we aim at learning vector representations for text entries in database tables. Here the objective is to 1) reflect the semantics of the text value specifically referred to in the database and 2) to fit into the vector space of the given basis word embedding model.

3 RELATIONAL RETROFITTING

In this paper, we extend idea proposed in [5] and formulate the *relational retrofitting* approach that learns a matrix of vector representations $W = (\boldsymbol{v_1}, \dots \boldsymbol{v_n})$ corresponding to text values $T = (\boldsymbol{t_1}, \dots \boldsymbol{t_n})$ where each $\boldsymbol{v_i} \in \mathbb{R}^D$ represents a unique text value in a specific column of the database. To find an initial vector representation for every text value, we tokenize the text values based on the vocabulary of the basis word embedding model and build centroid vectors which is a convenient way to obtain a representation of text values consisting of multiple tokens [1, 11]. These vectors are stored in a matrix $W^0 = (\boldsymbol{v_1'}, \dots \boldsymbol{v_n'})$ forming the basis for the retrofitting process. Besides, *columnar and relational connections* are extracted from the database (see Section 3.1). This encompasses semantic relations between text values, which are derived from the relational schema. Those connections are used to create a representation capturing the context of the text value in the database (e.g. “Brazil” in the column “movie.title” is considered as a movie) and thus helps to preserve their semantics more accurately compared to a plain word embedding representation. The core procedure of the relational retrofitting is the adaption of the basis vectors W^0. This is performed by solving an optimization problem detailed further in Section 3.2.

3.1 Extracting Relational Information

One can derive different structural relations from the alignment of text values in the relational schema.

Columnar Connections: Text values with the same attribute, i.e. appearing in the same column, usually form hyponyms of a common hypernym (similar to subclass superclass relations). Thus, they share a lot of common properties which typically leads to similarity. We capture this information and assign each text value t_i to its column $C(i)$.

Relational Connections: Relations exhibit from the co-occurrence of text values in the same row as well as from foreign key relations. Those relations are important to characterize the semantics of text value in the database. We define a set of relation types R for each specific pair of related text value columns. Those columns are related because they are either part of the same table or there exists a foreign key relationship between their tables. For every relation type $r \in R$ there is a set E_r containing the tuples of related text value ids. Relation types are directed. Accordingly, there is an inverted counterpart $\bar{r}$ for each relation r with $E_{\bar{r}} = \{(j,i)|(i,j) \in E_r\}$.

3.2 Optimization Problem

Retro considers relational and columnar connections (see Section 3.1) to retrofit an initial embedding. Accordingly, we define a loss function Ψ adapting embeddings to be similar to their basis word embedding representation W^0, the embeddings appearing in the same column, and related embeddings.

$$\Psi(W) = \sum_{i=1}^{n} \left[\alpha_i ||\boldsymbol{v_i} - \boldsymbol{v_i'}||^2 + \beta_i \Psi_C(\boldsymbol{v_i}, W) + \Psi_R(\boldsymbol{v_i}, W) \right] \quad (2)$$

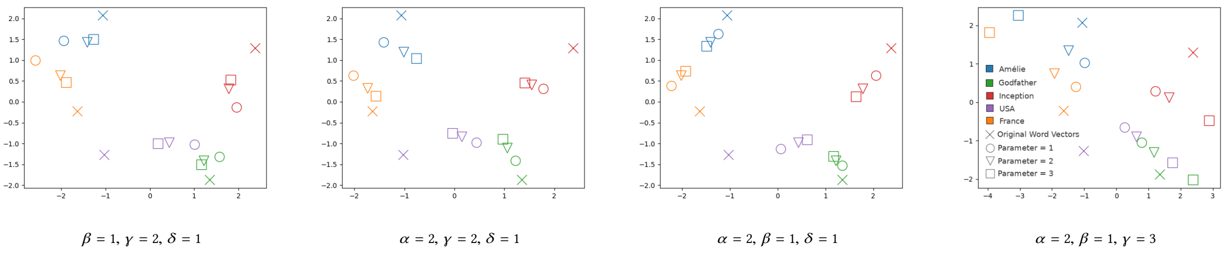
The columnal loss is defined by Ψ_C and treats every embedding $\boldsymbol{v_i}$ to be similar to the constant centroid $\boldsymbol{c_i}$ of the basis embeddings of text values in the same column $C(i)$.

$$\Psi_C(\boldsymbol{v_i}, W) = ||\boldsymbol{v_i} - \boldsymbol{c_i}||^2 \quad \boldsymbol{c_i} = \frac{\sum_{j\in C(i)} \boldsymbol{v_j'}}{|C(i)|} \quad (3)$$

The relational loss Ψ_R treats embeddings v_i and v_j to be similar if there exists a relation between them and dissimilar otherwise. E_r is the set of tuples where a relation $r \in R$ exists. $\widetilde{E_r}$ is the set of all tuples $(i,j) \notin E_r$ where i and j are part of relation r. Thus, each of both indices has to occur at least in one tuple of E_r.

$$\Psi_R(\boldsymbol{v_i}, W) = \sum_{r\in R} \left[\sum_{\substack{j:(i,j)\\ \in E_r}} \gamma_i^r ||\boldsymbol{v_i} - \boldsymbol{v_j}||^2 - \sum_{\substack{k:(i,k)\\ \in \widetilde{E_r}}} \delta_i^r ||\boldsymbol{v_i} - \boldsymbol{v_k}||^2 \right] \quad (4)$$

α_i, β_i, γ_i and δ_i are hyperparameters. Ψ should be a convex function. In [6] we proved the convexity of Ψ for hyperparameter



(a) Influence of $\alpha = 1, 2, 3$ **(b) Influence of $\beta = 1, 2, 3$** **(c) Influence of $\gamma = 1, 2, 3$** **(d) Influence of $\delta = 0, 1, 2$**

Figure 2: Examples for Different Hyperparameter Settings

configurations fulfilling the following inequation:

$$\forall r \in R, i \in \{1, \dots, n\} \quad (\alpha_i \geq 0, \quad \beta_i \geq 0, \quad \gamma_i^r \geq 0) \qquad (5)$$
$$\forall \boldsymbol{v_i} \in W \quad \Big(4\alpha_i - \sum_{r\in R} \sum_{j:(i,j)\in\widetilde{E_r}} \delta_i^r \geq 0\Big)$$

In practice, however, other parameter configurations that do not comply might work as well. The impact of the hyperparameter values on the retrofitting result is shown in Section 4.1. The retrofitting algorithm iteratively executes for all $\boldsymbol{v_i} \in V$ the following equation, which is derived from the root of the partial derivative $\frac{\partial \Psi(W)}{\partial \boldsymbol{v_i}}$.

$$\boldsymbol{v_i} = \frac{\alpha_i \boldsymbol{v_i'} + \beta_i \boldsymbol{c_i} + \sum_{r\in R}\Big[\sum\limits_{\substack{j:(i,j)\\ \in E_r}} (\gamma_i^r + \gamma_j^{\bar{r}})\boldsymbol{v_j} - \sum\limits_{\substack{k:(i,k)\\ \in \widetilde{E_r}}} (\delta_i^r + \delta_k^{\bar{r}})\boldsymbol{v_k}\Big]}{\alpha_i + \beta_i + \sum_{r\in R}\Big[\sum\limits_{\substack{j:(i,j)\\ \in E_r}} (\gamma_i^r + \gamma_j^{\bar{r}}) - \sum\limits_{\substack{k:(i,k)\\ \in \widetilde{E_r}}} (\delta_i^r + \delta_k^{\bar{r}})\Big]} \qquad (6)$$

Given the property of convexity, such an iterative algorithm can be used to minimize Ψ illustrated in more details in the next section.

3.3 Retrofitting Algorithm

The retrofitting algorithm can be expressed as a set of matrix operations that can be solved with linear time complexity according to the number of text values in W. We update all vectors at once using a recursive matrix equation. $\Psi(W)$ can be minimized by iteratively calculating W^k according to (7).

$$W_R = \sum_{r\in R} \Big[((\gamma_{ij}^r) + (\gamma_{ij}^{\bar{r}})^T) - ((\delta_{ij}^r) + (\delta_{ij}^{\bar{r}})^T)\Big] W^k$$
$$W' = \alpha W^0 + \beta \boldsymbol{c} + W_R$$
$$D = diag\Big(\alpha + \beta + \sum_{r\in R}\Big[\sum_{\substack{j:(i,j)\\ \in E_r}} (\gamma_i^r + \gamma_j^{\bar{r}}) - \sum_{\substack{k:(i,k)\\ \in \widetilde{E_r}}} (\delta_i^r + \delta_k^{\bar{r}})\Big]\Big)$$
$$W^{k+1} = D^{-1} W'$$
$$\boldsymbol{c} = (\boldsymbol{c_1}, \dots, \boldsymbol{c_n}) \quad \alpha = (\alpha_1, \dots, \alpha_n) \quad \beta = (\beta_1, \dots, \beta_n) \qquad (7)$$

More details are outlined in [6].

4 EVALUATION

Retro is a fully functional system built on top of PostgreSQL. Given an initial configuration including the connection information for a database and the hyperparameter configuration, Retro fully automatically learns the retrofitted embeddings and adds them to the given database. We created two databases based on the Movie Database[2] (TMDB) and the Google Play Store Apps dataset[3] (GPSA). TMDB consists of 15 tables containing 493,751 unique text values, whereas the GPSA database has 7 tables and 27,571 unique text values (details are outlined here[45]). Both of them are available as CSV files and are imported in our Retro PostgreSQL database system.
One baseline we compare our retrofitted embeddings to, are plain word vectors (PV) that have no notion of the relational schema. The counterpart to this would be embeddings that just rely on the the structural information given by the database. Here we use the node embedding technique DeepWalk [9] (DW) that is learned based on a graph representation of the database relations. Moreover, we applied the original retrofitting approach [5] leading to another baseline embedding dataset (MF).

4.1 Hyperparameter Analysis

The influence of the hyperparameters is visualized in Figure 2: We learned 2-dimensional embeddings for a small example dataset containing three movies and the country where those movies have been produced. Accordingly, there are two columnar (movie and country) and one relational connection (see Section 3.1). "Amélie" was produced in "France", the other movies in the "USA". Usually the hyperparameters for each vector are derived from four global hyperparameters α, β, γ, and δ as detailed in [6]. We set the hyperparameters α, β, γ, and δ to different values and performed the relational retrofitting.
As shown in Figure 2a, the learned embeddings stay closer to their original embeddings when the α values increasing. Higher values of β make it easier to cluster the categories from each other, e.g. reduce the distances between the movie vectors of "Inception" (red), "Godfather" (green), and "Amélie" (blue). The γ value controls the influence of relational connections. This brings the representations of text values which share a relation closer together. The δ factor causes vectors with different relations to separate and thus prevent concentrated hubs of vectors with different semantic. One can see in Figure 2d how $\delta = 0$ causes all vectors to concentrate around the origin of the coordinate system. If δ is set to a high value like $\delta = \alpha = 2$, the algorithm places the vectors far from the origin of the coordinate system. However, related text values still get assigned to similar representations. In the example, the retrofitting algorithm is still converging for this configuration. Our analysis shows, that the exposed hyperparameters allow to steer the relational retrofitting

[2] https://www.kaggle.com/rounakbanik/the-movies-dataset
[3] https://www.kaggle.com/lava18/google-play-store-apps/
[4] https://github.com/guenthermi/the-movie-database-import
[5] https://github.com/guenthermi/google-play-dataset-import

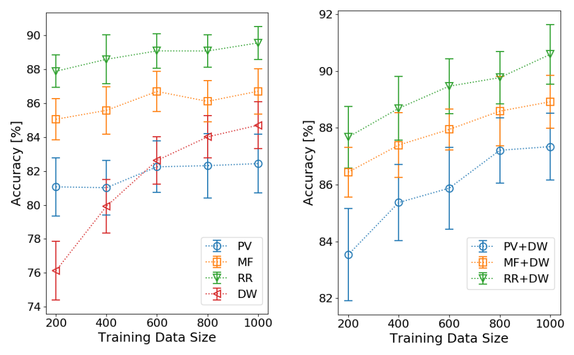


Figure 3: Classification of Birth Places of US-American Directors with Increasing Sample Size

process into different directions in a fine-grained manner, i.e. to adapt to different downstream tasks.

4.2 Machine Learning Tasks

Binary Classification. We implemented a binary classifier to label a set of directors of the TMDB dataset according to there citizenship. The classifier should decide between US-American and non-US-American directors. Since this information is not available from the TMDB dataset, we extract the citizenship from Wikidata [10] by using the SPARQL query service. We trained a feed-forward neural network (one hidden layer with 600 neurons; applying dropout and L2 regularization; Nadam optimizer [4]) on the different 300-dimensional embedding representations of the director names (full names). We used for the training 200 to 1,000 samples and validate the accuracy with 1,000 test samples. We compared the accuracies achieved when using plain word embeddings (PV), node embeddings (DW), simple retrofitted (MF) and relational retrofitted embeddings (RR). We ran the training and testing on the ANNs 20 times for each configuration with different sample sets.

The accuracy values and their standard deviation achieved by the classifiers are shown on the left in Figure 3. The best results are achieved with our relational retrofitting approach (RR) utilizing word embedding features of the directors name but indirectly also word embedding features of related text values like the movie titles directed by them. The influence of the training sample size is at lowest for the plain word embeddings (PV). DeepWalk (DW) needs a larger amount of training data to achieve comparable results. The right side of Figure 3 shows the accuracies achieved by running the same experiment but combining the previous embeddings with node embeddings by concatination. This leads to better results for all methods. Notably, the accuracies of the retrofitting methods are much better compared to methods where node embeddings (DW) and plain word embeddings (PV) are just concatenated.

Missing Value Imputation. Further, we built classifiers to predict app categories within GPSA database which can be used to impute missing values. Here,a feed-forward neural network (two hidden layer with 600 and 300 neurons; applying dropout and L2 regularization; Nadam optimizer [4]) is applied on the embeddings of the application names. The network was trained 10 times on 400 random samples to predict the one out of 33 categories. The category information and the genre information (which is often redundant) are omitted for the retrofitting. We trained the network on all embedding types and compared it to

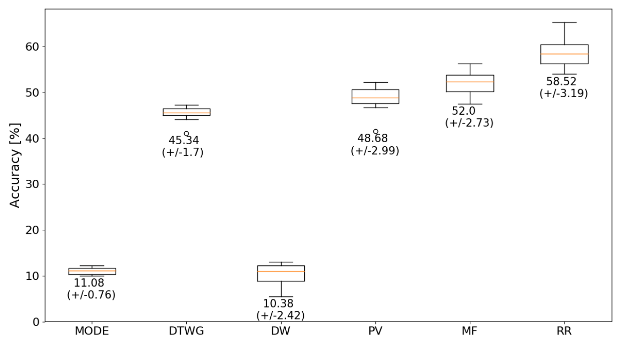


Figure 4: Imputation of Categories of Android Apps

MODE imputation, choosing always the most frequent category in the training data, and Datawig [2]. Figure 4 shows that best accuracy is achieved by relational retrofitting (RR).

5 CONCLUSION

In this paper, we presented Retro, a system that augments all terms in database tables by dense vector representations. Therefore, we employed the notion of retrofitting to modify word embedding representations to specialize for given relational schemas. We validated Retro experimentally by building standard feed-forward neural networks for different classification tasks. Our evaluation showed that the generated relational embeddings are ready-to-use for different ML tasks and even outperform state-of-the-art techniques such as the approach of Faruqui et al. or DeepWalk [9].

ACKNOWLEDGMENTS

This work is funded by the German Research Foundation (DFG) within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907) and by Intel®AI Research.

REFERENCES

[1] Abdulaziz Alghunaim, Mitra Mohtarami, Scott Cyphers, and Jim Glass. 2015. A Vector Space Approach for Aspect Based Sentiment Analysis. In *Proc. of NAACL-HLT.* 116–122.
[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 *Proc. of the CIKM 2018.* ACM, 2017–2025.
[3] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *TACL* 5 (2017), 135–146.
[4] Timothy Dozat. 2016. Incorporating Nesterov Momentum into Adam. In *ICLR 2016 Workshop.*
[5] Manaal Faruqui, Jesse Dodge, Sujay Kumar Jauhar, Chris Dyer, Eduard Hovy, and Noah A Smith. 2015. Retrofitting Word Vectors to Semantic Lexicons. In *Proc. of NAACL-HLT 2015.* 1606–1615.
[6] Michael Günther, Maik Thiele, and Wolfgang Lehner. 2019. RETRO: Relation Retrofitting For In-Database Machine Learning on Textual Data. (2019). arXiv:1911.12674
[7] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *NIPS 2013,* C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3111–3119.
[8] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *EMNLP.* 1532–1543.
[9] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online Learning of Social Representations. In *Proc. of the 20th ACM SIGKDD.* ACM, 701–710.
[10] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: A Free Collaborative Knowledge Base. *Commun. ACM* 57, 10 (2014), 78–85.
[11] Xinjie Zhou, Xiaojun Wan, and Jianguo Xiao. 2015. Representation Learning for Aspect Category Detection in Online Reviews. In *29th AAAI Conference on Artificial Intelligence.*

Revisiting the Theory and Practice of Database Cracking

Fatemeh Zardbani
Aarhus University

Peyman Afshani
Aarhus University

Panagiotis Karras
Aarhus University

ABSTRACT

Database cracking (DBC) provides an adaptive data storage environment that meets the needs of modern applications in business and science, reorganizing data on demand and adapting indexes on the fly, automatically, and collaterally to query processing. Despite intensive research on cracking and other adaptive indexing variants, their theoretical side has scarcely been investigated. Yet, quite surprisingly, as we show, an antecedent of database cracking in a pure, no-frills form had been developed in the theory community 24 years ahead of its time by the name of *deferred data structuring* (DDS). While lacking system implementations, DDS corresponds to what we would call, by the terminology used in the database community, *materialization-based data-driven center* cracking for point lookup queries, as well as a stochastic variant thereof. Further, DDS has gone beyond regular cracking proposals by suggesting a policy that reorganizes index ranges along the median of a *sample* set, i.e., a *mediocre* element.

In this paper, we reanalyze state-of-the-art database cracking algorithms with the benefit of hindsight provided by deferred data structuring, and propose new alternatives that use a mediocre element as cracking pivot instead of a random or a median one. In a thorough experimental study, we determine that a logarithmic or linear sample size yields best performance on a standard benchmark across the board of cracking algorithms.

1 INTRODUCTION

Database Cracking (DBC) [1–3] addresses the needs of dynamic environments where workload knowledge and idle time are scarce, queries follow an exploratory path, and new data arrive continuously [1]; as a form of adaptive indexing [6], it paves the way to self-organizing database management systems, eschewing the need for human administration in physical database design. Cracking builds and refines index data structures for a column-oriented database incrementally, in response to queries and arriving data, without a need for human intervention; its core operation, applied within the select operator, reorganizes a column into pieces [5], handles updates [4] and invites security features [8]. A *stochastic* alternative [1] improves performance by refraining from *blindly* following queries; it also creates *random cracks* on its own, and thereby avoids the deterioration of performance that skewed workloads may cause.

Surprisingly, while database cracking has been studied over the last decade, an antecedent thereof had been investigated from a theory perspective two decades in advance by the name of **Deferred Data Structuring** (DDS) [7]. Specifically, DDS suggested that, instead of processing a data set in advance, we may instead process it while responding to queries. Traditionally, to answer the query *"is integer x in list ℓ?"*, we would scan ℓ in $O(n)$. If the number of queries is high, it pays off to sort $O(n \log n)$ in a pre-processing step and then perform binary search in $O(\log n)$ for each query. By DDS, we create a data structure that represents the list *while* processing queries, achieving better performance in all cases. While DDS was proposed for lookup queries on a static data set only, we will argue that its main logic uncannily resembles that of database cracking.

In this paper, we observe the resemblance between DBC and DDS and bring both concepts under the same roof. We conduct a thorough theoretical study of state-of-the-art DBC algorithms under the light of DDS methods. We implement existing DDS and DBC proposals and propose new intermediary, mediocre-based solutions, that inherit both the theoretical elegance of DDS and the practical applicability of DBC.

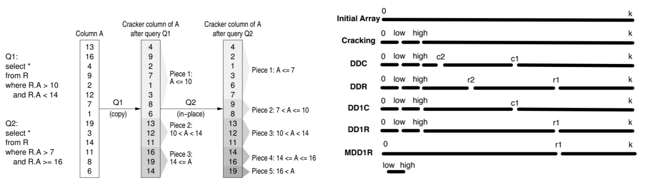
© 2020 Copyright held by the owner/author(s). Published in Proceedings of the 23rd International Conference on Extending Database Technology (EDBT), March 30-April 2, 2020, ISBN 978-3-89318-083-7 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

2 RELATED WORK

We discuss related works in two fields: database cracking and defferred data structuring.

2.1 Database Cracking

Database Cracking [2, 3] reorganizes and indexes columns in an adaptive manner triggered by user queries, within the SELECT operator. In the general case, a query requests all values within a range, [*low*, *high*]; when responding to that query, the a cracking system finds one or more *pieces* of the current index where the requested data resides, reorganizes (i.e., cracks) the column so as to bring the result values between query bounds *low* and *high* in a contiguous space, and updates the index accordingly. Figure 1a provides an example.



(a) Reorganizing a column. **(b) Cracking algorithms at work.**

Figure 1: Database cracking illustration [1].

Stochastic Cracking [1] maintains performance when faced with pathological workloads; in addition to cracking using query bounds as pivots, it creates additional cracks while traversing the index towards query bounds. Figure 1b indicates how several stochastic cracking algorithms work. We discuss the six main alternatives: DDC, DDR, DD1C, DD1R, MDD1R, and PMDD1R.

The *Data-Driven Center* (**DDC**) algorithm divides (i.e., cracks) each value range it encounters while traversing the index along the *middle* (i.e., median) of its value domain *recursively*, as in an ideal case of pivot selection by quicksort. This recursive splitting process terminates when pieces become smaller than a size threshold. Thereafter, DDC cracks on query bounds as usual. Algorithm 1 illustrates the process. On the other hand, the *Data-Driven Random* (**DDR**) algorithm avoids the median-finding overhead: it uses a random element instead of the median as pivot when cracking each value range in Line 6 of Algorithm 1.

Both DDC and DDR incur an overhead on the first few queries, as they *recursively* introduce many cracks on the way to query bounds. Two lightweight alternatives, **DD1C** and **DD1R**, eschew the recursion, i.e., crack only *once* at a median or random pivot, respectively, in addition to cracking at query bounds, by turning the *while* loop in Line 5 of Algorithm 1 to an *if* statement.

ALGORITHM 1: DDC [1]

```
   Result: Cracks at center of each relevant piece and bounds
1  int DDCCrack(C:array, v:value)
2      Find the piece Piece that contains value $v$;
3      $pLow$ = $Piece$.firstPosition();
4      $pHigh$ = $Piece$.lastPosition();
5      while ($pHigh - pLow >$ CRACK-THRESHOLD)
6          $pMiddle = (pHigh + pLow)/2$;
7          Introduce crack at $pMiddle$;
8          if ($v < C[pMiddle]$)
9              $pHigh = pMiddle$
10         else
11             $pLow = pMiddle$
12     $position$ = crack($C[pLow, pHigh]$, $v$);
13     return $position$;
   /* Main Body : DDC                                   */
   /* Crack array C on bounds a and b                   */
14 $positionLow$ = DDCCrack($C$, $a$);
15 $positionLow$ = DDCCrack($C$, $b$);
16 result = createView($C$, $positionLow$, $positionHigh$);
```

Still, the hitherto presented algorithms create cracks at each query's bounds, which may hurt performance without bringing a benefit in the long run. **MDD1R**, a variation of DD1R, dispels the cracking at query bounds as well, and simply *materializes* query results while creating exactly *one* random crack per query. Algorithm 2 shows the corresponding pseudocode.

ALGORITHM 2: MDD1R [1]

```
   Result: Cracks at a random point in one relevant piece and bounds
1  array split-and-materialize(Piece, a, b)
2      $L = Piece$.firstPosition;
3      $R = Piece$.lastPosition;
4      $result$ = newArray;
5      $X = C[L + rand()\%(R - L + 1)]$;
6      while ($L \leq R$)
7          while ($L \leq Q$ and $C[L] < X$)
8              if ($a \leq C[L]$ and $C[L] < b$)
9                  $result.Add(C[L])$
10             $L = L + 1$;
11         while ($L \leq R$ and $C[R] \geq X$)
12             if ($a \leq C[R]$ and $C[R] < b$)
13                 $result.Add(C[L])$
14             $R = R - 1$;
15         if ($L < R$)
16             swap($C[L]$, $C[R]$)
17     Add crack on $X$ at position $L$;
18     return $result$;
   /* Main Body: MDD1R                                  */
   /* Crack array C on bound a, b                       */
19 Find the piece $P1$ that contains value $a$;
20 Find the piece $P2$ that contains value $b$;
21 if ($P1 == P2$)
22     $result$ = split-and-materialize($P1$, $a$, $b$)
23 else
24     $res1$ = split-and-materialize($P1$, $a$, $b$);
25     $res2$ = split-and-materialize($P2$, $a$, $b$);
26     $view$ = createView($C$, $P1.lastPosition + 1$, $P2.firstPosition - 1$);
27     $result$ = concat($res1$, $view$, $res2$);
```

In more detail, the algorithm first finds the pieces where the two bounds are. If they are in the same piece, it partitions that piece with respect to a random pivot, while collecting the query results in an array. Should they be in different pieces, it partitions the pieces where each bound belongs and then concatenates the query results, as well as all pieces in between to produce the response to the query. Note that MDD1R maintains a data-driven character: even though not cracking at query bounds, it introduces random cracks in the pieces where those bounds are.

Even with MDD1R, the initial queries of a workload need to reorganize almost all the the data. **PMDD1R** is a *progressive* instantiation of MDD1R that takes the incremental nature of cracking one step further. MDD1R performs a reorganization task on a given piece in smaller units, performed with each query touching that piece. A percentage p determines how much of the pending reorganisation task is done with each relevant query, while materializing and returning the query result.

2.2 Deferred Data Structuring

Deferred Data Structures [7] are tree-like structures built in response to queries. Their objective and rationale resembles database cracking, even though they were introduced two decades earlier with a focused theoretical intent and no accompanying system implementation. Consider a list $\ell = \{x_1, x_2, x_3, \ldots, x_n\}$ and *existence* queries thereupon, $q = \{q_1, q_2, q_3, \ldots, q_r\}$. A conventional approach would sort the list and answer each query by binary search. DDS performs sorting through query answering: it answers each query in $O(n)$ time, and partition the list as well while doing so. Algorithm 3 illustrates DDS. By the terms of Section 2.1, Algorithm 3 corresponds to a **DDC** variant of database cracking specialized on point lookup queries, without a size threshold: it cracks recursively on the median, like DDC does, and reports the existence or absence of the lookup query value; were there a size threshold, it would correspond to an **MDDC** variant, yet without such a threshold **MDDC** degenerates to **DDC**.

ALGORITHM 3: DDS via recursive median finding [7]

```
   Result: Create a tree structure representation of list l while responding to queries
1  boolean SEARCH(v:node, q:query)
2      if(v is not labeled)
3          $EXPAND(v)$;
4      if(label(v) == q)
5          return true;
6      if(v is a leaf node)
7          return false;
8      if(q < label(v))
9          return $SEARCH(left\_child(v), q)$;
10     if(q > label(v))
11         return $SEARCH(right\_child(v), q)$;
12 void EXPAND(v:node)
13     $S \leftarrow set(v)$;
14     $m \leftarrow MEDIAN - FIND(S)$;
15     $label(v) \leftarrow m$;
16     if($\|S\|$ == 1)
17         return ;
18     $S_l \leftarrow [x \mid x \text{ in } S \text{ and } x < m]$;
19     $S_r \leftarrow [x \mid x \text{ in } S \text{ and } x > m]$;
20     $set(left\_child(v)) \leftarrow S_l$;
21     $set(right\_child(v)) \leftarrow S_r$;
   /* Main Body                                         */
22 initialize the tree $T_X$ with the n data keys at the root;
23 Get a query $q$;
24 Result $\leftarrow SEARCH(root, q)$;
25 Output the result;
26 Goto Line 23;
```

Further, DDS [7] comes along with a *randomized* proposal, which replaces the exact median-finding operation with a *mediocre* function, i.e., the median of small sampled set of values, whose computed rank passes a sanity test, instead of the whole set, as Algorithm 4 shows; this choice improves the cost per query while it still creates a well-balanced tree structure in the long term. Once again, the rationale is reminiscent of what we would call **MDDR** in database cracking terms. In particular, a randomized DDS where the size of the sampled set is one element corresponds to an MDDR cracking algorithm specialized on point lookup queries and without a size threshold.

ALGORITHM 4: Mediocre finding function [7]

```
   Result: Finds mediocre of set $T$
1  int mediocreFind(T:set of values)
2      $t \leftarrow size(T)$;
3      Pick a random of sample $S$ of size $2 * \lceil t^{5/6} \rceil + 1$ from T;
4      $m \leftarrow MEDIAN - FIND(S)$;
5      Compute rank(m) by comparing with each element of $T - S$;
6      If rank(m) is not in the range $(t/2) \pm t^{2/3}$;
7      return $m$;
```

3 THEORETICAL ANALYSIS

Here, we analyze state-of-the-art database cracking algorithms by the tools of deferred data structuring, assuming a cracking size threshold of zero and point lookup queries. We also propose, study, and build upon an alternative stochastic cracking algorithm, **DDM**, which uses a *mediocre* element as cracking pivot, as in Algorithm 4, instead of a random one, as DDR algorithms do, or a median one, as DDC algorithms do.

Theorem 3.1. *DDC: The number of operations needed to process r queries on a list of n points is no more than $\lambda(n, r)$:*

$$\lambda(n,r) = \begin{cases} 3n\log r + r\log n, & \text{if } r \le n \\ (3n+r)\log n, & \text{if } r > n \end{cases}$$

Proof. In case $r \le n$, at any level of the tree, at most r nodes are expanded. For the top $\log r$ levels, the total cost is less than $3n\log r$, since all nodes have to be expanded. The creation of a crack includes finding the median, which requires $3|set(node)|$. The cost of node expansion at level i of the tree for $i > \log r$, is $O(rn/2^i)$, since the expansion of a node at this level costs at most $3n/2^i$. Summing over all but the first $\log r$ levels, we get an $O(n)$ cost, which is dominated by $3n\log r$. Searching for each query costs $O(\log n)$, since the tree is balanced, hence the $r\log n$ term. When $r > n$, expansion will complete the tree, with a cost of $3n\log n$, while search follows the same principles. □

Theorem 3.2. *DDR: The number of operations needed to process r queries on a list of n points is no more than $\lambda(n, r)$:*

$$\lambda(n,r) = n^2 + rn$$

and is expected to be no more than $\lambda(n, r)$ operations:

$$\lambda(n,r) = \begin{cases} 1.39n\log r + r\log n, & \text{if } r \le n \\ (1.39n+r)\log n, & \text{if } r > n \end{cases}$$

Proof. As there is no guarantee that the created tree will be balanced, in the worst case, the random numbers chosen to create cracks yield a completely unbalanced tree. A query may cause the entire tree to be created in $O(n^2)$. Due to the lack of balance, search can take up to n operations, producing the rn term. In the average case, we expect performance similar to quicksort [9], with 1.39 in place of 3 in Theorem 3.1. □

Theorem 3.3. *DDM: The number of operations needed for processing r queries in a list of n points is no more than $\lambda(n, r)$:*

$$\lambda(n,r) = \begin{cases} (1+\alpha)(n\log r + r\log n), & \text{if } r \le n \\ (1+\alpha)(n+r)\log n, & \text{if } r > n \end{cases}$$

with probability greater than $1 - \frac{\log r}{\beta n}$, where $\alpha \ll 1$ and β depends on the value of α.

Proof sketch. The proof follows from [7]. The height of the tree created only differs from $\log n$ by a constant, so the search operations are $r\log n$. Then, the probability of the first sample chosen rendering a median that passes the test is higher than $(1 - \frac{1}{4|set(node)|})$, which leads to the conclusion that the total cost of testing for mediocrity is at most $(1+\alpha)n\log r$ with probability higher than $1 - \frac{\log r}{k^2 n}$, where α and k are small constants, and β depends on α. The total cost of finding the medians for the first $\log r$ levels is $O(n^{\frac{5}{6}} r^{\frac{1}{6}})$ with probability higher than $1 - \frac{\log r}{\beta n}$, from which the complexity for $r \le n$ follows. If $r > n$ the tree will be complete with some extra costs, as in the DDR case. □

Theorem 3.4. *DD1C: The number of operations needed to process r queries in a list of n points is no more than $\lambda(n, r)$:*

$$\lambda(n,r) = \begin{cases} 14n + r\log n, & \text{if } r \le n \\ (3n+r)\log n, & \text{if } r > n \end{cases}$$

Proof. As the first query, q_1 is made to a list of length n, the median should be found in $3n$. Then, one of the two crack pieces is chosen and another crack is made, with regards to the query. The process of partitioning takes at most $\frac{n}{2}$ comparisons. For the second query, up to two comparisons are made based on the crack created in the first query, to choose a chunk of size at most $n/2$, find its median, and crack one of the resulting pieces, yielding $\frac{3n}{2} + \frac{n}{4}$. Following the same pattern until the rth query, we get a cost of:

$$\sum_{i=1}^{i=r} \frac{3n}{2^{i-1}} + \sum_{i=1}^{i=r} \frac{n}{2^i} = 7n \sum_{i=1}^{i=r} \frac{1}{2^i} < 14n$$

while the search component costs $r\log n$. When the number of queries reaches n, the tree will be complete, and the cost is as in the proof of Theorem 3.1. □

Theorem 3.5. *DD1R :The number of operations needed to process r queries in a list of n points is no more than $\lambda(n, r)$:*

$$\lambda(n,r) = \begin{cases} 3rn + r, & \text{if } r \le n \\ n^2 + rn, & \text{if } r > n \end{cases}$$

and is expected to be no more than $\lambda(n, r)$ operations:

$$\lambda(n,r) = \begin{cases} 7.56n + r\log n, & \text{if } r \le n \\ (1.39n+r)\log n, & \text{if } r > n \end{cases}$$

Proof. In the worst case in terms of random pivot choices, for query q_1 we pick a random point and partition based on it, in at most n operations. The pieces may be of size 1 and $n-1$. One of those two is partitioned based on the query in $n-1$ comparisons. The second query will partition a piece of size as large as $n-2$ for the random crack and one as large as $n-3$ for the query bound, and so on, in $[n-2(i-1)] + [n-(2i-1)]$ operations for the i^{th} query. Going all the way to q_r, we have

$$\sum_{i=0}^{i=2r-1} (n-i) = 2rn - 2r^2 + r$$

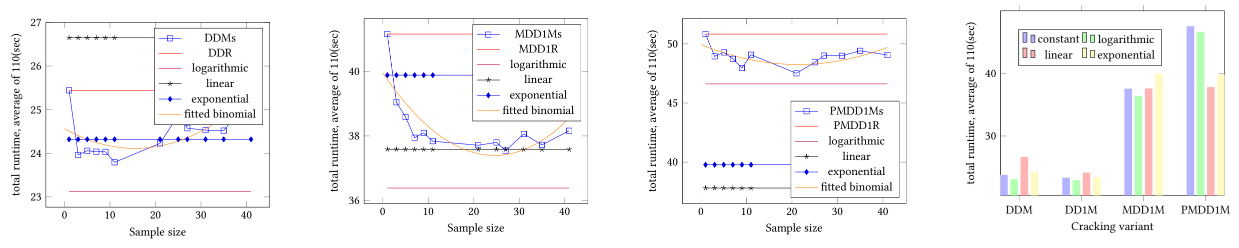
operations for expansion and rn operations for search. Should the number of queries exceed n, the tree is completed, hence n^2 operations. In the expected case, the tree resembles a balanced tree and results follow Theorem 3.4, with the quicksort complexity factor $1.39n$ replacing $3n$ in calculations, as in Theorem 3.2. □

Theorem 3.6. *DD1M: The number of operations needed to process r queries in a list of n points is no more than $\lambda(n, r)$:*

$$\lambda(n,r) = \begin{cases} (1+\alpha)(n\log r + r\log n), & \text{if } r \le n \\ (1+\alpha)(n+r)\log n, & \text{if } r > n \end{cases}$$

with probability greater than $1 - \frac{\log r}{\beta n}$.

Proof. By Lemma 3 in [7], with high probability we only need to compute a median over a sample once, as the first attempt passes the mediocrity test. For the first r queries, we compute the medians of nodes that will overall lead to $O(n^{\frac{5}{6}} r^{\frac{1}{6}})$ (Lemma 5 in [7]). The cost testing for mediocrity at level i, denoted by c_i, is proven to be less than $(1+\alpha)n\log r$. After cracking at the



(a) **Recursive crack** (b) **Crack with materialization** (c) **Progressive materialization** (d) **All variants, constant** $m = 21$

Figure 2: Total runtime, 160K queries; linear: $m = \frac{size}{1000}$**, logarithmic:** $m = log(size)$**, exponential:** $m = size^{\frac{5}{6}}$**.**

mediocre, we scan the list to find the query point and create another crack there, in at most s_i (Lemma 1 in [7]), yielding:

$$n^{\frac{5}{6}} r^{\frac{1}{6}} + \sum_{i=1}^{i=r} c_i + 1.5 s_{i+1} \leq$$

$$n^{\frac{5}{6}} r^{\frac{1}{6}} + (1+\alpha) n \log r + 2n(1-(0.5)^r) + 20 n^{\frac{2}{3}} \frac{1-2^{\frac{2r}{3}}}{1-2^{\frac{2}{3}}}$$

The cost of search remains similar to that of DDM. □

Overall, materialization-based algorithms have similar complexities as their default counterparts due to their common bases.

4 EXPERIMENTAL STUDY

We assess the performance of mediocre-based variants database cracking algorithms [1] inspired from our study of deferred data structures [7]. We conduct experiments on a Ubuntu Linux server release 18.04 machine with a 10-core 3.1GHz Intel E5-2687W processor and 377GB of RAM. All methods are implemented in C++ upon the code[1] of [1]; our code is also available[2] online. We use the 4TB SkyServer[3] data and workload [1], derived from an astronomy project mapping the universe. We filter selection predicates from 160K chronologically ordered queries using the right ascension attribute of the Photoobjall table, which contains 500 million tuples. Query patterns are complex, as users tend to focus in a specific area of the sky before moving on.

4.1 Compared Algorithms

We compare the state-of-the-art stochastic cracking algorithms presented in Section 2, namely DDR, DD1R, MDD1R, and PMDD1R with $p = 0.1$ and $CRACK\_THRESHOLD = 128$ to their median-based counterparts and to counterparts that use a mediocre cracking pivot, i.e., the median of a random *sample* set of a cracked piece, rather than a median or random one: DDM, DD1M, MDD1M, and PMDD1M, respectively.

The median-based counterparts of DDR, DD1R and MDD1R are DDC, DD1C and MDD1C respectively; we included those three in our study, but they proved to be too expensive. We do not include a median-based counterpart of PMDD1R, since median-finding operations are hard to render progressive. The mediocre-based policy is reduced to the median-based one for m equal to piece size, and to the randomized one for $m = 1$.

4.2 Results

We apply each algorithm on the same 160K-query workload and measure the total runtime, juxtaposing mediocre-based variants to their randomized and median-based counterparts, where such exist. Figure 2 shows the results when varying the sample set size m from 1 to 41, and as a linear, logarithmic, or exponential function of piece size, with the mediocre-based policy, averaging over 110 runs and foregoing the mediocrity check [7]. The case of single crack without materialization is very similar to that of single crack with materialization in Figure 2b, hence we omit a separate figure for that case.

In the case of a constant sample size, the cost of calculating the median of a small sample set is initially a worthwhile price to pay for the benefits it brings, yet the cost to benefit ratio deteriorates as m grows; binomial fit curves visualize the trends in Figure 2a, 2b, and 2c. However, cases where sample size is a simple function of piece size achieve the best performance in all variants. The logarithmic function is the best performer with recursive and simple crack with and without materialization. In the variant with progressive materialization, a linear function, $m = \frac{size}{1000}$, performs best.

5 CONCLUSION

We revisited the theory and practice of *database cracking*, which has been intensively studied in practice, yet scantily examined in theory. We provided the first thorough study of the complexity of the all state-of-the-art stochastic cracking algorithms, drawing from an overlooked 32-year-old study that introduced analogous concepts under the name of *deferred data structuring*. Inspired from deferred data structuring, we introduced a refined stochastic cracking policy that uses a sample-based *mediocre* pivot, rather than an arbitrary random or median one, for data-driven cracking. We showed that variants of state-of-the-art stochastic cracking algorithms using the mediocre-based policy have lower complexity than their median-based and randomized counterparts with high probability, and demonstrated experimentally that they stand out in terms of cumulative time efficiency.

REFERENCES

[1] 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.

[2] Stratos Idreos. 2010. *Database Cracking: Towards Auto-tuning Database Kernels.* Ph.D. Dissertation. CWI.

[3] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database cracking. In *CIDR*. 68–78.

[4] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Updating a cracked database. In *SIGMOD*. 413–424.

[5] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2009. Self-organizing tuple reconstruction in column stores. In *SIGMOD*. 297–308.

[6] Stratos Idreos, Stefan Manegold, Harumi 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.

[7] Richard M. Karp, Rajeev Motwani, and Prabhakar Raghavan. 1988. Deferred Data Structuring. *SIAM J. Comput.* 17, 5 (1988), 883–902.

[8] Panagiotis Karras, Artyom Nikitin, Muhammad Saad, Rudrika Bhatt, Denis Antyukhov, and Stratos Idreos. 2016. Adaptive Indexing over Encrypted Numeric Data. In *SIGMOD*. 171–183.

[9] Kurt Mehlhorn and Peter Sanders. 2008. *Algorithms and Data Structures: The Basic Toolbox.* Springer, Berlin.

[1] https://github.com/felix-halim/scrack
[2] https://gitlab.com/fatemeh.zardbani/adaptive-indexing
[3] http://cas.sdss.org/

Boosting Blocking Performance in Entity Resolution Pipelines: Comparison Cleaning using Bloom Filters

Leonardo Gazzarri
University of Stuttgart
Stuttgart, Germany
leonardo.gazzarri@ipvs.uni-stuttgart.de

Melanie Herschel
University of Stuttgart
Stuttgart, Germany
melanie.herschel@ipvs.uni-stuttgart.de

ABSTRACT

Entity Resolution (ER) allows to identify different virtual representations of entities that refer to the same real world entity. When applied to highly heterogeneous data, ER relies on schema-agnostic blocking techniques to improve efficiency while yielding good effectiveness. A drawback of schema-agnostic blocking is the potentially high number of redundant pairwise comparisons. This has led to the introduction of additional efficiency layers beyond blocking in the overall ER pipeline, which all aim at pruning comparisons to reduce the unnecessary time overhead.

This paper proposes a novel technique based on Bloom filters that integrates in such an efficiency layer. In addition to avoiding redundant comparisons, it further prunes superfluous comparisons that are unlikely to result in matches when actually compared. Experiments on benchmark datasets show that our approach improves existing approaches in space and time efficiency, with insignificant changes in effectiveness.

1 INTRODUCTION

Entity resolution (ER) is the problem of identifying or *matching* different digital representations of the same real-world entity (e.g., the same person, manufactured part). It represents a fundamental task in data integration and data cleaning. While ER has mostly been studied for homogeneously structured data (to which we refer to as *structured ER*) [6], recent work has been extended to *unstructured ER*, i.e., ER when it is not possible or useful to transform heterogeneous entities to match a common schema [5, 10]. This for instance applies when considering ER for the Web of Data or for data stored in data lakes.

For large datasets, handling the inherently quadratic complexity of ER generally becomes computationally prohibitive. Therefore, ER solutions commonly adopt blocking techniques [6, 12] that reduce the total number of pairwise comparisons by performing comparisons only between entity representations placed in the same block according to some criteria. This typically prunes a significant number of comparisons between entity descriptions that do not match anyway, to which we refer to as *superfluous* comparisons. In structured ER, blocking techniques (and ER in general) heavily rely on a fixed schema among all entity representations. As this assumption does not hold in unstructured ER, schema-agnostic blocking techniques have recently been proposed for unstructured ER [5, 8, 10].

Figure 1 depicts a general pipeline for unstructured ER [12]. It comprises three layers enabling efficient and effective ER. *Block building* places entity representations, each denoted as e_i, into blocks. One problem of schema-agnostic blocking is that the

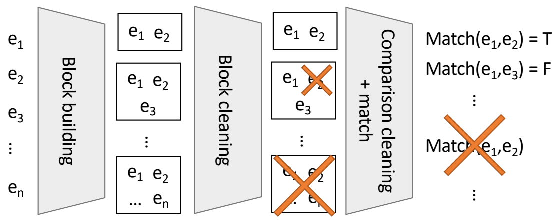


Figure 1: Unstructured ER pipeline.

resulting blocks may significantly overlap and thus yield *redundant* comparisons and the distribution of entity descriptions over blocks may still yield too many pairwise comparisons. These two problems have resulted in two further techniques for unstructured ER. *Block cleaning* acts at a block level and either prunes entire blocks (e.g., too large and thus most likely resulting in mostly superfluous comparisons) or entity descriptions within blocks (e.g., descriptions appearing in too many blocks are removed from the least important blocks). Finally, *comparison cleaning* considers pairs of entity descriptions resulting from the cleaned block collection. It prunes pairs if they are identified as either redundant or superfluous. Otherwise, pairs of entity descriptions are compared using a *match* function to determine whether they represent the same entity or not. Examples of comparison cleaning techniques are *comparison propagation* [9] and *meta-blocking* approaches [4, 11, 14].

Contribution. This paper presents a novel comparison cleaning approach that prunes both redundant and superfluous comparisons. It relies on (i) a favorable order of blocks being processed, for which we validate a heuristic that works in practice, and (ii) false positives that are, in our context, a useful feature of *Bloom filters (BFs).* Indeed, while false positives are typically undesired, we shall see that we can turn them to our advantage when coupled with a favorable order. As Bloom filters are static data structures that need to be correctly set up upfront, we further extend our method to use *scalable Bloom filters (SBFs).* Our experimental validation on several real-world benchmark datasets shows that comparison cleaning using SBFs is robust to different block collection characteristics and provides a good trade off between efficiency and effectiveness of comparison cleaning that improves on baseline and state-of-the-art solutions.

Structure. Section 2 introduces our novel approach based on Bloom filters and its extensions. Section 3 presents our experimental evaluation. We conclude in Section 4.

2 BLOOM FILTER ENHANCED BLOCKING

BFs have been previously used in ER, for example, to obscure sensitive data [15] or to summarize the blocking structure of a dataset [7]. In our work, we employ BFs for comparison cleaning. In this section, we describe our algorithm leveraging BFs (Section 2.1), the block ordering heuristic allowing to turn false positives inherent to BFs to our advantage (Section 2.2), and details on extending our algorithm with SBFs (Section 2.3).

Algorithm 1: Comparison cleaning using a Bloom filter

```
Input: $p_{max}$, $s$, $B$
1 initialize($BF$, $p_{max}$, $s$);
2 while $B$ has further pairs to process do
3     $p_{i,j}$ ← next pair according to order defined by $P$;
4     if !lookup($BF$, key($i$, $j$)) then
5         insert($BF$, $key(i, j)$);
6         submit(match($p_{i,j}$));
```

2.1 General BF based comparison cleaning

To discuss our algorithm that uses BFs, we first provide relevant background on BFs. We refer readers to [2, 3] for further details.

A BF is a space-efficient probabilistic data structure that offers two operations: *insert(k)* inserts the key k in the filter and *lookup(k)* answers if the key k has been inserted with some probability, or definitely if it has not been inserted. False positive probability (the probability of a lookup returning true even though k has not been inserted) can be computed and it increases with insertions of unique keys. A Bloom filter BF is initialized by providing a maximum acceptable false positive probability p_{max} and the number of expected keys to be inserted, denoted s.

In our context, a key $k_{i,j}$ uniquely identifies a pair $p_{i,j} = (e_i, e_j)$. We assume that e_i and e_j are unique identifiers of entity descriptions. We generate $k_{i,j}$ using a function $key : Integer \times Integer \longrightarrow Integer$. The pairs to be sequentially inserted into the BF (by inserting their integer key) are pairs produced from a block collection $B = [b_1, \ldots, b_n]$ that comprises n blocks of various sizes. The maximum number of keys possibly being inserted is bounded by $O(|b_1|^2 + |b_2|^2 + \ldots + |b_n|^2)$. Given that the false positive probability increases with the number of keys that have been inserted into a BF, we assume that we can iterate over pairs resulting from B in a specific order. That is, our algorithm uses an iterator to retrieve pairs in the order given by

$$P = \left[(e_i^k, e_j^k)\,\middle|\, e_i^k e_j^k \in b_k, 1 \leq k \leq n, 1 \leq i < |b_k|, i < j \leq |b_k|\right]$$

where the three ranges specified for k, i, and j are to be interpreted as three nested loops with k being the outer and j the most inner loop. How to sort B in a good order for efficient and effective comparison cleaning is further discussed in Section 2.2.

Using the above assumptions and notation, Algorithm 1 summarizes how we leverage BFs to perform comparison cleaning. It first initializes a Bloom filter BF, given p_{max} and s. While p_{max} is generally user defined (we will see how to best set it in practice in Section 3), s can also be computed as the upper bound of comparisons based on B (which we do in all experiments). The algorithm then iterates over pairs retrieved from B in the order given by P, retrieving each pair $p_{i,j}$ one at a time. In line 4, we check if $p_{i,j}$ has already been inserted in BF. To this end, a unique key $k_{i,j}$ is generated for pair $p_{i,j}$. If looking up $k_{i,j}$ in BF returns false, we know that the pair has not been compared before. So we insert the key into BF, perform the pairwise comparison of the actual pair of entity descriptions using a *match* function, and propagate the result (could be e.g., a boolean, a similarity score, or a match probability) for further processing via a submit method.

Time analysis. Time to insert or search a key in a BF is $O(|H|)$, with $|H|$ the number of hash functions used to fingerprint the key in the BF [3]. Defining $\bar{c}$ the average cost for comparing two entities, the desiderata is that $O(|H|) \ll \bar{c}$. This typically holds in practice because hashing a pair of integers is less expensive than fetching and matching two entity representations.

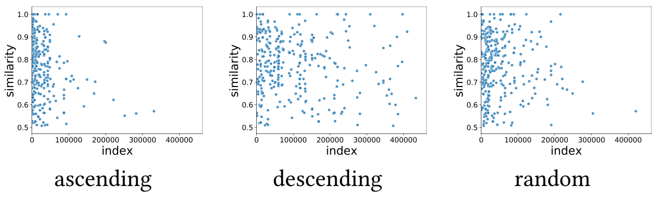


Figure 2: Distribution of pairwise similarities per iteration for different order heuristics

Space analysis. The minimum bits size m of the filter is determined by $m \geq 1.44s \cdot log_2(1/p_{max})$ [3]. For instance, assuming $p_{max} = 0.1$, we have $m \approx 4.78s$ bits.

In our algorithm, we can easily replace the BF by any other dictionary data structure implementing an $insert(\cdot)$ and a $lookup(\cdot)$ method. We opt for BFs for two reasons. First, a BF is a well known space efficient data structure and we validate experimentally that in comparison to other dictionary data structures, it exhibits better performance for comparison cleaning (see Section 3). Second, a BF allows us to additionally prune superfluous comparisons. This slightly more hidden benefit is rooted in both p_{max} inherent to Bloom filters and the order of entity pairs determined by P. This interplay is further discussed next.

2.2 Block ordering heuristic

The false positive probability inherent to a BF results in three possible cases when a pair $p_{i,j}$ is processed by Algorithm 1:

True negative (*tn*) $lookup(BF, p_{i,j})$ returns false, so $p_{i,j}$ is not redundant

True positive (*tp*) $lookup(BF, p_{i,j})$ returns true and the pair is indeed redundant

False positive (*fp*) $lookup(BF, p_{i,j})$ returns true even though $p_{i,j}$ has not yet been inserted

While *tn* and *tp* are desired cases, *fp* may be problematic for the effectiveness of ER as it may reduce the recall of ER if a wrongly pruned $p_{i,j}$ had, if compared, resulted in a match. We refer to this case as *miss-match*. On the other hand, if $p_{i,j}$ would have resulted in a non-match upon comparison, the comparison was a superfluous one, i.e., a comparison we actually want to prune.

When the number of keys inserted in the BF increases, the false positive rate of the BF increases up to p_{max}. The basic idea of our Bloom filter based comparison cleaning is to put this "effect" of BFs to good use by determining an order of pairs for P that has a high probability of processing true matches early (to minimize the negative effect of miss-match cases) and using the increasing false positive probability coming with later processing to prune superfluous comparisons. Such behavior is intuitively obtained when finding an order of pairs where the match probability decreases with increasing number of iterations. Clearly, the performance of our approach relies on both identifying a suited order that mimics the behavior described above, and on the parameter p_{max}. Given a block collection, [12] postulated that similar entity descriptions are more likely to be found in small blocks rather than in big blocks. We experimentally validate this heuristic on several real-world datasets (see Table 1). Figure 2 shows representative results for three order heuristics resulting from sorting blocks in ascending, descending, and random order of the number of entities they contain on the CDDB dataset. The plots show the first occurrence of a pair $p_{i,j}$ at position (x, sim) when it is the x-th pair to be processed and the similarity computation yields a similarity sim. We cut off similarities below 0.5 for better readability. As is common in ER, we assume that the match

Dataset	Size(D1)	Size(D2)	Duplicates	Brute-force
AG Products	1354	3039	1104	4.11e06
CdDb	9763	//	299	4.77e07
Movies	27615	23182	22813	6.40e08

Table 1: Dataset characteristics (same as in [1]).

	Dataset	Redundancy	Comparisons	Recall
E1	AG Products	79%	1.9e7	1.0
E2	Movies	15%	6.5e7	0.98
E3	Movies	7%	9.7e6	0.96
E4	CdDb	36%	2.2e7	0.99
E5	CdDb	7%	4.6e5	0.99

Table 2: Experiment settings.

function determines $p_{i,j}$ to match if its similarity is above a similarity threshold. So the higher the similarity, the more likely it is we determine a match. Clearly, the ascending order of block sizes best mimics the desired behavior for our order heuristic. This experimentally validates the claim that in practice, sorting the block collection B in ascending order of its block sizes is suited to approximate the desired behavior on match probability.

2.3 Extension using scalable Bloom filters

We have seen in Section 2.1 that initializing BF requires setting both p_{max} and s, which are key in optimally setting the number of bits allocated to the Bloom filter. While s is in the order of sum of squares of individual block sizes, this is in practice a very loose upper bound for the expected number of *unique* keys to be inserted, especially when a high degree of redundancy can be expected. To avoid allocating unnecessary space to a BF and be less sensitive to variations of both the redundancy and match distribution in P, we explore how to extend our comparison cleaning algorithm with scalable Bloom filters. We provide relevant background on SBFs and refer readers to [2] for details.

A SBF is a list of Bloom filters $BF_0, \ldots, BF_n$. Initially, the list includes a single BF, denoted BF_0, with an associated initial capacity s_0 and maximum false positive probability p_0. More generally, each BF_i has an associated capacity s_i and a false positive probability p_i. Given BF_i the last BF in the list, a new key is inserted into BF_i. When BF_i becomes full (i.e., new keys cannot be inserted without exceeding p_i), a new BF_{i+1} is inserted in the list. Given a tightening ratio θ with $0 < \theta < 1$ and a growth ratio $\sigma \geq 1$, BF_{i+1} is set with $p_{i+1} = p_0\theta^i$ and $s_{i+1} = s_i\sigma$. Overall, the capacity of the SBF gradually increases as needed while the compound false positive probability p_{fp} is bounded by $p_{fp} \leq p_0 \frac{1}{1-\theta}$. This means that we do no longer have to set s upfront, assuming the worst case in Algorithm 1. We can choose a more conservative initial capacity $s_0 << s$ to improve space efficiency and extend it if needed. The additional parameters of SBF can be fixed to $\theta = 0.9$ and $\sigma = 2$, following the recommendation of [2].

3 EVALUATION

We experimentally validate the comparison cleaning approaches presented in this paper. We both perform a parameter sensitivity study and a comparative evaluation to baseline and state-of-the-art methods. Performance metrics are runtime (time), memory footprint (space), and quality (recall over the set of executable comparisons considering a ground truth file).

All algorithms were implemented in Java 1.8 as extensions of the JedAI library [13] that supports numerous state-of-the-art unstructured ER solutions. We ran experiments on an OpenStack virtualized server (16 processors at 2.30GHz, 50GB RAM).

Our experiments use data from three benchmark datasets commonly used to evaluate unstructured ER (e.g., in [8, 11]). Their characteristics are summarized in Table 1. They are publicly available at the JedAI webpage together with ground truth files.

Our evaluation relies on different experiment settings. Each setting varies in the block collection B input to the evaluated comparison cleaning techniques, obtained by varying datasets and steps preceding comparison cleaning. Table 2 summarizes the characteristics of five settings E_1 through E_5. Here, "Redundancy" reports the fraction of redundant pairs produced by B. "Comparisons" is the total number of pairs resulting from B, i.e., $|P|$. "Recall" reports the maximum possible recall that can be obtained based on B.

3.1 Parameter sensitivity

We study the effect of parameter variations on performance by varying the parameters for both BF and SBF as follows:

BF Capacity $s = |P|$ (see Comparison column in Table 2), $p_{max} \in \{0.1, 0.5, 0.8\}$

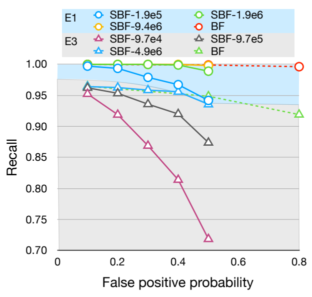
SBF Initial capacity $s_0 \in \{0.01|P|, 0.1|P|, 0.5|P|\}$ and $p_0 \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$

For all considered parameter settings, we studied their effect on time, space, and recall. Due to space constraints, we only report representative results on settings E_1 and E_3, two extreme cases with high and low redundancy. The remaining results are analogous and in line with our discussion.

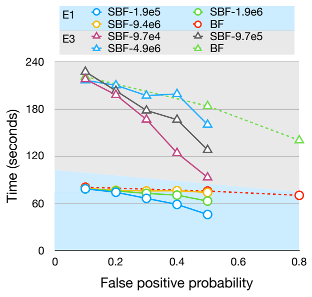
Figure 3(a) studies recall when varying the false positive probability for different configurations of the capacity in both E_1 and E_3. For setting E_1, we observe that recall is stable across all tested configurations. The reason is that E_1 has high redundancy, so even the most aggressive configuration setting for a SBF (low initial capacity, high initial false positive probability) has an acceptable low loss in recall because typically, it only prunes comparisons that are indeed unnecessary. In E_3, where we have low redundancy, an aggressive solution based on SBFs with a very small capacity and high false positive probability loses much more in effectiveness (25% total loss in recall) compared to a solution using an higher capacity and same false positive probability or a solution that uses a smaller false positive probability. This is due to the fact that a SBF with such aggressive configuration converges faster to the maximum compound false probability than the other solutions, resulting in more miss-matches.

Figure 3(b) shows runtime for the same configurations as the previous experiment. In E_1, the aggressive configuration mentioned before using SBF with very low initial capacity and high false positive probability is particularly efficient because it starts to remove non-redundant pairs earlier than the other solutions but in a point where the match probability is already very low. This positive effect on runtime decreases as the initial capacity of a SBF increases, until it eventually converges to the behavior of a BF. In E_3 and similarly in other settings with low to moderate redundancy, the behavior is similar, yet the slopes are more evident. This is due to the fact that in such settings, higher false positive probabilities prune more pairs that are not redundant (and thus pruned by any configuration).

Considering space (no graphs shown for conciseness), all configurations require less than 25MB of memory, making both SBFs and BFs space efficient data structures. Considering BFs, as expected by the formula in Section 2, the space decreases with



(a) E1-E3 recall



(b) E1-E3 time

Figure 3: Recall and runtime for parameter configurations

increasing p_{max}. Considering SBFs, we observe that required space is lower than for BFs only when the degree of redundancy is sufficiently high.

From our parameter sensitivity analysis, we observe that good performance across all settings is obtained when SBF capacity is set between 0.1 and 0.5 times $|P|$ (the size of a BF) coupled with a high/moderate false positive probability (0.3 to 0.5). As BFs do not suffer from efficiency or space degradation caused by SBF extensions, their false positive probability can be set to even higher values (as these high values kick in late in the process where typically, most matches have already been processed), for which we recommend the range between 0.5 and 0.8.

3.2 Comparative evaluation

Given our above conclusion, we now select two good configurations: BF* with $s = |P|$, $p_{max} = 0.5$ and SBF* with $s_0 = |P| * 0.5$, $p_0 = 0.5$). We compare them to two other comparison cleaning approaches. The first baseline uses a hash set (HS) instead of a BF. The second approach applies comparison propagation (CP) [9]. CP uses inverted indexes to avoid redundant comparisons. The inverted index is basically a hash-map where the keys are entity identifiers and the list of values associated to the key identifies the block indexes where the entity appears. Given a processing order of the blocks $b_1, b_2, ..., b_n$, two entities are compared in a block b_i only if the lowest common index of their associated block indexes is i. We again evaluate time, space, and recall. We do not consider meta-blocking [11] as a competitor because it is not integrable in Algorithm 1, which interleaves pairwise similarity computation with comparison cleaning.

Quality comparison. Both HS and CP yield the maximum possible recall (see Table 2) as they exclusively prune redundant comparisons. The possibly lower recall of our Bloom filter based solutions is attributed to miss-matches caused by false positives (see Section 2). Throughout all settings E_1 through E_5, the recall of BF* (SBF*) does not reduce by more than 4%.

Time comparison. Table 3a shows the runtime for the compared approaches in all setings. We further report on the runtime of ER without any comparison cleaning in the "Nothing" column. When high redundancy occurs like in E1, all the comparison cleaning solutions outperform the approach that compares all redundant pairs (Nothing). Also, we observe that HS and CS consistently have comparable runtimes. In scenarios with low redundancy, HS and CS are worse than (or comparable to) Nothing, as their overhead time to find redundant pairs is higher than the time avoided by not comparing them (see E_3 and E_4). Both BF* and SBF* outperform the baseline approaches thoughout all settings, improving runtime between 7% and 31% over the best competitor.

	BF*	SBF*	CP	HS	Nothing
E1	75	74	95	81	540
E2	1235	1161	1557	1566	2038
E3	183	160	255	218	221
E4	77	71	103	112	120
E5	4	4	5	5	4

(a) Time (in seconds)

	BF*	SBF*	CP	HS	Dataset
E1	6	3	4	223	3
E2	23	38	55	5063	40
E3	3	5	39	785	40
E4	7	13	12	1281	6
E5	<1	<1	5	39	6

(b) Space (in MB)

Table 3: Results for time and space

Space comparison. Table 3b shows space required by the compared comparison cleaning approaches. Unsurprisingly, HS performs worst. CP performs better because the size of its inverted index depends on the number of entity descriptions of the datasets, the number of blocks in B, and the level of redundancy of entities in multiple blocks. We observe that the space used by BF* and SBF* further usually improves on CP (up to 90%) while, at the same time, improving efficiency and maintaining high quality.

4 CONCLUSIONS

We proposed a novel approach comparison cleaning approach in entity resolution based on Bloom filters that removes both redundant and superfluous comparisons to improve efficiency. The technique relies on a validated heuristic that pairs a decreasing match probability with an increasing false positive probability. We further present an extension to our approach, using scalable bloom filters. Our experimental validation demonstrates that our approach outperforms state-of-the art algorithms in space and time, while maintaining high effectiveness.

ACKNOWLEDGMENTS

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program 'Services Computing'.

REFERENCES

[1] [n.d.]. JedAI. https://github.com/scify/JedAIToolkit/, accessed 29.11.2019.
[2] P. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. 2007. Scalable bloom filters. *Inform. Process. Lett.* 101 (2007), 255–261.
[3] A. Broder and M. Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet mathematics* 1 (2004), 485–509.
[4] V. Efthymiou, G. Papadakis, G. Papastefanatos, K. Stefanidis, and T. Palpanas. 2017. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Information Systems* 65 (2017), 137–157.
[5] V. Efthymiou, K. Stefanidis, and V. Christophides. 2016. Benchmarking blocking algorithms for web entities. *IEEE Transactions on Big Data* (2016).
[6] A. Elmagarmid, P. Ipeirotis, and V. Verykios. 2007. Duplicate record detection: A survey. *TKDE* 19 (2007), 1–16.
[7] D. Karapiperis, A. Gkoulalas-Divanis, and V.S. Verykios. 2018. Summarization Algorithms for Record Linkage.. In *EDBT*.
[8] G. Papadakis, E. Ioannou, C. Niederée, and P. Fankhauser. 2011. Efficient entity resolution for large heterogeneous information spaces. In *WSDM*.
[9] G. Papadakis, E. Ioannou, C. Niederée, T. Palpanas, and W. Nejdl. 2011. Eliminating the redundancy in blocking-based entity resolution methods. In *JCDL*.
[10] G. Papadakis, E. Ioannou, T. Palpanas, C. Niederee, and W. Nejdl. 2013. A blocking framework for entity resolution in highly heterogeneous information spaces. *TKDE* 25 (2013), 2665–2682.
[11] G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. 2014. Meta-blocking: Taking entity resolutionto the next level. *TKDE* 26 (2014), 1946–1960.
[12] G. Papadakis, J. Svirsky, A. Gal, and T. Palpanas. 2016. Comparative analysis of approximate blocking techniques for entity resolution. *PVLDB* 9 (2016), 684–695.
[13] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, and M. Koubarakis. 2018. The return of jedAI: end-to-end entity resolution for structured and semi-structured data. *PVLDB* 11 (2018), 1950–1953.
[14] G. Simonini, S. Bergamaschi, and H.V. Jagadish. 2016. BLAST: a loosely schema-aware meta-blocking approach for entity resolution. *PVLDB* 9 (2016), 1173–1184.
[15] D. Vatsalan, P. Christen, and E. Rahm. 2016. Scalable privacy-preserving linking of multiple databases using counting bloom filters. In *ICDMW*.

Short paper

Disco: Efficient Distributed Window Aggregation

Lawrence Benson[1] Philipp M. Grulich[2] Steffen Zeuch[2,3] Volker Markl[2,3] Tilmann Rabl[1]

[1]Hasso Plattner Institute, University of Potsdam [2]Technische Universität Berlin [3]DFKI GmbH

{lawrence.benson,tilmann.rabl}@hpi.de,{grulich,steffen.zeuch,volker.markl}@tu-berlin.de

ABSTRACT

Many business applications benefit from fast analysis of online data streams. Modern stream processing engines (SPEs) provide complex window types and user-defined aggregation functions to analyze streams. While SPEs run in central data centers, wireless sensors networks (WSNs) perform distributed aggregations close to the data sources, which is beneficial especially in modern IoT setups. However, WSNs support only basic aggregations and windows. To bridge the gap between complex central aggregations and simple distributed analysis, we propose Disco, a distributed complex window aggregation approach. Disco processes complex window types on multiple independent nodes while efficiently aggregating incoming data streams. Our evaluation shows that Disco's throughput scales linearly with the number of nodes and that Disco already outperforms a centralized solution in a two-node setup. Furthermore, Disco reduces the network cost significantly compared to the centralized approach. Disco's tree-like topology handles thousands of nodes per level and scales to support future data-intensive streaming applications.

1 INTRODUCTION

Modern business use-cases often require the analysis of high-volume data streams. To efficiently process such large amounts of data, stream processing engines (SPEs) provide complex windows and aggregations. But to perform these complex aggregations, state-of-the-art SPEs such as Apache Flink [2], Apache Spark Streaming [16], and Apache Storm [12] require the data to be collected in a single data center. Current research approaches to improve the performance of window aggregation, such as Scotty [13, 14], Cutty [3], and Pairs [7] also require data to be centrally available. However, efficiently analyzing an ever-increasing data volume requires streams to be processed on multiple nodes as the central collection of data quickly becomes a limiting factor in processing latency and network cost [17]. While SPEs require data to be available centrally, wireless sensor networks (WSN) perform aggregations on multiple nodes close to the data sources. WSNs drastically reduce the network costs by actively leveraging the distribution of incoming data streams. However, previous work on WSNs, such as TAG [10] or Cougar [15], provides only simple aggregation operations on basic window types.

To bridge the gap between complex central aggregation in SPEs and distributed basic aggregation in WSNs, we propose Disco, a distributed complex window aggregation approach. While SPEs processes incoming data in one central stream, Disco leverages the distribution of incoming data streams to perform distributed window creation and aggregations closer to the sources. We propose multiple strategies to efficiently merge distributed windows and their respective aggregates. With this approach, we benefit from both the reduced network cost of WSNs and the complex analysis of SPEs.

In this paper, we make the following contributions: 1.) We propose Disco, an approach for distributed complex window aggregation that does not require raw data collection on single nodes. 2.) We introduce window merging strategies to distribute the processing of common window types while maintaining correct aggregation semantics. 3.) We evaluate Disco and show that the throughput of our approach scales linearly with the number of nodes as well reduces the network cost drastically compared to state-of-the-art central window aggregation techniques.

The rest of the paper is structured as follows. In Section 2, we present the foundations of window aggregation that Disco is built upon. In Section 3, we present our distributed aggregation approach for arbitrary window types and aggregation functions. Before concluding, we present our evaluation of these distributed aggregation concepts implemented in our prototype in Section 4.

2 BACKGROUND

In this section, we present the background of distributed window aggregation. We first present the two window types on which we build, as well as two classes of aggregation functions. We then introduce stream slicing for efficient window aggregations.

Window Types. For this work, we distinguish between two types of windows, *context-free* and *context-aware* windows [9, 14]. In this case, context refers to the state that is required to calculate the start- and end-bounds of all the windows up to time t.

Context-free (CF) windows do not require any state and their bounds can be computed statically. For a time t, all window bounds can be computed without processing a single event. Examples of CF windows are tumbling and sliding windows [1, 9].

Context-aware (CA) windows require some kind of state to determine the window bounds, i.e., the windowing function needs to see either previous or future events to know when a window ends. An example of a CA window is a session window [1]. The windowing function needs to see future events after t in order to know that a session has terminated at t.

Aggregation Functions. For distributed aggregation, we distinguish between two classes of aggregation functions, *decomposable* and *holistic* (or *non-decomposable*) [4, 5]. Decomposable functions are aggregations for which the aggregation can be performed on subsets of the data and merged afterwards, e.g., $\texttt{sum}\{1, 2, 3, 4\} = \texttt{sum}\{1, 2\} + \texttt{sum}\{3, 4\}$. We call the aggregation values on subsets of the data *partial aggregates*. Other examples of decomposable functions are `avg`, `count`, and `max`.

Holistic functions are all functions that are not decomposable, i.e., they require all values to perform the correct aggregation. There is no partial aggregation state for these functions. Examples of holistic functions are `median` and `quantiles`.

Stream Slicing. In Disco, we apply general stream slicing [14] to efficiently aggregate windows. Stream slicing divides windows into non-overlapping slices. The slices then store either *i)* a running, partial aggregate or *ii)* the individual events, depending on the aggregation function. The final aggregation result is computed from the individual slices that fall in the window's range. For commutative, decomposable aggregations only partial aggregates are needed, leading to a large memory reduction [14].

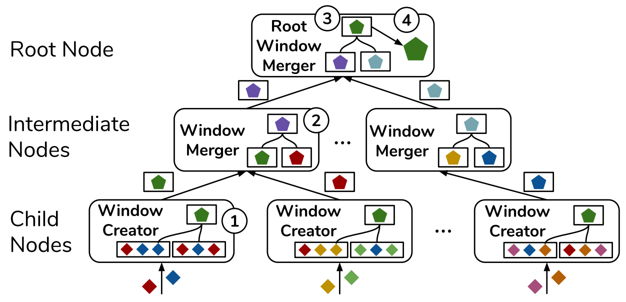


Figure 1: Disco's Architecture.

Each event belongs to exactly one slice, thus avoiding redundant storage and computation compared to other windowing techniques such as *Buckets* [9]. Also, slices represent logical event groups that can be transferred between nodes more efficiently than individual events. We refer to previous work [8, 14] for a detailed description of stream slicing and its advantages.

3 THE DISCO APPROACH

Efficiently aggregating windowed streaming data is a core task for modern SPEs. State-of-the-art window aggregation techniques require data to be available centrally on a single node. We introduce Disco, an approach for distributed window aggregation, which overcomes the central collection of raw events. Disco processes the incoming streams independently of each other and creates independent windows and aggregations on multiple nodes. These windowed aggregations are then merged to produce the final aggregation result. Disco's components communicate in a tree-like structure, in which leaf nodes (called *child nodes* in Disco) create independent windows and inner nodes (called *intermediate nodes*) merge them. This independence allows Disco to scale by adding new child nodes to process more events and by adding intermediate nodes to process more child nodes.

In the remainder of this section, we provide an architecture overview of Disco (Section 3.1) followed by a detailed description of Disco's window merging strategies (Section 3.2) and its distributed computation of aggregation functions (Section 3.3).

3.1 Architecture

We show Disco's architecture in Figure 1. Disco consists of three main components, the *root node*, the *intermediate nodes*, and the *child nodes*. These individual components communicate in a tree-like structure, where each node, except the root, communicates with exactly one parent node and each node can have many children, limited only by its network and processing power. Each event stream connects to one child node, which in turn receives the raw events. The *Window Creator* on each child node creates slices and windows according to user-specified queries ①. The child nodes then pass on partial aggregates for each window to their parent. On the intermediate nodes, a *Window Merger* then merges incoming windows according to the strategies presented in Section 3.2 ②. On the root node, the *Root Window Merger* finally merges all partial windows ③ and performs the final aggregation to retrieve the result for a given window ④. As the nodes process events and windows independently, there can be an arbitrary number of child nodes and intermediate levels, depending on the scale that the system requires.

Our implementation relies on the Scotty library[1] [14] for the window and slice creation step ①. Because Scotty requires data to be available centrally, we extend its windowing concepts to support distributed windows.

[1]https://github.com/TU-Berlin-DIMA/scotty-window-processor

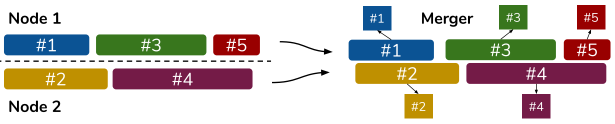


Figure 2: No merging for unique windows.

3.2 Distributed Window Merging

Depending on the window characteristics (i.e., type and measure), Disco selects the appropriate window creation and merging strategy. In particular, Disco differentiates between the two window types: *context-free* and *context-aware* (as described in Section 2). Disco processes incoming event streams and resulting windows independently on child nodes. In order to create the correct *global* window result over all streams, Disco merges the individual windows. To this end, Disco has to determine which partial windows belong to the same global window. We present three merging strategies for distributed window aggregation in Disco, *i)* unique window merging, *ii)* context-free merging, and *iii)* context-aware merging. For our explanation, we assume there are k independent child nodes producing windows and one *Root Window Merger* that merges the windows to produce a global result.

Window Merging Operations. In order to perform distributed aggregations, we use the three operations: *lift*, *combine*, and *lower* to model our aggregation functions [11]. *lift* converts a single input value x to an aggregation into a partial aggregate, e.g., $x = 5 \longrightarrow \langle \text{sum: } 5, \text{count: } 1\rangle$ for avg. *combine* merges two partial aggregates into a new partial aggregate, e.g., $\langle 5, 1\rangle \oplus \langle 7, 2\rangle \longrightarrow \langle 12, 3\rangle$. *lower* converts the partial aggregate to the final result, e.g., $\langle 12, 3\rangle \longrightarrow 12/3 = 4$.

To merge multiple windows, we extend the *slice-merge* operation [14] for windows. The *window-merge* operation $\uplus$ takes two windows w_1 and w_2 and creates a new window w with a merged aggregate state ($w.state = \texttt{combine}(w_1.state, w_2.state)$), as well as the according bounds ($w.start = min(w_1.start, w_2.start)) \land w.end = max(w_1.end, w_2.end)$).

Unique Window Merging. The first merging strategy that Disco uses is based on *unique* windows. A unique window is a window for which data is present in only one stream. With no matching data, there are no further windows to merge the unique window with. This strategy is applicable to queries where there are no overlapping keys on different nodes and the query is defined on individual keys only. We show this in Figure 2. A Window Merger that receives a unique window w, emits w unchanged and the Root Window Merger calls $\texttt{lower}(w.state)$ to retrieve the final aggregation result. For unique windows, we benefit from a distributed aggregation compared to central processing, as we do not need to transfer raw data to the root node and we distribute the aggregation cost across all child nodes. An example for a unique window is a smart home setting in which we calculate the average temperature per house.

Context-Free Window Merging. Disco determines to which global window a partial context-free window belongs by its start and end bounds. As the bounds are statically computed and require no context, they are identical on all nodes. Thus, equivalent partial windows have equal start and end bounds. We show this for a sliding window on two nodes in Figure 3. For each global window w with $w.start = x$ and $w.end = y$, we collect the k matching partial windows out of all windows W, $\{w_i \in W \mid w_i.start = x \land w_i.end = y\}$ and merge them into a

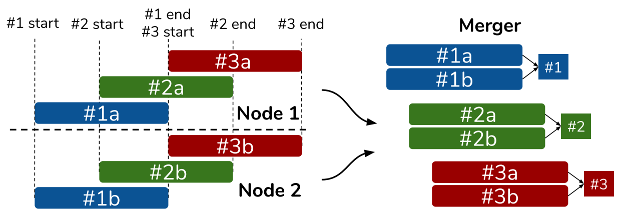


Figure 3: Merging distributed context-free windows.

new global result window $w = w_1 \uplus w_2 \uplus \ldots \uplus w_k$. The global result is then calculated by calling lower($w.state$) in the *Root Window Merger*. An example of a context-free window is a smart home setting in which we calculate the average temperature of all houses for the last hour.

Context-Aware Window Merging. Unlike context-free windows, the creation of context-aware windows cannot generally be distributed. As CA windows require some form of context, Disco cannot make assumptions about whether this context needs to be viewed centrally or can be viewed independently. If the window requires a global view of the context, we cannot create windows independently on multiple nodes. An example of such a global view is a count-based window, which uses a global event count to compute the window bounds.

However, certain CA windows, such as session windows, can benefit from a distributed aggregation. Session windows terminate if the event stream contains a period of inactivity (gap). In Disco, we define a period of inactivity across all distributed streams as a *global window gap*. A timestamp t is *active* in the set of all windows W if $\exists w \in W \;:\; w.start < t \land w.end > t$. A timestamp t is thus *inactive* if it is not active, i.e., there are no windows spanning across t. A period of inactivity is bound by two timestamps t_{start} and t_{end} between which there are no active timestamps. Disco leverages this knowledge to merge session windows in a distributed manner. Session windows are created independently on child nodes and the *Window Merger* then checks if there is an overlap between received windows. If an incoming window w_{in} overlaps with an existing window w_{ex}, it is merged to produce a new window $w_{new} = w_{ex} \cup w_{in}$ in their place. Two windows overlap if, and only if $w_{old}.start \leq w_{new}.end + gap \land w_{new}.start \leq w_{old}.end + gap$. For session windows, an overlap between two windows needs to take the session gap into account as events within the gap would cause the session to continue in a global stream. An arriving window can merge multiple windows, e.g., if it is a very long window or it closes the gap between two currently non-overlapping windows. We show this in Figure 4, where window #1b combines the windows #1a and #1c because it overlaps with both of them. An example for a context-aware window is a smart home setting in which we calculate the average temperature of certain houses as long as they are actively heating.

3.3 Distributed Window Aggregation

While the window type generally decides whether a window is computed centrally or distributed, the class of aggregation function determines which data needs to be transferred between nodes. For decomposable functions (e.g., `sum`, `avg`, `count`), Disco transfers partial aggregate values. The partial aggregates are then merged by the window merger.

As holistic functions require all data for a correct aggregation, Disco needs to transfer all events to the root. However, Disco does not send individual events but *slices*. For holistic functions, the slices contain the raw events and Disco uses these slices to send logical event groups between nodes. As each slice is immutable and uniquely identifiable, Disco sends it only once, regardless of how many windows it appears in. As all holistic functions require central processing, the root node is the only node that needs to store slices to performs aggregations on the events. Intermediate nodes do not process the data in the slices and thus, do not need to store the slices. Intermediate nodes only keep track of which slices they have sent to avoid duplicate transmission.

In summary, local and context-free windows can generally be distributed, while context-aware windows require certain data characteristics for distribution. If a window or aggregation function requires a global ordering of events (e.g., count-based windows), Disco cannot distribute the window creation across multiple nodes but requires central processing. For holistic aggregations, Disco transfers slices between nodes and for decomposable functions, it sends only partial aggregate values.

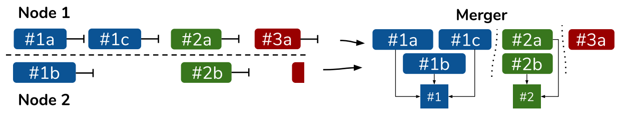


Figure 4: Merging distributed context-aware windows.

4 EVALUATION

In this section, we experimentally evaluate Disco's scalability (Section 4.1) and its network efficiency (Section 4.2). We choose `avg` and `median` as representative decomposable respectively holistic aggregation functions, as they show similar characteristics to other functions of the same group. We execute our experiments on a cluster consisting of 20 nodes. Each node has an AMD Opteron 6128 @ 2.0 GHz with 16 physical cores and 32 GB of RAM. All nodes are running Ubuntu 18.04.3 LTS, OpenJDK 12.0.2 64 bit, and are connected via Gigabit LAN. Furthermore, Disco and our experiments are available on GitHub[2].

4.1 Scalability

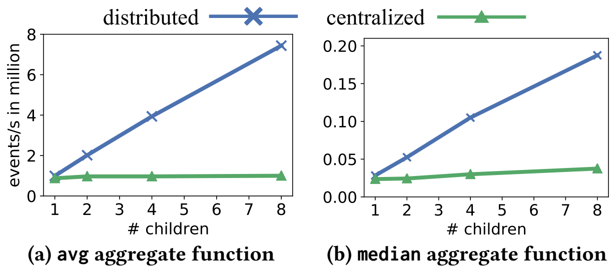
In this experiment, we compare the scalability of Disco's distributed window aggregations to a centralized implementation.

Workload. We evaluate the throughput of a one-second tumbling window query for a decomposable as well as a holistic aggregate function. We define throughput as sustainable if the system can handle incoming events without an ever-increasing backlog at the sender [6]. In the centralized implementation, the child nodes forward the raw events without processing.

Results. In Figure 5, we observe that Disco scales nearly linearly with the number of child nodes for both aggregation types. For decomposable aggregation functions (e.g., `avg` in Figure 5a), each child node can process around one million events per second. In the centralized approach, the root node becomes the bottleneck, as it processes all raw data centrally. As a consequence, it is limited by the single node performance (~1 million events/s). In contrast, Disco's root node receives only one partial window per second per child node instead of all raw events. Thus, Disco allows for linear scaling as the majority of the window aggregations is processed independently across all child nodes.

Furthermore, Disco provides scalability for holistic window aggregation functions (e.g., `median` in Figure 5b). Overall, the `median` aggregation performs significantly worse than the `avg` aggregation as it requires the centralized accumulation of all

[2]https://github.com/hpides/disco



(a) avg aggregate function (b) median aggregate function

Figure 5: Scalability of different aggregations in Disco.

events at the root node. As a consequence, the centralized approach is limited to a throughput of 0.025 million events/s. In contrast, Disco also performs window creation and merging of holistic states on multiple nodes in parallel. Thus, the root node only receives up to eight windows per second in this experiment. Consequently, Disco is able to scale nearly linearly for the holistic `median` function as the final `median` calculation on the root does not become a bottleneck for Disco.

Further scalability experiments show that the root node can process thousands of holistic and tens of thousands of decomposable windows per second before it becomes a bottleneck. Thus, Disco scales to support thousands of child nodes.

Summary. In this experiment, we showed that Disco scales linearly with the number of nodes for both decomposable as well as holistic aggregation functions. Even for functions that require central aggregation but can be windowed independently, Disco outperform centralized approaches significantly.

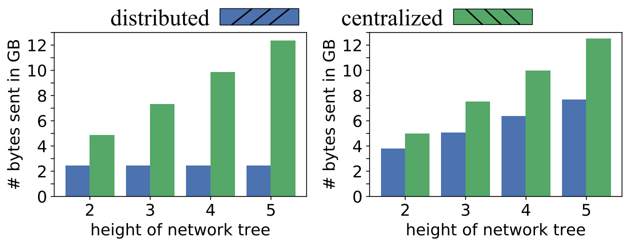
4.2 Network Cost

In this experiment, we evaluate the overall network costs of distributed and centralized aggregations.

Workload. We scale the height of the network topology to evaluate the network impact of the number of hops between child nodes and the root. We measure the total bytes sent for a one-second tumbling window query on 100 million events.

Results. Overall, we observe in Figure 6 that Disco has a significantly lower network footprint compared to a centralized approach. For decomposable distributed aggregations, the network consumption of the central approach scales linearly with the height of the network tree (see Figure 6a). In contrast, the network consumption of Disco stays nearly constant. Disco is independent of the height of the network topology, as all raw events are sent exactly once from a sensor to a child node. Beyond that, the intermediate nodes only exchange one partial aggregate and some window metadata per second. This causes an additional network traffic of only 1 MB per node level. As a consequence, a network height of five levels causes up to 6x less network traffic (2GB) compared to the centralized approach (12GB).

For holistic aggregations (see Figure 6b), Disco needs to send all raw events to the root node. Already for a tree of height two, we observe that Disco sends fifty percent more data than for decomposable aggregations. However, by sending events as groups of slices instead of individually, we avoid the additional TCP overhead of a purely centralized approach. While all single events are sent at each level for a centralized approach, slices become the smallest message unit in the distributed approach. In this scenario, we can save 50% per level compared to the centralized approach, which has a large effect once the tree becomes significantly deep. For five levels, we send only 7.5 GB of distributed slice data compared to 12.4 GB centralized single events.



(a) avg aggregate function (b) median aggregate function

Figure 6: Network cost for different aggregations in Disco.

Summary. Sending raw events is the dominant factor of the network cost. In Disco, we avoid sending individual events between nodes, which results in a high reduction of network traffic. For decomposable functions, Disco completely avoids sending individual events, which drastically reduces the network load. Even for holistic functions, we reduce TCP overhead by combining and sending events in slices.

5 CONCLUSION

In this paper, we present Disco, a distributed complex window aggregation approach. Disco combines the distributed data aggregation concepts from wireless sensor networks with the complex windowing and aggregation semantics from modern stream processing engines. This allows us to reduce network traffic while providing efficient aggregations on arbitrary windows. Our evaluation shows that Disco's throughput scales linearly and that Disco significantly reduces network costs compared to a centralized approach. With the ever increasing number of sensors in the IoT, Disco lays a foundation for efficient, application transparent, distributed stream processing.

Acknowledgements: Work partially funded by DFG Priority Program "Scalable Data Management for Future Hardware" (MA4662-5) and the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (01IS18025A and 01IS18037A)

REFERENCES

[1] Tyler Akidau et al. 2015. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB* 8, 12 (2015), 1792–1803.
[2] Paris Carbone et al. 2015. Apache Flink(TM): Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 38, 4 (2015), 28–38.
[3] Paris Carbone et al. 2016. Cutty: Aggregate Sharing for User-Defined Windows. In *CIKM*. 1201–1210.
[4] Jim Gray et al. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *DMKD* 1, 1 (1997), 29–53.
[5] Paulo Jesus et al. 2015. A Survey of Distributed Data Aggregation Algorithms. *IEEE Communications Surveys & Tutorials* 17, 1 (2015), 381–404.
[6] Jeyhun Karimov et al. 2018. Benchmarking Distributed Stream Processing Engines. In *ICDE*. 1507–1518.
[7] Sailesh Krishnamurthy et al. 2006. On-the-Fly Sharing for Streamed Aggregation. In *SIGMOD*. 623–634.
[8] Jin Li et al. 2005. No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *ACM SIGMOD Record* 34, 1 (2005), 39–44.
[9] Jin Li et al. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*. 311–322.
[10] Samuel Madden et al. 2002. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*. 131–146.
[11] Kanat Tangwongsan et al. 2015. General incremental sliding-window aggregation. *PVLDB* 8, 7 (2015), 702–713.
[12] Ankit Toshniwal et al. 2014. Storm@twitter. In *SIGMOD*. 147–156.
[13] Jonas Traub et al. 2018. Scotty: Efficient window aggregation for out-of-order stream processing. In *ICDE*. 1304–1307.
[14] Jonas Traub et al. 2019. Efficient Window Aggregation with General Stream Slicing. In *EDBT*. 97–108.
[15] Yong Yao et al. 2002. The cougar approach to in-network query processing in sensor networks. *ACM SIGMOD Record* 31, 3 (2002), 9–18.
[16] Matei Zaharia et al. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*. 423–438.
[17] Steffen Zeuch et al. 2020. The NebulaStream Platform: Data and Application Management for the Internet of Things. In *CIDR*.

Short paper

Explaining Missing Query Results in Natural Language

Daniel Deutch
Tel Aviv University
danielde@post.tau.ac.il

Nave Frost
Tel Aviv University
navefr@mail.tau.ac.il

Amir Gilad
Tel Aviv University
amirgilad@mail.tau.ac.il

Tomer Haimovich
Tel Aviv University
tomerh@mail.tau.ac.il

ABSTRACT

We propose in this paper a novel approach for explaining query non-answers in Natural Language within the context of Natural Language Interfaces to Databases (NLIDBs). Such interfaces allow non-expert users to pose queries over an underlying database; our goal is to further allow users to ask why some results that they have expected to see, are missing from the output. In a nutshell, our approach is to "marry" NLIDBs with an existing model for explaining missing query results by pinpointing the last query operator that is "responsible" for the missing result. We observe that one can often trace the parts of the original NL question that correspond to these operators. This paves the way for intuitive explanations of the non-answers, that are based on highlighting the relevant parts of the question. Our architecture is generic and is not coupled with a specific NLIDB, and our solution yields clear explanations in interactive speed.

```
return authors who published papers in database
conferences after 2005
```

(a) NL Question

```
SELECT DISTINCT author.aname
FROM author, writes, pub, conf,
WHERE conf.domain = `databases'
    AND pub.pyear > 2005
    AND author.aid = writes.aid
    AND writes.pid = pub.pid
    AND pub.cid = conf.cid
```

(b) SQL Query

```
return authors who published papers in database
conferences after 2005
```

(c) Word highlight explanation for "Why not Marge?"

Figure 2: NL query, SQL translation, and our why-not explanations

1 INTRODUCTION

Natural Language (NL) interfaces to database systems are often used as easy-to-understand gateways for accessing complex databases [1, 8, 13]. The rise of NL interfaces allows non-experts to access and query complex databases, without writing formal queries or understanding execution plans.

Yet often, the answers returned by such queries do not quite match the expectations of the users who formed them. Users are then faced with the problem of understanding the gap between their expectations and the result. Previous work has dealt with presenting the users the reason for the presence of a certain tuple in the result set in a manner that does not require technical proficiency [5, 6]. When users ask about a *missing* tuple, however, a different form of explanation is required. Explaining missing tuples, or non-answers, is termed why-not provenance and has been the focus of multiple previous works [2, 3, 10]. Such information is crucial for understanding the result, debug and improve the query and/or the input database. However, all of these works have provided this information in the form of *internal representation*, not suitable for non-experts.

Rel. *author*

aid	aname	oid
1	Marge	1
4	Bart	1

Rel. *pub*

pid	cid	jid	ptitle	pyear
5	11	-	"Paper x"	2001

Rel. *conf*

cid	cname	dname
11	DBDonut	databases

Rel. *writes*

aid	pid
1	5

Figure 1: MAS database instance

Example 1.1. Assume the NL question depicted in Figure 2a which was correctly translated by an NL interface to a formal query (shown as SQL in Figure 2b) and executed on the database instance depicted in Figure 1. The user is presented with a result set, but is surprised to see that Marge is missing. A why-not explanation for this missing tuple could be the predicate `pub.pyear > 2005` filtering Marge out, or a modified formal query without this predicate. However, understanding such explanations requires, at the very least, SQL knowledge.

In this paper, we aim to provide explanations for non-answers through natural language. The setting is unique in the sense that the query is given in NL and the user is not familiar with the technical details of the query execution, and the explanation should be tailored to bridge this knowledge gap. We rely on the frontier picky model [3] to provide explanations to non-expert users. Our main observation is that, if we use this model, when the original query was given in NL, we can in many cases trace back the responsible query operator to the part of the NL query that corresponds to it.

Example 1.2. For the NL query in Figure 2a, the SQL in Figure 2b, and the why-not query "why not Marge?". If we employ the mapping from the words in the NL query to the SQL one we can find that the words "after 2005" are connected to the operator `pub.pyear > 2005`, and reveal that these words in the NL query caused the removal of the result Marge.

To the best of our knowledge, presenting why-not provenance to non-experts was not previously studied. This form of explanations is fundamentally different from existing models that show SQL operators or other technical representations, as it allows users without technical knowledge to understand the gap between their expected result and the one they received. We claim that such explanations are of even greater importance in the context of NLIDBs, because of the cumulative errors that arise in such systems. In this setting, users have to specify their intent in a form of an NL sentence; a failure to specify certain conditions or a too specific sentence might result in tuple loss, simply because the user performed an error in the NL formulation. Since the user has no means of viewing or understanding the SQL query, this error may go unnoticed.

The solution is based on word highlighting in the original sentence form: we highlight the reason for the missing tuples, manifested as one or more words in the original NL query.

Example 1.3. In our running example, an operator-based why-not model would return the selection operator filtering tuples before 2005 as the explanation, i.e. `pub.pyear > 2005`. Using our system, the user will be shown her original NL query, with an emphasis on “after 2005" as the relevant words that caused this absence (see Figure 2c). This enables the user to better understand the query, validate the translation process and the credibility of the database, and reformulate her NL query accordingly.

We have implemented our solution and demonstrated it [7]. We now provide an experimental evaluation, showing multiple use-cases where the system provides useful explanations and showing that generating such explanations does not incur substantial time cost compared to the other steps in the computation.

Related Work: NLIDBs are aimed at bridging the gap between database systems that use formal query languages such as SQL for interaction, and users who are not experts in forming formal queries, yet posses domain knowledge [1, 8, 13, 16]. These interfaces allow users to form questions in English, and be presented with a set of results which satisfy the query. The whole process of converting the NL question to a formal SQL query can be treated by the users as a black box. Explaining query results that were returned in NL has been the focus of previous work [4].

There are multiple approaches and models for explaining why a certain piece of information that is expected to be returned from a query was actually discarded. We can broadly divide the approaches by their type.

(1) *Operator-based explanation* models (e.g. [2, 3]) aim to provide one or more query operators that are responsible for omitting a tuple in the query execution process.
(2) *Query modification* models (e.g. [9, 17]) try to broaden the given query to include the missing tuple. Depending on the model and the query, the change to the query may be as minor as replacing a constant, or even modifying the query completely by joining other tables etc.
(3) *Tuple modification / generation* models (e.g. [10, 11]) create or modify factoids in a way that will ensure that the required tuple will be returned from the query evaluation.

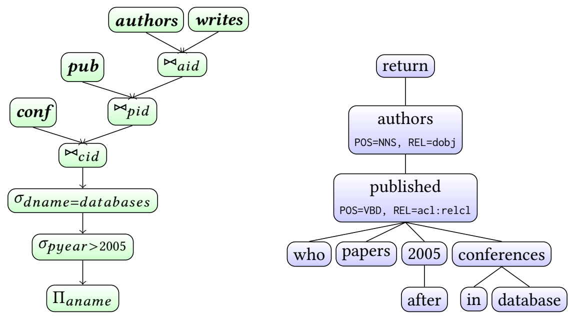
We focus here on explaining non-answers using the frontier picky model showing the “last” responsible operator [3]. To the best of our knowledge, no previous work in this field has dealt with the challenge of explaining missing tuples to non-expert users.

2 MODEL

In this section we review necessary notions in Natural Language Processing, formal queries and relevant provenance models.

From Natural Language to Formal Queries: This work relies on an NL interface named `NaLIR` [13] for translating English questions to SQL queries. The translation utilizes a data structure called *query dependency tree*, designed for conveying the relations between words in a sentence and their syntactic roles. A dependency tree $T = (V, E, L)$ is a node-labeled tree where labels consist of two components, as follows: (1) Part of Speech (*POS*): the syntactic role of the word [12] ; (2) Relationship (*REL*): the grammatical relationship between the word and its parent in the dependency tree [14].

After the translation phase, an SQL query Q is being generated along with a *query plan*. The query plan is a directed tree $T_Q = (V_Q, E_Q)$, where V_Q is the set of query operators in Q and the database table names which Q takes as input, and E_Q is a set of directed pairs (u, v). Table names form the leaves of T_Q and an edge (u, v) indicates that during the evaluation of Q over the database, all tuples outputted by the operator u are inputted to the operator v. Finally, the output implied by T_Q is the output of Q. Importantly, `NaLIR` maps the words in the NL query to their respective operators in the generated query plan. Reversing this mapping and augmenting it allows us to map why-not provenance back to NL.



(a) Query Plan **(b) NL query dependency tree**

Figure 3: Query plan and dependency tree

Definition 2.1. Given an NLIDB, a dependency tree $T_d = (V_d, E_d, L_d)$ and a query plan $T_Q = (V_Q, E_Q)$, a *operator-to-word mapping* is a partial function $M_{ops} : V_Q \rightarrow V_d$, where M_{ops} is the reverse of the mapping created by the NLIDB during the mapping from NL query to SQL.

Example 2.2. Consider the MAS database instance in Figure 1, and the NL query in Figure 2a. `NaLIR` parses this sentence to create the dependency tree shown in Figure 3b and returns the SQL query shown in Figure 2b; its query evaluation plan depicted in Figure 3a. The operator-to-word mapping, M_{ops}, includes the mappings: $M_{ops}[\sigma_{pyear>2005}]$ = “2005”, $M_{ops}[\sigma_{dname=databases}]$ = “*database*”, and $M_{ops}[\Pi_{aname}]$ = “*authors*”.

Why-not provenance: We start by defining a *why-not question WNQ*. Intuitively each hypothetical output tuple of Q that matches the criteria of WNQ is a tuple whose non-existence in the answer we wish to explain.

Definition 2.3. Given an database instance D and an SQL query Q, a why-not question WNQ is an SQL query such that (1) Q and WNQ have the same result schema, and (2) $Q(D) \cap WNQ(D) = \emptyset$.

The why-not question might be more complex than the initial query, as long as the query it represents has the same result schema. This allows the user to pose constraints on other attributes as well.

Example 2.4. Reconsider the database instance in Figure 1 and the SQL query in Figure 2b. Assume the user expects to see the author Marge in the result set, and surprised when she is not a part of the answer. The user may form a why-not question, “Why not Marge?", which would be translated by `NaLIR` to the query:

```
SELECT DISTINCT author.aname FROM author
WHERE author.aname = 'Marge'
```

The query is over the projected attribute of the original query (author names), and that the author Marge is returned by WNQ.

Some why-not provenance models allow users to define questions over attributes not in the result schema. We focus here on the subset of queries over the projected attributes, as we think that it is the most natural approach for posing why-not questions.

We will focus on *frontier picky* model [3]. This model for why-not provenance focuses on explaining non-answers using a single query operator. Intuitively, the last operator to contain in its input a tuple matching the why-not predicate. The motivation for explaining tuple loss with the frontier picky operator is that the answer is compact, easy-to-understand and provides a real value in the sense that it is necessary to modify the returned operator in order to include the tuple in the result list.

Definition 2.5 (adapted from [3]). Given a database D, a query Q with its query plan $T_Q = (V_Q, E_Q)$, a WNQ, and a tuple $t \in WNQ(D)$, a *picky operator* w.r.t. t is a node $v \in V_Q$ that gets t (or a predecessor of t in the evaluation process) through one of its incoming edges, as its input, and does not output it through its outgoing edge. An operator $v \in V_Q$ is *frontier picky* if:

(1) v is a picky operator for at least one tuple in $WNQ(D)$.
(2) There is no tuple $t \in WNQ(D)$ in the input of any operator $u \in V_Q$ such that u is a successor of v in T_Q.

According to this model, the answer to a why-not question is the frontier picky operator. Notice that the frontier picky operator might be different even if for the same query, evaluated with different query plans, as it depends on the structure of the query plan, i.e., the ordering of the operators in the plan.

Example 2.6. Consider the SQL query in Figure 2b and its query plan in Figure 3a, with the WNQ from Example 2.4. If Marge has not published papers after 2005, the operator node $\sigma_{pyear>2005}$ is picky w.r.t. the tuple that contains *Marge*. As all successors of this operator do not contain a tuple with $aname = Marge$, the node $\sigma_{pyear>2005}$ is the frontier picky operator w.r.t. this tuple.

3 HIGHLIGHT ALGORITHM

Our approach is composed of two stages: find the frontier picky operator for every removed tuple, and, given a why-not questions, find the relevant words that correspond to the frontier picky operator of the tuple in question.

Provenance-Aware Query Evaluation: As a first step, we evaluate the query while storing why-not provenance. We start by translating the NL query to a formal one, via NaLIR [13] augmented so that we keep track of which word in the original NL query has been mapped to which operator of the formal query, as done in [4]. The reverse of this mapping is stored in the data structure M_{ops}. Then, the query is evaluated. During evaluation, whenever a tuple is removed (due to a selection operator or as part of a filtering join), we update the mapping M_{filter}, which maintains the relation between the query operators and the tuples that were removed by them.

Example 3.1. Reconsider the NL query translated by NaLIR to the query in Figure 2b. First, we store the operator-to-word mapping between the query operators and their respective words in the original NL query, shown in Example 2.6. During evaluation, M_{filter} includes intermediate tuples such as (Marge, Paper x, 2001) and its respective picky operators $\sigma_{pyear>2005}$ (additional attributes, such as *pid*, are omitted for brevity).

Finding Relevant Words to Answer Why-Not: After viewing the evaluation results, the user now formulates a why-not query in NL. Algorithm 1 gets as input the results of the evaluation, i.e., the result, $Q(D)$, the mapping M_{filter}, the mapping M_{ops}, the why-not query formulated in NL Q_{WN}, the dependency tree of the NL query T_d, and the database D. Its output is the set of word indices that correspond to the reason for excluding the tuple of interest. Algorithm 1 operates as follows. It uses a sub-mechanism of NaLIR to convert the NL why-not query Q_{WN} into a formal why-not selection query WNQ (line 1). In lines 2–3 it checks whether WNQ is valid, i.e. if there are indeed no output tuples satisfying both the why-not query and the original query (this is a sort of sanity check). If this is not the case, it finds the frontier picky operator (line 4) for each of the tuples satisfying the formal why-not query in WNQ and returns the last of them which is the frontier picky operator w.r.t WNQ. We may get NULL as the operator in the case where there is no match to WNQ in the input; in this case we consider the last projection operation that took place to be the "reason" (lines 5–6). For instance, if someone asked about "why not Krusty?", who is not an author in the example database, the reason would be Π_{aname} which is mapped to the *author* table.

We then use the mapping outputted by the evaluation process to trace back the words corresponding to the operator. The *map*() function exploits the data structure M_{ops}, and gets as input variable names, values or table names that can be mapped back into words in the original query. Lines 13-18 are used to help link back the join operators which could not be directly mapped to words. The main idea is to exploit the typical scenario where unmapped joins are used as means for the query generation engine to link two other relations, which are directly referenced in the NL query. The algorithm traverses the ancestor and successor join operations of the given join operator (by repeatedly calling *GetJoinedRelations*()), until two operators, one ancestor and one successor of the join, which can be mapped into words $(left, right)$, are found. The returned indices are not of these two words, but rather of the words between $left$ and $right$ in T_d (by calling *GetPath*()), corresponding to the intuition that the answer to the why-not query is an unmapped operator between the two mapped operators.

Example 3.2. Consider the NL query in Figure 2a, its SQL query plan in Figure 3a, and the database in Figure 1. The author Bart exists in the *author* table, but has no published papers. Therefore, the frontier picky operator for the NL why-not query "why not Bart?" is the join operator $\bowtie_{aid}$ (see Figure 3a), connecting the *author* table with the *writes* table, which is a join table, containing author ids and publication ids. Algorithm 1 will first convert Q_{WN} to its SQL form and check that Bart is indeed not in the result set of Q in Figure 2b (lines 1–3). It will then get the frontier picky operator $\bowtie_{aid}$, depicted in Figure 3a, which took the tuple (4, *Bart*, 1) as input and did not output it (line 4). Since the output of this line is not NULL, it will continue to line 9. The *writes* table cannot be mapped into a word in the NL query in Figure 2a because the word *writes* does not even appear in it. The reason for the join operation being included is that NaLIR has used this table as a link between the authors table and the papers table. Lines 13-18 are then used to trace back the joined relations. Left and right would be "authors" and "papers" respectively, and the returned path would include only the word "published" as this is the word connecting "authors" and "papers" as seen in the dependency tree of the NL query in Figure 3b, thus our word highlight answer is "return authors who **published** papers in database conferences after 2005".

Algorithm 1: Highlight

```
    input  : $Q(D)$, $M_{filter}$, $M_{ops}$, $Q_{WN}$, $T_d$, and $D$
    output : Word highlight set
 1  $WNQ = NLToFormal(Q_{WN})$;
 2  if $Q(D) \cap WNQ(D) \neq \emptyset$ then
 3      return $\emptyset$;
 4  $op \leftarrow FrontierPicky(M_{filter}.find(WNQ(D)))$;
 5  if $op = NULL$ then
 6      $op \leftarrow GetProjectionOperator(M_{filter})$;
 7  if $op$ is $\sigma_{A=x}$ then
 8      return $map(A, M_{ops}) \cup map(x, M_{ops})$;
 9  if $op$ is $T \bowtie R$ where $R$ is a new input table then
10      if $map(R, M_{ops}) \neq \emptyset$ then
11          return $map(R, M_{ops})$;
12      else
13          $(left, right) \leftarrow GetJoinedRelations(R)$;
14          while $map(left, M_{ops}) = \emptyset$ do
15              $(left, \_) \leftarrow GetJoinedRelations(left)$;
16          while $map(right, M_{ops}) = \emptyset$ do
17              $(\_, right) \leftarrow GetJoinedRelations(right)$;
18          return $GetPath(left, right, T_d)$;
```

4 IMPLEMENTATION AND EXPERIMENTS

We describe various use cases and our scalability evaluation.
Use cases:

Table 1: Sample of Natural Language queries along with Why-Not questions and "Why-Not answers"

ID	NL Query With Highlight Explanation	Why-Not Question
	Selection Frontier Picky Operator	
1	Return authors who published in database conferences after **2015**	Catriel Beeri
2	Return authors who published in **database** conferences after 2015	Yishay Mansour
3	Return authors from **"Tel Aviv University"** who published in VLDB	Benny Kimelfeld
4	Return authors from "Tel Aviv University" who published in **VLDB**	Yishay Mansour
5	Return organizations of authors who wrote in **database** journals	AI University (org. without DB authors)
6	Return papers of authors in **"Artificial Intelligence"** after 2005 and before 2007	Active Views for Electronic Commerce
7	Return papers of authors in "Artificial Intelligence" after **2005** and before 2007	Stochastic Link and Group Detection
8	Return authors from **"North America"** who presented in VLDB in 2000	Tova Milo
9	Return authors from "North America" who presented in **VLDB** in 2000	Geoffrey E. Hinton
10	Return authors from "North America" who presented in VLDB in **2000**	Christopher Ré
11	Return publications about **graphics** after 2005	Overflow Controled SIMD Arithmetic
	Join Frontier Picky Operator	
12	Return organizations of **authors** who wrote in database journals	MadeUp College (org. without authors)
13	Return authors who **published** in database conferences	John Doe (author without publications)

Table 1 depicts representative examples of NL queries along with relevant why-not questions that were executed on the MAS database [15]. The first 11 examples demonstrate cases in which the reason for the tuple absence was a selection operator. Queries 12 and 13 demonstrate the operation of Algorithm 1 when the frontier picky operator is a join operation, this is often an indication of missing tuples in the dataset. Overall we can see that for the vast majority of NL queries and why-not questions the explanations supply valuable information that justify in concise manner the absence of the tuples in question.

Scalability: Here again we have used the MAS database whose total size is 4.7 GB, and queries 1–11 from Table 1, running the algorithm to generate word highlight and NL explanation. The computation steps execution times, for each query, are depicted in Figure 4. The computation times are given in nanoseconds and the y axis is log-scaled. As evident from the graph, most of the time is spent on NL to SQL conversion, query evaluation and identifying the relevant tuples for the why-not queries. Generating the why-not explanations incurs a negligible performance cost (less than a millisecond on average for selection frontier picky operators), and thus provides an interactive experience for the user.

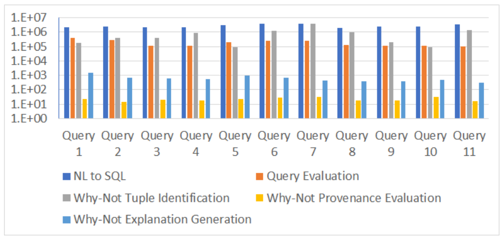


Figure 4: Average computation time by step (log scale, values in nanoseconds)

Acknowledgements. This research has been funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 804302), the Israeli Science Foundation (ISF) Grant No. 978/17, and the Google Ph.D. Fellowship. The contribution of Amir Gilad is part of a Ph.D. thesis research conducted at Tel Aviv University.

REFERENCES

[1] 2002. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *ICDE*.
[2] Nicole Bidoit, Melanie Herschel, and Aikaterini Tzompanaki. 2015. Efficient Computation of Polynomial Explanations of Why-Not Questions. In *CIKM*.
[3] Adriane Chapman and H. V. Jagadish. 2009. Why not?. In *SIGMOD*. 523–534.
[4] Daniel Deutch, Nave Frost, and Amir Gilad. 2016. NLProv: Natural Language Provenance. *PVLDB* 9, 13 (2016), 1537–1540.
[5] Daniel Deutch, Nave Frost, and Amir Gilad. 2017. Provenance for Natural Language Queries. *PVLDB* 10, 5 (2017), 577–588.
[6] Daniel Deutch, Nave Frost, and Amir Gilad. 2019. Explaining Natural Language query results. *The VLDB Journal* (2019).
[7] Daniel Deutch, Nave Frost, Amir Gilad, and Tomer Haimovich. 2018. NLProveNAns: Natural Language Provenance for Non-Answers. *PVLDB* 11, 12 (2018), 1986–1989.
[8] Enrico Franconi, Claire Gardent, Ximena I Juarez-Castro, and Laura Perez-Beltrachini. 2014. Quelo Natural Language Interface: Generating queries and answer descriptions. In *NLIWOD*.
[9] Z. He and E. Lo. 2014. Answering Why-Not Questions on Top-K Queries. *IEEE Transactions on Knowledge and Data Engineering* 26, 6 (2014), 1300–1315.
[10] Melanie Herschel and Mauricio A. Hernández. 2010. Explaining Missing Answers to SPJUA Queries. *Proc. VLDB Endow.* 3, 1-2 (2010), 185–196.
[11] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. 2008. On the Provenance of Non-answers to Queries over Extracted Data. *Proc. VLDB Endow.* 1, 1 (2008), 736–747.
[12] Dan Klein and Christopher D. Manning. 2003. Accurate Unlexicalized Parsing. In *Annual Meeting on Association for Computational Linguistics.*
[13] Fei Li and H. V. Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *PVLDB* 8, 1 (2014), 73–84.
[14] M. Marneffe, B. Maccartney, and C. Manning. 2006. Generating Typed Dependency Parses from Phrase Structure Parses. In *LREC*.
[15] MAS. 2016. http://academic.research.microsoft.com/.
[16] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a Theory of Natural Language Interfaces to Databases. In *IUI*. 149–157.
[17] Quoc Trung Tran and Chee-Yong Chan. 2010. How to ConQueR Why-not Questions. In *SIGMOD*. 15–26.

A Context-based Approach for Partitioning Big Data

Sara Migliorini
Dept. of Computer Science, University of Verona
sara.migliorini@univr.it

Alberto Belussi
Dept. of Computer Science, University of Verona
alberto.belussi@univr.it

Elisa Quintarelli
Dept. of Computer Science, University of Verona
elisa.quintarelli@univr.it

Damiano Carra
Dept. of Computer Science, University of Verona
damiano.carra@univr.it

ABSTRACT

In recent years, the amount of available data keeps growing at fast rate, and it is therefore crucial to be able to process them in an efficient way. The level of parallelism in tools such as Hadoop or Spark is determined, among other things, by the partitioning applied to the dataset. A common method is to split the data into chunks considering the number of bytes. While this approach may work well for text-based batch processing, there are a number of cases where the dataset contains structured information, such as the time or the spatial coordinates, and one may be interested in exploiting such a structure to improve the partitioning. This could have an impact on the processing time and increase the overall resource usage efficiency.

This paper explores an approach based on the notion of context, such as temporal or spatial information, for partitioning the data. We design a context-based multi-dimensional partitioning technique that divides an $n-$dimensional space into splits by considering the distribution of the each contextual dimension in the dataset. We tested our approach on a dataset from a touristic scenario, and our experiments show that we are able to improve the efficiency of the resource usage.

KEYWORDS

Big Data, Partitioning, Contextual dimensions

1 INTRODUCTION

A dataset analyzed using parallel data processing systems, such as Hadoop or Spark, is usually divided into chunks, or *splits*. The basic partitioning approach uses the amount of bytes (e.g., 64 or 128 MB) as splitting technique, without considering the content of the data. In case of batch processing, where the dataset is always analyzed entirely, this solution is reasonable. Nevertheless, if the dataset is analyzed using selective queries based on some attributes of the data, like time intervals or spatial regions [6, 11], such an approach may not be efficient, since the partitioning does not exploit the correlations in the data.

Consider for instance a dataset that collects the visits of tourists at different Points of Interests (PoIs). The tourists have a city pass which they swipe at the entrance of a PoI. Each swipe contains the identifier of the city pass, the name and location (coordinates) of the visited PoI, together with an entrance timestamp. One may analyze such a dataset considering the timestamp (How many tourists have been there in a specific day?), or the space (How many times has a specific PoI been visited?), or PoI type (Are modern-art museums preferred to science museums?). One may also combine different dimensions (How many tourists visited a specific PoI in a specific hour?).

As a general approach, we consider *context-aware* partitioning techniques. Assuming that a dataset can be analyzed w.r.t. several dimensions, the idea is to group in the same split records that are context-related [7]. For instance, if the context is defined by the space and time dimensions, a context-aware partitioning will include in the same split records that are nearby to each other from both a spatial and temporal point of view.

Previous works proposed partitioning approaches mainly based on spatial [9, 12] and spatio-temporal [1, 2] characteristics. In case of spatial partitioning, one may partition based on space (grid and Quad-tree), based on data (STR, STR+, K-d tree), or based on space filling curves (Z-curve, Hilbert curve) [8]. The selection of the partitioning technique is usually left to the user, and only few works automatically select the best partitioning technique based on the dataset distribution [5].

When two or more dimensions need to be combined, there are two possible approaches. The first one considers each dimension independently and builds a *multi-level partitioning*. This approach produces a list of n grids (one for each level) that are used for performing the partitioning, and it imposes an order between them. The chosen order can have a great impact on the nature and balancing of the resulting splits. For instance, ST-Hadoop [1] firstly divide the dataset based on temporal granularity, and then splits each portion based on spatial proximity. A query focused on spatial properties (e.g., Has PoI x been visited more than PoI y?) requires the analysis of all, temporally organized, splits.

The second approach considers all the dimensions together and builds a *multi-dimensional partition*, i.e., a n-dimensional grid. An example is HadoopTrajectory [2], in which partitions are 3D cubes where the three dimensions are space (planar coordinates) and time. Given a query focused on one dimension, this approach allows the exact selection of the splits that could be useful in answering the query. The challenge imposed by the multi-dimensional partitioning is to find the best size of the grid cells in each dimension, so that the amount of data in each cell is balanced. This can be a non trivial task, especially in the general case where data are not uniformly distributed [3, 5].

In this paper, we consider a context-based multi-dimensional partitioning approach, which takes as input a dataset D and the set of n contextual dimensions relevant to analyse D. We design a solution that automatically produces the most appropriate division of the $n-$dimensional space, considering the distribution of each contextual dimension inside D. The proposed technique could be adopted in case of recurrent queries, to drive the partitioning of the dataset that is stored permanently (e.g., HDFS), or it can be used in a dynamic scenario, where the dataset is kept in-memory (e.g., Spark), and it can be repartitioned based on the current set of queries.

We evaluate our solution on a real-world dataset containing the swipes of a city pass – with the characteristics previously described. The results show that we are able exploit the partitioning to efficiently process a set of representative queries.

2 PROBLEM FORMULATION

This section formalizes the context-aware partitioning problem.

Definition 2.1 (Dataset). A *dataset schema* $S = \langle a_1, \ldots, a_m \rangle$ is a list of attributes, each one belonging to a particular domain, denoted as $\Delta(a_i)$. A *dataset* $D = \{r_1, \ldots, r_n\}$ over a schema S is a collections of records $r_i = \langle v_1, \ldots, v_m \rangle$, where $\forall i \in \{1, \ldots, m\}\ v_i \in \Delta(a_i)$.

Definition 2.2 (Context). Given a dataset D over a schema $S = \langle a_1, \ldots, a_m \rangle$, the context is a subset of the attributes in S:

$$C = \{c_1, \ldots, c_k\} \subseteq \{a_1, \ldots, a_m\} \tag{1}$$

Definition 2.3 (Partitioning). Given a dataset $D = \{r_1, \ldots, r_n\}$, a partitioning P is a collection of subsets of D:

$$P = \{p_1, \ldots, p_h\} \text{ such that } \forall p_i \in P\,(p_i \subseteq D) \text{ and } D = \cup_i p_i \tag{2}$$

Definition 2.4 (Balanced partitioning). A partitioning $P = \{p_1, \ldots, p_h\}$ for a dataset D is said to be balanced if and only if:

$$\forall p_i, p_j \in P : \; abs(|p_i| - |p_j|) \leq \varepsilon \tag{3}$$

where $|p_i|$ denotes the cardinality of the partition p_i.

Definition 2.5 (Context-aware partitioning). A partitioning P_C for a dataset D is the minimal one for the context C and a context-based query q if and only if:

(1) it is a balanced partitioning, i.e. it is able to produce balanced partitions,
(2) it minimizes the number of splits to consider for answering the query q. Notice that a partition might contain also more than one split.

3 CONTEXT-BASED PARTITIONING

In order to partition a dataset D considering a context C with several dimensions of analysis, different approaches can be applied. For instance, in the multi-dimensional partitioning the subdivision of the elements in D is performed by defining an n-dimensional grid where $n = |C|$. Conversely, a multi-level partitioning could also be applied by using n grids, each one representing a level and corresponding to a dimension in C.

Notice that given a dataset D, a context C and a query q, it is possible to have multiple minimal partitionings. Indeed, the quality of the resulting set of partitions can highly depend on the distribution in D of the values of the dimensions belonging to C and those used in q. When our partitioning technique is not applied on-line before executing each query q, the minimality of a partitioning can be evaluated by considering the average number of splits used for a reference set of context-based queries.

Given a context C for a dataset D, the identification of the most appropriate partitioning requires an easy and efficient way for evaluating the skewness in D of each context dimension. Based on this evaluation, the right shape of each n-dimensional cell inside the n-dimensional grid can be determined.

The aim of this paper is to propose a partitioning technique able to capture the distribution of the dataset w.r.t. each context dimension and based on this to build the right set of partitions even for skewed datasets. For this purpose, we extend the idea originally proposed in [5] for the spatial domain to the management of a generic number n of context dimensions. For this reason, we present below the definition of the box-counting function $BC_r^q(D, a)$ for a given dataset D and a context dimension a, that is the fundamental notion for the skewness evaluation.

Definition 3.1 (Box-counting function for a dimension a). Given a dataset D, containing an attribute a belonging to a domain $\Delta(a)$, and a scale r representing the cell size of a mono-dimensional grid covering the range of values of $\Delta(a)$ appearing in D, the box-counting function $BC_r^q(D, a)$ is defined as:

$$BC_r^q(D, a) = \sum_i \delta_i(D, a)^q \quad \text{with } q \neq 1 \tag{4}$$

where $\delta_i(D, a)$ is the number of records in D whose value for a is contained in the cell i. The case $q = 1$ is excluded, since it does not depend on r and it equals the number of records in D. □

Intuitively, given a grid with cells of side r, the box-counting function with $q = 0$ counts the number of cells that contains at least one record of D. When q is greater than 1, the box-counting becomes the sum of the number of records that a cell contains, raised to q. This function can be used to detect the skewness of a dataset by computing it for $q = 0$ and $q = 2$, while varying the value of r. More specifically, the level of skewness of a dataset depends on how this value changes while increasing r.

Definition 3.2 (Box-counting plot). Given a dataset D, containing an attribute a belonging to a domain $\Delta(a)$, the box-counting plot is the plot of $BC_r^q(D, a)$ versus r in logarithmic scale.

On datasets representing fractals, since it can be derived from theory, and on real datasets, as shown in [5, 10], the following observation can be considered valid.

Observation 1. *For finite datasets representing fractals and real datasets the box counting plot reveals a trend of the box counting function that, in a large interval of scale values r, behaves as power law:*

$$BC_r^q(D, a) = \alpha \cdot r^{E_q} \tag{5}$$

where α is a constant of proportionality and E_q is a fixed exponent that characterizes the power law.

The power law exponent E_q for a given dataset D and an attribute a can be computed by starting from the box-counting plot, since it becomes the slope of the strait line that approximates $BC_r^q(D, a)$ in a range of scale (r_1, r_2), thus it can be computed by a linear regression procedure.

We can observe that E_0 and E_2 could be chosen as reference descriptors about the distribution of the values of the attribute a in the dataset D. Indeed, E_0 can be an indicator of the cases where the dataset leaves empty some areas of the range of values of a covered by D, while E_2 can also be affected by the concentration of the values in some areas with respect to other ones, i.e. the situations where there are no empty areas, but different data concentrations in different areas.

In order to optimize the computation of E_0 and E_2 for a big dataset D_{big} the following approach can be applied. First, we consider a sample of D_{big} (usually 10% of the records) for the computation of the n-dimensional histogram: it is composed of a n-dimensional cube counting the number of records falling in each cell (only the non-empty cells are represented). Second, the projection of the n-dimensional cube on each dimension produces n one-dimensional histograms to be used for the computation of E_0 and E_2 for each dimension. Third, considering the heuristics presented in [5], accordingly to E_0 and E_2 the suitable partitioning technique for each dimension is chosen (possible techniques are: regular grid, space-based partitioning, record-based partitioning). Finally, D_{big} is scanned and partitioned using the n-cube produced by intersecting the list of splitting planes obtained by applying the chosen techniques. Notice that for each dimension a different technique might have been chosen.

4 CASE STUDY EVALUATION

The proposed technique has been applied to a real-world dataset containing the swipes of a city pass, called *VeronaCard*, which is provided by the tourist office of Verona, a municipality in Northern Italy. The dataset contains about 1,200,000 records concerning 4 years. Each record reports beside to the identifier of the city pass and the name of the visited PoI: the location (coordinates) of the PoI, the entrance timestamp and the age of the tourist holding the card.

Fig. 1 shows the spatial distribution of the records: the size of the circle surrounding the PoI name represents the number of records regarding that location: bigger cycles represent PoIs with the higher number of visits. As you can notice, the records are not uniformely distributed w.r.t. space, since there are some PoIs, such as *Arena* and *Casa di Giulietta*, which have much more visits w.r.t. others, like *San Fermo*.

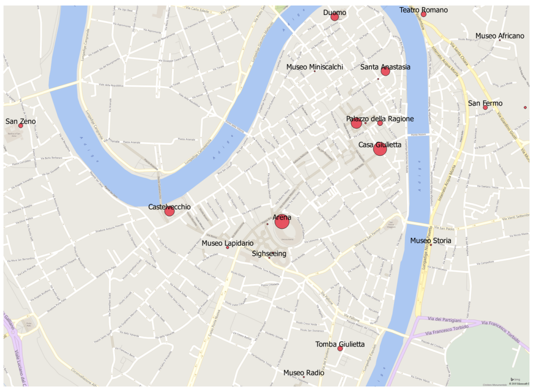


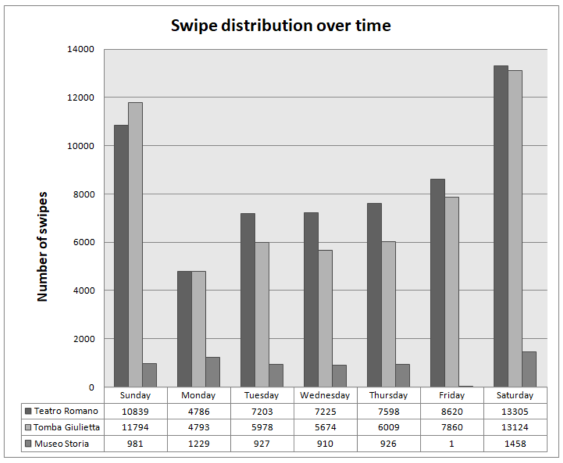
Figure 1: Spatial distribution of the swipes: bigger circlers represent an higher number of visits in that PoI (records).

A sample about the distribution of the records w.r.t. the time is represented in Fig. 2 by means of histograms. In particular, we show the distribution of three PoIs aggregated by the day of the week. Even in this case the distribution is not uniform, since there are some days in which a PoI is mostly visited than others (es. weekend days vs week days). Moreover, some PoI can have a closing day in which there are no visits at all.

Finally, the age distribution is reported in Fig. 3 by means of a Pie chart. Even in this case the distribution is not uniform: some ages more frequent than others, reflecting the fact that there are PoIs more suitable for some kinds of tourists than others.

These different distributions are recognized also by the exponents E_0 and E_2 of their box counting plot, as reported in Tab. 1 together with the chosen partitioning technique.

Tab. 2 illustrates some statistics about the partitions produced by the four considered partitioning techniques: *Rand* is the default partitioning technique traditionally supported by a MapReduce environment, it simply subdivides the dataset into parts with homogeneous size in bytes. MD_{grid} is a multi-dimensional uniform grid partitioning technique which essentially subdivides the dataset by using uniform d-dimensional cells for data aggregation, while ML_{grid} is a multi-level uniform grid partitioning



	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
■ Teatro Romano	10839	4786	7203	7225	7598	8620	13305
▨ Tomba Giulietta	11794	4793	5978	5674	6009	7860	13124
■ Museo Storia	981	1229	927	910	926	1	1458

Figure 2: Number of swipes for 3 PoIs by day of the week.

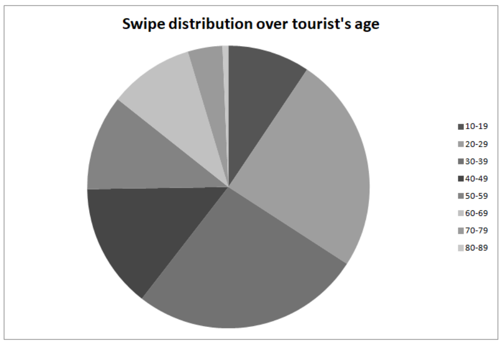


Figure 3: Number of swipes for each age value.

Table 1: Exponents E_0 and E_2 and technique choice.

Context attribute	$-E_0$	E_2	Index
time	1.041	0.963	Regular Grid
x	0.041	0.001	Space-based partitioning
y	0.054	0.154	Space-based partitioning
age	0.506	0.474	Record-based partitioning

which considers only a dimension at each phase during the uniform subdivision. Finally, *CBP* is the context-based partitioning technique proposed in this paper which uses the computation of the box counting plot and the corresponding exponents E_0 and E_2, in order to produce the most appropriate d-dimensional grid in order to accomodate also non uniform data.

Table 2: Experimental results: RAND is the random partitioning, %RSD is the relative standard deviation with respect to the dimension analysis.

Index	#parts	#splits	%RSD	%RSD	%RSD	%RSD	%RSD
			#size	time	x	y	age
Rand	69	69	1%	57%	33%	35%	58%
MD_{grid}	64	109	64%	25%	4%	17%	24%
ML_{grid}	73	117	71%	26%	4%	14%	23%
CBP	88	88	26%	20%	2%	3%	19%

The first consideration we can done is that while *Rand* is able to produce the minimum number of splits with respect to the dataset dimension and the split size (#parts = #splits), the other techniques can produce a greater number of splits which is equal to: $\lceil \sqrt[d]{ds/sp} \rceil$ where d is the number of dimensions, ds is the dataset size in bytes and sp is the split size in bytes. Notice that since the dataset is not uniformly distributed: (1) some cells could be empty, so they are not produced (i.e., MD_{grid} has 64 partitions and ML_{grid} has 73 partitions). (2) Some cells could be overpopulated, so they have to be split in order to comply with the split size prescription, i.e., at the and MD_{grid} has 109 splits, while ML_{grid} has 117 splits. Conversely, *Rand* and *CBP* has a number of splits equal to the number computed of partitions. *CBP* produces more partitions than *Rand* due to the additional subdivision of the n-dimensional space in case of clustered data.

The non uniformity of the dataset distribution has a direct effect also on the split size variability. In order to evaluate the variability of a given feature, we use the %RSD, which is the relative standard deviation, namely it is a statistical measurement describing the spread of data with respect to the mean and the result is expressed as a percentage. Clearly, the *Rand* technique is able to produce very balanced splits, since the partitioning is guided only by this parameter. Conversely, MD_{grid} and ML_{grid} produce very unbalanced splits due to the data skeweness. With *CBP*, we obtain quite good results in terms of balancing.

The second aspect to consider is the variability of each dimension values inside the splits. Columns 5-8 in Tab. 2 reports the average %RSD for each considered dimension. As you can notice, all the last three techniques generally improve the performances of the *Rand* technique. In particular, as regards to the temporal dimension, since it is quite uniformly distributed and also it is the first considered dimension by the ML_{grid}, its average spread is the same for MD_{grid} and ML_{grid}, while for the other dimensions the spread is less for MD_{grid} since it considers all dimensions at the same level for producing the partitioning. *CBP* produces splits with less variability in each dimension w.r.t. all the other techniques.

Given the partitions induced by the four considered partitioning techniques, we have performed some representative range queries in order to evaluate their performances. The performances are evaluated as the number of splits that have to be processed in order to produce the desired result, namely in the filtering capabilities induced by each partitioning technique.

We consider 4 queries, the first one has a condition on all the context dimensions, while the other ones contain conditions on less dimensions. The results are reported in Tab. 3.

Q_1: *find all visits performed around the Arena during spring 2015 by young tourists.* The spatial location is defined by a buffer around the Arena, while the period *spring 2015* is defined by a temporal interval, and the *young tourists* are identified by an age range. Since the condition regards all the four dimensions, the performance of MD_{grid} and ML_{grid} are quite similar.

Q_2: *find all visits performed by teenager in 2016 everywhere in Verona.* In this case the spatial dimensions are not considered in the filter, while a pruning is performed on the temporal dimension. Since this represents the first level for ML_{grid}, it sightly outperforms MD_{grid}.

Q_3: *find all visits around Arena performed by senior tourists.* In this case the temporal dimension is not considered, while the spatial and age dimensions are considered. Differently to the previous case, the advantage of ML_{grid} is loss, since no filtering can be performed at the first level.

Q_4: *find all the visits performed by adult tourists.* As you can notice here the filter is applied only on the 4th dimension, so

Table 3: Experimental results. Numbers represents #splits.

Query	*Rand*	MD_{grid}	ML_{grid}	*CBP*
Q1	69	3	3	2
Q2	69	32	30	14
Q3	69	6	6	5
Q4	69	85	94	80

ML_{grid} performs worst than MD_{grid}, because it has to scan all the levels before applying a filter.

5 CONCLUSION

In MapReduce frameworks the partitioning of a dataset into independent splits is a critical operation, since the degree of parallelism and the overall performances directly depend from the initial partitioning technique. This is particularly true in case of context-based applications, where data present correlations and consequently data could be aggregated and filtered in order to reduce the amount of work to be done during the analysis. Moreover, beside the need for a context-based partitioning technique, in order to produce balanced splits, it is necessary to consider the distribution of the dataset w.r.t. the analysis dimensions. This paper proposes a context-based partitioning technique which takes care of the dimension distributions to produce the best partitioning for the dataset at hand. We also apply it to a real-world dataset and compare its performances w.r.t. existing partitioning techniques for highlighting its differences and benefits. The obtained preliminary results confirm the goodness of the approach and encourage further research in this direction, for instance as regards to the managament of multi-accuracy data [4].

ACKNOWLEDGMENTS

This work was partially supported by the Italian National Group for Scientific Computation (GNCS-INDAM). and by “Progetto di Eccellenza” of the Computer Science Dept., Univ. of Verona, Italy.

REFERENCES

[1] L. Alarabi, M. F. Mokbel, and M. Musleh. 2018. ST-Hadoop: a MapReduce framework for spatio-temporal data. *GeoInformatica* 22, 4 (2018), 785–813.

[2] M. Bakli, M. Sakr, and Taysir H. A. S. 2019. HadoopTrajectory: a Hadoop spatiotemporal data processing extension. *Journal of Geographical Systems* 21, 2 (2019), 211–235.

[3] A. Belussi, D. Carra, S. Migliorini, M. Negri, and G. Pelagatti. 2018. What Makes Spatial Data Big? A Discussion on How to Partition Spatial Data. In *10th Int. Conf. on Geographic Information Science (GIScience 2018).* 2:1–2:15.

[4] A. Belussi and S. Migliorini. 2012. A Framework for Integrating Multi-Accuracy Spatial Data in Geographical Applications. *Geoinformatica* 16, 3 (2012), 523–561.

[5] A. Belussi, S. Migliorini, and A. Eldawy. 2018. Detecting Skewness of Big Spatial Data in SpatialHadoop. In *Proceedings of the 26th ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems.* 432–435.

[6] A. Belussi, S. Migliorini, M. Negri, and G. Pelagatti. 2015. Validation of Spatial Integrity Constraints in City Models. In *Proc. of the 4th ACM SIGSPATIAL Int. Workshop on Mobile Geographic Information Systems.* 70–79.

[7] C. Bolchini, E. Quintarelli, and L. Tanca. 2013. CARVE: Context-aware automatic view definition over relational databases. *Inf. Syst.* 38, 1 (2013), 45–67.

[8] A. Eldawy, L. Alarabi, and M. F. Mokbel. 2015. Spatial Partitioning Techniques in SpatialHadoop. *Proc. VLDB Endow.* 8, 12 (2015), 1602–1605.

[9] A. Eldawy and M. F. Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *2015 IEEE 31st Int. Conf. on Data Engineering.* 1352–1363.

[10] C. Faloutsos, B. Seeger, A. Traina, and C. Traina, Jr. 2000. Spatial Join Selectivity Using Power Laws. *SIGMOD Rec.* 29, 2 (2000), 177–188.

[11] S. Migliorini, A. Belussi, M. Negri, and G. Pelagatti. 2016. Towards Massive Spatial Data Validation with SpatialHadoop. In *Proc. of the 5th ACM SIGSPATIAL Int. Workshop on Analytics for Big Geospatial Data.* 18–27.

[12] J. Yu, Z. Zhang, and M. Sarwat. 2019. Spatial Data Management in Apache Spark: The GeoSpark Perspective and Beyond. *Geoinformatica* 23, 1 (2019), 37–78.

SlideSide: A fast Incremental Stream Processing Algorithm for Multiple Queries

Georgios Theodorakis
Imperial College London
grt17@imperial.ac.uk

Peter Pietzuch
Imperial College London
prp@imperial.ac.uk

Holger Pirk
Imperial College London
pirk@imperial.ac.uk

ABSTRACT

Aggregate window computations lie at the core of online analytics in both academic and industrial applications. To efficiently compute sliding windows, the state-of-the-art algorithms utilize incremental processing that avoids the recomputation of window results from scratch. In this paper, we propose a novel algorithm, called SlideSide, that extends `TwoStacks` for multiple concurrent aggregate queries over the same data stream. Our approach uses different yet similar processing schemes for invertible and non-invertible functions and exhibits up to 2× better throughput compared to the state-of-the-art incremental techniques in a multi-query environment.

1 INTRODUCTION

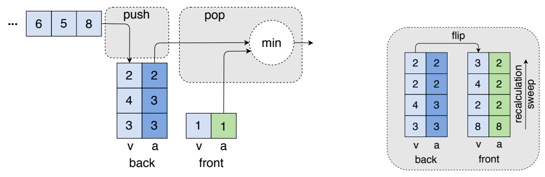
An ever-growing amount of data needs to be analyzed in real-time. Applications ranging from credit card fraud detection to clickstream analytics are not supported by "classic" relational systems and algorithms. Consequently, streaming applications have become increasingly important. One of the key operators in stream processing is window aggregation [1], i.e., the calculation of running aggregates over the continuous data stream.

Since data streams are conceptually infinite, they are partitioned into finite subsets of elements, called *windows.* A window has a *definition*, which maps each input tuple to a window *instance.* Upon aggregation, each window instance yields a result. Windows can be distinguished by whether their instances are disjoint ("tumbling windows") or not ("sliding windows"). Tumbling (a.k.a. fixed) windows slice up the input stream into segments with a fixed size temporal length (static *window size*). Sliding (a.k.a. hopping) windows generalize tumbling windows by specifying a *slide* parameter in addition to the size that specifies the distance between the start of two windows.

While tumbling windows are amenable to classic "relational" queries implementation techniques, the performance of sliding windows is more challenging to compute efficiently. Incremental algorithms introduce inherent control dependencies in the CPU instruction stream, as intermediate results from previous window instances have to be used to compute efficiently the next result. This is amplified in the case of multiple-queries applying computations over the same data stream, which has not been explored comprehensively. An example of the latter scenario is a live-visualization dashboard that plots line charts of aggregates on time-series data at different zoom levels [9].

Our contributions are the following:

- We study the performance of the best-performing incremental algorithms, as reported in recent literature [4, 6]. We determine sections of the problem space in which different approaches perform best (focusing specifically on multi-query processing).



(a) Pushing and Popping (b) Flipping

Figure 1: The TwoStacks algorithm

- We propose a novel approach for incremental processing in multi-queries scenarios, called SlideSide. Our solution extends the logic of `TwoStacks` [4] based on the insight that the algorithm maintains a running `preffix-`/`suffix-scan` over the input stream. SlideSide optimizes its performance for associative aggregation functions and can serve as a drop-in replacement for the aggregation operator in a streaming system.
- Finally, we demonstrate that SlideSide is competitive with highly optimized single-query algorithms, while it yields up to 2× better throughput and comparable latency in the multi-query scenario.

The remainder of the paper is organized as follows: in Section 2 we survey the state-of-the-art in incremental window computation. Section 3 introduces our novel incremental algorithm. The paper finishes with evaluation results (Section 4), and conclusions (Section 5).

2 BACKGROUND

In this section, we provide background on the underlying concepts of incremental processing. We also review current approaches and provide in Table 1 a summary of their complexities.

Algebraic Properties. In this work, we focus on associative algebraic aggregations [3] and consider the following properties:

- Invertibility: $(x \oplus y) \ominus y = x, \forall x, y$. This property can be exploited by introducing the following function: `inverse(a: Agg, b: Agg): Agg`, which removes the oldest partial aggregate from the window result with an incremental operation.
- Commutativity: $x \oplus y = y \oplus x, \forall x, y$

Partial Aggregates are smaller units of computation that compose the aggregate functions. Partial aggregation allow us to buffer and apply inexpensive aggregation and trivially parallelize the computation (e.g., using SIMD instructions [8]), when there are no data dependencies. This idea is applied in the form of *window slicing* [9] over a stream of data, where a slice is defined as the largest sequence of tuples that offer no sharing potential.

Incremental Aggregation Techniques. After the computation of partial results, the final aggregation step has to be applied to generate the query results. For that, streaming systems utilize the incremental aggregation techniques we describe next and summarize in Table 1.

Subtract-on-Evict (SoE) [4] is the best-performing approach in the case of invertible functions, i.e., AGG_{sum}. With SoE, the result of the previous window instance is re-used to compute the next

Algorithm		Time				Space	
		Single Query		Multi-Queries		Single-Query	Multi-Queries
		Amort	Worst	Amort	Worst		
SoE [4]	Inv	2	2	q	q	n	qn
	Non-Inv	n	n	qn	qn	n	qn
FlatFAT [7]		$log(n)$	$log(n)$	$qlog(n)$	$qlog(n)$	$2n$	$2n$
TwoStacks [4]		3	n	q	qn	$2n$	$2qn$
SlickDeque [6]	Inv	2	2	$2q$	$2q$	n	$q+n$
	Non-Inv	<2	n	q	qn	2 to $2n$	2 to $2n$
SlideSide	Inv	3	n	q	q	3n	$3n$
	Non-Inv	3	n	q	qn	$2n$	$2n$

Table 1: Algorithmic Complexities (partial aggregates: n, queries: q)

in constant time, by removing the expired elements and merging in the new data. However, SoE cannot efficiently compute non-invertible functions (e.g., AGG_{min}), as the whole window needs to be rescanned in the worst-case scenario.

TwoStacks [4] can be used for non-invertible functions. Figure 1 illustrates an example of the TwoStacks algorithm, which maintains a *back* and a *front* stack to store the input values and the aggregates required to produce the window results. When a new input value v arrives, its aggregate is computed based on the value of the back stack's top element and it is **push**ed onto the back stack. For every **pop** operation, the top from the front stack is removed and a result is produced by aggregating its value with the top of the back stack. Whenever the front stack is empty, the algorithm **flip**s the back onto the front stack, reversing the order of the values and recalculating the aggregates. However, as this happens infrequently, it exhibits $O(1)$ amortized complexity.

Multi-Query Algorithms. The previous algorithms are not designed to efficiently share intermediate results between multiple window definitions over the same stream, in contrary to approaches such as FlatFAT [7] and SlickDeque [6]. FlatFAT uses a pointer-less binary tree structure to store the partials, which results in $O(logn)$ complexity for a single query. SlickDeque proposes a different solution for invertible and non-invertible functions. For invertible functions, SlickDeque generalises SoE to answer multiple queries, by maintaining multiple instances of the original algorithm with partials that share the same memory space. In the case of non-invertible functions, instead of using a queue implemented by two stacks, SlickDeque uses a deque structure for insertions/removals of aggregates and answering queries with $O(1)$ amortized complexity. The time and space complexities of the non-invertible functions (see Table 1) depend on the input, with the worst-case scenario being a stream that is ordered in the opposite way of the aggregate operator order (i.e., if AGG_{max} the input is ordered descendingly).

3 SLIDESIDE

Let us now, describe SlideSide, our novel algorithm for accelerating incremental aggregation in a multi-query environment. It aggressively reuses intermediate results with data structures that have a sequential memory layout. Fundamentally, SlideSide is an extension of the TwoStacks algorithm. However, it uses different processing schemes for invertible and non-invertible functions.

Regarding the algebraic properties of the aggregate functions, SlideSide has the same requirements as the state-of-the-art algorithms described in Section 2 (associative aggregate functions). SlideSide can be applied to FIFO windows (in-order data).

3.1 Invertible Aggregates

The simpler case are invertible combiners, such as AGG_{sum}. The natural approach of evaluating multiple simultaneous windows would be to run multiple loop-fused instances of SoE. However, we found that the TwoStacks algorithm can be extended to support this case as well, yielding a more cache efficient approach. Somewhat surprisingly, this can be implemented using only two stacks (illustrated in Figure 2). Like the single query case, the elements of the back and front stacks share the same memory space and their aggregates are kept separately. Next, we will explain the algorithm and provide an example with two queries.

Algorithm 1: SlideSide (Inv) Pseudocode

```
Input: A set of aggregate queries Q, a combiner operation ⊕, an inverse operation ⊖
Output: The results of the window queries in Q
1 windowSize = Q.getMaxWindowSize()
2 backStack [windowSize+1] = {neutralVal} // used for prefix-scan
3 frontStack [windowSize+1] = {neutralVal} // used for suffix-scan
4 elements [windowSize] = {neutralVal} // used for input stream
5 curPos = 0
6 foreach val: stream do
7     insert(backStack, frontStack, elements, curPos, windowSize, val)
8     emitResults(backStack, frontStack, curPos, windowSize, Q)
```

Algorithm 2: Algorithm for insert(...)

```
1 // compute the suffix-scan
2 if (curPos==0) then
3     for i=0,1,..., windowSize do
4         frontStack[i+1] = frontStack[i] ⊕ partials[windowSize-i-1]
5 elements[curPos] = val
6 backStack[curPos+1] = elements[curPos] ⊕ backStack[curPos]
7 curPos = (curPos+1) % windowSize // wrap around the circular buffer
```

During the initialization phase of Algorithm 1, the *back* stack (blue row), the *front* stack (green row) and a circular buffer of *elements* (light-blue row) are allocated with size equal to the largest window from a given set of queries, Q, and initialized with the neutral element of the aggregate function (lines 1-4). For every input value *val* from the stream, we call the *insert* function and then compute the results for every query in Q with *emitResults* in line 6-8.

Upon the arrival of a new element, using the *insert* function (Algorithm 2), its *val* is stored in the next available slot in the circular buffer, defined by the *curPos* variable in line 5. The back stack is used for maintaining the prefix-scan of the input with every insert (line 6). If we reach the end of the elements buffer, we wrap around to the beginning and compute a suffix-scan over the input (lines 2-4) before applying the new insertion. Note that, as in TwoStacks, the computation of the suffix-scan occurs infrequently and the algorithm has $O(1)$ amortized complexity.

Algorithm 3: Algorithm for emitResults(...)

```
1 foreach query q : Q do
2     curWindowSize = q.getSize();
3     hasWrapped = false;
4     endPtr = curPos;
5     if (endPtr == 0) then
6         endPtr = windowSize
7     startPtr = endPtr - curWindowSize;
8     if (startPtr < 0) then
9         hasWrapped = true // the window wraps around the circular buffer;
10        startPtr + = windowSize;
11    if (!hasWrapped && startPtr == 0) then
12        res = backStack[endPtr] // use the result from prefix-scan;
13    else if (hasWrapped) then
14        res = backStack[endPtr] ⊕ frontStack[windowSize - startPtr];
15    else
16        res = backStack[endPtr] ⊖ backStack[startPtr];
17    forward answer res to query q;
```

After the insertion, the *emitResults* function (Algorithm 3) is called for computing the results for each query with the set

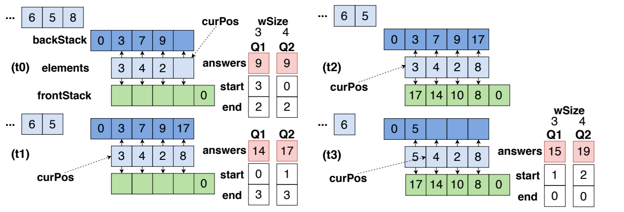


Figure 2: SlideSide (Inv)

Q. Based on the values of *curPos* and the window size of each query, this algorithm first computes the `start` and `end` pointers (lines 2-10) that are used for "bookkeeping". Next, we distinguish three different cases based on the value of these pointers: ***a)*** if the start pointer is 0, the result is already computed by the `prefix-scan` and returned in line 12; ***b)*** if the start pointer is greater than 0 and the end pointer does not wrap around the beginning of the elements buffer, the result value is computed by applying the *inverse* aggregate function on the values from the back stack in the positions of start and end pointers (line 16); ***c)*** if the end pointer has wrapped, the result is computed by combining `backStack[endPtr]` and `frontStack[winSize-startPtr]` in line 14.

In Figure 2, we present an example of SlideSide for two windows of size 3 ($Q1$) and 4 ($Q2$) and slide 1. The red boxes represent the result for the queries on each phase. The white boxes hold the values of the start and end pointers we discussed above. At our initial phase $t0$, the values 3, 4 and 2 have been already inserted in the *elements* buffer (from left to right) and their `prefix-scan` is computed in the back stack above (3, 7, 9). In the next phase $t1$, we have an insertion in the last slot of the *elements* buffer, which triggers the computation of the `prefix-scan` again for `backStack[4]` by combining the values 9 (previous result) and 8.

After the insertion, the algorithm is ready to emit results for both queries. For $Q2$, the result contains all the elements and the window starts from position 0 (case ***a)*** from above). Thus the result is already computed by the `prefix-scan` and can be obtained by accessing the value of `backStack[3]` which is 17. For $Q1$, the result contains only the latest three elements and the window points at positions 1 and 3 (case ***c)***). Thus, the result is computed by applying the *inverse* aggregate function on the elements from the back stack placed at that positions (17-3=14).

When we reach the end of the elements buffer ($t2$), we wrap around to the beginning and we compute a `suffix-scan` over the input using the front stack. In phase $t3$, we have the first eviction, where the latest input value 5 replaces value 3 and `backStack[1]` becomes equal to 5. Now, both query windows have wrapped (case ***b)***). This operation is going to return the `suffix-scan` of the remaining elements after evictions using the front stack and the current running aggregate from the back stack, which results in 15 and 19 respectively for queries $Q1$ and $Q2$.

3.2 Non-Invertible Aggregates

Processing multiple non-invertible functions can be performed with the Algorithm 1, but the `suffix-scans` have to be triggered more frequently. The algorithm is omitted to conserve space, but the logic is similar. The intuition behind this approach is that, while each of the queries maintains and operates on its own pair of stacks, these can be overlayed and start at the same memory address. In effect, the smallest stack is stored in the same memory region as the bottom part of the next larger, and so forth. To preserve correctness among the results, the computation of the `suffix-scan` is triggered every time the query with the smallest window size starts to evict.

In addition to the previous observations, we can apply optimizations proposed for single-query evaluation in Hammer-Slide [8], such as maintaining only the top value of the back stack. During the `suffix-scan` computation we can also stop propagating the changes from the current position until the end of the stack, if our computations do not alter the aggregate values, which reduces greatly the overhead of multiple flip phases and result in constant amortized complexity (see Table 1).

4 EVALUATION

In this section, we evaluate SlideSide for both invertible and non-invertible functions to show the benefits of our incremental strategy. To evaluate the efficiency of different aggregation algorithms, we run our experiments as a standalone prototype. We compare SlideSide to `SlickDeque` (for non-invertible functions we tradeoff performance with memory by using a fixed size deque), `TwoStacks` (using optimizations from [8]), `SoE` and `FlatFAT` when it's applicable (e.g., `SoE` is evaluated only for invertible functions). Each prototype maintains sliding windows with slide 1 by performing an eviction, an insertion and producing a result. We start our evaluation with the case we focus on: multi-queries and we demonstrate that SlideSide achieves higher performance. After that, we study the performance in the single query case, in which our solution exhibits only small performance loss.

4.1 Experimental setup and workloads

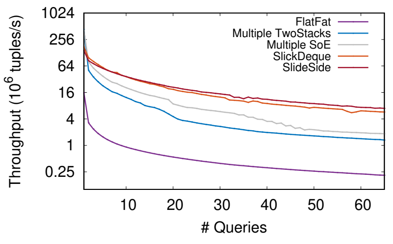
Hardware. All experiments are performed on a server with 2 Intel Xeon E5-2640 v3 2.60 GHz CPUs, a 20MB LLC cache and 64 GB of memory. We used Ubuntu 18.04 with 4.15.0-50-generic Linux kernel and compiled all experiments with clang++ version 9.0.0 and optimization level `-O3`.

Workload. Our workload emulates an anomaly detection scenario using the energy consumption trace from a smart electricity grid. This trace contains smart meter data from electrical devices in households [5] (32 bytes tuple size). We use two queries to perform analysis over the stream and detect outliers: SG_1, an aggregation that computes a sliding global AGG_{sum} and SG_2, which computes a sliding global AGG_{min} over the meter load.

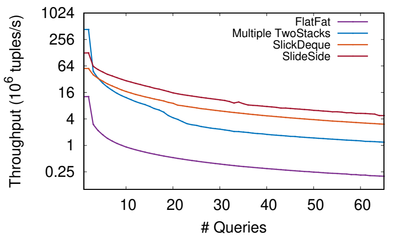
4.2 Multi-Query Evaluation

In the multi-query experiments, we generate queries of uniformly random window sizes (within the range [1, 32768] of tuples), while maintaining a constant window slide of 1 tuple for all of them. In this setup, we created workloads that contain from 1 up to 65 concurrent queries. `TwoStacks` and `SoE` can not be used to evaluate multiple queries, so we replicate their data-structures for every single window definition, as illustrated in Table 1.

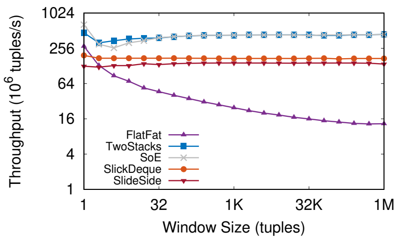
Invertible Functions. For invertible functions, we are computing query SG_1 over different window definitions. Figure 3a demonstrates that `SoE` is the fastest algorithm and outperforms the multi-query solutions by up to 2.5× for a single query. However, as the number of queries increases, the overhead of maintaining multiple data-structure replicates becomes noticeable. Thus, we observe that the multi-query algorithms perform nearly 4×. Comparing SlideSide with `SlickDeque` reveals a small performance benefit that reaches up to 40% with the increase of query concurrency. Our approach allows the compiler to generate more efficient code, because of the simpler CPU instruction stream, while providing more predictable memory access.



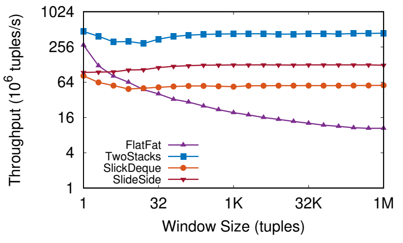
(a) Throughput of multi-queries (AGG$_{\mathrm{sum}}$)



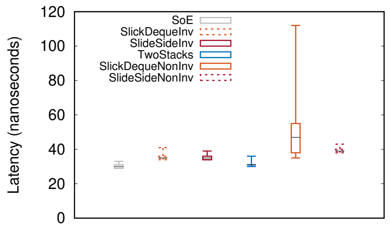
(b) Throughput of multi-queries (AGG$_{\mathrm{min}}$)



(c) Throughput of single-query (AGG$_{\mathrm{sum}}$)



(d) Throughput of single-query (AGG$_{\mathrm{min}}$)



(e) Latency in nanoseconds

Figure 3: Comparison of incremental techniques

Non-Invertible Functions. For the non-invertible functions we are computing the AGG$_{\mathrm{min}}$ over the generated windows. In Figure 3b, we observe that the multiple instances of TwoStacks outperform both SlideSide and SlickDeque for the first two and three workloads respectively. After that point, SlideSide is from 70% up to 2.2× faster compared to SlickDeque and more than 4× compared to the other two techniques. This illustrates that even though SlideSide requires more memory compared to SlickDeque, its CPU-cache-friendly data layout scales better with the number of queries in comparison to the deque data structure.

4.3 Performance Overhead for Single-Query

In this section, we present the efficiency of SlideSide for single-query workloads. We use queries SG$_{1\text{-}2}$ to measure throughput and latency of the aforementioned approaches.

Throughput. For this experiment, we use SG$_1$ and SG$_2$ over windows with window sizes that vary between 1 and 1048576 tuples. Figure 3c illustrates the throughput penalty introduced by our algorithm for invertible functions in a single query scenario. SlideSide exhibits throughput nearly 3× worse than SoE and TwoStacks. In Figure 3d, we observe that TwoStacks is the best-performing non-invertible algorithm for different window sizes (440 million tuples/sec). In contrary, SlideSide is 3× worse but exhibits better performance than SlickDeque, because of its underlying data structure with sequential memory layout.

Latency. To measure the latency of all the previous approaches, we use a fixed window size of 32K tuple and window slide of 1. In Figure 3e, we omit the latency of FlatFAT, as it consistently is an order of magnitude higher than the other algorithms. We show that SlideSide exhibits latency that is comparable to the best-performing solutions for both invertible and non-invertible functions (minimal overhead) and better compared to the other multi-query solution, SlickDeque.

Overall, we observe that for single query evaluation SlideSide ends up exhibiting nearly 3× worse performance in throughput and similar latency compared to the best-performing approaches. This is the result of the memory pressure from maintaining extra dependencies (not needed by a single-query) along with a more complex CPU instruction stream that hinders optimizations.

5 CONCLUSION

In this paper, we presented a novel algorithm for highly efficient evaluation of multiple aggregate queries by maintaining a prefix- and a suffix-scan over the input. Our algorithm can be used as a drop-in replacement for any associative aggregation operator in a commercial streaming system, such as Flink [2] (e.g., as an aggregate store for Scotty[9]). SlideSide outperforms the state-of-the-art algorithms in multi-query scenarios by up to 2× in throughput, while exhibiting better latency. However, our study reveals that current window aggregation techniques do not exhibit robust performance across different types of aggregation functions and concurrency levels. Thus, a streaming engine will either perform poorly for different points within this design space or have to maintain multiple algorithms with a cost model.

REFERENCES

[1] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (June 2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[2] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™ : Stream and Batch Processing in a Single Engine. (2015). https://doi.org/10.1109/IC2EW.2016.56

[3] J. Gray, A. Bosworth, A. Lyaman, and H. Pirahesh. 1996. Data cube: a relational aggregation operator generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS. In *Proceedings of the Twelfth International Conference on Data Engineering*. 152–159. https://doi.org/10.1109/ICDE.1996.492099

[4] Martin Hirzel, Scott Schneider, and Kanat Tangwongsan. 2017. Sliding-Window Aggregation Algorithms. *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems - DEBS '17* (2017), 11–14. https://doi.org/10.1145/3093742.3095107

[5] Zbigniew Jerzak and Holger Ziekow. 2014. The DEBS 2014 Grand Challenge. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. ACM, New York, NY, USA, 266–269. https://doi.org/10.1145/2611286.2611333

[6] Anatoli U Shein, Panos K Chrysanthis, and Alexandros Labrinidis. 2018. SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation. Section 4 (2018), 397–408. https://doi.org/10.5441/002/edbt.2018.35

[7] Kanat Tangwongsan and Martin Hirzel. 2015. General Incremental Sliding-Window Aggregation. *Pvldb* 8, 7 (2015), 702–713. https://doi.org/10.14778/2752939.2752940

[8] Georgios Theodorakis, Alexandros Koliousis, Peter R. Pietzuch, and Holger Pirk. 2018. Hammer Slide: Work- and CPU-efficient Streaming Window Aggregation, See [8], 34–41. http://www.adms-conf.org/2018-camera-ready/SIMDWindowPaper_ADMS%2718.pdf

[9] J. Traub, P. M. Grulich, A. Rodriguez Cuellar, S. Bress, A. Katsifodimos, T. Rabl, and V. Markl. 2018. Scotty: Efficient Window Aggregation for Out-of-Order Stream Processing. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1300–1303. https://doi.org/10.1109/ICDE.2018.00135

Efficient Enumeration of Four Node Graphlets at Trillion-Scale

Yudi Santoso
University of Victoria
BC, Canada
santoso@uvic.ca

Venkatesh Srinivasan
University of Victoria
BC, Canada
srinivas@uvic.ca

Alex Thomo
University of Victoria
BC, Canada
thomo@uvic.ca

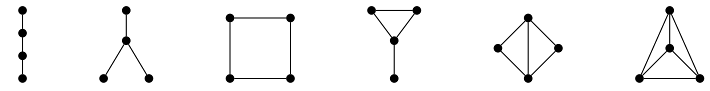


Figure 1: Four node graphlets: a 3-path, a 3-star, a rectangle or 4-cycle, a tailed-triangle, a diamond, and a 4-clique.

ABSTRACT

Graphlet enumeration is known to be a challenging task in graph analysis. This is because the cost is exponential in the order of the graphlet. Triangle is a graphlet of order three that has received special attention because it is relatively small but non-trivial, and can still be enumerated quite fast even for massive graphs of millions of nodes and edges. In this paper, we propose an efficient algorithm for enumerating four node graphlets, such as 4-cycles, 4-cliques, diamonds, etc by leveraging the most efficient algorithm for triangle enumeration. We show that despite the belief that any such enumeration algorithm cannot terminate in reasonable time, our method can handle large graphs containing trillions of such graphlets, using a single commodity machine, within a reasonable amount of time.

1 INTRODUCTION

It is commonly thought that graphlets, beyond three nodes, are difficult to enumerate. This is because the number of possible instances grows as $O(n^k)$, where k is the order of the graphlet and n is the order of the graph. Thus, for massive graphs, it was believed that an enumeration algorithm, which has to touch each graphlet, cannot terminate in a reasonable time [11]. Indeed, previous methods, such as Fanmod [18] and Rage [8], do not scale well and take a very long time to run on million scale graphs. Other proposed solutions, such as Arabesque [17] and PGD [1] use distributed platforms. However, our focus is to explore the limits of what can be achieved using single-machine algorithms.

There are several algorithms proposed in the literature to count the number of the graphlets. They are either estimates using approximation methods, such as Graft [13], or exact counting without full enumeration, notably Orca [6] and Escape [11]. However, what if we need to find each of the graphlet instances? Knowing where the graphlets are is useful in analysing the local structures of the graph. For example, enumerating graphlets is important in detecting cancer through differential graphlet communities [19]. Also, enumeration can yield graphlet degree counts which are useful for uncovering biological network functions [9].

It is worth noting that there have been plenty of studies on triangle enumeration. It was found that triangles can be enumerated quite efficiently using the compact forward edge-iterator algorithm [7]. In general, graphlets of order k can be enumerated using an algorithm with runtime $O(nd^{k-1})$ where d is the maximum degree [15]. However, in [14] it was shown that through careful preprocessing, triangle enumeration using edge-iterator can be significantly faster than $O(nd^2)$ time. Can we achieve a better runtime than $O(nd^{k-1})$, for higher order graphlets?

In this paper we show that efficient enumeration for triangles can be leveraged to enumerate higher order graphlets, in particular four node graphlets. Our algorithm achieves a significantly improved runtime, which depends on the number of three-node graphlets and is able to handle large graphs efficiently on a single machine. Moreover, unlike most in the literature, our solution yields the counts of *all* four node graphlets in a *single* run.

Our contributions are as follows:

(1) We propose a new algorithm to enumerate *all* types of four node graphlets of an undirected graph on a single run. Enumeration is done carefully so that no graphlet is listed more than once.
(2) We provide detailed analyses on the algorithm correctness and time complexity. We refine the time upper-bound of enumeration to depend on the number of three-node graphlets and thus be significantly better than $O(nd^3)$ for real-world networks.
(3) We create an efficient implementation of the algorithm for a single machine. Our algorithm is able to run on graphs of millions nodes and edges, which contain trillions of graphlets, within reasonable time.

2 RELATED WORK

Chiba and Nishizeki published several subgraph listing algorithms [5] which can be considered as a pioneering work on graphlet enumeration. Milo et al. [10] analysed frequent subgraph patterns, and called them network motifs. Since then, there have been many studies on how to find and count small subgraphs within a graph or network, including those we already discussed in the Introduction. Also, Silvestri [16] provided another complexity analysis on subgraph enumeration. To the best of our knowledge, there has not been a solution using the method that we propose here, to simultaneously, and fully, enumerate all the graphlets (of order four) through triangles and wedges, and that can scale to large graphs using a single machine.

3 PRELIMINARIES

In this paper we solely work on simple undirected graphs. We denote a graph by $G(V, E)$ where V is the set of nodes and E is the set of edges. Let $n = |V|$ and $m = |E|$. The degree of a node is the number of edges incident on it. For simple graphs, there is no self-loop and the degree is equal to the number of neighbours. We

Algorithm 1 Triangle Enumeration

```
Input: An undirected graph $G(V, E)$ in an adjacency list representation
1: for all vertex $u \in V(G)$ do
2:     for all vertex $v \in N(u)$ do
3:         if $v > u$ then
4:             for all $w \in N(u) \cap N(v)$ do
5:                 if $w > v$ then
6:                     EnumerateTriangle $(u, v, w)$
```

denote the set of neighbours of node u by $N(u)$, and the degree of u by $d(u) = |N(u)|$. A *subgraph* of $G(V(G), E(G))$ is a graph $H(V(H), E(H))$ such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. We use the notation $H \subseteq G$ to say that H is a subgraph of G. A subgraph $H \subseteq G$ is an *induced* subgraph if any edge (u, v) (which $= (v, u)$ for undirected graphs), with $u, v \in V(H)$, is in $E(H)$ if and only if (u, v) is in $E(G)$. A subgraph is *connected* if every pair of nodes in it is connected by a path of edges. We assume the following definition: A *graphlet* is an induced connected subgraph.

There are two kinds of graphlets of three nodes: wedge (Henceforth labeled as g_1) and triangle (g_2). Note that for induced subgraphs, wedges and triangles cannot be on top of each other (i.e., a wedge and a triangle cannot have the same set of three nodes within the same graph.). For four nodes, we have six types of graphlets, depicted in Figure 1. These are 3-path (or four-node-path) (g_3), 3-star (g_4), 4-cycle or rectangle (g_5), tailed-triangle (g_6), diamond (g_7), and 4-clique (g_8). The labels that we use here follow the common labelling used by some papers in the literature [2, 12].

Notice that g_6, g_7 and g_8 contain triangle(s). A g_6 contains one triangle, a g_7 contains two triangles, and a g_8 contains four triangles. This fact suggests that we can find them through the triangles in the graph. Whenever we find a triangle, we can check if this triangle is a part of any g_6, g_7 and/or g_8. Similarly, g_3, g_4 and g_5 contain two, three and four wedges, respectively. Therefore, we can find them through wedges.

We list graphlets by their nodes. Thus, for example, $(u, v, w, z)_8$ is a g_8 with nodes u, v, w and z. In enumeration, some care is needed to avoid multiple listing. Without lost of generality, we can use label 1, 2, 3, 4 to represent the nodes in a graphlet. Clearly, $1 < 2 < 3 < 4$, so 1 represents the smallest node.

There are $3! = 6$ permutations of three nodes. Therefore, for **wedges**, we have $(1, 2, 3)_1$, $(1, 3, 2)_1$, $(2, 1, 3)_1$, $(2, 3, 1)_1$, $(3, 1, 2)_1$ and $(3, 2, 1)_1$. However, $(1, 2, 3)_1$ is the same wedge as $(3, 2, 1)_1$, $(1, 3, 2)_1$ is the same as $(2, 3, 1)_1$, and $(2, 1, 3)_1$ is the same as $(3, 1, 2)_1$. Thus, we have only three possible wedges, only one can be present (for induced case). Our convention is to list with the smaller leg first, i.e. $(1, 2, 3)_1$, $(1, 3, 2)_1$, and $(2, 1, 3)_1$. We can divide these into two types: those with the smallest node at the center of the wedge (type 1), i.e., $(2, 1, 3)_1$, and those with the smallest node at one of the legs (type 2), i.e., $(1, 2, 3)_1$ and $(1, 3, 2)_1$. We will see that they require separate treatment. For **triangles**, all six permutations are isomorphic. Therefore, we only need one to list. We choose the one with the nodes ordered ascendingly: $(1, 2, 3)_2$.

Now for four nodes, there are $4! = 24$ permutations. For **3-paths**, by symmetry we only need half (i.e. twelve) of them. For **3-stars**, we have four distinct ones depending on which one is the centre. For **4-cycles**, the cyclic symmetry gives us a factor of four, while the clockwise counter-clockwise symmetry gives

Algorithm 2 Graph-Prep

```
Input: An undirected graph $G(V, E)$
1: Sort $V$ based on the degrees, in ascending order.
2: Relabel the vertices according to their new order.
3: Build adjacency list of the sorted and relabeled vertices.
4: Cut out the smaller neighbours from each neighbour list.
```

us a factor of two. Therefore, we have only $24/8 = 3$ distinct permutations. For **tailed-triangles**, the distinguishing nodes are the end node and the centre node, giving us $\binom{4}{2}$ or twelve distinct configurations. For **diamonds**, we have a pair of triangles. Let us call the two end nodes of the shared edge as the connecting nodes, and the other two nodes as the opposing nodes. There are symmetries between the two opposing nodes, and between the two connecting nodes, giving us $24/2/2 = 6$ distinct configurations. For **4-cliques**, we can exchange any pair of nodes and get the same clique. Thus there is only one unique configuration, and we choose to list the nodes in order: $(1, 2, 3, 4)_8$.

4 THE ALGORITHMS

The algorithm that we use for triangle enumeration is an edge iterator algorithm (Algorithm 1) combined with nodes ordering. This combination is similar to the Compact-Forward algorithm [7] but with the ordering done in a pre-processing before the enumeration (Algorithm 2).

Algorithm 3 Triangle and Wedge Enumeration

```
Input: An undirected graph $G(V, E)$ in an adjacency list representation
1: for all vertex $u \in V(G)$ do
2:     for all vertex $v \in N(u)$ do
3:         if $u < v$ then
4:             for all $u' \in N(u)$ and $v' \in N(v)$ do
5:                 if $(u' > u) \wedge (v' > u)$ then
6:                     if $u' = v' > v$ then
7:                         EnumerateTriangle $(u, v, u')$
8:                     if $(u' < v') \wedge (u' > v)$ then
9:                         EnumerateWedgeType1 $(v, u, u')$
10:                    if $u' > v'$ then
11:                        EnumerateWedgeType2 $(u, v, v')$
```

The graph preprocessing is based on the following observations: (i) Because of lines 3 and 5 of Algorithm 1 we need to consider only bigger neighbours of every vertex, i.e., $N^{>}(u) = \{v \in N(u) | v > u\}$. (ii) The triangle count in a graph will not change if we relabel the vertices.

Algorithm 1 can be modified to enumerate the wedges as well. This is shown in Algorithm 3. Notice that by condition on line 3 we assure that u is always smaller than v. To avoid multiple listing, when we iterate neighbours of v we consider only those that are bigger than u (line 5). However, we need to include smaller neighbours of v (i.e. those between u and v) to catch all of the wedges.

We extend each of the EnumerateTriangle and EnumerateWedge functions above to search for four-node graphlets. Whenever we find a triangle, $(u, v, w)_2$, we call the ExploreTriangle function (Algorithm 4), which checks for the intersections among the neighbour sets of the three triangle nodes, $N(u)$, $N(v)$ and $N(w)$. If we find a $z \in N(u) \cap N(v) \cap N(w)$, then (u, v, w, z)

Algorithm 4 Explore Triangle

```
Input: Given triangle $(u, v, w)_2$, $u < v < w$: $N(u)$, $N(v)$, $N(w)$.
1: Compute intersections among the three neighbour sets.
2: for all $z \in N(u) \cap N(v) \cap N(w)$ with $z > w$ do
3:     EnumerateCLIQUE $(u, v, w, z)_8$
4: for all $z$ in two sets and $z >$ opposite node do
5:     EnumerateDiamond $(.)_7$
6: for all $z$ in one set only do
7:     EnumerateTailedTriangle $(.)_6$
```

is a four-clique (i.e. g_8). A node z that is in two of the three neighbour sets gives us a diamond (i.e. g_7), while a z that is in only one of the three neighbour sets gives us a tailed triangle (i.e. g_6). For 4-cliques, we can use the sets of larger neighbours. For diamonds, we can use the sets of neighbours larger than u. For the tailed triangles, however, we need to include all of the neighbours. Due to this last case we lost some of the advantage of the graph preprocessing. As a result, the runtime might be much longer compared to the triangle enumeration time, depending on the maximum degree.

Algorithm 5 Explore Wedge Type-1

```
Input: Given wedge $(v, u, w)_1$, $u < v < w$: $N^{>u}(u)$, $N^{>u}(v)$, $N^{>u}(w)$.
1: Compute intersections among the three neighbour sets.
2: for all $z \in N^{>u}(v) \cap N^{>u}(w)$ with $z \notin N^{>u}(u)$ do
3:     EnumerateRectangle $(u, v, z, w)_5$
4: for all $z \in N^{>u}(u)$ only do
5:     if $z > w$ then
6:         Enumerate3Star $(u, v, w, z)_4$
7: for all $z \in N^{>u}(v)$ only do
8:     Enumerate3Path $(w, u, v, z)_3$
9: for all $z \in N^{>u}(w)$ only do
10:    Enumerate3Path $(v, u, w, z)_3$
```

For the wedges, we call two different functions depending on the type of the wedge, either Algorithm 5 or 6. In this two functions we only need sets of neighbours that are larger than u, but this is not done in a pre-processing. Notice that in Algorithm 6 w can be smaller than v, which is the center of the wedge.

Algorithm 6 Explore Wedge Type-2

```
Input: Given wedge $(u, v, w)_1$, $u < v$, $u < w$: $N^{>u}(u)$, $N^{>u}(v)$, $N^{>u}(w)$.
1: Compute intersections among the three neighbour sets.
2: for all $z \in N^{>u}(v)$ only do
3:     if $z > w$ then
4:         Enumerate3Star $(v, u, w, z)_4$
5: for all $z \in N^{>u}(w)$ only do
6:     if $z \neq v$ then
7:         Enumerate3Path $(u, v, w, z)_3$
```

5 ANALYSIS

Theorem 1. *Algorithm 3 correctly enumerates wedges and triangles in an undirected graph.*

Proof. Each edge (u, v), with $u < v$, is iterated once and only once. For each, we enumerate all the intersecting neighbours (i.e., triangles), and non-intersecting neighbours (i.e., wedges). Thus, all wedges and triangles in the graph would be found. For triangles, we avoid multiple listing by imposing condition $u' = v' > v$. For type-1 wedges we impose condition $u' > v$. For type-2 wedges, since $u < v'$ there will be no double counting. □

Theorem 2. *Algorithms 4, 5 and 6, combined with algorithm 3, correctly enumerate all four node graphlets in an undirected graph.*

Proof. As proven above, all triangles and wedges are enumerated once. For each triangle, the three neighbour sets are checked. Each node that is in only one of the sets yields a tailed-triangle. All tails would be found in the sets. A node that is in the intersection of two sets yields a diamond. By asserting that this node is larger than the opposite node in the diamond we assure that any diamond would be listed just once. A node that is in the intersection of all three sets yields a 4-clique. We assert that this node is larger than any node in the triangle to assure that the clique has not been listed in any previous iteration. For wedges, similarly, all four node graphlets attached to each wedge would be found. Multiple listing is avoided by considering only 3-paths, 3-stars and 4-cycles, and by careful conditions on the node ordering. For the 3-paths we make sure that the smallest node is always in the first half of the path. For the 3-stars we make sure that the fourth node is greater than the third node. The center node does not need to be the smallest. For 4-cycles we make sure that the fourth node is opposite to the first node. □

Theorem 3. *The runtime of the four node graphlet enumeration is bounded by $O((N_\Delta + N_\angle)\, d_{\max} + T_{3g})$, where N_Δ ($N_\angle$) is the number of triangles (wedges), and T_{3g} is the time to enumerate triangles and wedges.*

Proof. For each triangle and wedge the algorithm runs through the neighbor sets to check the intersections with cost $\leq (d(u) + d(v) + d(w))$. □

Note that in general $(N_\Delta + N_\angle) \lesssim nd^2_{\max}$, with the upper value is satisfied by a regular graph. However, for all real world networks, we have $(N_\Delta + N_\angle) \ll nd^2_{\max}$. Also, $T_{3g} \ll nd^2_{\max}$ using efficient enumeration. Therefore, in practice, our runtime is much less than worst case bound of $O(nd^3_{\max})$.

6 EXPERIMENTS

The networks that we study are listed in Table 2. All of the datasets were downloaded from the Laboratory for Web Algorithmics [3, 4], http://law.di.unimi.it/datasets.php. We symmetrized them and got rid of any loops to get simple undirected graphs. We implemented our code in Java with parallel streams, and use Webgraph library [4]. We used a Linux machine with dual Xeon E5-2620 processors of 24 threads and 128 GB of RAM. We notice, however, that the memory usage is < 1 GB throughout the experiment.

The graphlet counts are listed in Table 1. We can check that for all of these graphs, $N_\Delta + N_\angle \ll nd^2_{\max}$ using their $d_{\max}$ values from Table 2. For example, for **amazon**, $N_\Delta + N_\angle \approx$ 42M and $nd^2_{\max} \approx$ 853B, a four order of magnitude difference. For all of the graphs that we consider here the difference is from three to five orders of magnitude.

The runtimes are shown in Table 3. We include the triangle enumeration time, T_Δ, for comparison. As wedges cannot take full advantage of the pre-processing, they take longer time for enumeration, hence T_{3g} is larger than T_Δ. Notice that the preprocessing time, T_{Prep}, is just about the same magnitude as T_Δ.

Table 1: Counts of the graphlets. The dewiki dataset needs longer than our time limit to terminate.

Graph	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8
enron	40,309,453	1,067,993	2,511,039,670	8,043,804,283	21,598,984	582,841,848	46,141,288	5,001,773
cnr	7,798,287,209	20,977,629	6,118,026,632	41,392,015,937,553	37,876,822,234	79,429,334,745	42,974,515,602	159,814,399
dblp	81,529,950	7,005,235	2,678,518,695	3,545,925,764	1,483,611	543,447,587	21,608,538	40,910,658
amazon	38,015,403	4,464,791	372,366,885	609,961,827	2,689,696	9,232,707	13,096,219	4,192,682
dewiki	51,141,107,679	88,611,282	..	..	..	..	..	..
ljournal	8,726,048,197	411,155,444	1,812,284,632,329	8,847,128,736,944	8,551,292,956	189,716,360,703	26,962,410,402	16,129,080,442

Note that T_{4g}, the time required to enumerate all 3 and 4-node graphlets, does not strongly depend on the size of the graph, but rather on the degrees and the numbers of triangles and wedges, validating our analysis. For example, comparing **ljournal** with **amazon**, the ratio of their $(N_\angle + N_\Delta)\, d_{\max}$ values is about four thousand, while the ratio of their T_{4g} values is about six thousand, i.e. approximately the same order. This observation experimentally validates the statement of Theorem 3 relating the runtime to the $(N_\angle + N_\Delta)\, d_{\max}$ value.

Interestingly, **cnr** requires longer runtime than **ljournal**. Even though it is smaller by an order of magnitude it has more graphlets. The **amazon** dataset, which has relatively small maximum degree can be processed in merely 14 seconds. The **dewiki** dataset has enormous number of wedges and large maximum degree and the algorithm did not terminate even after running for four days.

Table 2: The undirected graphs. Here, $d^{\mathrm{BG}}_{\max}$ is the effective maximum degree when only larger neighbours are included after the preprocessing.

Dataset	n	m	$d_{\max}$	$d^{\mathrm{BG}}_{\max}$	d_{avg}
enron	69,244	254,449	1,634	87	7.35
cnr	325,557	2,738,969	18,236	85	16.83
dblp	986,324	3,353,618	979	118	6.80
amazon	735,323	3,523,472	1,077	16	9.58
dewiki	1,532,354	33,093,029	118,246	490	43.19
ljournal	5,363,260	49,514,271	19,432	756	18.46

Table 3: The runtime, in seconds, for preprocessing, for triangle enumeration, for wedges and triangles together, and for all three and four node graphlets together.

Graph	T_{Prep}	T_Δ	T_{3g}	T_{4g}
enron	0.87	1.03	3.49	76.50
cnr	1.93	1.75	57.03	176K
dblp	4.87	1.93	3.45	62.05
amazon	5.80	1.73	2.53	14.05
dewiki	26.45	12.79	517.3	> 300K
ljournal	46.68	32.96	257.1	82K

7 CONCLUSIONS

In this study we have shown that it is possible to enumerate all types of four node graphlets simultaneously with runtime $O((N_\Delta + N_\angle)\, d_{\max} + T_{3g})$. Wedges and triangles can be enumerated relatively fast (in a pre-run) and the result can be used to estimate the time needed to enumerate the four node graphlets. We found that the runtime upper bound depends more on the maximum degree than on the size of the graph. Our algorithm can finish the enumeration in seconds when the maximum degree is around 1K. Moreover, it does not require large memory space, and it would run for even larger graphs (provided that we allow enough time). Notably, we were able to process massive graphs of millions of nodes and edges and enumerate about 40 trillions graphlets in a single run, within a reasonable amount of time.

REFERENCES

[1] Nesreen K Ahmed, Jennifer Neville, Ryan A Rossi, and Nick Duffield. 2015. Efficient graphlet counting for large networks. In *2015 IEEE International Conference on Data Mining*. IEEE, 1–10.

[2] Mansurul A Bhuiyan, Mahmudur Rahman, Mahmuda Rahman, and Mohammad Al Hasan. 2012. Guise: Uniform sampling of graphlets for large graph analysis. In *2012 IEEE 12th International Conference on Data Mining*. 91–100.

[3] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 587–596.

[4] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[5] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM Journal on computing* 14, 1 (1985), 210–223.

[6] Tomaž Hočevar and Janez Demšar. 2014. A combinatorial approach to graphlet counting. *Bioinformatics* 30, 4 (2014), 559–565.

[7] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.* 407, 1-3 (2008), 458–473. https://doi.org/10.1016/j.tcs.2008.07.017

[8] Dror Marcus and Yuval Shavitt. 2012. Rage–a rapid graphlet enumerator for large networks. *Computer Networks* 56, 2 (2012), 810–819.

[9] Tijana Milenković and Nataša Pržulj. 2008. Uncovering biological network function via graphlet degree signatures. *Cancer informatics* 6 (2008).

[10] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.

[11] Ali Pinar, C Seshadhri, and Vaidyanathan Vishal. 2017. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1431–1440.

[12] Nataša Pržulj, Derek G Corneil, and Igor Jurisica. 2004. Modeling interactome: scale-free or geometric? *Bioinformatics* 20, 18 (2004), 3508–3515.

[13] Mahmudur Rahman, Mansurul Alam Bhuiyan, and Mohammad Al Hasan. 2014. Graft: An efficient graphlet counting method for large graph analysis. *IEEE Transactions on Knowledge and Data Engineering* 26, 10 (2014), 2466–2478.

[14] Yudi Santoso. 2018. *Triangle counting and listing in directed and undirected graphs using single machines.* Master's thesis. University of Victoria.

[15] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten Borgwardt. 2009. Efficient graphlet kernels for large graph comparison. In *Artificial Intelligence and Statistics*. 488–495.

[16] Francesco Silvestri. 2014. Subgraph enumeration in massive graphs. *arXiv preprint arXiv:1402.3444* (2014).

[17] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 425–440.

[18] Sebastian Wernicke and Florian Rasche. 2006. FANMOD: a tool for fast network motif detection. *Bioinformatics* 22, 9 (2006), 1152–1153.

[19] Serene WH Wong, Nick Cercone, and Igor Jurisica. 2015. Comparative network analysis via differential graphlet communities. *Proteomics* 15, 2-3 (2015), 608–617.

Short paper

Band Joins for Interval Data

Panagiotis Bouros
Johannes Gutenberg University
Mainz, Germany
bouros@uni-mainz.de

Konstantinos Lampropoulos
University of Ioannina, Greece
klampropoulos@cs.uoi.gr

Dimitrios Tsitsigkos
University of Ioannina, Greece
Athena Research Center, Greece
dtsitsigkos@imis.
athena-innovation.gr

Nikos Mamoulis
University of Ioannina, Greece
nikos@cs.uoi.gr

Manolis Terrovitis
Athena Research Center, Greece
mter@imis.athena-innovation.gr

ABSTRACT

Interval data are found in a wide range of applications (e.g., validity intervals in temporal databases, ranges of uncertain values in probabilistic databases, etc.) We study the efficient (parallel) evaluation of *band joins* for interval data. Specifically, given two collections R and S of intervals, the objective is to find all pairs (r, s), such that $r \in R$, $s \in S$, and the *difference gap* between r and s is at most equal to a given threshold ϵ. We first show how this problem can be solved by directly applying the state-of-the-art domain-based partitioning approach for interval joins, after extending the intervals by ϵ. Then, we propose a novel partitioning strategy for the original intervals, which defines the partitions using the threshold ϵ and achieves much better performance, on most datasets, for reasonably large values of ϵ.

1 INTRODUCTION

The evaluation of joins with non-equality predicates finds many applications, especially in data domains where values are approximate by nature (e.g., temporal data). For example, one application is finding pairs of events whose time difference is not greater than a given threshold ϵ. This bounded-difference join is also called *band join* [6], since for each value v in one join input, the objective is to find the values in the other input, which are inside an $[-\epsilon, \epsilon]$ band around v.

Although the evaluation of band joins has already been studied for offline (disk-resident) [6, 10] and streaming data [1, 7], previous work focuses on joins between collections of values (not intervals). In addition, the possibilities of parallel evaluation using modern hardware are not fully explored. In this paper, we study the evaluation of band joins between two collections of intervals. Specifically, given two collections R and S of intervals and a band constraint ϵ, our objective is to find all pairs (r, s) of intervals, such that $r \in R$, $s \in S$, and the *difference gap* between r and s is at most ϵ. More precisely, for two intervals $r = [r.start, r.end]$ and $s = [s.start, s.end]$ to qualify the join, it should be $s.start \leq r.end + \epsilon$ and $r.start \leq s.end + \epsilon$.

To our knowledge, this problem has not been studied before, although it has important applications. For example, the user of a temporal database [9] may often be interested in finding pairs of intervals that qualify some overlap or distance constraints (e.g., find pairs of flights which do not have a difference gap larger than 2 hours). Band joins can also be useful for *coalescing* pairs of intervals that overlap or they are close to each other [2]. Another application is in probabilistic databases, where uncertain values are often approximated by confidence intervals [5, 11]. Finding pairs of values that differ less than a threshold ϵ can be modeled and solved as a band interval join problem. XML queries can be modeled as joins between intervals that capture the positions and ancestor-descendant relationships scope of nodes in XML trees [4]. Queries over data streams [1] also constrain the time differences between events, which could be instantaneous or with a temporal duration (i.e., intervals); hence, streaming data analytics could benefit from fast band join evaluation algorithms.

In our previous work [3], we studied the parallel evaluation of *interval overlap joins*, where the objective is to find the pairs of intervals from two collections that *overlap* (i.e., share at least one value). This is a special case of the problem that we study here for $\epsilon = 0$. We proposed a *domain-based partitioning* approach, which divides the data from the two collections into partitions and processes the partitions independently and in parallel, while avoiding duplicate results.

In this work, we extend our framework to evaluate interval band joins. An intuitive and straightforward approach in this direction is to *expand* the intervals in both collections by ϵ (e.g., by adding ϵ to endpoint $x.end$ of each interval x). The interval overlap join between the two collections of expanded intervals is equivalent to the interval band join on the original data inputs and, hence, we can directly apply the original approach of [3]. On the other hand, expanding the intervals increases data replication, which could slow down the evaluation of the join.

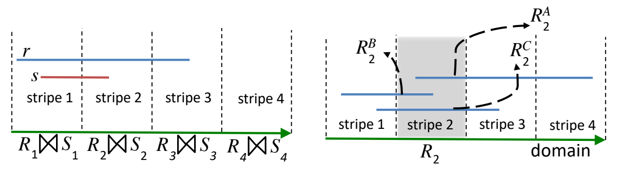
This motivated us to design an alternative approach that partitions the original intervals, as in an overlap join, but sets the width of each partition to ϵ. Our new algorithm joins each partition R_i from R with exactly two partitions from S, S_i and S_{i+1} (and vice versa), corresponding to the same and the next ϵ-wide stripes of the domain. As we show, the $R_i \bowtie S_i$ band join reduces to a cross-product, while the $R_i \bowtie S_{i+1}$ band join can be processed very efficiently, after further dividing the intervals in each partition into classes, based on the way they intersect the corresponding stripe.

We evaluate the proposed algorithm on four real datasets and varying ϵ thresholds and confirm that the ϵ-wide stripes approach is superior to the baseline adaptation of [3] on most datasets, for reasonably large values of ϵ.

2 BACKGROUND

In this section, we review the domain-based partitioning approach of [3] for interval overlap joins, which is necessary for understanding our solutions to the band interval join problem.

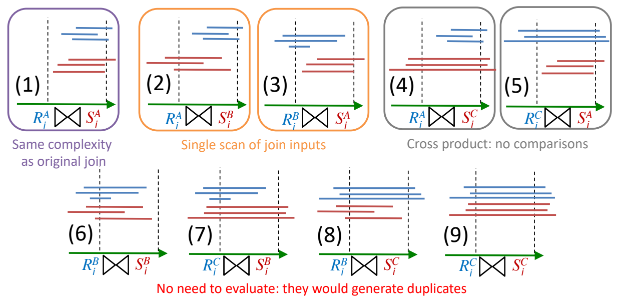
In order to process the join efficiently and in parallel, this approach first divides the data domain into disjoint regions (stripes),



r
s
stripe 1
stripe 2
stripe 3
stripe 4
$R_1 \bowtie S_1$ $R_2 \bowtie S_2$ $R_3 \bowtie S_3$ $R_4 \bowtie S_4$
R_2^A
R_2^B
R_2^C
R_2
domain

(a) Domain-based partitioning (b) mini-partitions

Figure 1: Example of domain-based partitioning



(1) $R_i^A \bowtie S_i^A$
Same complexity as original join
(2) $R_i^A \bowtie S_i^B$
(3) $R_i^B \bowtie S_i^A$
Single scan of join inputs
(4) $R_i^A \bowtie S_i^C$
(5) $R_i^C \bowtie S_i^A$
Cross product: no comparisons
(6) $R_i^B \bowtie S_i^B$
(7) $R_i^C \bowtie S_i^B$
(8) $R_i^B \bowtie S_i^C$
(9) $R_i^C \bowtie S_i^C$
No need to evaluate: they would generate duplicates

Figure 2: Breakdown of $R_i \bowtie S_i$ into mini-joins

the union of which covers the entire domain. For each of the two input collections R and S, we then define one partition per stripe and assign each interval to all stripes that the interval overlaps. Hence, an interval may span multiple partitions. After this *partitioning phase*, partition R_i (respectively, S_i) includes all intervals from R (respectively, S) which overlap the i-th stripe. During the *join phase*, each partition R_i only has to be joined with the corresponding partition S_i. Moreover, each of the $R_i \bowtie S_i$ partition-to-partition joins can be evaluated independently from the others and the joins can be processed in parallel. Figure 1(a) illustrates an exemplary domain partitioning to four stripes and two intervals; $r \in R$ is assigned to partitions R_1, R_2, and R_3 and $s \in S$ is assigned to partitions S_1 and S_2.

However, a brute-force implementation of the *join phase* may produce *duplicate results*. For example, in Figure 1(a), join pair (r, s) would be reported by both $R_1 \bowtie S_1$ and $R_2 \bowtie S_2$. Duplicates can be avoided by reporting a pair (r, s) in a partition-to-partition join $R_i \bowtie S_i$ only if at least one of r or s starts inside the i-th stripe. Otherwise, the pair would also be detected in the join of a previous stripe. Hence, in our example, pair (r, s) is reported by $R_1 \bowtie S_1$, but the pair is *pruned*, after being detected by $R_2 \bowtie S_2$.

Instead of eliminating duplicates this way, the approach of [3] goes one step further, by avoiding the generation of such duplicates overall. The idea is to further divide each partition R_i into three *mini-partitions* R_i^A, R_i^B, and R_i^C; R_i^A takes all intervals in R_i which start in stripe i, R_i^B takes all intervals in R_i which start before stripe i and end inside stripe i, and R_i^C takes all intervals that start before stipe i and end after stripe i. Figure 1(b) shows examples of three intervals from R_2 that go to different mini-partitions. Now, each $R_i \bowtie S_i$ can be computed by performing 5 *mini-joins* between the mini-partitions, as shown in Figure 2:

- $R_i^A \bowtie S_i^A$ is computed as a typical interval overlap join;
- $R_i^A \bowtie S_i^B$ and $R_i^B \bowtie S_i^A$ are computed as a special case of an interval join, where the start point of every interval in S_i^B (resp. R_i^B) precedes all start points of all intervals in R_i^A (resp. S_i^A) [3].
- $R_i^A \bowtie S_i^C$ and $R_i^C \bowtie S_i^A$ are cross products, hence their computation requires no comparisons;
- $R_i^B \bowtie S_i^B$, $R_i^B \bowtie S_i^C$, $R_i^C \bowtie S_i^B$, and $R_i^C \bowtie S_i^C$ *do not have to be computed* because they would produce duplicate join results (guaranteed to be found in previous stripes).

Mini-joins are evaluated by an optimized version of a *forward scan* algorithm based on plane-sweep (also proposed in [3]).

3 EVALUATING BAND JOINS

3.1 Evaluation based on interval overlap joins

As discussed in the Introduction, a baseline evaluation algorithm for interval *band* joins transforms the problem to an interval *overlap* join. For this purpose, it expands the intervals from both input collections by ϵ. Without loss of generality, each interval $r \in R$ and $s \in R$ becomes $r' = [r.start, r.end + \epsilon]$ and $s' = [s.start, s.end + \epsilon]$, respectively. We can easily show that if r' overlaps s' then $s.start \leq r.end + \epsilon$ and $r.start \leq s.end + \epsilon$ hold, i.e., pair (r, s) satisfies the band join predicate.

This baseline can be straightforwardly implemented using the approach of [3]. The expansion of the input intervals takes place before they are assigned to the partitions, while the mini-joins breakdown operates exactly as discussed in the previous section.

3.2 Evaluation on ϵ-wide partitions

Despite its simplicity, the baseline exhibits two shortcomings. First, due to expanding intervals by ϵ,data replication increases (compared to the replication in the overlap interval join problem). This increases the cost of the partition-to-partition joins. The second drawback is that the domain-based partitioning is agnostic to the input parameter ϵ. For instance, a pair of intervals located at the two different ends of a stripe may not qualify the band join predicate; nevertheless, they need to be checked.

To address these issues, we next propose our second solution for band joins. The key idea of the method is to split the domain into disjoint ranges (stripes), such that the width of each stripe is ϵ (in case the domain cannot be divided exactly by ϵ, the last stripe is narrower). The input intervals from R (resp. S) are not expanded, but directly assigned to every partition R_i (resp. S_i) they intersect, as described in Section 2.

Since the width of each stripe i is (at most) ϵ, it is guaranteed that every pair of interval (r, s) with $r \in R_i, s \in S_i$ forms a result of the band join. In other words, $r.end + \epsilon < s.start$ or $s.end + \epsilon < r.start$ cannot hold; otherwise, r and s would not have been assigned to the same partition. Hence, we can directly report all pairs (r, s) with $r \in R_i, s \in S_i$ as results. However, the same pair of intervals could co-exist in other stripes as well (e.g., the $(i-1)$-th and/or the $(i+1)$-th). Therefore, as in the interval overlap join case, we should only report a pair if it is not a join result in a previous stripe, i.e., if at least one of r or s start inside stripe i. To avoid this test, we can divide each partition R_i (and S_i) again into three mini-partitions R_i^A, R_i^B, R_i^C (and S_i^A, S_i^B, S_i^C), as explained in Section 2 and then evaluate *all* $R_i^A \bowtie S_i^A$, $R_i^A \bowtie S_i^B$, $R_i^B \bowtie S_i^A$, $R_i^A \bowtie S_i^C$, and $R_i^C \bowtie S_i^A$ mini-joins as *cross-products*.

However, we are not done yet. There could also be band join results (r, s), such that r *ends* in stripe i, s *starts* in stripe $i + 1$ and $r.end + \epsilon \leq s.start$ (and the symmetric case). The current decomposition has no explicit partition for all intervals that end in stripe i, i.e., the mini-partition R_i^A does not distinguish between the intervals $r \in R$ that end in stripe i from those that end after stripe i. To this end, we define an *additional* mini-partition R_i^{A1}, which contains the intervals $r \in R_i$ that both *start* and *end* inside stripe i. Mini-partition R_i^{A1} is a subset of R_i^A, but their contents are sorted differently to enhance the join evaluation, as we discuss in the next paragraph. Figure 3 shows examples of four intervals

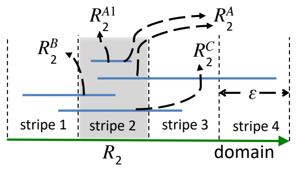


Figure 3: Mini-partitions for band joins

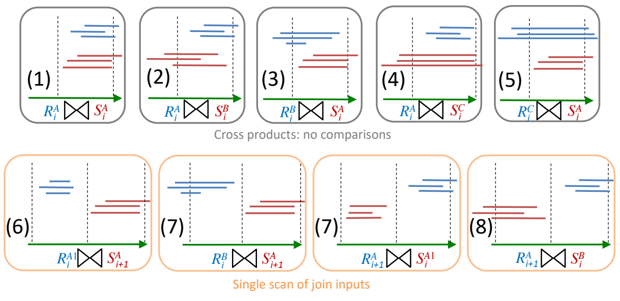

(1)
$R_i^A \bowtie S_i^A$
(2)
$R_i^A \bowtie S_i^B$
(3)
$R_i^B \bowtie S_i^A$
(4)
$R_i^A \bowtie S_i^C$
(5)
$R_i^C \bowtie S_i^A$
Cross products: no comparisons
(6)
$R_i^{A1} \bowtie S_{i+1}^A$
(7)
$R_i^B \bowtie S_{i+1}^A$
(7)
$R_{i+1}^A \bowtie S_i^{A1}$
(8)
$R_{i+1}^A \bowtie S_i^B$
Single scan of join inputs

Figure 4: Mini-joins breakdown in band joins

which are assigned to R_2^A, R_2^B, R_2^C, and R_2^{A1}. Observe that intervals which are assigned to R_2^{A1} are also assigned to R_2^A.

In Figure 4, we illustrate the set of mini-joins that have to be evaluated by the new algorithm. Besides the 5 cross-products between mini-partitions in the same stripe, we have to perform four mini-joins $R_i^{A1} \bowtie S_{i+1}^A$, $R_i^B \bowtie S_{i+1}^A$, $S_i^{A1} \bowtie R_{i+1}^A$, $S_i^B \bowtie R_{i+1}^A$, at every pair i and $i+1$ of neighboring partitions. All these mini-joins can be evaluated efficiently in a similar manner as mini-join $R_i^B \bowtie S_i^A$ of the interval overlap join problem (illustrated in Figure 2). Specifically, the intervals in mini-partitions R_i^{A1}, R_i^B, S_i^{A1}, S_i^B are sorted by $r.end$ or $s.end$ and the intervals in mini-partitions R_i^A and S_i^A are sorted by $r.start$ and $s.start$, respectively. Then, at each of the four mini-joins, we can compute the result by *concurrently scanning* the join inputs only once.

Consider, for instance, the mini-join between R_i^{A1} (sorted by $r.end$) and S_{i+1}^A (sorted by $r.start$). We first check whether $r.end + \epsilon \geq s.start$ holds for the first r in R_i^{A1} and the first s in S_{i+1}^A. If so, (r, s) is a band join result. At the same time, we can conclude that for all intervals r' that follow r in R_i^{A1}, (r', s) is also a band join result, since $r'.end \geq r.end$ holds (due to sorting). Note that all these join results are generated without any comparisons. Afterwards, we advance to the next interval $s \in S_i^A$ and repeat the test $r.end + \epsilon \geq s.start$. If the test is negative, we advance to the next interval $r \in R_i^{A1}$ and repeat the test, etc. As soon as either r or s is out of bounds, i.e., the input mini-partitions are fully scanned, the join algorithm terminates.

For example, in Figure 5, we first consider intervals r_1 and s_1. As $r_1.end + \epsilon \geq s_1.start$ holds, all intervals in R_i^{A1} form band join pairs with s_1. Next, we examine s_2 and repeat the test to find again that $r_1.end + \epsilon \geq s_2.start$. Hence, we also report all intervals in R_i^{A1} paired with s_2 as band join results. However, when we advance to s_3, we observe that $r_1.end + \epsilon < s_3.start$, so (r_1, s_3) is not a join result. At this point, we consider the next intervals r from R_i^{A1} while $r.end + \epsilon < s.start$ holds and stop at r_3, where we have $r_3.end + \epsilon \geq s_3.start$. Again, we report join pairs (r_3, s_3) and (r_4, s_3) and advance to s_4. Since $r_3.end + \epsilon < s_4.start$, we finally advance to r_4; since, $r_4.end + \epsilon \geq s_4.start$ holds, we report join pair (r_4, s_4). At this point, both mini-partitions are completely scanned and the algorithm terminates.

The number of comparisons conducted by the above algorithm (applied for all mini-joins which are not cross products) equals

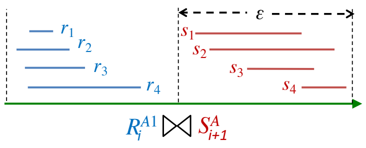


Figure 5: Mini-join between neighboring partitions

Table 1: Statistics of datasets

	INFECTIOUS	BOOKS	TAXIS	WEBKIT
Cardinality	415,912	2,312,602	14,212,261	2,347,346
Domain duration (secs)	6,946,360	31,507,200	2,592,000	461,829,284
Distinct domain points	81,514	5,330	2,229,932	174,471
Shortest interval (secs)	20	1	1	1
Avg. interval dur. (secs)	20	2,201,320	685	33,206,300
Longest interval (secs)	20	31,406,400	1,816,164	461,815,512

the total number of intervals in its two inputs (e.g., $|R_i^{A1}| + |S_{i+1}^A|$), which means that mini-joins are evaluated very efficiently.

Parallel evaluation. As shown in [3], the best approach to parallelize interval joins based on domain-based partitioning is to treat every mini-join as an independent task. Each task is scheduled to one of the available CPU threads. To maximize load balancing, the tasks are greedily assigned to threads in decreasing order of their expected costs (based on the size of the involved partitions).

4 EXPERIMENTAL EVALUATION

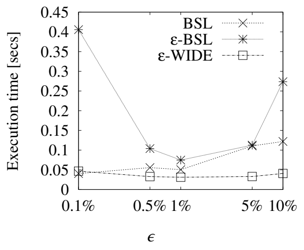
Our evaluation was conducted on a machine with 384 GBs of RAM and a dual Intel(R) Xeon(R) CPU E5-2630 v4 clocked at 2.20GHz. All methods were implemented in C++, compiled using `gcc` (v4.8.5) with flags `-O3`, `-mavx` and `-march=native`. We activated hyper-threading, allowing us to run up to 40 threads and used OpenMP for multi-threaded processing. Every interval contains two 64-bit domain point attributes (i.e., `start` and end) while the workload accumulates the number of result pairs. All data reside in main memory.

Methods. We compare our ϵ-wide partitioning join (denoted by ϵ-WIDE) to the baseline (denoted by BSL). In addition, we include a version of BSL (denoted by ϵ-BSL), which uses ϵ-wide domain partitions, but it conducts an overlap join using the extended intervals, instead of the method described in Section 3.2. Both variants of the baseline use our optimized forward scan based plane sweep method from [3] and all our optimizations to improve load balancing in domain-based partitioning.

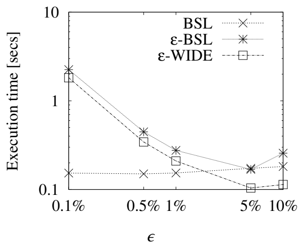
Datasets. Table 1 details our 4 real-world experimental datasets. INFECTIOUS [8] stores contact intervals between visitors at an exhibition at the Science Gallery in Dublin from 2009/05 to 2009/07. BOOKS [3] includes periods of book lent outs at Aarhus public libraries in 2013 (https://www.odaa.dk). TAXIS (https://www1.nyc.gov/site/tlc/index.page) stores durations of taxi trips in NYC, in Jan 2013. WEBKIT [3] records durations of file versions in the git repository of the Webkit project from 2001 to 2016 (https://webkit.org).

Tests. To assess the performance of the methods, we measure their response time while varying (i) threshold ϵ as a fraction of the domain duration inside $\{0.001, 0.005, 0.01, 0.05, 0.1\}$ and (ii) the number of available CPU threads inside $\{5, 10, 15, 20, 25, 30, 35, 40\}$. We also experimented with uniformly sampled subsets of the dataset as R and set the entire dataset as S; for this purpose, we varied the $|R|/|S|$ ratio inside $\{0.25, 0.5, 0.75, 1\}$.

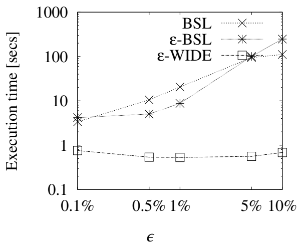
Results. Figures 6-8 summarize our experimental results. When varying ϵ, we observe the following. First, ϵ-WIDE is consistently faster than ϵ-BSL. This shows that applying the mini-joins



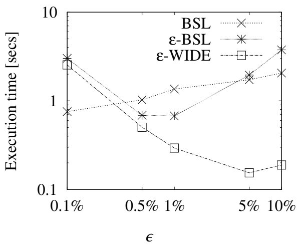
(a) INFECTIOUS



(b) BOOKS

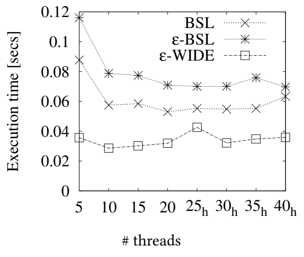


(c) TAXIS

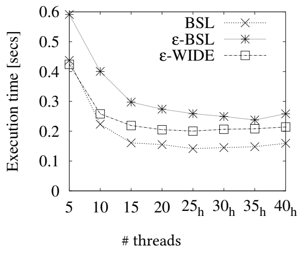


(d) WEBKIT

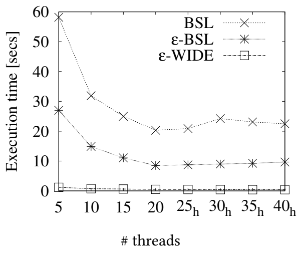
Figure 6: Total execution time while varying threshold ϵ as fraction of the domain duration; 20 threads, $|R| = |S|$.



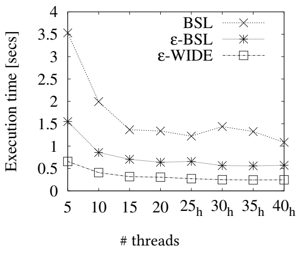
(a) INFECTIOUS



(b) BOOKS

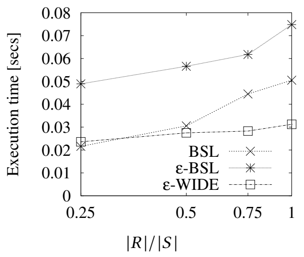


(c) TAXIS

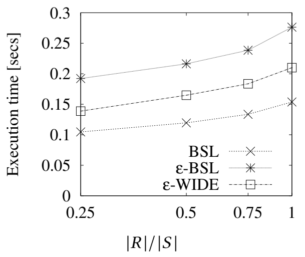


(d) WEBKIT

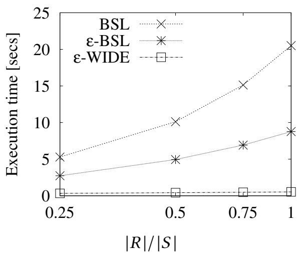
Figure 7: Total execution time while varying the number of CPU threads; $\epsilon = 0.01$ of the domain size, $|R| = |S|$.



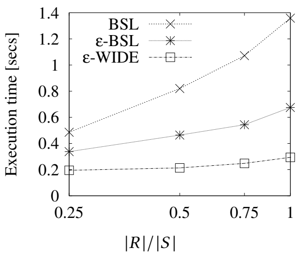
(a) INFECTIOUS



(b) BOOKS



(c) TAXIS



(d) WEBKIT

Figure 8: Total execution time while varying the $|R|/|S|$ ratio; $\epsilon = 0.01$ of the domain size, 20 threads.

described in Section 3.2 on the original intervals is faster than applying the mini-joins of [3] on the ϵ-extended intervals. However, the number of ϵ-wide partitions can become extremely large for small values of ϵ, which renders these approaches slower compared to BSL. Note that BSL chooses the number of partitions, according to the number of threads that can run in parallel as suggested in [3]. On WEBKIT, TAXIS and INFECTIOUS, ϵ-WIDE is up to a few time faster compared to BSL and ϵ-BSL for a wide range of ϵ/domain ratios. On the other hand, on dataset BOOKS, ϵ-WIDE is faster than the competition for ϵ/domain ratios larger than 2%. When varying the number of threads, we observe that all methods scale well until when the number of threads becomes 20, after which hyper threading comes into effect. The join on INFECTIOUS is already very cheap and does not benefit from increasing parallelism. Last, as expected all methods are affected by increasing $|R|/|S|$; their execution time rises.

5 CONCLUSION

In this short paper, we studied the evaluation of band joins between two collections of intervals. We extended our framework for interval overlap joins [3] in two directions; a baseline approach that expands all intervals by ϵ and then evaluates an overlap join and a novel approach that uses ϵ to define the partitions and then conducts cheaper joins between partitions. Our experimental findings show that the second approach is more efficient, unless ϵ is very small compared to the domain size.

ACKNOWLEDGEMENTS

Partially funded by the European Union's Horizon 2020 research - innovation programme under grant agreement No 657347 and the European Regional Development Fund - GSRT's project "Moving from Big Data Management to Data Science" (MIS 5002437/3) and Research–Create–Innovate project "Proximiot" (T1EDK-04810).

REFERENCES

[1] Pankaj K. Agarwal, Junyi Xie, Jun Yang, and Hai Yu. 2005. Monitoring Continuous Band-Join Queries over Dynamic Data. In *ISAAC*. 349–359.
[2] Michael H. Böhlen, Richard T. Snodgrass, and Michael D. Soo. 1996. Coalescing in Temporal Databases. In *VLDB*. 180–191.
[3] Panagiotis Bouros and Nikos Mamoulis. 2017. A Forward Scan based Plane Sweep Algorithm for Parallel Interval Joins. *PVLDB* 10, 11 (2017), 1346–1357.
[4] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. 2002. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*. 310–321.
[5] Reynold Cheng, Ben Kao, Sunil Prabhakar, Alan Kwan, and Yi-Cheng Tu. 2005. Adaptive Stream Filters for Entity-based Queries with Non-Value Tolerance. In *VLDB*. 37–48.
[6] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. 1991. An Evaluation of Non-Equijoin Algorithms. In *VLDB*. 443–452.
[7] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and Adaptive Online Joins. *PVLDB* 7, 6 (2014), 441–452.
[8] Lorenzo Isella, Juliette Stehlé, Alain Barrat, Ciro Cattuto, Jean-François Pinton, and Wouter Van den Broeck. 2011. What's in a crowd? Analysis of face-to-face behavioral networks. *Journal of Theoretical Biology* 271, 1 (2011), 166–180.
[9] Christian S. Jensen and Richard T. Snodgrass. 2018. Temporal Data Models. In *Encyclopedia of Database Systems, Second Edition*.
[10] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. 2015. Lightning Fast and Space Efficient Inequality Joins. *PVLDB* 8, 13 (2015), 2074–2085.
[11] Weining Zhang and Ke Wang. 2000. An Efficient Evaluation of a Fuzzy Equi-Join Using Fuzzy Equality Indicators. *TKDE* 12, 2 (2000), 225–237.

Personalized Page Rank on Knowledge Graphs: Particle Filtering is all you need!

Denis Gallo
University of Trento
denis.gallo@alumni.unitn.it

Matteo Lissandrini
Aalborg University
matteo@cs.aau.dk

Yannis Velegrakis
Utrecht University
i.velegrakis@uu.nl

ABSTRACT

Graphs are everywhere. Personalized Page Rank (PPR) is a particularly important task to support search and exploration within such datasets. PPR computes the proximity between query nodes and other nodes in the graph. This is used, among others, for entity exploration, query expansion, and product recommendation. Graph databases are used for storing knowledge graphs. Unfortunately, the exact computation of PPR is computationally expensive. While different solutions have been proposed to compute PPR values with high precision, these are extremely complex to implement, and in some cases require heavy preprocessing. In this work, we sustain that a better approach exists: *particle filtering*. Particle filtering methods produce ranks with sufficient precision while exploiting what graph databases architectures are already optimized for: navigating local connections. We present the implementation of such an approach in a popular commercial database and show how this outperforms the already implemented functionality. With this, we aim to motivate future research to optimize and improve upon this research direction.

1 INTRODUCTION

Graphs are everywhere [20], in particular, Knowledge Graphs (KG) [17] gained increasing attention thanks to their ability to represent entities and their relationships in many domains. Knowledge graphs model entities as nodes and the relationships among them as labelled edges. They are used to store the relationships about products, customers, events, locations, and more.

Personalized Page Rank [4, 5, 9, 12] (PPR) is a particularly important task to support search and exploration within graphs. At a high level, PPR extends the well known Page Rank [18] by computing a *local* popularity (or *proximity*) instead of a global importance score for nodes. In practice, given a small set of query entities, PPR returns a ranked list of other relevant entities based on a computed *random-walk proximity* to the query nodes. Famous examples of the application of PPR are the Twitter *Who To Follow* [8] that suggest to users other users to follow and the Pinterest related pins suggestions [14]. In other contexts, it can suggest related scientific articles, related entities, or suggest products to buy.

When it comes to PPR applications to KGs, a number of aspects become important, namely: the possibility to use a small set of nodes as query (e.g., for product recommendation or query expansion), the ability to include edge weights in the computation (given that in a KG different edges have different semantics), and fast response times for top-k queries (since in practical cases only a small set of high ranked nodes are required in contrast to computing such values for all nodes in the graph).

Currently, graph data management systems (GDBMS) [13] are the de-facto solution for managing knowledge graphs in transactional settings, thus, they need to support PPR queries. Unfortunately, the exact calculation of PPR is computationally expensive. In the literature many solutions have been proposed to provide fast computation of PPR values [2, 6, 7, 10, 11, 15, 19, 22–25]. Yet, they have been designed with the needs of social networks in mind, i.e., they focus on single source queries, unlabeled edges, and high-precision PPR value computations (required for community detection [26]). In many cases, they require pre-computation of indexes and other data-structures. In short, most existing methods are not designed to focus on the real needs for KG search, neither they take into account the requirements for being implemented in real-world Graph DBMS.

In this work, we study PPR computation solutions designed around the needs of Knowledge Graph search and the strengths of graph databases. We proposed a much simpler approach for computing Personalized Page Rank queries with multiple sources and heterogeneous edge weights. To achieve a significant performance in computing the top-k PPR, we extend the *damping function* template [4] with the *particle filtering procedure* [12], extended to correctly take into account the *teleportation probability* and account for the non-uniform edge importance typical of KGs. Currently, the only implementation of a similar approach has been proposed for an in-memory research prototype [16]. In our work, instead, we show how real word commercial graph databases can support this functionality.

Our experiments, on real large graphs, demonstrate the superiority of this approach (which we have made available as open-source[1]) against the currently implemented version in a major commercial GDBMS (Neo4j[2]). Furthermore, this direction is open to interesting challenges and can foster the development of new techniques to improve real-world graph databases.

2 RELATED WORKS

Personalized Page Rank [9] has been initially proposed as an alternative to global Page Rank since it computes local proximity to query nodes based on random-walks. Since that seminal work, many different approaches and implementations have followed. In general, they follow three alternative strategies, namely (1) matrix computation, (2) Monte-Carlo simulations, or (3) local search. For matrix computation, the graph is represented as its adjacency matrix, as in the original formulation, and different matrix multiplications are performed to compute the final values. These approaches [7, 10, 11, 22] are typically computationally expensive, hence they tend to be optimized through heavy pre-computations and large scale indexing. Matrix computation approaches are impractical for real-world graph databases, since GDBMS do not usually represent a graph as an adjacency matrix, and would require to maintain the pre-computed results.

[1]https://github.com/DenisGallo/Neo4j-ParticleFiltering
[2]https://neo4j.com/docs/graph-algorithms/

Monte-Carlo approaches, on the other hand, simulate a number of random traversals of the graph in order to compute the final PPR values. Employing this strategy offers few guarantees on the final output and on the actual response time. In general, a large number of random traversals are required to obtain reliable results. Hence, also in this case, precomputed values and indexes are employed [23, 24], suffering though from the same shortcomings discussed earlier.

Local search approaches, start, instead, from the query nodes and spread a rank value following the local neighborhoods until some stopping condition verifies. Usually, once this condition is encountered, they employ some additional Monte-Carlo traversals to refine the obtained approximate results and ensure higher precision [2, 5, 6, 15, 19, 24, 25]. Yet, most of these approaches have focused on either source-target queries, i.e., compute the PPR value of a target node given a source node, or single source queries with focus on computing high precision PPR values for very large portions of nodes in the graph. Both these directions, while useful in cases like social networks, impose an unnecessary burden on the system (since they focus on the actual PPR value instead of just computing a ranking of nodes) and neither address other important required features, like the necessity to differentiate edge types, the possibility to have multiple source nodes in the query, or the requirement of returning a ranked list of nodes of usually small size (i.e., top-k queries).

Contrary to most recent literature, following promising preliminary results from knowledge graph exploration [16], we study a method to enable multi-source, edge-weighted, top-k Personalized Page Rank queries within a Graph DBMS. We explicitly focus on the now prevalent need of Knowledge Graph search and on fully exploiting the ability of graph databases to efficiently query the local neighborhood of nodes [13]. Our method extends the *damping function template* [4], yet it implements an approach similar to *particle filtering procedure* [12] with two main differences: it has been extended to correctly take into account the *teleportation probability* that was not accounted for there, and it provides the ability to include edge relevance in the calculation, a feature that has been largely neglected in the literature. This solution has the advantage that it does not require any internal ad-hoc data-structure (our solution is a stored procedure of about two hundreds lines of code) and obtains both fast response time and high-quality results in practice.

3 APPROXIMATE PERSONALIZED PAGE RANK COMPUTATION

Given a graph $G : \langle V, E\rangle$ with V nodes and E edges, the result of a Personalized Page Rank (PPR) [9] given a set of query nodes $Q \subset V$ computes a proximity value of every node in V to the nodes in Q. Formally, the result of the computation is represented as a vector $\mathbf{v}$, with size $|V|$ representing the stationary distribution of the Markov chain [9] with state transition given by the equation

$$(1-c)A\mathbf{v} + c\mathbf{p} \tag{1}$$

Given the column normalized transition probability matrix A, the teleportation probability c, and the preference vector $\mathbf{p}$. The matrix A (of size $|V| \times |V|$) contains values between 0 and 1 according to the probability that an edge is traversed (hence the column normalization), where a value of 0 corresponds to non-existing edges. In general, a KG is a graph with edges of different types (an edge-labelled graph) and in many cases, different edge types are assigned different relevance scores (i.e., a weight in [0,1]). Furthermore, $\mathbf{p}$ is an $|V| \times 1$ column vector, which serves

Algorithm 1 PPR by Particle Filtering

```
Require: Graph $G$; Query nodes $\mathbf{Q}$
Require: Restart probability $c \in [0,1]$; Threshold $\tau \in [0,1]$
Require: Query value $k$
Ensure: Ranked Top-K nodes
 1: $\mathbf{p} \leftarrow \{\}$
 2: for each $q_i \in \mathbf{Q}$ do
 3:     $\mathbf{p}[q_i] \leftarrow 1/\tau$                          ▹ Initialize Particles
 4: while $\exists\ n_i \in \mathbf{p} \mid \mathbf{p}[n_i] \neq 0$ do
 5:     temp $\leftarrow \{\}$
 6:     for each $n_i \in \mathbf{p} \mid \mathbf{p}[n_i] \neq 0$ do
 7:         $particles \leftarrow \mathbf{p}[n_i] \times (1-c)$
 8:         for each $e : (n_i \rightarrow n_j) \in G$ do    ▹ Sorted by Weight
 9:             if $particles \leq \tau$ then
10:                 break
11:             $passing \leftarrow$MAX($particles \times e$.weight(),$\tau$)
12:             $\mathbf{temp}[n_j] \leftarrow \mathbf{temp}[n_j] + passing$
13:             $particles \leftarrow particles - passing$
14:     $\mathbf{p} \leftarrow \mathbf{temp}$
15:     for each $n_i \in \mathbf{p}$ do
16:         $\mathbf{v}[n_i] \leftarrow \mathbf{v}[n_i] + \mathbf{p}[n_i] \times c$         ▹ Update score
17: return top-k($\mathbf{v}$)
```

as the normalized preference vector, for which $\mathbf{p}[n]{\neq}0$ and in particular $0{<}\mathbf{p}[n]{\leq}1$ *iff* $n{\in}Q$. Finally, the *teleportation* probability $c \in (0, 1)$ is typically ≈ 0.15 in the literature [18].

In practice, the goal of the PPR value is to rank nodes, hence *the exact value of the PPR is not necessary as far as the ranking is preserved.* We propose an approximation of this process [4] and apply an approach similar to the *weighted particle filtering procedure* [12] to consider the non-uniform edge weights.

The approach simulates a set of $1/\tau$ floating particles (lines 2-3, Algorithm 1) starting from each node in the query set Q. At each iteration (lines 5-16), the particles distribute among the neighbors of the current node (minus the number of particles that *restart*, line 7). An important optimization is to prevent particles to split to arbitrarily small sizes, limiting them to a minimum of τ (lines 9-11). When distributing the particles among the neighbors, the algorithm gives preference to the edges with higher weights (line 8). Since the weight is normalized on the edges of each node, this operation matches the damping function framework [4]. The restart probability c will dissipate part of the particles at every iteration (line 8), and the algorithm will stop when no more particles can be distributed. During the process, a vector $\mathbf{v}$ accumulates the total amount of particles visiting each node (lines 15-16). Hence, the final list of top-k nodes is based on $\mathbf{v}$.

Here, we argue that for the case of Knowledge Graphs, the Particle Filtering approach for PPR computation is the best-suited approach to extend GDBMS functionalities for retrieving Top-K nodes.

The benefits of this approach are that (1) it does not require any complex preprocessing nor any additional persistent data structure, (2) it is directly implementable within any graph databases by direct use of core operations that are already optimized (namely local node neighbor traversal [13]), (3) it returns a ranking that strictly correlates with the actual ranking, (4) it can account for heterogeneous edge types and importance, (5) it does not require to traverse the entire graph and its exploration rate (and hence running time) can be fine-tuned through the τ and c parameters.

4 EXPERIMENTS

We investigated the performance and the quality of the ranking of Particle Filtering (PF) compared to the current exact implementation of Personalized Page Rank (PPR) in a commercial database. In particular, we implemented particle filtering (**PF** – Algorithm 1) as a Java stored procedure in Neo4j[3] v3.5.5, and compared it against the current implemented exact solution (**PPR** - Graph Algorithms v3.5.2)[4]. Experiments have been executed on a server with 12 cores and 128GB RAM.

Datasets: We compared the two alternatives on 4 different KGs from different domains and different sizes, widely used in the literature (Table 1). These KGs have been obtained from a dataset of movies, including also information about actors, directors, and genres[5] (*Movies*); from an established benchmark for triplestores [1] representing an heterogenous product catalog (*WatDiv*); an open domain knowledge graph (*DBpedia* [3]); and finally a knowledge base about drugs, their composition, and their interactions (*DrugBank* [21]). Of these, Movies, DBpedia, and DrugBank are real-world datasets, while WatDiv is synthetic.

Queries, Parameters, and Evaluation Metrics: For each dataset, we extracted 5 sets of 20 queries. Each set contains 20 queries of the same size (i.e., number of source nodes). We generated through sampling queries of size 1, 5, 10, 20, and 100 nodes, for a total of 100 queries. For each query, we recorded the execution time (average of 3 runs), and for the queries executed with PF, we recorded NDCG score at top-5, 50, 100, and 500. We executed queries both with the weighted (i.e., assigning weights based on label informativeness [16]) and unweighted (i.e., uniform weights) version of the algorithm. We tested the PF with three values of τ: 0.1, 0.05, and 0.01.

Results: Due to space constraints, we report here only results for the two largest graphs, namely WatDiv and DBpedia, for the labelled case. The full set of experimental results can be found in the extended version of this document[6]. Nonetheless, we also comment on some of the findings of the excluded experiments.

Quality of Ranking: Over all the datasets, the NDCG score for PF with $\tau = 0.01$ has the best quality, and often close to the perfect ranking (i.e., between 0.8 and 1.0) with queries containing up to 10 nodes. In most cases, also PF with $\tau = 0.05$ obtains a good quality ranking (NDCG>0.65). With more than 10 query nodes, the quality of ranking is subject to high variability, especially depending on the dataset. On DBpedia and WatDiv, still, we obtain NDCG scores above 0.65 for both τ=0.05 and 0.01 with 20 nodes in input at top-500, while for 100 nodes, we need τ=0.01 on DBpedia. This confirms the suitability for KG exploration cases.

Running Time: Compared to the running time of exact PPR, the PF algorithm provides a speedup between 1 and 4 orders of magnitudes, i.e., returning on average in 1-30 seconds while the exact solution requires 3-6 minutes (on DBpedia). In general, the speedup is proportional to the value of τ, i.e., τ=0.01 is between 10 and 100 times slower than τ=0.1. Yet, for 100 query nodes on DBpedia, we report that τ=0.01 rarely achieves sensible improvements due to the high number of particles generated.

Effect of Weights: when considering weights particle transitions are skewed towards more relevant nodes (through more informative edges). This has a noticeable effect on the running time because for high degree edges only the most informative edges are traversed (given the fact that they are prioritized), and many less relevant edges are skipped. For a similar reason, we notice that the NDCG score when weighted edges are considered is higher because for nodes with very high degrees, there are not enough particles to visit all the neighbors, hence in the weighted case the PF prioritization is consistent with the importance of the node, while in the unweighted case all of them should be visited.

	Movies	WatDiv	DBpedia	DrugBank
#Nodes	63K	5.2M	11.6M	391K
#Edges	106K	95.8M	216.7M	1M
#Edge types	4	31	13k	68
Density	3.93E-05	3.48E-06	1.61E-06	6.82E-06
#Con. Components	433	1	2	1
Min size CC	3	5.2M	59.3k	391K
Max size CC	58.2k	5.2M	11.5M	391K
Avg size CC	142	5.2M	579K	391K
Median size CC	6	5.2M	579K	391K
Max Out-degree	71	345	7.2k	423
Avg Out-degree	2.13	18.4	22.6	3.3
Median Out-degree	1	1	9	2
Max In-degree	92	585K	3.3M	316K
Avg In-degree	9.18	20.3	20.1	2.8
Median In-degree	8	1	2	1

Table 1: Size and characteristics of the datasets

Open Challenges: In our experiments, we noticed that when starting nodes are hubs (i.e., nodes with high degree) or are near hubs, the weighted traversal is the bottleneck. The reason is that current GDBMS can very quickly retrieve all the neighbors of a node, but then we require to sort them by weight. Hence, we identify the opportunity in GDBMS for implementing *sorted edge iterators for weighted edges* to speed up this step and other similar cases. Moreover, automatic tuning of the τ threshold depending on the query is an open research challenge.

5 CONCLUSIONS

In this paper, we argue that, when computing the Personalized Page Rank value in a knowledge graph, the approximate computation framework offered by the Particle Filtering approach, provides substantial advantages in terms of running time and ease of implementation, while ensuring good ranking quality. Our implementation can provide an efficient solution for extending existing graph databases since it exploits the strengths of these systems. While this approach is simple and effective for queries with few input nodes and limited to the first few hundreds nodes (typical of on-line exploration settings), we believe that the algorithm can be further expanded to assure high-quality ranking also for nodes in the long tail and larger queries.

REFERENCES

[1] Güneş Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified Stress Testing of RDF Data Management Systems. In *ISWC'14*. 197–212.

[2] Reid Andersen, Fan Chung, and Kevin Lang. 2006. Local Graph Partitioning Using PageRank Vectors. In *FOCS '06*. 475–486.

[3] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. DBpedia: A Nucleus for a Web of Open Data. In *ISWC'07*. 722–735.

[4] Ricardo Baeza-Yates, Paolo Boldi, and Carlos Castillo. 2006. Generalizing PageRank: Damping Functions for Link-based Ranking Algorithms. In *SIGIR '06*. ACM, 308–315.

[5] Soumen Chakrabarti. 2007. Dynamic Personalized Pagerank in Entity-relation Graphs. In *WWW '07*. ACM, 571–580.

[6] Dániel Fogaras and Balázs Rácz. 2004. Towards Scaling Fully Personalized PageRank. In *Algorithms and Models for the Web-Graph*. Springer, 105–117.

[7] Yasuhiro Fujiwara, Makoto Nakatsuji, Takeshi Yamamuro, Hiroaki Shiokawa, and Makoto Onizuka. 2012. Efficient Personalized Pagerank with Accuracy Assurance. In *KDD '12*. ACM, 15–23.

[8] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. 2013. WTF: The Who to Follow Service at Twitter. In *WWW '13*.

[3]https://neo4j.com/download-center/#community

[4]https://github.com/neo4j-contrib/neo4j-graph-algorithms

[5]https://neo4j.com/developer/example-data/

[6]http://people.cs.aau.dk/~matteo/pdf/EDBT2020-pf-long.pdf

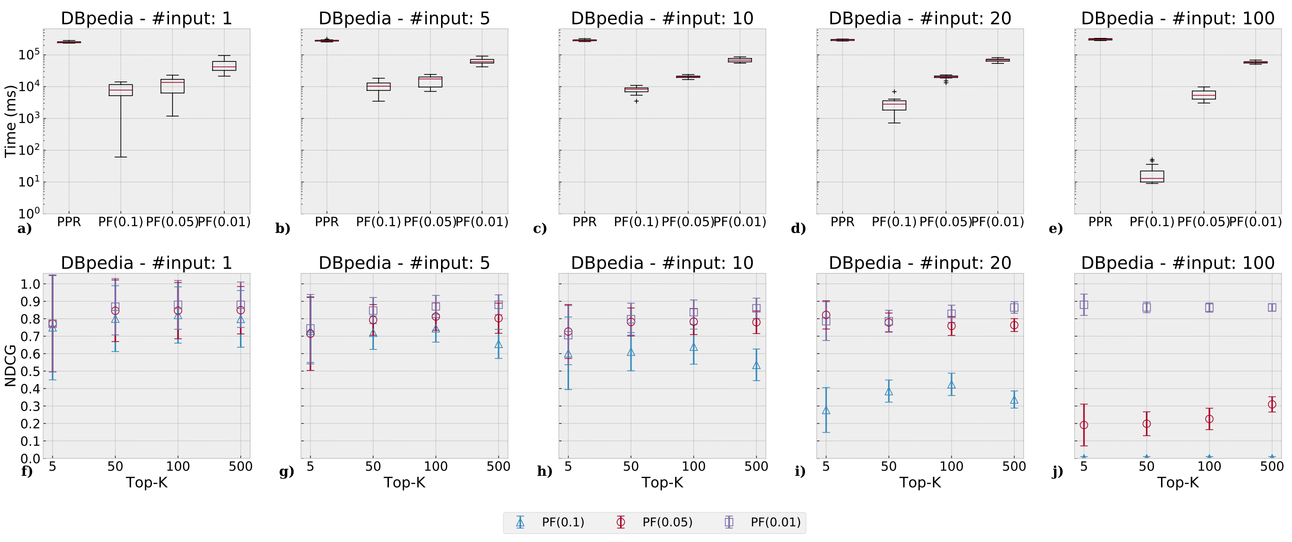


Figure 1: Running time (top) and NDCG (bottom) vs. weighted exact PPR on DBpedia, varying τ and # of input nodes.

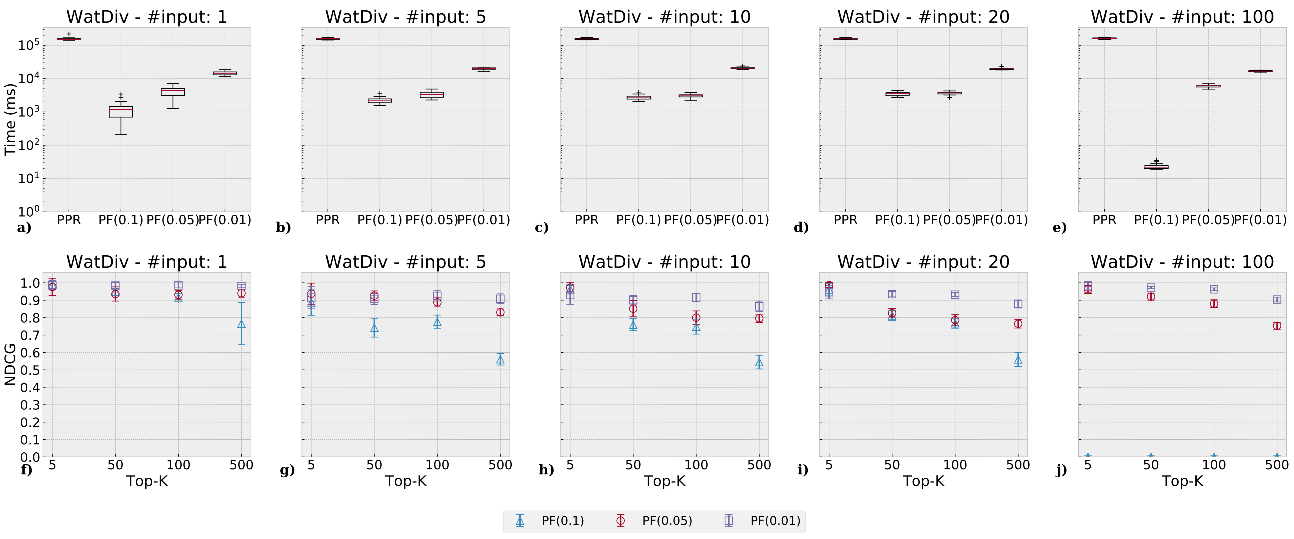


Figure 2: Running time (top) and NDCG (bottom) vs. weighted exact PPR on WatDiv, varying τ and # of input nodes.

[9] Glen Jeh and Jennifer Widom. 2003. Scaling Personalized Web Search. In *WWW '03*. ACM, 271–279.

[10] Jinhong Jung, Namyong Park, Sael Lee, and U Kang. 2017. BePI: Fast and Memory-Efficient Method for Billion-Scale Random Walk with Restart. In *SIGMOD '17*. ACM, 789–804.

[11] Jinhong Jung, Kijung Shin, Lee Sael, and U Kang. 2016. Random Walk with Restart on Large Graphs Using Block Elimination. *ACM Trans. Database Syst.* 41, 2 (2016), 12:1–12:43.

[12] Ni Lao and William W. Cohen. 2010. Fast Query Execution for Retrieval Models Based on Path-constrained Random Walks. In *KDD*. ACM, 881–888.

[13] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond Macrobenchmarks: Microbenchmark-based Graph Database Evaluation. *PVLDB* 12, 4 (2018), 390–403.

[14] David C. Liu, Stephanie Rogers, Raymond Shiau, Dmitry Kislyuk, Kevin C. Ma, Zhigang Zhong, Jenny Liu, and Yushi Jing. 2017. Related Pins at Pinterest: The Evolution of a Real-World Recommender System. In *WWW '17*. 583–592.

[15] Peter Lofgren, Siddhartha Banerjee, and Ashish Goel. 2016. Personalized PageRank Estimation and Search: A Bidirectional Approach. In *WSDM '16*. ACM, 163–172.

[16] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. 2016. Exemplar Queries: A New Way of Searching. *The VLDB Journal* 25, 6 (2016), 741–765.

[17] Natasha Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, and Jamie Taylor. 2019. Industry-scale Knowledge Graphs: Lessons and Challenges. *Queue* 17, 2, Article 20 (2019), 48–75 pages.

[18] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[19] Jia-Yu Pan, Hyung-Jeong Yang, Christos Faloutsos, and Pinar Duygulu. 2004. Automatic Multimedia Cross-modal Correlation Discovery. In *KDD '04*. ACM, 653–658.

[20] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *PVLDB* 11, 4 (2017), 420–431.

[21] Muhammad Saleem, Gábor Szárnyas, Felix Conrads, Syed Ahmad Chan Bukhari, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2019. How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks. In *WWW '19*. ACM, 1623–1633.

[22] Hanghang Tong, Christos Faloutsos, Christos Faloutsos, and Jia-Yu Pan. 2006. Fast Random Walk with Restart and Its Applications. In *ICDM '06*. 613–622.

[23] Sibo Wang, Youze Tang, Xiaokui Xiao, Yin Yang, and Zengxiang Li. 2016. HubPPR: Effective Indexing for Approximate Personalized Pagerank. *PVLDB* 10, 3 (2016), 205–216.

[24] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. 2017. FORA: Simple and Effective Approximate Single-Source Personalized PageRank. In *KDD '17*. ACM, 505–514.

[25] Zhewei Wei, Xiaodong He, Xiaokui Xiao, Sibo Wang, Shuo Shang, and Ji-Rong Wen. 2018. TopPPR: Top-k Personalized PageRank Queries with Precision Guarantees on Large Graphs. In *SIGMOD '18*. ACM, 441–456.

[26] J. J. Whang, D. F. Gleich, and I. S. Dhillon. 2016. Overlapping Community Detection Using Neighborhood-Inflated Seed Expansion. *TKDE* 28, 5 (2016).

Towards Planning of Regular Queries with Memory

Thomas Mulder, Nikolay Yakovets, George Fletcher
Technische Universiteit Eindhoven
The Netherlands
[t.mulder,hush,g.h.l.fletcher]@tue.nl

ABSTRACT

We investigate efficient evaluation of regular path queries with memory (RQM), which are an extension of regular path queries (RPQ) with additional constraints on data encountered along a path in a graph. We show how *Waveguide*, a state of the art system capable of planning RPQs, can be extended to facilitate RQM planning. Furthermore, we show that RQM planning is not as trivial as finding a conventional optimal join order and adding-on data constraints. Rather, we showcase that efficient evaluation of RQMs poses a number of non-trivial novel challenges.

1 INTRODUCTION

Regular path queries with memory (RQM) are subgraph-matching queries that allow the definition of constraints on both *topology* and *data* in graphs [3]. RQMs extend the expressive power of regular path queries (RPQ) [1] in many useful ways while maintaining an acceptable PSPACE-complete combined complexity.

An example of a query that is expressible as an RQM but not as an RPQ is to find all pairs of people (x, y) such that x directly or indirectly knows y and all people along the chain of acquaintance have the same age.

The initial study of RQMs [3], provides an extension of regular expressions called regular expressions with memory (REM) that are used to write RQMs, along with a procedure for constructing a *k-register data path automaton* that can be used for the evaluation of an RQM. While k-register data path automata are an excellent tool for the investigation of the expressive power and complexities of RQMs, they do not represent effective query plans, and do not provide opportunities for query optimisation.

We aim to develop the first practical evaluation engine for RQMs [4]. Here we take first steps towards this goal by (1) studying the shortcomings of the proposed automata from a query planning- and evaluation perspective, (2) addressing these shortcomings by proposing a more expressive type of automata that can be used to represent query plans and (3) showing that optimising such plans for *topological*- and *data* constraints are *orthogonal* problems. That is, a query plan that is optimal for evaluating only the topological constraints of a query on a particular graph, and another query plan that is optimal for evaluating the topological- *and* data constraints of the same query on the same graph, need not consider the topology of the query in the same order.

We extend Waveguide [5], a cost-based optimizer for property paths which builds query plans called *waveplans* that guide query evaluation. Waveplans are based on automata, which allows us to combine concepts from waveplans and k-register data path automata to obtain k-register waveplans.

2 REGULAR QUERIES WITH MEMORY

Data graphs are defined over a finite alphabet Σ and a countably infinite set of data values $\mathcal{D}$ as a triple $G = \langle V, E, \rho \rangle$, where:

- V is a finite set of nodes;
- $E \subseteq V \times \Sigma \times V$ is a set of labelled edges; and
- $\rho : V \to \mathcal{D}$ is a function that assigns a data value to each node in V.

To write regular expressions that can specify paths in data graphs, regular expressions with memory (REMs) over a finite alphabet Σ and set of variables $x_1, \ldots, x_k$ are introduced and defined by the grammar

$$e := \epsilon \mid a \mid e + e \mid e \cdot e \mid e^+ \mid e[c] \mid \downarrow \bar{x}.e \mid (e), \tag{1}$$

where $a \in \Sigma$, c is a condition and $\bar{x}$ a tuple of variables from $x_1, \ldots, x_n$. A condition, in turn, is defined by the grammar

$$c := x_i^= \mid x_i^{\neq} \mid z^= \mid z^{\neq} \mid c \wedge c \mid c \vee c \mid \neg c \mid (c),\ 1 \leq i \leq k, \tag{2}$$

where z is a data value in $\mathcal{D}$, also referred to as a constant.

As an example, consider the following REM:

$$(\mathtt{owns} \cdot \downarrow x_1.\mathtt{isLocatedIn})^+ \cdot \mathtt{hasCapital}[x_1^=] \tag{3}$$

This REM specifies paths where we encounter at least one sequence of two edges labelled `owns` and `isLocatedIn`, followed by an edge labelled `hasCapital`. Additionally, the data value associated with the source of an edge labelled `isLocatedIn` is stored in the first register, which is subsequently compared to the data value associated with the target of an edge labelled `hasCapital`.

A regular query with memory is defined as an expression of the form $Q := x \xrightarrow{e} y$ where e is an REM, and x and y are variables that are mapped to nodes in a data graph. Hence, the evaluation of Q amounts to finding pairs $(u, v) \in V \times V$ such that there exists a path from u to v that adheres to e. See Section 3.3 of [3] for the formal semantics of REMs and RQMs.

3 QUERY PLANNING

Query planning in the most general sense, is the process of finding an ordering of operations that, when executed, produce the solution to the given query. In the context of regular queries with memory, this means finding an ordering of *edge labels*, *assignments*, *conditions* and *projections*.

3.1 Automata as Query Plans

Query plans for relational database systems are often represented as trees. Due to the recursive nature of RQMs introduced by the Kleene plus ($^+$) operator, it is more convenient to represent query plans for RQMs as *automata* instead.

Regular Data Path Automata (RDPA) [3] have been proposed as a representation of RQMs. These RDPAs, however, capture the operations necessary for the evaluation of an RQM $Q = x \xrightarrow{e} y$ only in the order corresponding to the left-deep parsing of e. Hence, RDPAs are not a suitable formalism for representing query plans for RQMs.

Instead, we will consider *Waveguide* [5], a state of the art query optimiser for RPQs. It represents query plans as *waveplans*, a automaton-based formalism that allows for a rich variety of query plans, due to its *inverse transitions* and *transitions over views*. Conceptually, a waveplan consists of one or more *wavefronts*, which are automata that are used to compute (part of) the solution to a query. We extend wavefronts with assignments, conditions and projection over data values and registers.

Inverse- and view transitions. Let Σ be a finite alphabet, and L a finite set of state labels disjoint from Σ. We define $\Sigma^* = \bigcup_{a \in \Sigma \cup L} \{/a, a/\}$ as a set of labels. A transition with label $/a \in \Sigma^*$ is said to *append* edges labelled a to an intermediate result, whereas a transition labelled $a/ \in \Sigma^*$ is said to *prepend* edges labelled a to an intermediate result. The latter is referred to as an *inverse* transition. Inverse transitions allow our plans to represent many different orders of edge labels, such as a right-deep order.

A transition with label $/2 \in \Sigma^*$ appends paths computed by the state with label 2 to an intermediate result. Such a transition is called a *transition over a view*. Transitions over views further extend the orders of edge labels our plans can express by including *bushy* plans.

Projection. *Data paths* are defined as a sequence of interleaving nodes and edge labels that always start and end with a node [3]. Consider the REM e from (3) and a data path:

$$\pi = v_1 \, \texttt{owns} \, v_2 \, \texttt{isLocatedIn} \, v_3 \, \texttt{hasCapital} \, v_4$$

Checking whether or not π is accepted by e in a right-deep order means first finding edges labelled `hasCapital`, then amongst those finding edges that are preceded by edges labelled `isLocatedIn`, etc. This is a valid order of evaluating the edge labels and may be more efficient than a left-deep order, depending on the input graph. This order provides a problem with respect to the assignment- and condition in this REM. Namely, we would like to check that $\rho(v_4) = x_1$ where x_1 is the value in the first register. We can only do so once the first register has been assigned the value $\rho(v_2)$. Hence, to make this ordering of the edge labels work, $\rho(v_4)$ will have to be stored until the assignment has been made. To indicate which data- and register values must be stored at which point during query evaluation, we associate with each state in our automata finite sets $P_{\mathfrak{D}} \subset \mathbb{N}$ and $P_r \subset \mathbb{N}$ that contain the positions of nodes in a path and the indices of registers that must be kept, respectively. For instance, to indicate that $\rho(v_4)$ and x_1 (i.e. the value of the first register) must be kept, we would set $P_{\mathfrak{D}} = \{4\}$ and $P_r = \{1\}$.

3.2 k-Register Waveplans

We will refer to the extension of waveplans and wavefronts with assignments, conditions and projections as *k-register waveplans* and *k-register wavefronts*, respectively.

Let Σ^* be a finite labelling alphabet, k a natural number and C a finite set of conditions. Formally, a *k-register wavefront* is a tuple $w_l = \langle l, S, Q, q_0, \Pi_{\mathfrak{D}}, \Pi_r, \delta, F, \tau_0 \rangle$, where

- l is a wavefront label,
- S is a seed,
- Q is a set of states,
- q_0 is the starting state $q_0 \in Q$,
- $\Pi_{\mathfrak{D}} : Q \longrightarrow 2^{\mathbb{N}}$ is a function assigning a projection $P_{\mathfrak{D}}$ of path positions corresponding to data values to each state,
- $\Pi_r : Q \longrightarrow 2^{\mathbb{N}}$ is a function assigning a projection P_r of register indices to each state,

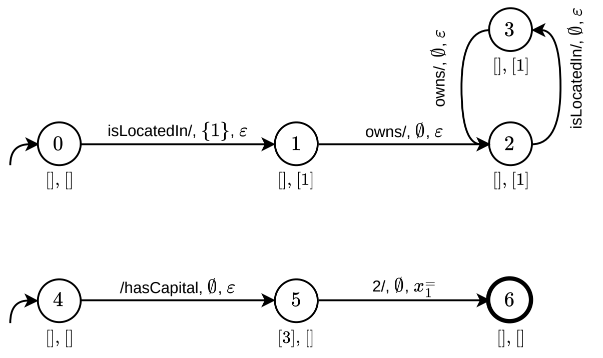

isLocatedIn/, $\{1\}$, ε
owns/, $\emptyset$, ε
owns/, $\emptyset$, ε
isLocatedIn/, $\emptyset$, ε
0
1
2
3
[], []
[], [1]
[], [1]
[], [1]
/hasCapital, $\emptyset$, ε
2/, $\emptyset$, $x_1^=$
4
5
6
[], []
[3], []
[], []

Figure 1: A k-register waveplan for (3)

- δ is a transition relation $\delta : Q \times \Sigma^* \times 2^{[k]} \times C \times Q$,
- $F \subseteq Q$ is a set of accepting states, and
- $\tau_0 \in \mathfrak{D}_{\perp}^{k}$ is the initial configuration of the registers.

A tuple $d = (q_1, a, K, c, q_2) \in \delta$ consists of source- and target states q_1 and q_2, label a, a set $K \subseteq \{1, ..., k\}$ indicating which registers are to be assigned a value during this transition and a condition c which is to be checked during this transition.

A *k-register waveplan* p is an ordered set of wavefronts. Consider any pair of wavefronts $w, w' \in p$ such that
$w = \langle l, S, Q, q_0, \Pi_{\mathfrak{D}}, \Pi_r, \delta, F, \tau_0 \rangle$ and
$w' = \langle l', S', Q', q_0', \Pi_{\mathfrak{D}}', \Pi_r', \delta', F', \tau_0 \rangle$. Then p defines an order $<_p$ on wavefronts as follows:

$$\forall w, w' \in p \mid w <_p w' \; : \; l' \notin S \wedge l' \notin L$$

Given this order, lower wavefronts cannot use labels of higher wavefronts in their seeds or transitions.

The role of *seeds* in Waveguide is quite complex. Here it suffices to say that seeds are necessary to ensure semantically correct plans when dealing with multiple wavefronts in a waveplan, or single wavefronts implementing a closure over an expression.

Figure 1 shows a k-register waveplan for the REM from (3). It consists of two wavefronts. The first wavefront computes the result to the expression $e' = (\texttt{owns/}{\downarrow}\; x_1.\texttt{isLocatedIn})^+$ in a right-deep order. It assigns a value to the first register (i.e. x_1 is set to the data value at the source of an edge labelled `isLocatedIn`). The second wavefront computes the result to (3) by first computing $e'' = \texttt{hasCapital}$, storing the data value at the target of edges labelled `hasCapital` in state 5. The result of state 2 is prepended to the result of state 5, and x_1 is checked for equality against the data value at the end of the resulting paths (i.e. at the target of an edge labelled `hasCapital`).

3.3 Topological Order

Let e be a regular expression with memory. We will refer to t_e as the topology of e which is a regular expression obtained by recursively replacing

- every sub-expression $e_1[c]$ of e by e_1, and
- every sub-expression $\downarrow \overline{x}.e_1$ of e by e_1

such that t_e no longer contains assignments and conditions.

We can now define the order in which a k-register waveplan p considers the topology of a query as follows. Consider a wavefront $w \in p$. We define the *topological order* λ_w of w as a sequence $\langle t_0, ..., t_n \rangle$ where t_i is the sub-expression of t_e for which the state with label i in p returns an intermediate result. We

ignore all states that are part of a cycle in w, except for accepting states, when constructing any sequence λ_w. We define the topological order λ_p of a k-register waveplan p as a sequence of sequences $\langle\langle t_0, ..., t_l\rangle, ..., \langle t_m, ..., t_n\rangle\rangle$, where each inner sequence corresponds to the topological order of the wavefronts that make up p, in the order defined by $<_p$. Consider the waveplan in Figure 1. Its topological order is
$\langle\langle \epsilon, \texttt{isLocatedIn}, (\texttt{owns/isLocatedIn})^+\rangle,$
$\langle \epsilon, \texttt{hasCapital}, (\texttt{owns/isLocatedIn})^+\texttt{/hasCapital}\rangle\rangle$.

3.4 Query Evaluation

As per [5], query evaluation is done using a procedure based on breadth-first search in which a waveplan guides the search (i.e. determines which edges are to be explored based on their label and possibly the satisfaction of conditions) until a fix-point is reached.

Edge walks. The metric by which we will judge the performance of a plan on the query evaluation task is that of the total number of *edge walks*. Every distinct tuple that is added to the queue during the search is considered one edge walk. Notice that a tuple that is added to the queue is an n-tuple with $n \geq 3$. A tuple contains at least a pair of nodes representing the end-points of a path and the state of the waveplan that produced this pair. Additionally, for a query $Q := x \xrightarrow{e} y$, a tuple can also contain up to k register values and up to l data values, where l is bounded by the number of sub-expressions of e of the shape $e_1[c]$. Hence $n \leq k + l + 3$.

Optimality of query plans. - Consider a pair of RQMs $Q := x \xrightarrow{e} y$ and $Q' := x \xrightarrow{t_e} y$. These queries are topologically equivalent (since t_e is the topology of e), but Q may contain assignments and conditions. Let P_Q and $P_{Q'}$ be sets of k-register waveplans for Q and Q', respectively. For every plan $p \in P_Q$ there exists a plan $q \in P_{Q'}$ such that $\lambda_p = \lambda_q$ because Q and Q' are topologically equivalent. Hence, we can construct a relation $R \subset P_Q \times \mathbb{N} \times P_{Q'} \times \mathbb{N}$ where $(p, n, q, m) \in R$ if and only if $\lambda_p = \lambda_q$ and be assured that there exists a tuple $r \in R$ for every $p \in P_Q$ such that p is part of r. The values n and m denote the total number of edge walks produced by p and q, respectively.

We can define sets $OPT_Q \subseteq P_Q$ and $OPT_{Q'} \subseteq P_{Q'}$ as

$$OPT_Q = \{\, p \mid \exists (p, n, q, m) \in R \wedge \forall (p', n', q', m') \in R : n \leq n' \,\} \tag{4}$$

$$OPT_{Q'} = \{\, q \mid \exists (p, n, q, m) \in R \wedge \forall (p', n', q', m') \in R : m \leq m' \,\} \tag{5}$$

That is, OPT_Q and $OPT_{Q'}$ are the sets of plans for Q and Q' that produce a minimal number of edge walks. We say that any $q \in OPT_{Q'}$ is *optimal with respect to topology*, and any $p \in OPT_Q$ is *optimal with respect to topology and data.*

We define a *simple* query Q as a query where

$$\forall (p, n, q, m) \in R \mid q \in OPT_{Q'} \Rightarrow p \in OPT_Q$$

which states that a simple query is one where optimality with respect to topology guarantees optimality with respect to topology and data.

We investigate the performance difference between a plan $p_2 \in OPT_Q$ and a plan $p_1 \notin OPT_Q$ for which there exists $(p_1, n_1, q_1, m_1) \in R$ such that $q_1 \in OPT_{Q'}$. Such an investigation will yield insights into the performance improvements that are neglected when query plan- and graph topology are assumed to be the determining factors in query performance.

To this end, we define a performance ratio between the edge walks produced by such plans p_1 and p_2. Formally, φ is defined over Q (from which R, OPT_Q and $OPT_{Q'}$ are derived) as

$$\varphi(Q) \begin{cases} \frac{n_1}{n_2}, & \text{if } \exists (p_1, n_1, q_1, m_1), (p_2, n_2, q_2, m_2) \in R \mid \\ & \quad q_1 \in OPT_{Q'} \wedge p_1 \notin OPT_Q \wedge p_2 \in OPT_Q \\ 1, & \text{otherwise} \end{cases} \tag{6}$$

Notice that $\varphi(Q) = 1$ if and only if Q is simple.

4 EXPERIMENTAL SETUP

In order to show that optimising k-register waveplans with respect to the topology of a query is orthogonal to optimising these plans with respect to the query's data constraints, we will construct a workload $\mathcal{W}$ consisting of pairs (Q, G) where Q is a regular path query with memory, and G is a data graph. We will count the number of pairs (Q, G) such that Q is *simple* on G, and investigate the average- and worst-case improvements in performance that are neglected when the topology of a query plan and graph are assumed to be the determining factors in query performance.

4.1 Query Pattern

The queries in $\mathcal{W}$ will be based on instances of the pattern:

$$((a\cdot \downarrow x_1.b)^+) \cdot c[x_1^=] \tag{7}$$

The motivation for the choice of this particular pattern is threefold:

(1) it contains interactions with data through a register (i.e. an assignment and condition) *inside* of a closure,
(2) it is simple in terms of the number of edge labels it contains and registers it uses, and
(3) its interactions with data apply to nodes that are neither:
  - part of the pairs in the result of the query evaluation problem, or
  - guaranteed to have the same data value associated with them

The first point is important because any pattern that does not interact with data at all, or does so only outside of closures can be modelled as a (C)RPQ [1], and is therefore not an example of the increased expressive power of RQMs. The second point is more practical in that it is possible to find many different instances of a pattern which contains few edge labels and uses few registers in real graph data. Thirdly, a pattern such as $(\downarrow x_1.a\cdot b\cdot c[x_1^=])^+$ would also satisfy the first two points. However, since all its interactions with data pertain to the data values in nodes that are either part of the pairs in the result of the query evaluation problem, or pertain to nodes that have the same data value associated with them (i.e. the nodes with an incoming edge labelled c and an outgoing edge labelled a in the closure), such queries are too selective in practice.

Seventeen concrete combinations of edge labels for a, b and c are obtained from the semantic knowledge graph Yago2s [2]. Example instances are:

- $(\texttt{isLocatedIn}\cdot \downarrow x_1.\texttt{dealsWith})^+ \cdot \texttt{hasCapital}[x_1^=]$
- $(\texttt{owns}\cdot \downarrow x_1.\texttt{isLocatedIn})^+ \cdot \texttt{hasCapital}[x_1^=]$
- $(\texttt{isLocatedIn}\cdot \downarrow x_1.\texttt{owns})^+ \cdot \texttt{isConnectedTo}[x_1^=]$

4.2 Data Graphs

A data graph $G = \langle V, E, \rho \rangle$ is extracted from the semantic knowledge base Yago2s. Yago2s consists of RDF-triples, derived from Wikipedia, WordNet and GeoNames [2].

Let U denote the set of all distinct subjects and objects in Yago2s' RDF-triples. Similarly, let Σ denote the set of all distinct predicates in Yago2s' RDF-triples. We identified $P \subset \Sigma$ as a set of 13 predicates such that for all triples (s, p, o) with $p \in P$ it holds that o is a numerical value. No other predicates in the Yago2s data set could be identified that co-occur with edge labels for (7) and have numerical values. The range of values over all objects o was categorized into three equally sized categories. Thus, the data domain for each $p \in P$ is set to $\{0, 1, 2\}$ where we interpret the data values 0, 1 and 2 as low, medium and high, respectively. Consider an RDF-triple (s, p, o) where $p \in P$. Let $\gamma(o) \in \{0, 1, 2\}$ denote the category that o was assigned to.

Since $|P| = 13$ but the data graph model only allows a single data value per node, we can construct multiple data graphs G_i from Yago2s. Note that the query pattern from (7) requires data values at the source of edges labelled b and at the target of edges labelled c. Hence, for all 13^2 pairs from $P \times P$ we construct a data graph G_i.

Let (q_1, q_2) be an arbitrary pair from $P \times P$. To construct G_i we:

- add a node v to V for every $u \in U$. Let $\eta(u) = v$ denote the node in G that corresponds to u;
- for every triple (s, p, o) with $p \in \Sigma - P$ we add $(\eta(s), p, \eta(o))$ to E;
- for every pair of triples $(s, p, o), (s, b, o')$ where $p \in P$ and b in some instance of (7) we set $\rho(\eta(s)) = \gamma(o)$;
- for every pair of triples $(s, p, o), (s', c, s)$ where $p \in P$ and c in some instance of (7) we set $\rho(\eta(s)) = \gamma(o)$;
- for every $v \in V$ where v is not yet in the domain of ρ we set $\rho(v) = 3$.

Thus the labelling alphabet of each G_i is $\Sigma - P$ and the set of data values $\mathcal{D}$ is $\{0, 1, 2, 3\}$. When checking equality for two data values $d_1, d_2 \in \mathcal{D}$, we will maintain that $d_1 = 3 \lor d_2 = 3 \Rightarrow d_1 \neq d_2$. That is, if neither of the data values were obtained from Yago2s we consider them unknown and therefore unequal.

4.3 Query Workload

The combination of 17 instances of query pattern (7) and the $13^2 = 169$ data graphs allows for a maximum workload size of $17 * 169 = 2873$. However, many of these combinations will yield empty result sets on the query evaluation problem because there existed no RDF-triples in Yago2s that produce data values from $\{0, 1, 2\}$ for the source- or target nodes of edges labelled b or c, respectively. Instead, the workload $\mathcal{W}$ consists of 579 distinct pairs of RQMs and data graphs where the combination of edge labels for a, b and c from $\Sigma - P$ and properties q_1 and q_2 from P resulted in at least one $v \in V$ where $\rho(v) \in \{0, 1, 2\}$.

5 RESULTS

Out of the 579 pairs of regular queries with memory and data graphs (Q, G) only 87 queries Q are simple on G (as shown in Table 1). Because a large majority of the queries is not simple we conclude that, for the given query pattern and data set, the topology of a data graph and RQM are not, by themselves, determining factors in the performance of query plans.

On average, k-register waveplans that are optimal with respect to topology (but not necessarily optimal with respect to data) produce close to 2.5 times the number of edge walks to evaluate a query as do those k-register waveplans that are optimal with respect to topology and data.

	simple	non-simple	total
# of queries	87	492	579
% of queries	15.03%	84.97%	100%

Table 1: The number- and percentage of simple and non-simple queries

	min	max	mean	std
$\varphi(Q)$	1.0	29.08	2.42	5.30

Table 2: A breakdown of the values for $\varphi(Q)$ over $\mathcal{W}$

Moreover, for the worst-case ratio $\varphi(Q)$ observed in $\mathcal{W}$, a plan that is optimal with respect to topology performed just over 29 times more edge walks than a plan for the same query, on the same graph, that is optimal with respect to topology and data. A breakdown of the minimum, maximum, mean and standard deviation of $\varphi(Q)$ over $\mathcal{W}$ is presented in Table 2.

A caveat to the results obtained in this way is the following; the evaluation procedure employs a time-out mechanism whereby a query plan that has produced more edge walks than the best performing (i.e. fewest edge walks producing) plan thus far for the same query is terminated, even if it has not yet completed its evaluation. Hence, the observed ratios are a lower-bound on the actual performance ratios.

We have presented evidence of the orthogonality that exists between optimising query plans for RQMs with respect to topology and data, showing that such optimization involves novel- and non-trivial challenges that go beyond finding an optimal join order for edge labels. Ignoring this orthogonality leads to significant decreases in performance, both on average and in worst-case scenarios.

6 CONCLUDING REMARKS

In our experimental study we found that (1) a large majority of queries is not simple, from which we can conclude that optimising for topology and data are orthogonal problems; (2) on average, plans that are optimal w.r.t. topology (but not necessarily w.r.t. data) perform 2.5 times worse than plans that are optimal overall; and, (3) plans that are optimal w.r.t.topology only can perform up to 29 times worse than plans that are optimal overall

Looking ahead, a main direction of future work is to continue our study of RQM query optimization in the context of a fully-fledged property graph query engine.

REFERENCES

[1] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets. 2018. *Querying Graphs.* Morgan & Claypool Publishers.

[2] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. 2013. YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. *Artif. Intell.* 194 (Jan. 2013), 28–61.

[3] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. 2016. Querying Graphs with Data. *J. ACM* 63, 2, Article 14 (March 2016), 53 pages.

[4] Thomas Mulder. 2019. *Regular Queries with Memory: From Theory to Practice.* MSc thesis. Technische Universiteit Eindhoven.

[5] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2016. Query planning for evaluating SPARQL property paths. In *Proceedings of the 2016 International Conference on Management of Data.* ACM, 1875–1889.

Accurate Demand Forecasting for Retails with Deep Neural Networks

Shanhe Liao
Tongji University, Shanghai, China
shhliao@tongji.edu.cn

Xianghong Zhou
Tongji University, Shanghai, China
xhz7@tongji.edu.cn

Weixiong Rao
Tongji University, Shanghai, China
wxrao@tongji.edu.cn

ABSTRACT

Demand forecasting, which aims to predict future product sales, is one of the most important tasks in retail markets. With the help of time series prediction models, the literature works either perform the prediction of each individual product item separately, or adopt a multivariate time series forecasting approach. However, none of them leveraged the structural information of product items, such as product brands and multi-level categories. Moreover, various product items have significantly different temporal characteristics, such as periodicity. In this short paper, we propose a deep learning-based prediction model to find inherent inter-dependencies and temporal characteristics among product items for more accurate prediction. Evaluation on two real-world datasets validates that our model can achieve much higher accuracy compared with state-of-the-art methods.

1 INTRODUCTION

Demand forecasting, which aims to predict future product sales, is one of the most important tasks in retail markets. The accurate prediction is important to avoid either insufficient or excess inventory in product warehouses for a typical retail market which typically sells at least thousands of product items.

Traditional works adopt either univariate time series models or multivariate time series model. The univariate time series models, such as the autoregressive integrated moving average (ARIMA) [1], autoregression (AR) [1], moving average (MA) [1] and autoregressive moving average (ARMA) [1], treat different product items separately. ARIMA is rather time consuming especially when there are thousands of products or more. In addition, ARIMA assumes that the current value of time series is a linear combination of historical observations of itself and a random noise. It is hard for ARIMA to capture non-linear relationships and inter-dependencies of different product items. Some machine learning models can also be applied to demand forecasting problems, such as linear regression [11] and linear support vector regression (SVR) [2]. Nonetheless, these machine learning models suffer from the similar weaknesses as ARIMA [15].

Multivariate time series models instead take into account the inter-dependencies among product items. For example, as an extension of ARIMA, vector autoregression (VAR) [9] can handle multivariate time series. However, the model capacity of VAR grows linearly over temporal window size and quadratically over the number of variables, making it hard to model thousands of product items with a long history. More recently, deep learning models have demonstrated outstanding performance in time series forecasting problems. There are basic recurrent neural network (RNN) [5] and its variants including long short-term memory (LSTM) network [10] and gated recurrent unit (GRU) [4]. On this basis, a recent work LSTNet [7] combines convolutional neural network (CNN) [8] and GRU to perform multivariate time series forecasting. The LSTNet model uses a special recurrent-skip component to capture very long-term periodic patterns. However, it assumes that all variables in the multivariate time series have same periodicity, which is invalid for most real datasets.

Other than the aforementioned weaknesses, the existing prediction methods ignore that product items have inherent structural information, e.g., the relations between product items and brands, and the relations among various product items (which may share the same multi-level categories). Our work is motivated by a clustering algorithm [3] to segment product items with help of a so-called *product tree*. This tree structure takes product categories as internal nodes and product items as leaf nodes. Beyond that, we extend the product tree by incorporating product brands and then construct a *product graph* structure. This structure explicitly represents the structural information of product items. Figure 1 illustrates an example of the graph structure of four product items. We can easily find that the brand *Master Kong* has three products, which belong to two different subcategories. Consider that a customer prefers the brand *Master Kong* and recently bought a product item *Master Kong Jasmine Tea*. It is reasonable to infer that he will try another product item *Master Kong Black Tea*, especially when *Master Kong Black Tea* involves a sale promotion campaign.

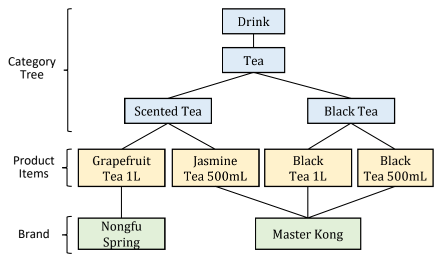


Figure 1: An example of Product Graph Structure involving Categories, Items and Brands.

Without the product graph structure as prior, previous methods either treat all product items equally or have to implicitly infer the inherent relationship but at the cost of accuracy loss. To overcome the issues, with help of the graph structure, we propose a new deep learning model to precisely predict product demands in a multivariate time series forecasting manner, called Structural Temporal Attention Network (STANet). This network incorporates both the product graph structure (see Figure 1) and temporal characteristics of product items. In particular, we note that the inter-dependencies of products and temporal dependencies (e.g., temporal periodicity) may change over time. Thus, we leverage attention mechanism [12] to deal with these variations. In this way, STANet assigns various weights with respect to different inputs involving the variations. Based on graph attention network (GAT) [13], GRU, and a special temporal attention, STANet performs better than existing methods. As a summary, we make the following contributions.

- We give adequate analysis on two real datasets to validate the motivation of our model, including the product structural inter-dependencies and temporal dependencies.
- The proposed STANet leverages GAT to capture the product structural inter-dependencies and GRU to capture temporal patterns. Moreover, a temporal attention mechanism is adopted on the hidden states of GRU to deal with diverse temporal characteristics of product items. Thus, the two attention mechanisms, i.e., GAT and temporal attention, work together to comfortably learn the product structural inter-dependencies and temporal dependencies.
- Evaluations on two real-world sales datasets show that STANet achieves the best results compared with several state-of-the-art methods.

The rest of this paper is organized as follows. Section 2 gives the problem formulation, and Section3 describes the proposed approach STANet. Then, Section 4 reports evaluation results on two real-world datasets. Finally, Section 5 concludes the paper.

2 PROBLEM FORMULATION

We consider a data set of transaction records in a market. Each transaction record contains 3 fields: the transaction timestamp, item ID, and amount of sold items. Moreover, via the item ID, we can find a list of product categories (in our dataset, each product item is with a list of 4-level categories) and an associated product brand. In this way, we augment each transaction record by totally 8 (=3+4+1) fields. Given a certain time horizon (e.g., one day or one week), we pre-process the transaction records into a multivariate time series of the volumes of sold product items. In addition, for a certain category (or brand), we sum the volumes of all product items belonging to the category (or brand). In this way, we have the multivariate time series of the volumes of product items, categories, and brands. Meanwhile, the product graph structure is stored in an adjacency matrix, where an element 1 indicates an edge between two nodes (such as a product item and its brand), otherwise 0.

Formally, we denote the number of product items by N_p and the total number of items, brands and categories as N. Given the augmented multivariate time series $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T\}$, $\boldsymbol{x}_t \in \mathbb{R}^{N\times 1}$, $t = 1, 2, \ldots, T$ and adjacency matrix $M \in \mathbb{R}^{N\times N}$, we aim to predict future product sale volume $\boldsymbol{x}_{T+h}$ where h is the desirable horizon ahead of the current time stamp. More specifically, to train a model using historical data, we use a time window of size τ to split existing data into fixed length inputs, where each input is expressed as $\{\boldsymbol{x}_t, \ldots, \boldsymbol{x}_{t+\tau-1}\}$ and $\boldsymbol{x}_{t+\tau-1+h}$ is the label. The adjacency matrix M is fixed. In this way, the demand forecasting problem is equivalent to learning a function $f_M : \mathbb{R}^{N\times\tau} \to \mathbb{R}^{N\times 1}$. On the testing stage, we only need to calculate evaluation metric for the N_p real product items.

3 FRAMEWORK

In this section, we present the detail of the proposed model STANet. Figure 2 gives the framework of STANet.

3.1 Graph Attention Component

For multivariate time series forecasting, one of the most crucial tasks is to capture the inter-dependencies between different variables. What's more, as shown in Section 4.1, the inter-dependencies may change over time. To explore inter-dependencies

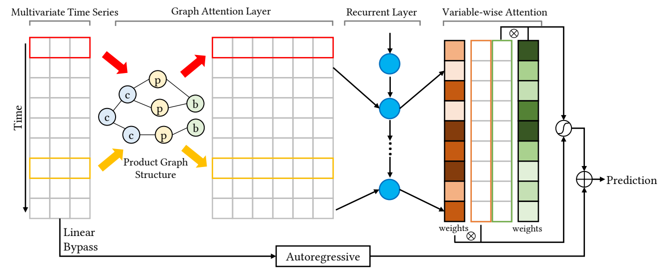


Figure 2: The framework of STANet.

from product graph structure, we need graph neural network. Furthermore, to handle dynamic inter-dependencies, we choose attention mechanism because it can assign various weights with respect to different inputs. As a result, the first layer of STANet is a graph attention layer [13]. Given the input time series $X \in \mathbb{R}^{N\times\tau}$ and adjacency matrix $M \in \mathbb{R}^{N\times N}$, we use a multi-head graph attention layer to process X time step by time step. Formally, this operation at time step t is given by

$$\boldsymbol{h}_t^i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i}\alpha_{ij}^k W^k x_t^j\right), \tag{1}$$

where x_t^j is the sale of variable (product, or brand, or category) j at time step t, W^k is a linear transformation to obtain sufficient expressive power, $\mathcal{N}_i$ refers to all the adjacent nodes of variable i given by M, K is the number of multi-head attention and σ is an activation function. α_{ij}^k is the coefficient of attention mechanism computed by

$$\alpha_{ij}^k = \frac{exp\left(LeakyReLU\left(f_a\left(W^k x_t^i, W^k x_t^j\right)\right)\right)}{\sum_{\ell\in\mathcal{N}_i} exp\left(LeakyReLU\left(f_a\left(W^k x_t^i, W^k x_t^\ell\right)\right)\right)}, \tag{2}$$

where f_a is a scoring function to evaluate relevance, and in our model it is a single-layer feedforward neural network.

Suppose $W^k \in \mathbb{R}^{F\times 1}$, with aforementioned X and M, the output of the graph attention component is $X_G \in \mathbb{R}^{FN\times\tau}$.

3.2 Recurrent Component

When the variable-to-variable relationships have been processed, X_G is fed into the recurrent component to capture temporal patterns. Here we use gated recurrent unit (GRU) [4] as the recurrent layer. Compared with vanilla recurrent neural networks (RNN) [5], GRU is more capable to capture long-term patterns. Suppose the hidden size of GRU is d_r, then the output of recurrent component is $X_R \in \mathbb{R}^{d_r\times\tau}$.

3.3 Variable-Wise Temporal Attention

After the graph attention component and recurrent component, the model has successfully captured inter-dependencies and basic temporal patterns. Nonetheless, temporal patterns could also be dynamic. Therefore, as a commonly used technique in RNN, temporal attention can be added to the model as

$$\boldsymbol{\alpha}_{t+\tau-1} = f_a\left(H_{t+\tau-1}, \boldsymbol{h}_{t+\tau-1}\right), \tag{3}$$

where $\boldsymbol{\alpha}_{t+\tau-1} \in \mathbb{R}^{\tau\times 1}$, f_a is a scoring function and $\boldsymbol{h}_{t+\tau-1}$ is the last hidden state of RNN, $H_{t+\tau-1} = [\boldsymbol{h}_t, \ldots, \boldsymbol{h}_{t+\tau-1}]$ is a matrix stacking the hidden states of RNN.

However, we will show in Section 4.1 that various products may have rather different temporal characteristics such as periodicity. Instead of using the same attention mechanism for all product items, we propose a variable-wise temporal attention

Table 1: Datasets statistics, where T is length of time step, P is the time interval, N_p, N_b, N_c are numbers of products, brands, categories respectively, $N = N_p + N_b + N_c$, and sparsity means proportion of zero value in the data.

Datasets	T	P	N_p	N_b	N_c	N	Sparsity
Dataset-1	572	1 day	1878	433	612	2923	53%
Dataset-2	833	1 day	1925	289	771	2985	39%

mechanism to compute the attention for each variable independently as

$$\alpha^i_{t+\tau-1} = f_a\left(H^i_{t+\tau-1}, h^i_{t+\tau-1}\right). \tag{4}$$

Equation (4) is similar to Equation (3), except a superscript $i = 1, 2, \ldots, d_r$, indicating that the attention mechanism is calculated for a particular GRU hidden variable. In this way, our model could deal with different temporal characteristics such as periodicity for different variables. With the coefficients $\boldsymbol{\alpha}^i_{t+\tau-1}$, the weighted context vector of i^{th} hidden variable is calculated as

$$c^i_{t+\tau-1} = H^i_{t+\tau-1}\alpha^i_{t+\tau-1}, \tag{5}$$

where $H^i_{t+\tau-1} \in \mathbb{R}^{1\times\tau}$ and $\boldsymbol{\alpha}^i_{t+\tau-1} \in \mathbb{R}^{\tau\times 1}$. Let $\boldsymbol{c}_{t+\tau-1}$ be context vector of all hidden variables, then we can calculate the final output for horizon h as

$$\boldsymbol{y}_{t+\tau-1+h} = W\left[\boldsymbol{c}_{t+\tau-1}; \boldsymbol{h}_{t+\tau-1}\right] + b, \tag{6}$$

here $W \in \mathbb{R}^{N\times 2d_r}$ and $b \in \mathbb{R}^{1\times 1}$ are parameters of a fully connected layer.

3.4 Autoregressive Component

Similar to LSTNet [7], we add an autoregressive component to capture the local trend of product demands. This component is a linear bypass that predicts future demands directly from the input data to address the scale problem. This linear bypass will fit all products' historical data with a single linear layer.

The final prediction of STANet is then obtained by integrating the outputs of the neural network part and the autoregressive component using an automatically learned weight.

4 EXPERIMENTS AND EVALUATIONS

We first analyze two used real-world datasets to motivate STANet and then compare STANet against 6 counterparts.

4.1 Datasets and Analysis

We use two real-world datasets collected from two medium size stores of a chain retail in Shandong Province, China. Table 1 summarizes the statistics. As shown in this table, the sparsity of dataset-1 is greater than 50%, which makes the forecasting task rather challenging. Both datasets are split into training set (70%), validation set (15%) and testing set (15%) in chronological order. To explore the inter-dependencies and temporal characteristics of datasets, we give following analysis.

We consider two variables have inter-dependencies if the history of one variable can help forecasting another. Assuming two univariate time series $\boldsymbol{x} = x_1, x_2, \ldots, x_T$, $\boldsymbol{y} = y_1, y_2, \ldots, y_T$ and a specific time lag m, we model $\boldsymbol{y}$ as a regression of itself and $\boldsymbol{x}$:

$$y_t = a_0 + a_1 y_{t-1} + \ldots + a_m y_{t-m} + b_1 x_{t-1} + \ldots + b_m x_{t-m}. \tag{7}$$

If $\boldsymbol{y}$ gets the best fitting when all $b_i = 0$ for $i = 1, \ldots, m$, we believe that $\boldsymbol{x}$ cannot help forecasting $\boldsymbol{y}$. To test whether $\boldsymbol{x}$ can help forecast $\boldsymbol{y}$, we use the Granger causality test [6], and for each lag m the result maybe differ. We use

$$GR_{\boldsymbol{x},\boldsymbol{y}} = \frac{number\ of\ m\ where\ \boldsymbol{x}\ helps\ forecast\ \boldsymbol{y}}{total\ number\ of\ m} \tag{8}$$

to represent importance of $\boldsymbol{x}$ to $\boldsymbol{y}$, $GR_{\boldsymbol{x},\boldsymbol{y}} \in [0, 1]$. We select one category and two concrete products to verify the inter-dependencies in Figure 3.

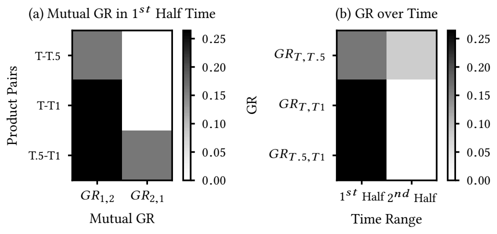


Figure 3: GR among three variables, where $T.5$ is the product item "Black Tea 500mL", $T1$ is the item "Black Tea 1L" and T is the category "black tea". Time range is split into two parts. (a) Mutual GR of three pairs. (b) Change of dependencies over time.

From Figure 3(a), we can find that $GR_{T,T.5}$ and $GR_{T,T1}$ are quite darker while $GR_{T.5,T}$ and $GR_{T1,T}$ are almost white. This figure means that the historical data of the category "black tea" helps forecasting the demand of its children and instead the children's history data is trivial to the forecast of their parent category. $GR_{T.5,T1}$ and $GR_{T1,T.5}$ involve the different degree of darkness. It means that $GR_{T.5,T1}$ can improve the forecast of $GR_{T1,T.5}$ and vice versa. However, the improvement degree is not identical. Figure 3(b) shows that the inter-dependencies is dynamically changing over time.

To verify that various variables have different temporal characteristics, we use Fast Fourier Transform (FFT) to plot the periodogram [14] for the category "black tea" and its two children product items in Figure 4. We can easily find that they have rather different periodicity at the rectangles. Thus, we cannot simply apply the same attention onto all variables.

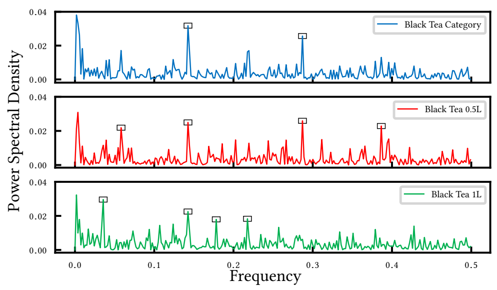


Figure 4: Periodogram of one category and two products. The rectangles highlight points for periodicity candidates.

4.2 Methods and Metric

We compare our model with five other methods, and their variants. For AR, Ridge and LSVR, we train models for each product separately, while neural network models take multivariate data as input directly.

- AR [1]: A classic univariate time series modeling method.
- Ridge [11]: Linear regression with $\ell 2$ regularization.
- LSVR [2]: Linear SVR, another machine leaning method for regression.

Table 2: RSE of six methods and their variants on two datasets: boldface indicates the best result on each dataset.

Methods	Dataset-1	Dataset-2
AR [1]	0.8016	0.5313
Ridge [11]	0.7981	0.5628
LSVR [2]	3.2446	7.8246
GRU [4]	0.8535	0.9067
LSTNet [7]	0.7907	0.7570
LSTNet-IAttn	0.8230	0.5399
STANet	**0.7783**	**0.5233**
STANet-oStructure	0.8907	0.6179
STANet-oCategory	0.8256	0.5569
STANet-oBrand	0.8640	0.5471
STANet-oAR	0.8327	0.8850

- GRU [4]: Recurrent neural network using GRU cell.
- LSTNet [7]: A state-of-the-art model composed of CNN, GRU and highway network.
- LSTNet-IAttn: LSTNet improved by the variable-wise temporal attention.
- STANet-oStructure: STANet without structural information.
- STANet-oCategory: STANet without category information.
- STANet-oBrand: STANet without brand information.
- STANet-oAR: STANet without autoregressive component.

We measure the forecasting performance by root relative squared error (RSE).

$$RSE = \frac{\sqrt{\sum_{i=1}^{N}\sum_{t=t_0}^{t_1}\left(y_{i,t} - \hat{y}_{i,t}\right)^2}}{\sqrt{\sum_{i=1}^{N}\sum_{t=t_0}^{t_1}\left(y_{i,t} - mean(Y)\right)^2}}, \tag{9}$$

where y and $\hat{y}$ are ground truth and predicted value respectively, t_0 and t_1 are start and end time of testing set, and $Y \in \mathbb{R}^{N\times(t_1-t_0)}$ represents the matrix of all labels y in testing set. RSE can be regarded as RMSE divided by standard deviation of testing set, so scale differences between different datasets can be ignored. Lower RSE generally means better forecasting performance.

4.3 Results of Different Methods

Table 2 provides the RSE of aforementioned methods on two datasets for horizon = 1. Our proposed model STANet outperforms others on both datasets. It is because STANet leverages the attention mechanism and inherent product structural information to precisely capture structural and temporal dependencies. LSVR performs worst on both datasets. Vanilla GRU suffers from worse performance than univariate models, because not every product has inter-dependency with each other and simply adding irrelevant data would harm the forecasting task. LSTNet can achieve lower errors than AR and Ridge on dataset-1, but not on dataset-2. It is mainly because the product items in dataset-1 exhibit much stronger structural inter-dependencies than those in dataset-2. For most methods except LSVR and GRU, the RSE on dataset-2 is much lower than that on dataset-1. It is due to the fact that dataset-1 contains much sparser data than dataset-2.

In terms of the ablation experiments of STANet, we can find that the structural information and the AR component play the major contribution in the forecasting task. For example, the result of STANet-oStructure indicates that the removal of structural information could greatly harm the forecasting accuracy. Incorporating the brand and category information will benefit the forecast and their corresponding contribution heavily depends upon the datasets. Both parts work together to the best performance. Also the results of STANet-oAR without the AR component indicate that its RSE is much higher than the original STANet especially on dataset-2. This is because the AR component can more comfortably capture the local trend in dataset-2 with a lower sparsity than the one in dataset-1. Finally, by comparing the results of LSTNet and LSTNet-IAttn, we find that LSTNet-IAttn by incorporating the variable-wise attention mechanism can greatly improve the forecast performance on dataset-2.

5 CONCLUSIONS

In this paper, we propose a novel deep learning-based forecasting model STANet in a multivariate time series manner. The model integrates the four components of GAT, GRU, variable-wise attention mechanism and auto-regression to precisely capture the inherent product structural information and temporal periodicity for more accurate prediction. Our analytic result on two real datasets demonstrates that the two real datasets exhibit strong product structural information and temporal periodicity. The evaluation result on the two datasets validates that STANet outperforms 6 counterparts and 4 variants of STANet. As the future work, we plan to further improve STANet and provide more experimental results on both online and offline transaction data.

ACKNOWLEDGMENTS

This work is partially supported by National Natural Science Foundation of China (Grant No. 61572365, No. 61772371, No. 61972286).

REFERENCES

[1] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung. *Time series analysis: forecasting and control.* John Wiley & Sons, 2015.
[2] L.-J. Cao and F. E. H. Tay. Support vector machine with adaptive parameters in financial time series forecasting. *IEEE Transactions on neural networks*, 14(6):1506–1518, 2003.
[3] X. Chen, J. Z. Huang, and J. Luo. Purtreeclust: A purchase tree clustering algorithm for large-scale customer transaction data. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 661–672. IEEE, 2016.
[4] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.
[5] J. L. Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.
[6] M. Gregorova, A. Kalousis, and S. Marchand-Maillet. Learning predictive leading indicators for forecasting time series systems with unknown clusters of forecast tasks. *arXiv preprint arXiv:1710.00569*, 2017.
[7] G. Lai, W.-C. Chang, Y. Yang, and H. Liu. Modeling long-and short-term temporal patterns with deep neural networks. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 95–104. ACM, 2018.
[8] Y. LeCun, Y. Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.
[9] H. Lütkepohl. *New introduction to multiple time series analysis.* Springer Science & Business Media, 2005.
[10] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal. Long short term memory networks for anomaly detection in time series. In *Proceedings*, page 89. Presses universitaires de Louvain, 2015.
[11] G. A. Seber and A. J. Lee. *Linear regression analysis*, volume 329. John Wiley & Sons, 2012.
[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
[13] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.
[14] M. Vlachos, P. Yu, and V. Castelli. On periodicity detection and structural periodic similarity. In *Proceedings of the 2005 SIAM international conference on data mining*, pages 449–460. SIAM, 2005.
[15] J. Yin, W. Rao, M. Yuan, J. Zeng, K. Zhao, C. Zhang, J. Li, and Q. Zhao. Experimental study of multivariate time series forecasting models. In *ACM CIKM 2019*, pages 2833–2839.

Efficient Skyline Computation in High-Dimensionality Domains

Rui Liu
University of Tours, France
rui.liu@univ-tours.fr

Dominique Li
LIFAT Laboratory, University of Tours, France
dominique.li@univ-tours.fr

ABSTRACT

We present a dimension indexing based algorithm for skyline computation. We first show that the dominance tests required to determine a skyline tuple can be sufficiently bounded to a subset of the current skyline, and then propose the algorithm SDI, of which the time complexity is better than the best known algorithm in high-dimensionality domains with reasonably low cardinality. Our performance evaluation on synthetic and real datasets shows that SDI outperforms the state-of-the-art skyline algorithm in both low-dimensionality and high-dimensionality domains.

1 INTRODUCTION

The *skyline computation problem* aims at retrieving the complete set of dominating tuples from multidimensional data, with respect to a monotonic *preference order* on all dimensions. Over several past decades, many algorithms have been developed, which can be categorized into sorting based [1, 4, 6, 7, 14] and partitioning based [2–4, 8–13, 15]. Most the existing skyline algorithms have been designed for low-dimensionality domains because of the quadratic issue raised by the worst case time complexity.

In this paper, we present a dimension indexing based skyline algorithm SDI (Scalable Dimension Indexing) that is efficient in high-dimensionality domains as well as in low-dimensionality domains. We show that by indexing all dimensions, it is sufficient to test a tuple only with the existing skyline tuples on an arbitrary dimension instead of with the complete set of skyline tuples. We also show that any skyline tuple can be used as a *stop line* that traverses the indexed dimensions to stop the computation, which is much performant than the calculation of *stop point* mentioned in SaLSa [1]. Furthermore, SDI adopts the *weak incomparability checking* to take the incomparability between tuples into account, which is the most important feature of the state-of-the-art skyline algorithm BSkyTree [10]. Our analysis shows that the worst time complexity of SDI is better than the best known one [13] in high-dimensionality domains with reasonably low cardinality, and our performance evaluation shows that SDI outperforms BSkyTree on both low and high dimensional data, but less efficient than BSkyTree on medium dimensional data.

The rest of this paper is organized as follows. Section 2 presents the SDI algorithm with preliminary definitions. We show our theoretical analysis of the computational complexity of SDI in Section 3. Section 4 reports the performance evaluation of SDI on both synthetic and real datasets. We conclude in Section 5.

2 THE SDI APPROACH

Let t be a d-dimensional tuple, we denote $t[i]$ the *dimension value* of t on the dimension i, where $1 \leq i \leq d$. We define the *preference order*, denoted by $\prec$, as a total order that covers each dimension such that given two tuples t and u, $t[i]$ is *better than* $u[i]$ if $t[i] \prec u[i]$; $t[i]$ is *equal to* $u[i]$ if $t[i] = u[i]$; $u[i]$ is *not worse* than $t[i]$ if $(t[i] \prec u[i]) \vee (t[i] = u[i])$ holds, denoted by $t[i] \preceq u[i]$. We say that a tuple t *dominates* a tuple u, denoted by $t \prec u$, if and only if for each dimension i, we have $t[i] \preceq u[i]$, and for at least one dimension k we have $t[k] \prec u[k]$. We denote $t \not\prec u$ that t does not dominate u, and $t \nsim u$ that t and u are *incomparable*, that is, $(t \not\prec u) \wedge (u \not\prec t)$. Considering a d-dimensional database $\mathcal{D}$ and a preference order $\prec$ on $\mathcal{D}$, a tuple $t \in \mathcal{D}$ is a *skyline tuple* if and only if $\nexists u \in \mathcal{D}$ such that $u \prec t$. The *skyline* of $\mathcal{D}$ is the complete set $\mathcal{S}$ of skyline tuples such that $\mathcal{S} = \{t \in \mathcal{D} \mid \nexists u \in \mathcal{D}, u \prec t\}$. We have $s \nsim t$ for any two skyline tuples s and t.

Given a database, the *dimension index*, denoted $\mathcal{I}$, is the set of d ordered lists on a preference order $\prec$, in which each list $I_i \in \mathcal{I}$ is a *dimensional subindex* that contains all dimension values sorted with respect to $\prec$.

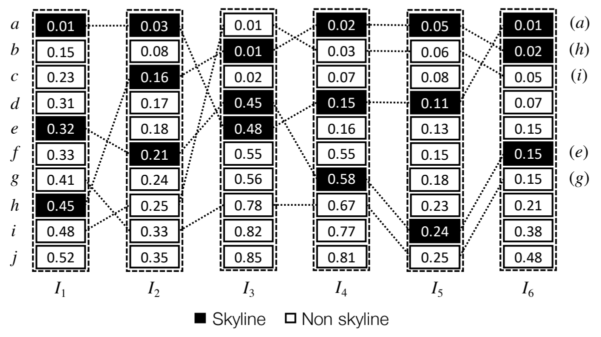

a b c d e f g h i j
I1: 0.01 0.15 0.23 0.31 0.32 0.33 0.41 0.45 0.48 0.52
I2: 0.03 0.08 0.16 0.17 0.18 0.21 0.24 0.25 0.33 0.35
I3: 0.01 0.01 0.02 0.45 0.48 0.55 0.56 0.78 0.82 0.85
I4: 0.02 0.03 0.07 0.15 0.16 0.55 0.58 0.67 0.77 0.81
I5: 0.05 0.06 0.08 0.11 0.13 0.15 0.18 0.23 0.24 0.25
I6: 0.01 0.02 0.05 0.07 0.15 0.15 0.15 0.21 0.38 0.48
(a) (h) (i) (e) (g)
■ Skyline □ Non skyline

Figure 1: A dimension index example.

Let us consider the 5 tuples $\{a, e, g, h, i\}$ presented in Figure 1 that shows a dimension index example consisting of 10 tuples and 6 subindexes, where the dashed lines link the tuple among the subindexes. Obviously, a is a skyline tuple, which can be independently concluded from I_1, I_2, and I_6 because no tuple can dominate a; if we regard only I_4, h is immediately a skyline tuple and in order to determine whether i, the second tuple in this subindex, is a skyline tuple, it is enough to compare i with h because any tuple x located after the position of i in I_4 cannot dominate i since we have $i[4] \prec x[4]$ in I_4. Nevertheless, if we focus on I_3, we see that $h \prec i$ and $i \prec h$ must be first tested in order to decide whether h or/and i shall be skyline tuple(s) since $h[3] = i[3]$ (indeed we have $h \prec i$ and $i \not\prec h$). That is also the case in I_6, where 3 tuples contain the same dimension value 0.15, so all these 3 tuples must be first locally compared in order to filter the potential skyline tuples (in our example, e). We call such tuples as h in I_3 and e in I_6 the *local skyline tuple*. It is easy to see that any tuple is a local skyline tuple on a given dimension if there are no identical dimension values.

Theorem 1. *Let $\mathcal{I}$ be the dimension index of a database $\mathcal{D}$ and $I_i \in \mathcal{I}$ be an arbitrary subindex of the dimension i. Then, a local* *skyline tuple t is a skyline tuple if and only if (1) there is no tuple u such that $u[i] \prec t[i]$, or (2) for any skyline tuple s such that $s[i] \prec t[i]$, we have $s \not\prec t$.*

Proof. Let t be a local skyline tuple. If t is the first tuple in I_i, then t is a skyline tuple because no tuple is better than t. Otherwise, let s be a skyline tuple such that $s[i] \prec t[i]$, then if $s \prec t$, t cannot be a skyline tuple since s dominates t; if $s \not\prec t$, then there exists at least on dimension $k \neq i$ such that $t[k] \prec s[k]$ so no tuple dominated by s dominates t. Hence, for each skyline tuple s such that $s[i] \prec t[i]$, if we have $s \not\prec t$, then no tuple in the database dominates t, thus, t is a skyline tuple. If $t[i] \prec s[i]$, then $s \not\prec t$ so it is meaningless to compare t with s. □

Theorem 1 allows to determine whether a tuple is a skyline tuple only with a subset of the existing skyline. Furthermore, once the dimension index has been constructed, Theorem 1 allows to switch among subindexes so that the best one containing the least of known skyline tuples can always be selected. Indeed, let p be a skyline tuple, then any tuple t such that $p \prec t$ can be pruned from the database in order to reduce the computation time. In this paper, we propose the notion of stop line based on the dimension index of which the effectiveness can be guaranteed.

Theorem 2. *Let $\mathcal{I}$ be a dimension index of a d-dimensional database $\mathcal{D}$ and $p \in \mathcal{D}$ be an arbitrary skyline tuple. Let $o_i(p)$ denote the largest offset of any tuple x such that $x[i] = p[i]$ in the dimensional subindex $I_i \in \mathcal{I}$, if all offsets $o_i(p)$, $1 \leq i \leq d$, have been reached by following a top-down traversal on all dimensions, then the complete set of skyline tuples has been identified and the computation can be terminated.*

Proof. Let p and $t \in \mathcal{D} \backslash p$ be two skyline tuples in $\mathcal{D}$, we have: (1) $t \nsim p$; or (2) t and p have identical values on all dimensions. We denote $L_p = \bigcup_{1 \leq i \leq d} o_i(p)$ the set of all offsets $o_i(p)$. In the first case, $t \nsim p \Rightarrow \exists k$ such that $p[k] \prec t[k]$, i.e. $o_k(p) < o_k(t)$, hence, if the index traversal reaches all offsets in L_p, t must have been identified at least in the dimension k. In the second case, we have $p[i] = t[i]$ on any dimension i. In both cases, if all offsets in L_p have been reached, all skyline tuples have been identified. □

We call the set $L_p = \bigcup_{1 \leq i \leq d} o_i(p)$ a *stop line* that can safely terminate the skyline computation of SDI. In theoretical, any skyline tuple can be selected as a stop line, however, different stop lines behave differently in pruning irrelevant tuples. We propose therefore a function min_{stop} to find the best stop line L_p^*, defined as:

$$L_p^* = min_{stop}(p) = \arg\min_p (max\{o_i(p)\}, \sum_{i=1}^{d} o_i(p)),$$

where d is the dimensionality of the data. The function min_{stop} sorts first by the maximum offset, then by the sum of offsets in all dimensions, so the skyline tuple having the minimized value is the best stop line. The best stop line L_p can be dynamically maintained by keeping $min_{stop}(p) < min_{stop}(t)$ for any two skyline tuples p and t.

The *incomparability checking* is taken into account while a dominance test is proceeding. In our approach, we consider that a d-dimensional dominance test runs in $O(d)$ time, so any tuple comparison better than $O(d)$ time shall improve the efficiency of SDI. Indeed, to efficiently determine $s \nsim t$ in stead of testing $s \prec t$ in the case of $s \not\prec t$ is an essential time-costly task while comparing t with all existing skyline tuples in a dimensional subindex. In this paper, we propose a *weak* checking mechanism of the incomparability between a skyline tuple s and a testing tuple t in a dimensional subindex I_i as following.

Theorem 3. *Let $\mathcal{I}$ be a dimension index and s be a skyline tuple present in a dimensional subindex $I_i \in \mathcal{I}$. Given a tuple t such that $s[i] \prec t[i]$, we sufficiently have $s \nsim t$ if $max(s) > max(t)$, or $min(s) > min(t)$, or $sum(s) > sum(t)$.*

Proof. The sets L_s and L_p (see the proof of Theorem 2 for the definition) can be considered as the coordinates of two curves in a two-dimensional space. In Euclidean geometry, $max(s) > max(t)$, or $min(s) > min(t)$, or $sum(s) > sum(t)$ are sufficient conditions for the existence of at least one intersection of the curves formed by s and t since we have $s[i] \prec t[i]$, i.e. $o_i(s) < o_i(t)$, that is, according to the definition of dominance, $s \nsim t$. □

Note that the maximal value, the minimal value, and the sum of a tuple can be pre-calculated while constructing the dimension index, so Theorem 3 can efficiently determine $s \nsim t$. However, Theorem 3 shows in fact 3 sufficient conditions for $s \nsim t$, hence a dominance test is necessary to determine $s \nsim t$ in the cases that are not covered by Theorem 3, for which we call Theorem 3 a weak incomparability checking. The sketch of SDI is listed in Algorithm 1.

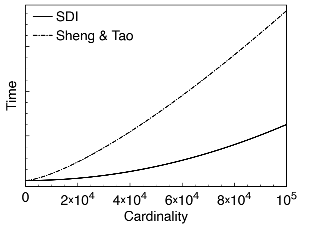
Algorithm 1: SDI

```
   Input: Dimension index $\mathcal{I}$
   Output: Skyline $\mathcal{S}$
1  while true do
2      $I_{best} \leftarrow BestSubindex(\mathcal{I})$
3      while $T \leftarrow NextLocalSkyline(I_{best})$ do
4          if $T = null$ then
5              return $\mathcal{S}$
6          foreach $t \in T$ and $t \notin \mathcal{S}$ do
7              if $\mathcal{S}_{best} \not\prec t$ then
8                  $\mathcal{S}_{best} \leftarrow \mathcal{S}_{best} \cup t$
9                  $\mathcal{S} \leftarrow \mathcal{S} \cup t$
10         if Found new skyline tuples then
11             Update StopLine
12             break
13     if StopLine is reached then
14         return $\mathcal{S}$
```

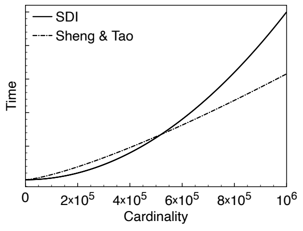
Extensions. (1) SDI computes the skyline in the categorical domains as long as the preference order $\prec$ can be defined; (2) SDI can be immediately adapted to subspace skyline computation by skipping unrelated dimensions; (3) SDI can be extended to the skyline maintenance by dynamically constructing the dimension index; (4) SDI can handle the top-k skyline query by finding the skyline tuples having the best positions in the dimension index.

3 THEORETICAL ANALYSIS

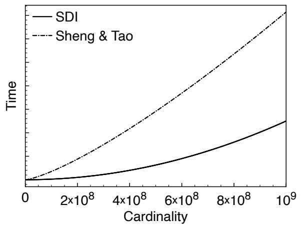
We denote d the dimensionality and N the cardinality of the data, and M the size of the skyline. We discuss without duplicate values on any dimension, but if K duplicate values are present in a dimensional subindex, $O(dK^2)$ shall be considered in assuming that BNL is applied to compute local skylines. Note that we consider $O(d)$ time for a d-dimensional dominance test, which implies the tests of $s \prec t$ and $t \prec s$, hence, the dominance test



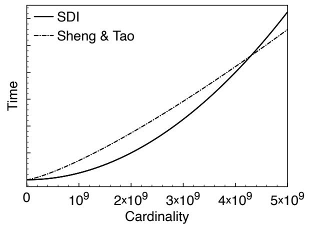
(a) $d = 6, N = 1 \times 10^5$.



(b) $d = 6, N = 1 \times 10^6$.



(c) $d = 8, N = 1 \times 10^9$.



(d) $d = 8, N = 5 \times 10^9$.

Figure 2: Numerical simulation for complexity study in the worst case.

within SDI is considered in $O(d/2)$ time since $s < t$ is sufficient. We also note that the construction of dimension index requires $O(dN \log N)$ time with respect to general $O(N \log N)$ sorting algorithms on d dimensions, and we do not consider the two heuristics of stop line and incomparability checking.

The average time complexity of SDI is measured on M skyline tuples uniformly distributed in d dimensional subindexes.

Theorem 4. *Considering M skyline tuples, SDI computes the skyline of a d-dimensional database with the cardinality N in*

$$O(dN \log N + \frac{M(2N - M - d)}{4}).$$

Proof. With M skyline tuples uniformly distributed on each dimension, $(N - M)/d$ non skyline tuples must be compared with M/d skyline tuples. For each dimension, in the worst case, $(M/d)(M/d - 1)/2$ dominance tests are required by skyline tuples, $((N - M)/d)(M/d)$ dominance tests are required by non skyline tuples, and each dominance test cost $O(d/2)$ time. Therefore, the average time complexity of SDI is $O(((M/d)(M/d - 1)/2 + ((N - M)/d)(M/d))(d/2)d)$, that is the result shown in Theorem 4. □

Theorem 5. *In the worst case, all N tuples in a d-dimensional database are skyline tuples. SDI computes the skyline in*

$$O(dN \log N + \frac{N^2 - dN}{4}).$$

Proof. The proof is immediate if we replace M in Theorem 4 by N. □

Comparing with the best known worst-case time complexity $O(N \log^{max(1,d-2)} N)$ proposed by Sheng and Tao [13], given $d > 2$, the following equation must be resolved:

$$N \log^{d-2} N > dN \log N + \frac{N^2 - dN}{4}.$$

The above equation belongs to transcendental equations that have no closed-form solutions. Our numerical simulation results presented in Figure 2 shows that while $d = 6$, SDI is better than the approach of Sheng and Tao for $N < 5 \times 10^5$; and while $d = 8$, the compared approach beats SDI only if $N > 4 \times 10^9$. SDI performs better in high-dimensionality domains with respect to a reasonable data cardinality.

4 PERFORMANCE EVALUATION

We evaluate the performance of SDI in comparison with BSkyTree implemented in SkyBench[1] [5] on both synthetic datasets and real world datasets. The synthetic datasets are generated by Skyline Benchmark Data Generator[2], including uniform independent (UI), correlated (CO), and anti-correlate (AC) data; the real world datasets include NBA, HOUSE, and WEATHER [5]. We implemented SDI[3] in C++ standard and all executables are compiled by LLVM Clang with -O3 option. All experiments have been conducted on an Intel Core i5 2.8 GHz processor with 16GB 1600 MHz DDR3 RAM, running macOS 10.15.1 operating system. We note that all results reported are the average performance over 5 iterations.

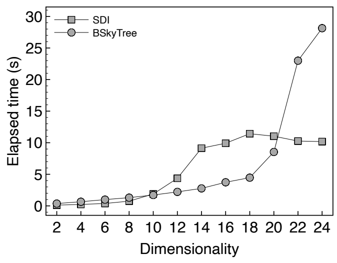
First, the effects of **(1) dimensionality** and **(2) cardinality** of data have been evaluated. For (1), the cardinality is fixed to 100K and for (2), the dimensionality of data is fixed to 24. Due to the space limitation, only overall elapsed time, that is, the sum of data loading time, data structure construction time, and query time, has been reported in this paper. Indeed, we consider that for single-round skyline queries, to focus on total processing time is much important than to focus only on the query time without looking at data structure building time.

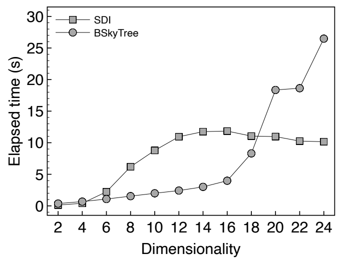
Dataset	$d = 2$	$d = 4$	$d = 6$	$d = 8$	$d = 10$	$d = 12$
UI	7	259	2,597	9,960	25,737	46,301
CO	1	5	31	120	449	790
AC	50	4,100	26,713	56,118	75,668	87,857
Dataset	$d = 14$	$d = 16$	$d = 18$	$d = 20$	$d = 22$	$d = 24$
UI	67,676	82,286	92,011	96,832	99,059	99,662
CO	1,439	2,941	5,471	8,936	14,498	16,948
AC	94,053	96,956	98,413	99,205	99,570	99,760

Table 1: Skyline size of synthetic datasets (N = 100K).

Table 1 lists the skyline size of synthetic datasets with the cardinality of 100K. We see that the higher the dimensionality of data, the closer to the worst case is likely to be. For instance, the skyline consists of 92% of tuples in UI data while $d = 18$, however while $d = 14$ of AC data, the skyline rate reaches already 94%. Figure 3 shows the effect of dimensionality on SDI where the cardinality N = 100K is reasonable in the most of use cases. SDI outperforms BSkyTree in both low-dimensionality and high-dimensionality domains on UI ($d \leq 10$ or $d > 20$) and AC data ($d \leq 4$ or $d > 20$), but is less efficient than BSkyTree in other dimensionalities. In fact, we note that the stop line takes no advantage in AC data because of the *anti-correlated* characteristics of data, however SDI systematically outperforms BSkyTree on the CO data because the stop line can efficiently determined with respect to the strong *correlation* in data. Figure 4 shows the effect of cardinality on SDI with the highest dimensionality in our experiments. In high-dimensionality domains, SDI outperforms BSkyTree in most cases except in AC data. Again, we confirm that the stop line does not show any advantage in high-dimensionality

[1]https://github.com/sean-chester/SkyBench
[2]http://pgfoundry.org/projects/randdataset
[3]https://github.com/skyline-sdi/sdi-bench


(a) The UI dataset.


(b) The AC dataset.

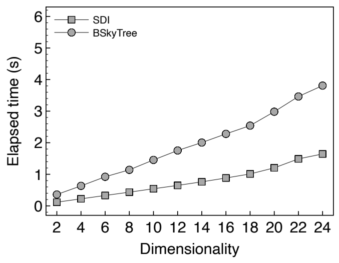
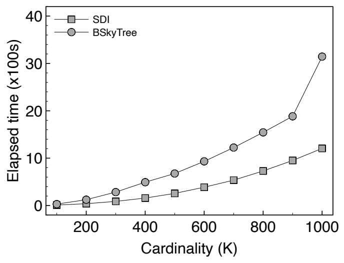
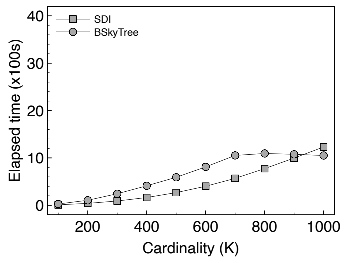

(c) The CO dataset.

Figure 3: Performance evaluation on the effect of dimensionality (N = 100K).


(a) The UI dataset.


(b) The AC dataset.

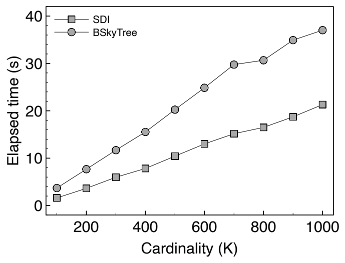

(c) The CO dataset.

Figure 4: Performance evaluation on the effect of cardinality (d = 24).

and high-cardinality AC data. Table 2 lists the pruned tuples by the stop line in different synthetic datasets while $N = 100$K, that are relevant to our experimental results.

Dataset	$d = 2$	$d = 4$	$d = 6$	$d = 8$	$d = 10$	$d = 12$
UI	99,849	88,011	63,260	30,576	12,381	8,110
CO	99,997	97,773	98,832	97,656	96,526	92,066
AC	36,731	5,022	2,227	240	30	27
Dataset	$d = 14$	$d = 16$	$d = 18$	$d = 20$	$d = 22$	$d = 24$
UI	3,794	297	933	35	3	8
CO	87,901	80,587	73,837	51,347	39,742	48,239
AC	13	25	0	10	0	0

Table 2: Pruned tuples by the stop line (N = 100K).

Table 3 shows the performance comparaison between SDI and BSkyTree on real datasets. SDI outperforms BSkyTree on NBA and HOUSE datasets but is much slower than BSkyTree on the WEATHER dataset because the huge number of duplicate dimension values in WEATHER makes $O(dK^2)$ (discussed in Section 3) an important factor.

Dataset	d	N	$\|\mathcal{S}\|$	SDI	BSkyTree
HOUSE	6	127,931	5,774	306 ms	839 ms
NBA	8	17,264	1,796	45 ms	155 ms
WEATHER	15	566,268	26,713	18,680 ms	11,641 ms

Table 3: Performance evaluation on real datasets.

5 CONCLUSION

In this paper, we presented an efficient Skyline computation algorithm. We proved that in multidimensional databases, skyline computation can be conducted on an arbitrary dimensional index which is constructed with respect to a predefined total order that determines the skyline. We further showed that any skyline tuple can be used to stop the computation process by outputting the complete skyline. Our experimental evaluation shows that SDI outperforms the state-of-the-art skyline algorithm in both low-dimensionality and high-dimensionality domains.

REFERENCES

[1] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. 2008. Efficient sort-based skyline evaluation. *ACM Transactions on Database Systems* 33, 4 (2008), 31.

[2] Jon L. Bentley, Kenneth L. Clarkson, and David B. Levine. 1993. Fast linear expected-time algorithms for computing maxima and convex hulls. *Algorithmica* 9, 2 (1993), 168–183.

[3] Jon L. Bentley, Hsiang-Tsung Kung, Mario Schkolnick, and Clark D. Thompson. 1977. On the average number of maxima in a set of vectors and applications. *J. ACM* (1977).

[4] S. Borzsony, D. Kossmann, and K. Stocker. 2001. The Skyline operator. In *ICDE*. 421–430.

[5] Sean Chester, Darius Šidlauskas, Ira Assent, and Kenneth S Bøgh. 2015. Scalable parallelization of skyline computation for multi-core processors. In *ICDE*. 1083–1094.

[6] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. 2003. Skyline with presorting. In *ICDE*, Vol. 3. 717–719.

[7] Parke Godfrey, Ryan Shipley, and Jarek Gryz. 2005. Maximal Vector Computation in Large Data Sets. In *VLDB*. 229–240.

[8] Donald Kossmann, Frank Ramsak, and Steffen Rost. 2002. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB*. 275–286.

[9] Hsiang-Tsung Kung, Fabrizio Luccio, and Franco P. Preparata. 1975. On finding the maxima of a set of vectors. *J. ACM* 22, 4 (1975), 469–476.

[10] Jongwuk Lee and Seung-Won Hwang. 2014. Scalable skyline computation using a balanced pivot selection technique. *Information Systems* 39 (2014), 1–21.

[11] Ken CK Lee, Wang-Chien Lee, Baihua Zheng, Huajing Li, and Yuan Tian. 2010. Z-SKY: an efficient skyline query processing framework based on Z-order. *The VLDB Journal* 19, 3 (2010), 333–362.

[12] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2005. Progressive skyline computation in database systems. *ACM Transactions on Database Systems* 30, 1 (2005), 41–82.

[13] Cheng Sheng and Yufei Tao. 2012. Worst-case I/O-efficient skyline algorithms. *ACM Transactions on Database Systems* 37, 4 (2012), 26.

[14] Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. 2001. Efficient Progressive Skyline Computation. In *VLDB*. 301–310.

[15] Shiming Zhang, Nikos Mamoulis, and David W Cheung. 2009. Scalable skyline computation using object-based space partitioning. In *SIGMOD*. 483–494.

Entity Matching with Transformer Architectures - A Step Forward in Data Integration

Ursin Brunner
Zurich University of Applied Sciences
Switzerland
ursin.brunner@zhaw.ch

Kurt Stockinger
Zurich University of Applied Sciences
Switzerland
kurt.stockinger@zhaw.ch

ABSTRACT

Transformer architectures have proven to be very effective and provide state-of-the-art results in many natural language tasks. The attention-based architecture in combination with pre-training on large amounts of text lead to the recent breakthrough and a variety of slightly different implementations.

In this paper we analyze how well four of the most recent attention-based transformer architectures (BERT[6], XLNet[33], RoBERTa[17] and DistilBERT [23]) perform on the task of entity matching - a crucial part of data integration. Entity matching (EM) is the task of finding data instances that refer to the same real-world entity. It is a challenging task if the data instances consist of long textual data or if the data instances are "dirty" due to misplaced values.

To evaluate the capability of transformer architectures and transfer-learning on the task of EM, we empirically compare the four approaches on inherently difficult data sets. We show that transformer architectures outperform classical deep learning methods in EM[7, 20] by an average margin of 27.5%.

1 INTRODUCTION

Entity Matching (EM) is the task of determining if two data instances refer to the same real-world object. As a simple example consider the entries in Table 1 and Table 2. We need to match two entities based on attributes like names, addresses or product information while dealing with various formats and differing, missing or wrong values. Given this uncertainty, the challenge of entity matching is to identify which existing entity pairs have the highest probability of being a match.

The task of entity matching is crucial especially for the process of data integration and data cleaning [4]. Due to the large amount of data produced every day, the challenge of data integration and therefore entity matching becomes more urgent than ever for corporations and large institutions with data stemming from multiple sources [27].

While early solutions often relied on rule-based approaches and hand-crafted heuristics, entity matching by now is mainly based on machine learning (ML) approaches. Large projects like *Magellan*[14] provide tools and libraries for the whole EM-pipeline with good results on structured and semi-structured data.

However, what if the data consists of large textual instances, such as product descriptions, posts on Reddit, Quora, Stackoverflow, or company descriptions? What if the data has structure, but attributes are dirty, e.g. the attribute "name" consists of a given name and a last name, while "given name" is empty? In those cases, traditional EM approaches provide only mediocre results or require large efforts in hand-crafted features [2, 13]. Therefore and due to the recent advances of deep learning in natural language processing (NLP), several papers suggest end-to-end deep learning architectures [7, 20, 34] for EM.

Table 1: Database A - structured product information, with *description* being a text-blob.

Title	Brand	Description	Price
iPhone XS	Apple	The brand new iPhone now available in white, red and silver.	899.99
ZenFone 4 Pro	Asus	Thin and light, yet incredibly strong, the ZenFone 4 Pro (ZS551KL) features an expansive 5.5-inch, Full HD AMOLED display	530.00

Table 2: Database B - textual product information.

Product description
Apple's new iPhone XS - a masterpiece of design. Available now with 64 or 128GB storage and in three colors: white, silver and red
A smart device for a decent price - Nokia's Pure View 9, powered by a pure android, is the gift you need for Christmas. With it's incredible robust design and a battery duration of two days under heavy load, you will love it from day one.

The above-mentioned deep learning architectures for EM apply different modern NLP techniques: [7] is based on a bi-directional LSTM and word embeddings, [20] uses an attention mechanism on top of an LSTM and [34] introduces the power of pre-trained models.

While these deep learning approaches already lead to massive performance improvements on difficult data sets [20], none of them use modern transformer architectures for EM. Transformer architectures, largely based on the influential "Attention is All You Need" [30] paper, have proven to be effective on a large variety of NLP tasks. Especially in combination with pre-training on large text corpora, they almost always beat earlier deep learning approaches based on recurrent neural networks (RNNs) on popular NLP benchmarks [6].

The contributions of our paper are as follows:

- To the best of our knowledge, we are the first to compare four of the most recent transformer architectures, namely BERT[6], XLNet[33], DistilBERT[23] and RoBERTa[17] on the task of entity matching. We run experiments with all of them on five different EM data sets. Since the authors

of [20] have shown that most of the traditional, structured EM data sets are not challenging for advanced deep learning architectures anymore, we focus on "dirty" data sets and data sets with large textual data instances.
- We compare the results of modern transformer architectures with more traditional deep learning architectures [7, 20, 34]. Our experiments demonstrate that all transformer architectures clearly outperform classical deep learning methods in EM. On challenging datasets, the best transformer outperforms DeepMatcher[20] by an average margin of 27.5%.
- Finally, we analyze how much training is required to achieve good results on the EM tasks with heavily pre-trained transformers. To elaborate on this, we compare the results before fine tuning on the data set (zero-shot learning) and after each epoch. The results indicate that transformers reach already good results after only one epoch of training, while after 2-3 epochs they converge towards their optimum solution.

The paper is organized as follows. Section 2 reviews the related work on entity matching and gives a brief overview on transformer technologies that appear to be a valid technology for entity matching. Section 3 provides the background knowledge on attention-learning and the transformer architecture. Section 4 describes the four architectures and their pre-training algorithms in detail and explains the intuition behind each approach. In Section 5 we compare the four architectures based on their results on five selected EM data sets. Finally, we present our conclusion and future work in Section 6.

2 RELATED WORK

We can divide the work related to this paper roughly in two categories, namely *entity matching* and *transformer architectures*. We will discuss each of them in more detail.

2.1 Entity Matching

The entity matching process as a whole has been tackled over the years by many works [4, 7, 13, 14, 20], with [4] providing an excellent overview over the major challenges of entity matching. Recent projects like *Magellan*[14] focus not only on research, but provide a set of practical tools to solve entity matching in data integration processes. For a long time the field of entity matching focused on rather structured data records [8], with each attribute of the data records containing only relatively short amounts of text.

To calculate the similarity of such textual attributes, a variety of similarity functions has been developed [4]. Many of them focus on specific data structures as e.g. the *Jaro-Winkler* distance [11], known to work well on person names. The similarity values of such functions where then used as features in a binary classification problem (*match*/*no match*)[4].

With the emergence of deep learning in NLP, several papers [7, 20] started to apply new techniques such as *word embeddings* and *attention* to the task of entity matching. The advantages of such deep learning models is that they remove the need for handcrafted features. While recent work [34] introduced the power of *pre-training* for entity matching, we are to our knowledge the first paper approaching entity matching with modern *transformer architectures*.

2.2 Transformer Architectures

Based on the well-known paper "Attention is all you need"[30], transformers - a special type of neural networks - started by mid 2017 to become one of the most interesting techniques for NLP. BERT [6], combining the transformer architecture with massive unsupervised pre-training was the first paper to achieve state-of-the-art results in a large number of NLP tasks. Succeeding works by [16, 17, 33] achieved even higher results on said NLP benchmarks.

While transformers undeniably are one of the largest achievements in the 2018/2019 NLP landscape, they also started a controversy about the *more data/larger models/more computational power* mentality. Therefore, in addition to increasing model sizes like the authors of [26] did, research started also to develop more lean transformer models, which can be used on mobile devices or non-GPU servers at inference time [23].

Transformers have traditionally been used for NLP-tasks. However, we will apply this technology for entity matching - an important aspect of data integration.

3 ATTENTION AND TRANSFORMERS

In this section we provide detailed background information on a special type of neural networks called transformers. We will use this technology later on for entity matching.

The field of NLP has been dominated for a long time by architectures using recurrent neural networks (RNNs) at its core. The most popular approaches for machine translations were so-called *seq2seq*[29] or *encoder/decoder*[3] architectures, which basically consist of two RNNs, one for the encoder, and one for the decoder. Incrementally improved versions of these architectures are used in most translation software packages. For instance, Google Translate started using such models in late 2016. Even though the RNN architecture seems to be well-suited for building representations of sequential text data, several issues were discovered:

- The so-called *bottleneck problem*[1] makes it hard to transfer knowledge about the source sentence to the target sentence. The bottleneck is the context vector, which is basically all the decoder network gets as an input. By conditioning only on this single vector, the decoder then has to produce an output sequence of length N. It is intuitive to understand that the longer the sequence, the harder it is to get all necessary information of the source sentence in one single vector.
- An RNN is, due to its sequential nature, harder to parallelize and therefore takes longer to train than a simple feed forward network.
- Long range dependencies are hard to learn with RNNs, even though LSTMs and GRU (gate recurrent unit) architectures theoretically allow it. The authors of [30] describe this effect by the path length between two tokens (e.g. two words in a sentence), which is the number of steps the signal has to flow through the network. The longer the path, the harder to learn a dependency.

3.1 Attention

To overcome the bottleneck problem, [1] and [18] came up with a new technique called *attention*. Let us first elaborate on the idea of *self-attention*, to describe the intuition: A word can be represented as a weighted combination of its neighborhood. As an example, take the word *"it"* in Figure 1. For the human reader it is intuitive that the word *it* in this sentence can be represented by the words *The* and *animal*. By applying an attention mechanism, we can train a language model to pay attention to relevant words in its neighborhood in a similar way.

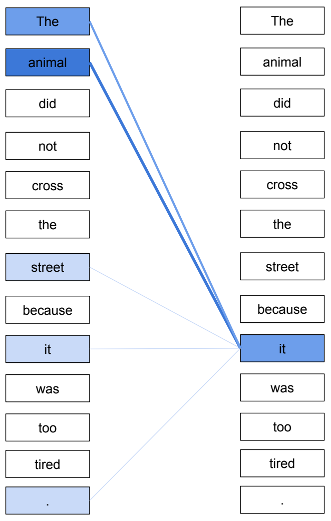


Figure 1: Self-Attention: The token "it" (see right side) can refer to the tokens "The", "animal", "street" etc. (see left side).

A sightly different version of attention was used by the first seq2seq model with attention mechanism[1]. This approach applied attention between the target sentence and the source sentence. So instead of only transferring knowledge between encoder and decoder via one context vector at the end of the encoding, the decoder has access to all the hidden states of the encoder.

Assume that we want to translate the English sentence *"The cat is sleeping"* to the German counterpart *"Die Katze schläft"*, as in Figure 2. It is helpful for the decoder to only consider information of the encoder's hidden state for *is* and *sleeping*, when the decoder is trying to produce the word *schläft*. By applying attention, the decoder will learn to focus on the important hidden states of the encoder at each given time step.

3.2 The Transformer

In 2018, Vaswani et al. published a paper with a more radical approach [30]: an architecture that completely relinquishes RNNs with the idea to build a model with attention only. The core of this paper, the so-called transformer, can be found in Figure 3. While the transformer still retains the classical encoder/decoder structure, it replaced the RNN by a so-called Multi-Head-Attention sub-layer. There are three of those attention sub-layers in each of the *N* stacked layers. The first two Multi-Head Attention sub-layers implement the self-attention mechanism. During the training, they will independently learn dependencies in the source and target sentence respectively, as previously shown in Figure 1. It is important to understand that in the decoder, the self-attention mechanism will only have access to all the words produced by the decoder so far, while all the words that still need to be processed are masked. Masking words with high negative values will basically "hide" them from the learning algorithm.

While the encoder contains only one Multi-Head Attention sub-layer per layer, the decoder contains a second one. This sub-layer performs attention over the output of the encoder stack.

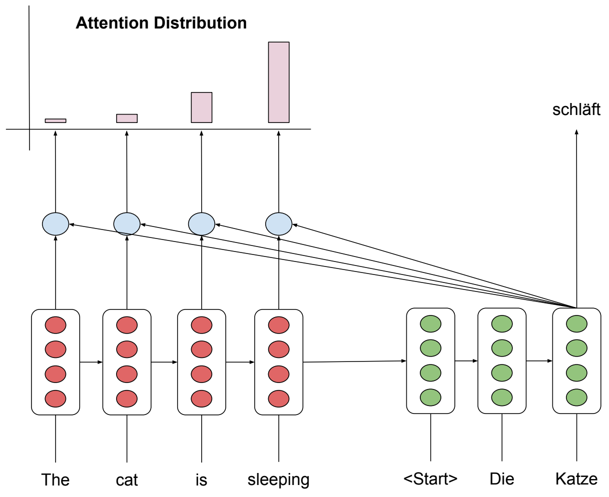


Figure 2: Sequence-to-sequence model with attention. The decoder (green) has access to all hidden states of the encoder (red) and learns how much attention to pay to each hidden state. To predict the word *schläft*, the decoder pays most attention to the hidden state of *is* and *sleeping*.

From a conceptual point of view, this sub-layer performs a similar task to the attention mechanism of classical seq2seq architectures (see previously discussed Figure 2). As a consequence, the Multi-Head Attention sub-layer allows the decoder to pay attention only to the relevant words of the source sentence.

Since a transformer model does not contain any recurrence, the authors had to introduce the idea of ordered, sequential inputs in a different way. They do this by using so-called *positional encoding* for each input word. This allows the model to make use of both the position of a word and the relative distance between two words. The authors of [30] use a combination of the functions sine and cosine with different frequencies to implement positional encodings.

A last detail to mention is the *residual connection* [9] around each sub-layer. The residual connection will, on the one hand, improve training performance, but even more importantly, it allows the positional encoding to flow untouched up to the higher layers.

4 BERT AND FRIENDS

The transformer architecture appearing first in 2017 triggered a wave of new papers based on this idea. With BERT [6] being the most popular one and setting a new state-of-the-art for many NLP tasks in 2018, other papers followed quickly and further improved performance on many tasks. In this section we will have a look at the four approaches used in our experiments: BERT, XLNet, RoBERTa and DistilBERT. The order of the subsections corresponds to the publication date of the papers.

4.1 BERT

BERT is a universal language model, pre-trained on large amounts of text data with the intention of fine-tuning it on downstream tasks (e.g. entity matching) in a supervised manner with relatively little data.

The abbreviation BERT stands for *Bidirectional Encoder Representations from Transformers*, with the emphasis on *bidirectional*.

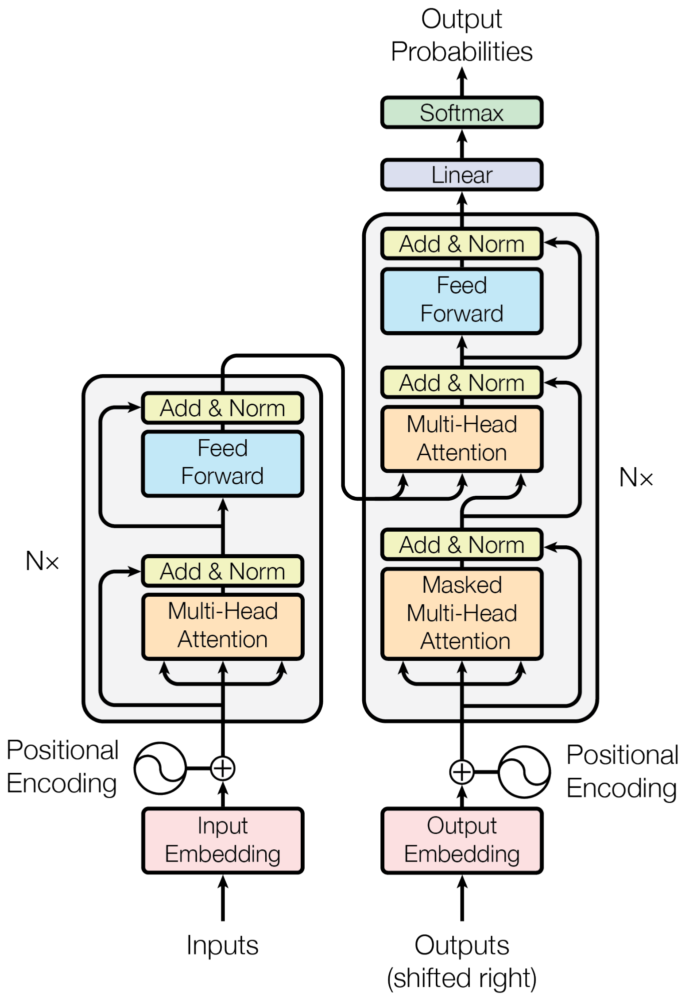


Figure 3: The original transformer architecture by Vaswani et al. as shown in the paper[30].

It is in its core a transformer language model as designed by [30], but contrary to other language models (e.g. OpenAI's GPT[21]) it is jointly conditioning on both the left- and right context of the query token during pre-training. This is somehow counter intuitive, as the most common training task of language models is to simply predict the next token (word).

Let us consider the following fraction of a sentence "[..] problems turning into banking crisis as [..]" (see Figure 4). Let us further assume that we want to predict the query token *"into"* by its left context, which is all the input to the left of the query token. With this training task one can by definition only use the left or the right context. By using both contexts together, the task would become trivial since you already know the next token.

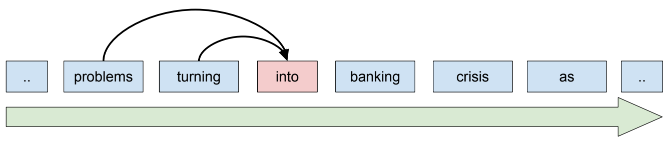


Figure 4: Left-to-right: Using the left context to predict the query token *into*.

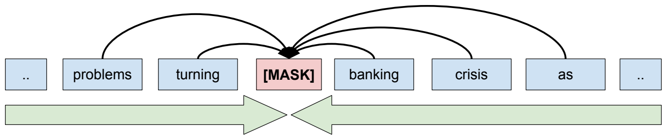


Figure 5: Bidirectional: Using both the left- and right context to predict a masked token.

To be able to condition on both the left and right context, BERT had to change the training task. Instead of predicting the next word, it tries to predict masked tokens as seen in Figure 5 by using both the left and right context. By predicting masked tokens, the BERT model is classified as an auto encoder. It learns to reconstruct the original data by restoring corrupted input, i.e. the masked tokens.

In a second training task, BERT performs *Next Sentence Prediction* (NSP). Here, the model receives two sentences as input, and has to predict if the first sentence is followed by the second one. This pre-training is necessary for all tasks which are based on the relationship between sentences. Typical examples of these tasks are *Question Answering*, *Natural Language Inference* or *Entity Matching*. It is important to understand that both training tasks are unsupervised tasks and do not require labels, but only large amounts of text data. Labeled data are only required for task-specific fine-tuning.

The ablation studies of the BERT paper demonstrate that using both the left and right context is the most important contribution of the paper. As a second contribution the BERT-team shows, that massive unsupervised pre-training on large data (BooksCorpus and Wikipedia) improves performance on a large number of tasks without the need for task-specific architectures. The BERT architecture further demonstrates to be very flexible as it allows simple fine-tuning on a range of downstream tasks such as *Question/Answering*, *Named Entity Recognition* or *Classification*.

4.2 XLNet and Transformer-XL

As seen in Section 4.1, BERT's most important contribution is its bidirectional representation. To achieve it, an auto encoder approach is necessary, with the training task to predict **[MASK]** tokens instead of predicting the next word. The XLNet [33] paper demonstrates the downsides of this approach and proposes a new architecture to solve it. According to XLNet there are two major downsides of the BERT approach:

- Predicting **[MASK]** tokens is what BERT learns during its pre-training phase. But those artificial symbols never occur in downstream tasks, which creates a pretrain-finetune discrepancy.
- BERT reconstructs all **[MASK]** tokens in parallel and independent of each other. This independence assumption is not justified as a simple example in Figure 6 demonstrates.

To overcome these flaws, XLNet returns to the more classical architecture of an autoregressive (AR) language model. In contrast to BERT's autoencoder approach, an AR model does not suffer from the downsides described above, as it does not introduce any artificial symbols and simply learns to predict the next token.

However, as we discussed in Section 4.1, an AR pre-training cannot use forward and backward contexts at the same time. To be able to capture a bidirectional context (as in BERT) and still

I went to [MASK] [MASK] and saw the [MASK] [MASK] [MASK].

I went to ***New York*** *and saw the* ***Empire State building.*** ✓

I went to ***San Francisco*** *and saw the* ***Golden Gate bridge.*** ✓

I went to ***San Francisco*** *and saw the* ***Empire State building.*** ✗

Figure 6: The independence assumption between the masked tokens is not justified.

have the advantages of an AR model, XLNet proposes a new *generalized autoregressive pre-training method*.

The core of this method is a new pre-train objective, which is called *permutation language modeling*. Instead of approaching a sequence of tokens only in a forward or backward manner, permutation language modeling takes into account all possible factorization orders of a sequence.

Assume that we have the following sentence "New York is a city". Further assume that we have already received the tokens "New" and "York". Next, we want to predict the token "is". Figure 7 shows how the model tries to predict the third token *"is"* with different factorization orders.

Let us discuss the three permutations in more detail:

- *Permutation 1* is the classical autoregressive left-to-right order. The model has access to the left side context (x_1, x_2) and tries to predict the query token x_3.
- In *Permutation 2* one can grasp the advantage of the permutation method: due to its order, the model now has access to the context x_4, x_2, x_1 to predict x_3. Be aware that in this sequence order, token x_5 is not accessible, as it appears to the right of token x_3.
- In *Permutation 3* the model has access to the tokens x_4 and x_5, as those appear to the left of token x_3 in the factorization order. By looking at this example it becomes intuitive that the token x_3 (that has to be predicted) has seen every other tokens in the sequence and will therefore, similar to an autoencoder model, capture the bidirectional context.

In addition to its core contribution, the *permutation language modeling*, XLNet includes two major improvements originally proposed in the Transformer-XL paper [5]. The Transformer-XL architecture is based on the original transformer [30], but with several improvements. The most important one, which is also implemented in XLNet, is the ability to learn dependencies beyond a fixed length without disrupting temporal coherence. This is possible due to a segment-level recurrence mechanism and a novel positional encoding scheme.

XLNet outperforms BERT on 20 tasks and achieves state-of-the-art results on 18 tasks including question answering, natural language inference, sentiment analysis, and document ranking [33].

4.3 RoBERTa

RoBERTa [17], a paper released at the end of July 2019, differs from other all the other papers as it does not come up with a new transformer approach, but new insights on BERT. The authors claim that BERT, without any major changes, can match or exceed every published model after it by just using the right hyperparameters and enough training data.

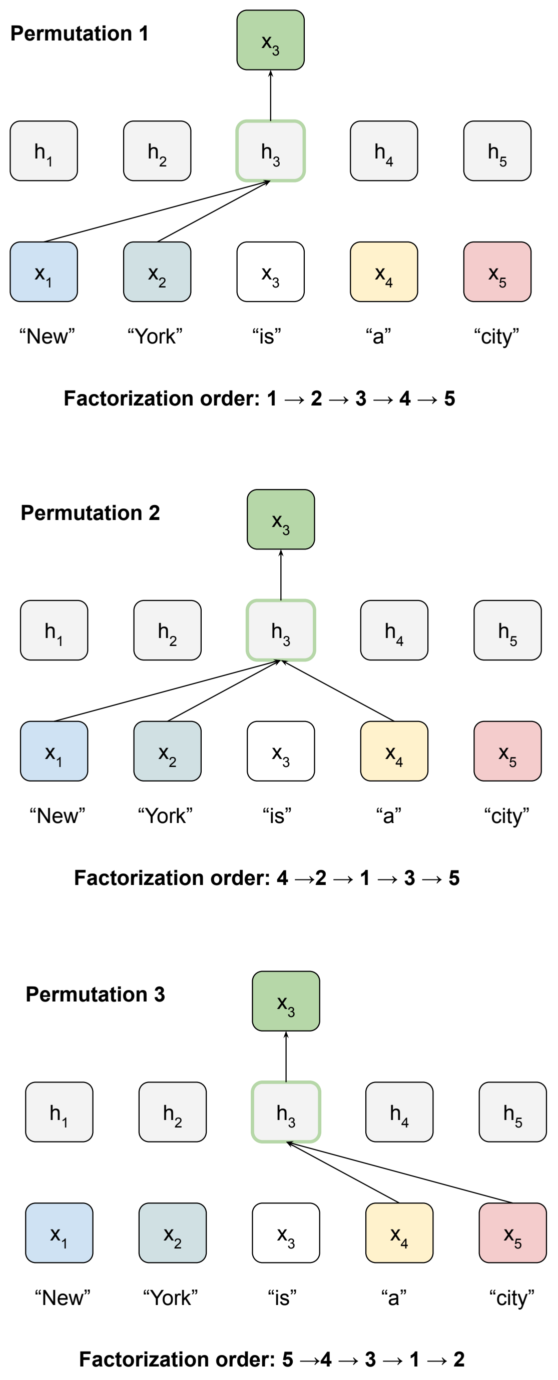

Permutation 1
x3
h1 h2 h3 h4 h5
x1 x2 x3 x4 x5
"New" "York" "is" "a" "city"
Factorization order: 1 → 2 → 3 → 4 → 5
Permutation 2
x3
h1 h2 h3 h4 h5
x1 x2 x3 x4 x5
"New" "York" "is" "a" "city"
Factorization order: 4 →2 → 1 → 3 → 5
Permutation 3
x3
h1 h2 h3 h4 h5
x1 x2 x3 x4 x5
"New" "York" "is" "a" "city"
Factorization order: 5 →4 → 3 → 1 → 2

Figure 7: Permutation language modeling: predicting x_3 given the same input sequence but with different factorization orders.

The authors find that BERT was significantly undertrained and propose the following modifications for maximal performance:

- **More training data.** The original BERT was trained on the BookCorpus and English Wikipedia, covering around 16 GB of text. Many of the subsequent papers used much larger data sets, some of them publicly available, others not. RoBERTa uses five English-language corpora with a total size of over 160 GB of text.

- **Longer training.** RoBERTa evaluates three training durations with 100K, 300K and 500K steps. They show that maximal training duration, together with the additional data, results in best performance.
- **Larger batch size.** While the original mini-batch size of BERT is 256, RoBERTa further evaluates batch sizes of 2,000 and 8,000 samples per mini-batch. The experiments on several downstream tasks indicate that a batch size of 2,000 is the best choice, given the learning rate is increased appropriately.
- **Get rid of the next sentence prediction (NSP) objective.** In contrast to [6], the authors of RoBERTa show that by removing the NSP loss, they achieve slightly better downstream task performance. They also show though that it is crucial to use the full attention span (the model input size, max. 512 tokens) during pre-training in order for BERT to learn long-range dependencies.
- **Change the masking pattern of training data dynamically.** While BERT uses a relatively static masking procedure applied during preprocessing, RoBERTa suggests dynamically masking of each sample during training before feeding it to the model.

With all these modifications, RoBERTa's performance is evaluated on the GLUE, SQuAD and RACE benchmarks. RoBERTa achieves state-of-the-art results on all three challenges, with slightly better results than XLNet (its closest competitor) on most challenges and clearly better than the original BERT [17].

4.4 DistilBERT - Smaller, Faster, Cheaper

While all three models discussed so far focused on improving the state-of-the-art, the authors of DistilBERT [23] pursue another objective. Instead of better results, DistilBERT aims for a smaller and faster models which can be used in real-world projects. Indeed, the size of recent transformer models is impressive:

- A BERT-Large model uses 340M parameters.
- RoBERTa works with 355M parameters while using 160 GB data for pre-training.
- NVIDIA trained a transformer language model called *MegatronLM*[26], using 8300M parameters.

The authors of DistilBERT see the challenge of modern transformer models mainly at inference time, when a model should be small and fast, so it can work on mobile devices or non-GPU servers.

There are several approaches to reduce model size while approximating performance, namely *quantization*, *weights pruning* and *distillation*. The authors of DistilBERT focus on knowledge distillation.

4.4.1 *Knowledge Distillation.*

The core idea of knowledge distillation is to train a smaller model (the *student*) which learns from the original model (the *teacher*). The method has been generalized by [10] and is sometimes also referred to as *teacher-student learning*.

The challenge of knowledge distillation is to learn the so-called *dark knowledge* of a model. Let us assume the example in Figure 8, where a language model predicts the last token in a sentence. While one or two predictions usually have a high probability, there will almost always be a long tail of tokens with low probability. This knowledge, even though the probabilities might be low, is crucial for a model to generalize. The challenge of knowledge distillation is therefore not only to learn the one prediction with high probability, but also the near-zero probabilities on other classes. In terms of distillation loss function, this near-zero probabilities are called *soft targets*.

Input:

'I', 'think', 'this', 'is', 'the', 'beginning', 'of', 'a', 'beautiful', **[MASK]**

Predicted token (top 16):

#1 token: 'day'	**p:** 0.213		**#9** *token:* 'summer'	**p:** 0.016
#2 *token:* 'life'	**p:** 0.183		**#10** *token:* 'adventure'	**p:** 0.013
#3 *token:* 'future'	**p:** 0.062		**#11** *token:* 'dream'	**p:** 0.012
#4 *token:* 'story'	**p:** 0.058		**#12** *token:* 'moment'	**p:** 0.012
#5 *token:* 'world'	**p:** 0.049		**#13** *token:* 'night'	**p:** 0.011
#6 *token:* 'era'	**p:** 0.045		**#14** *token:* 'beginning'	**p:** 0.010
#7 *token:* 'time'	**p:** 0.032		**#15** *token:* 'season'	**p:** 0.009
#8 *token:* 'year'	**p:** 0.017		**#16** *token:* 'journey'	**p:** 0.006

Figure 8: Predicting a masked token with BERT: The first two tokens show a high probability, followed by a long tail of near-zero possibilities.

4.4.2 *Loss Objective.*

To motivate a model to learn both high and near-zero probabilities, DistillBERT suggests a composite loss function as follows:

- **Distillation Loss** The student learns the probabilities of the teacher (*soft targets*) with $L = \sum_i t_i * \log(s_i)$ where t_i is a probability estimated by the teacher and s_i the probability estimated by the student. To control the smoothness of the output distribution, a further technique from [10] called *softmax-temperature* is used.
- **Masked Language Modeling (MLM) Loss [6]** As the teacher model used in DistilBERT is the original BERT model, the second loss function is the original MLM, with the goal to predict the masked tokens.
- **Cosine Embedding Loss** A rather technical loss to align the directions of the student and teacher hidden states vectors.

4.4.3 *Student Model Architecture.*

While using the original BERT architecture as teacher model, the student model architecture is a purged version of the BERT model. *Token-type embeddings* and the *pooler* have been removed and the number of layers have been reduced by factor 2. This reduces the overall size of the model by 40% [23].

Note that the distillation process happens on the general-purpose model, before applying fine-tuning on downstream tasks. This in contrast to task-specific distillation, which distills the model after already being fine-tuned on a specific task.

4.4.4 *Performance.*

To compare DistilBERTs performance, it has been fine-tuned on two of the usual benchmarks (GLUE, SQuAD). DistilBERT retains 97% of the original BERT model while reducing the model size by 40% and being 60% faster [23].

5 EVALUATION

In this section, we present and discuss the results of the experiments conducted with the transformer architectures described in Section 4. In particular, we will address the following research questions with respect to entity matching:

- How well do transformer architectures perform compared to traditional, non-deep-learning ML approaches?
- How well do transformer architectures perform compared to recent deep learning approaches?
- Which transformer architecture performs best on the task of entity matching?
- Transformer architectures are complex deep neural networks. How much training is necessary to fine-tune them on the entity matching task? Is the amount of training data in a traditional EM dataset sufficient to fine-tune a transformer?

5.1 Datasets

The authors of *DeepMatching*[20] demonstrated that modern deep learning methods do not perform better in entity matching tasks on structured data than traditional approaches. Traditional ML approaches such as e.g. *Magellan*[14] perform even slightly better and training time is magnitudes shorter than for deep learning methods. In addition, the scores of *Magellan* on structured data are high - it performs on those 11 datasets with an average F1-score of 86% - without the rather textual dataset *Amazon-Google* the F1-score is even 90%. The authors of DeepER[7], a second approach using deep learning for entity matching, confirm these results with F1-scores between 88% and 100% on these structured datasets.

We therefore focus our experiments on challenging datasets where other approaches showed relatively low performance. To evaluate our transformer approaches we use both *textual* and so-called *dirty* datasets provided by the Magellan team [20]. We use all publicly available datasets except the textual dataset *Company*, as this contains text blobs with lengths between 2000-3000 tokens, which exceeds the maximal attention spans of 512 tokens for transformer architectures. There have been recent approaches to extend this attention span to 8000+ tokens by [28]. However, we leave these challenges for future work.

For our entity matching experiments, we use the following five datasets to evaluate the transformer architectures (see Table 3): *(1) Abt-Buy*: Product data from Abt.com and Buy.com. The core attribute is *description*, which is a long text blob describing the product. We use no informative attribute (e.g. the *title*), but only the noisy *description* attribute, similar to [20]. *(2) iTunes-Amazon (Dirty):* music data from iTunes and Amazon. The data has been modified to simulate dirty data as done by [20]. They suggest for each attribute other than "title" to randomly move each value to the attribute "title" in the same tuple with a probability of $p = 0.5$. *(3) DBLP-ACM (Dirty):* Bibliographic data from DBLP and ACM. The data has been modified to simulate dirty data. *(4) DBLP-Scholar (Dirty):* Bibliographic data from DBLP and Google Scholar. The data has been modified to simulate dirty data. *(5) Walmart-Amazon (Dirty):* Product data from Walmart and Amazon. The data has been modified to simulate dirty data.

To evaluate our transformer architectures, we split all five datasets into into three parts with a ratio of 3:1:1. We use the 60% split of the data for training, and the two 20% splits for validation and test. All reported numbers in this paper show results on the test split.

Table 3: Datasets used in our experiments.

Dataset	Domain	Size	# Matches	# Attr.
Abt-Buy	Products	9,575	1,028	3
iTunes-Amazon	Music	539	132	8
Walmart-Amazon	Products	10,242	962	5
DBLP-ACM	Citation	12,363	2,220	4
DBLP-Scholar	Citation	28,707	5,347	4

5.2 Setup & Methods

In this section we will describe the hardware we used for our experiments as well as the implementation of the transformers along with the hyperparameters and pre-trained models.

5.2.1 Hardware. All experiments were executed on a single Nvidia TITAN Xp GPU (12GB Memory) with Intel(R) Xeon(R) CPU E5-2650 v4 (4 cores) and 8GB memory. The experiments are implemented using PyTorch and the transformer implementations are based on [32].

5.2.2 Methods. Figure 9 shows how an entity pair consisting of `Entity A` and `Entity B` is fed into a transformer architecture. This approach is used at both training and inference time. The data feeding approach looks similar in all four transformer architectures, while having minor differences in the use of separator tokens, position embeddings and the classification representation (`CLS`) in the last layer. The detailed description of the following section refers to BERT/RoBERTa and DistilBERT, while being very similar in XLNet.

We will now describe how we use the transformer for entity matching in detail. `Entity A` and `Entity B` contain a single text blob each. In case of textual data (*Abt-Buy*), the text blob consists of the single attribute *description.* For "dirty" datasets, all attributes of a data instance are concatenated, for instance `[name + brand + description + price]`. Next, the single text blob is tokenized. For each token, the corresponding embedding is looked up (for details on tokenizing and embedding see Section 5.2.3).

Each token embedding is then summed up with a positional embedding and a segment embedding. The segment embedding is used to distinguish between tokens of `Entity A` and `Entity B`. The sum of all three embeddings is then fed into the transformer. The maximum length of the input ($A_1 - A_N$ plus $B_1 - B_M$ plus the two artificial tokens *classification* and *separator*) has been empirically defined based on the longest data rows in the training data. It lies between 128 and 265 tokens, depending on the data set.

In one pass, a pair of two entities (all tokens of `Entity A` and `Entity B`) is processed by the transformer model. This leads to major performance improvements compared to classical deep learning models based on RNNs[20], where each token is depending on the tokens appearing earlier in the sequence. The output of a pass is then represented in the hidden state at position 0, the purple `CLS` symbol in Figure 9. This hidden state (a vector of size 768) is then passed into a single classification layer (*Classifier* in Figure 9).

Keep in mind that transformer models are universal language models, capable of performing different downstream tasks (e.g. classification, seq2seq, question/answering, NER, etc.). Hence, to feed the classification output of the transformer model via a

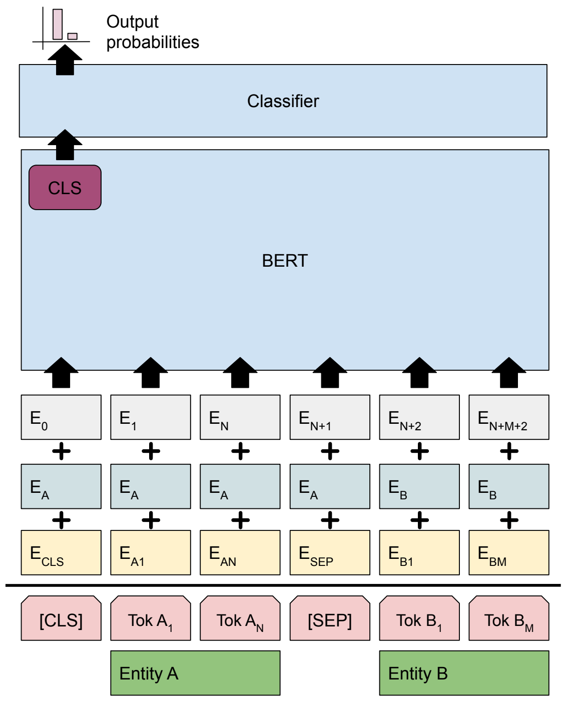


Figure 9: Transformer feeding approach. Entity A and Entity B are tokenized (e.g. `Tok A`$_1$`,red`). The embedding of each token (`E`$_{A1}$`,yellow`) is fed into the transformer together with positional- and segment embeddings (`E`$_0$`,gray/E`$_A$`,blue`). Special symbols are used to feed the output into the classification layer (`[CLS]`) and to separate the two entities (`[SEP]`)

designated representation symbol (CLS) into a specific classification layer, is an architectural decision of [6] to keep the model flexible. When using the model for other tasks, e.g. for NER, the ouput layer looks differently.

The classification layer is - in contrast to the rest of the model - not pre-trained and contains a fully connected layer with 768 neurons plus two output neurons. These two output neurons finally represent the two classes of an entity matching problem: *"Entity A and B match"* or *"Entity A and B do not match"*.

We use Adam [12] for optimization in combination with a linear learning rate. The choice of optimizer, learning rate and other hyperparameters are based on best practices for similar classification tasks, e.g. the GLUE-QQP benchmark [6, 17, 23, 31, 33].

With the implementation described above, we evaluate all four transformer architectures described in Section 4: BERT, XLNet, RoBERTa and DistilBERT. To reproduce the experiments, we provide source code and hyperparameters on GitHub[1].

5.2.3 Tokenization and Embeddings. Before feeding a pair of `Entity A` and `Entity B` into the transformer, tokenizing the input and a lookup of embeddings is necessary. Transformers like almost all modern language models work with embeddings as input values. Using embeddings increases performance over simple one-hot encoded vectors massively, as demonstrated by the influential *word2vec* paper [19].

We use the following tokenization/embedding techniques:

[1]https://github.com/brunnurs/entity-matching-transformer.

Table 4: Pre-trained models used in our experiments.

Transformer	Details
BERT	12-layer, 768-hidden, 12-heads, 110M parameters. BERT-base model. Trained on lower-cased English text.
XLNet	12-layer, 768-hidden, 12-heads, 110M parameters. XLNet English model
RoBERTa	12-layer, 768-hidden, 12-heads, 125M parameters. RoBERTa is using the BERT-base architecture.
DistilBERT	6-layer, 768-hidden, 12-heads, 66M parameters. The DistilBERT model is distilled from the BERT-base model.

- **BERT/DistilBERT:** We first split the whole input textblob into single tokens by simple white space- and punctuation splitting rules. In a second step, we create Wordpiece embeddings by applying the original algorithm from[24].
- **RoBERTa:** We split the whole input into tokens by using white spaces, punctuation and special abbreviations (`'s|'t|'re|'ve|'m|'ll|'d`). We then apply Byte-level Byte-Pair-Encoding [25].
- **XLNet:** In contrast to the other approaches, we do not pre-tokenize the input into word sequences, but directly input the raw text blob into a SentecePiece[15] subword tokenizer.

5.2.4 Pre-Trained Models. Due to their model sizes, all four transformer architectures require very resource-intensive pre-training on large amounts of data. As an example, the BERT$_{LARGE}$ model was trained on 64 TPU chips for 4 days while RoBERTa uses 1024 V100 GPUs for approximately one day [6, 17]. The transformers performance is therefore heavily dependent on the pre-trained model. In Table 4 we list all pre-trained models used in the experiments. We use the pre-trained models provided by the original papers [32] and gathered by Hugging Face[2]. Note that due to hardware constrains, we always used the smallest available pre-trained model. We expect larger pre-trained models to perform as good or even better.

5.3 Comparison with Classical Models

Table 5: F1-scores of the best performing Transformer model in comparison with Magellan (MG) and DeepMatcher (DeepM).

Dataset	MG	DeepM	T$_{BEST}$	ΔF_1
Abt-Buy	33.0	55.0	90.9	35.9
iTunes-Amazon$_{Dirty}$	46.8	79.4	94.2	14.8
Walmart-Amazon$_{Dirty}$	37.4	53.8	85.5	31.7
DBLP-ACM$_{Dirty}$	91.9	98.1	98.9	0.8
DBLP-Scholar$_{Dirty}$	82.5	93.8	95.6	1.8

Table 5 shows the performance of transformer models in comparison with Magellan and DeepMatcher. The results are measured, similar to other papers in the field [7, 20], by an F1 score where recall is the ratio of true matches predicted vs. all true

[2]https://github.com/huggingface/transformers

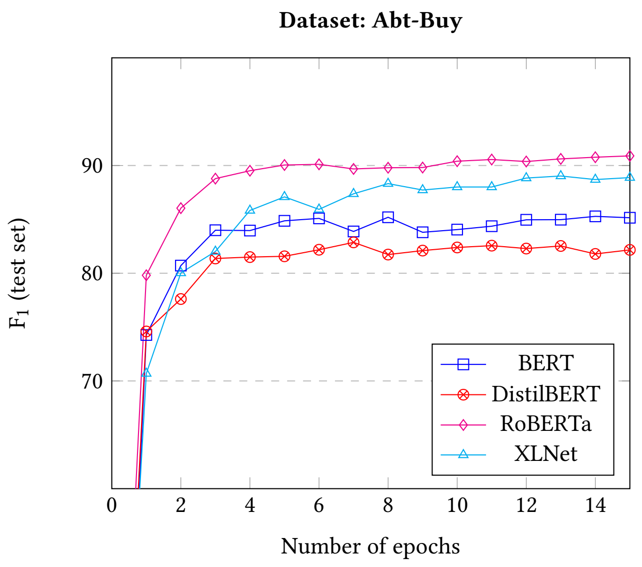


Figure 10: Performance of transformers on the Abt-Buy dataset, averaged over five runs. We only visualize F_1 scores in range 60 - 100% to emphasize the difference.

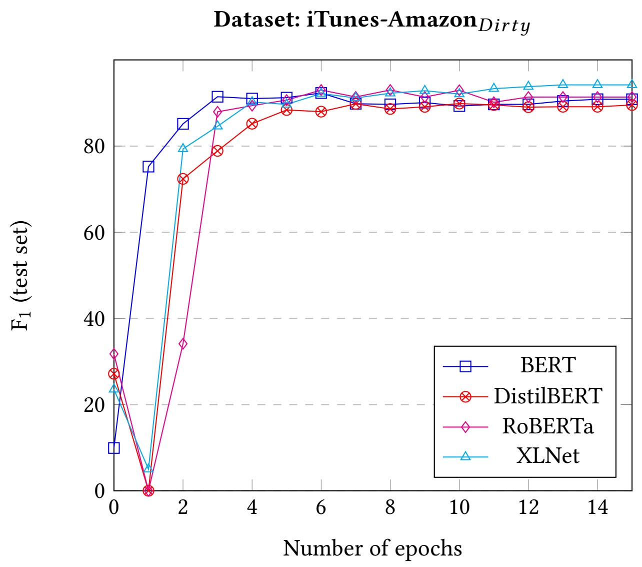


Figure 11: Performance of transformers on the iTunes-Amazon$_{Dirty}$ dataset, averaged over five runs. The effect of little training data is visible at epoch 1.

matches. For each dataset we report the best performing of the four DeepMatcher DL models, the result of Magellan and the best performing transformer. The transformer result is an average over five runs. **We see that transformer architectures outperforms both DeepMatcher and Magellan by a large margin on the challenging datasets *Abt-Buy* (35.9%), *iTunes-Amazon* (14.8%) and *Walmart-Amazon* (31.7%).**

We are also interested in how transformer architectures perform on datasets where DeepMatcher and Magellan perform very well, like on *DBLP-ACM* (DeepMatcher: 98.1%, Magellan: 91.9%) and *DBLP-Scholar* (DeepMatcher: 93.8%, Magellan: 82.5%). As expected, the transformers perform better than existing methods also on these datasets with a ΔF_1 of 0.8% and 1.8%, respectively. In comparison to the challenging datasets though, the improvements are relatively small. Since the results of DeepMatcher on these datasets are already very high (F1-score: 98.1% and 93.8%) we assume that better language models only slightly improve the performance on those tasks.

5.4 Transformers Architectures Head-to-Head

The performance of transformer architectures has been compared many times in recent papers [6, 17, 33], usually on large NLP benchmarks (e.g. GLUE, SQuAD [22, 31]). However, we are not aware of any related work where transformers were used for entity matching.

Figures 10 - 14 show the performance of the different transformer architectures on the datasets defined in Section 5.1. The results have been averaged over five runs. The main findings of our systematic, experimental evaluation are as follows:

- **All transformer architectures perform remarkably well on the task of entity matching.** Even the least performing transformer, *DistilBERT*, performs clearly better than classical approaches (see Section 5.3).
- After only **one epoch of training**, most experiments range within a 5% interval of their peak performance. After 3-5 epochs, almost all experiments converge to their peak performance. The effort to fine-tune a transformer on an entity matching task is therefore manageable, especially considering the training time per epoch in Section 5.5.
- Overall, *RoBERTa* performs best, also reaching high results already after only a few epochs. We conclude that the massive pre-training effort on a huge amount of data (compared to the other three approaches) gives RoBERTa a slight advantage in the task of entity matching.

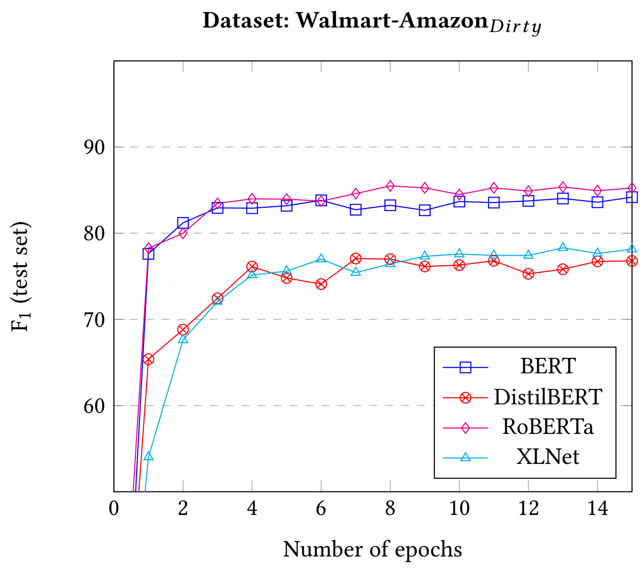


Figure 12: Performance of transformers on the Walmart-Amazon$_{Dirty}$ dataset, averaged over five runs. We only visualize F_1 scores in range 50 - 100% to emphasize the difference.

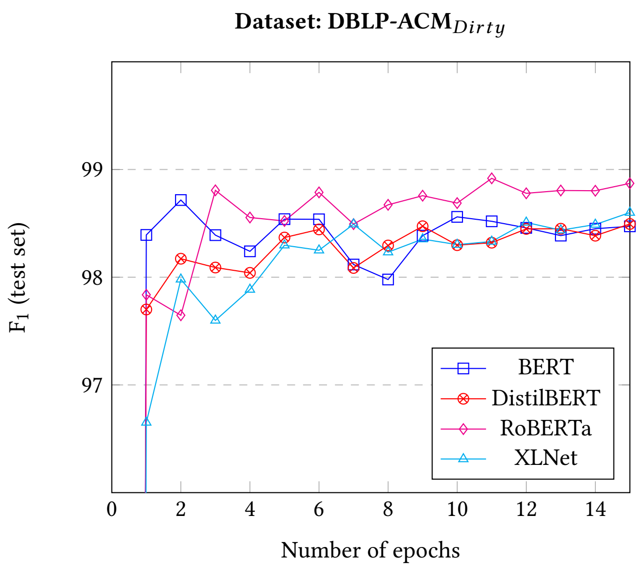


Figure 13: Performance of transformers on the DBLP-ACM$_{Dirty}$ dataset, averaged over five runs. We only visualize F_1 scores in range 96 - 100% to emphasize the difference.

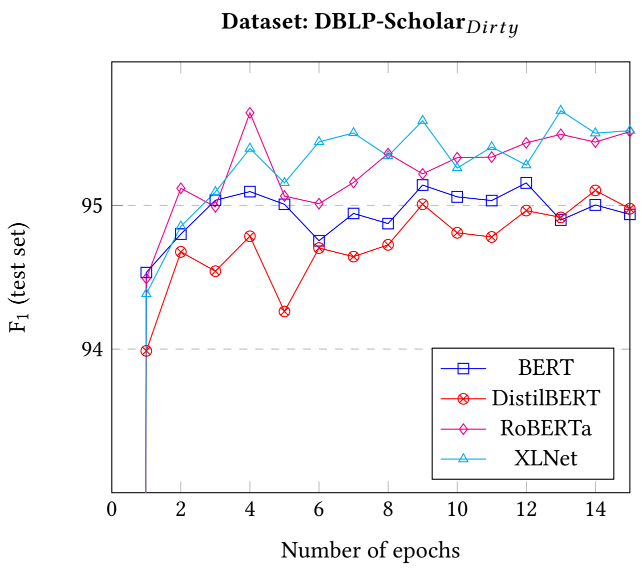


Figure 14: Performance of transformers on the DBLP-Scholar$_{Dirty}$ dataset, averaged over five runs. We only visualize F_1 scores in range 93 - 96% to emphasize the difference.

- *DistilBERT* performs worst, averaged on all tasks. This is explainable due to the comparably small model of DistilBERT. Still though the results of DistilBERT are close to its "big brothers" and higher than classical approaches (see Section 5.3).
- *XLNET* requires, on average, longer training times to reach peak results. We conclude the *permutation language modeling* eventually results in high performance (XLNet performs often almost equally well as RoBERTa), but due to the large amount of permutations, it requires more training time.

Table 6: Training time per epoch on each data set.

Dataset	BERT	XLNet	RoB.a	D.BERT
Abt-Buy	2m 42s	6m 15s	2m 43s	1m 22s
iTunes-Amazon$_{Dirty}$	7s	12s	7s	3.5s
Walmart-Amazon$_{Dirty}$	1m 41s	2m 29s	1m 41s	52s
DBLP-ACM$_{Dirty}$	2m 24s	4m 9s	2m 24s	1m 13s
DBLP-Scholar$_{Dirty}$	4m 5s	5m 57s	4m 13s	2m 6s

- The iTunes-Amazon$_{Dirty}$ dataset is extremely small containing only 132 matching records . As we see in Figure 11, this is influencing the training process. After epoch 1 almost all experiments have an F1-score of 0%, in comparison to the other datasets where after epoch 1 the results are already close to peak performance. We see, though, that even with this small amount of training data, the fine-tuning converges after 4-6 epochs of training.

5.5 Training Time

Transformer models are known to be resource-intensive architectures with millions of parameters. For instance, DistilBERT$_{SMALL}$ consists of 66 million parameters and RoBERTa$_{LARGE}$ has even 355 million parameters. As we have seen in Section 4, training a transformer is split in two tasks: *(1)* unsupervised *pre-training* on large amounts of unspecific data and *(2)* supervised *fine-tuning* on downstream tasks (e.g. entity matching) with task-specific data. While pre-training is doing a major part of finding these parameters (see Section 5.2.4), it is still a large model to fine-tune on a specific downstream task like entity matching.

Table 6 shows the training time for fine-tuning each transformer on the given datasets. The times reported are per epoch on the training set (which is roughly 60% of the total dataset).

As fine-tuning usually converges to maximum performance after 1-3 epochs (see Section 5.4), total training takes between **10s** (DistilBERT on iTunes-Amazon$_{Dirty}$, 3 epochs) and **11m 55s** (XLNet on DBLP-Scholar$_{Dirty}$, 2 epochs). It is important to note that the experiments ran on low budget hardware (see Section 5.2.1) without any parallelization. With state-of-the-art-hardware, these times can be reduced by an order of magnitude.

If we compare these training times to the reported results of DeepMatcher [20], we can draw two important conclusions:

(1) Compared to conventional, non-deep learning EM models as e.g. *Magellan*, fine-tuning a transformer is still relatively slow.
(2) Compared to training a deep learning model as proposed in DeepMatcher, fine-tuning a transformer is fast. DeepMatcher reports training times from **10min** to **11h**, depending on the dataset and solution. This is 1-2 magnitudes slower than fine-tuning a transformer as we propose, especially as DeepMatcher uses faster hardware than we do.

The reason fine-tuning a transformer is faster than training a comparably light-weighted deep learning model[20] is clearly the resource-intensive pre-training, which improves convergence on the downstream task (entity matching) enormously.

6 CONCLUSIONS

In this work, we show for the first time the strong impact of transformer architectures and pre-training for entity matching on datasets with large textual- or "dirty" data. We show that all transformers described can be used for EM out of the box, without the need for a task-specific architecture. Our experiments on five datasets show a significant improvement of F1-scores of up to 35.9% of transformers on challenging datasets compared to state-of-the-art approaches. In addition, we demonstrate that on relatively small and clean datasets, transformers still perform slightly better than earlier deep learning approaches.

We demonstrate that fine tuning a transformer on an EM task takes relatively little time and requires no particularly capable hardware, which might be contrary to expectations due to the large size of transformer models. Regarding the question on which of the four transformer models performs best on the EM task, experiments show that all of them deliver relatively similar results. In comparison with average results, RoBERTa shows slightly better and DistilBERT slightly worse performance, which is in line with the theoretical background of these approaches.

We consider transformer approaches as a valuable technology for entity matching. Using standard transformer architectures instead of designing EM-specific architecture does not only benefit from simplicity, but profits also directly from further advances in the NLP field of pre-training and transformers.

REFERENCES

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. http://arxiv.org/abs/1409.0473 cite arxiv:1409.0473Comment: Accepted at ICLR 2015 as oral presentation.

[2] U. Brunner and K. Stockinger. 2019. Entity Matching on Unstructured Data: An Active Learning Approach. In *2019 6th Swiss Conference on Data Science (SDS)*. 97–102. https://doi.org/10.1109/SDS.2019.00006

[3] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014). arXiv:1406.1078 http://arxiv.org/abs/1406.1078

[4] Peter Christen. 2012. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer Publishing Company, Incorporated.

[5] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. *CoRR* abs/1901.02860 (2019). arXiv:1901.02860 http://arxiv.org/abs/1901.02860

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[7] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq R. Joty, Mourad Ouzzani, and Nan Tang. 2017. DeepER - Deep Entity Resolution. *CoRR* abs/1710.00597 (2017). arXiv:1710.00597 http://arxiv.org/abs/1710.00597

[8] Lise Getoor and Ashwin Machanavajjhala. 2012. Entity Resolution: Theory, Practice &#38; Open Challenges. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 2018–2019. https://doi.org/10.14778/2367502.2367564

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[10] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the Knowledge in a Neural Network. *ArXiv* abs/1503.02531 (2015).

[11] Matthew A. Jaro. 1989. Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida. *J. Amer. Statist. Assoc.* 84, 406 (1989), 414–420. https://doi.org/10.1080/01621459.1989.10478785

[12] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. http://arxiv.org/abs/1412.6980 cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[13] Hanna Koepcke, Andreas Thor, and Erhard Rahm. 2010. Evaluation of Entity Resolution Approaches on Real-world Match Problems. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 484–493. https://doi.org/10.14778/1920841.1920904

[14] Pradap Konda, Jeff Naughton, Shishir Prasad, Ganesh Krishnan, Rohit Deep, Vijay Raghavendra, Sanjib Das, Paul C., AnHai Doan, Adel Ardalan, Jeffrey Ballard, Han Li, Fatemah Panahi, and Haojun Zhang. 2016. Magellan: Toward building entity matching management systems. *Proceedings of the VLDB Endowment* 9 (08 2016), 1197–1208. https://doi.org/10.14778/2994509.2994535

[15] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. *CoRR* abs/1808.06226 (2018). arXiv:1808.06226 http://arxiv.org/abs/1808.06226

[16] Guillaume Lample and Alexis Conneau. 2019. Cross-lingual Language Model Pretraining. *CoRR* abs/1901.07291 (2019). arXiv:1901.07291 http://arxiv.org/abs/1901.07291

[17] 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). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[18] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. *CoRR* abs/1508.04025 (2015). arXiv:1508.04025 http://arxiv.org/abs/1508.04025

[19] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3111–3119.

[20] 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 *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 19–34. https://doi.org/10.1145/3183713.3196926

[21] Salimans Radford, Narasimhan and Sutskever. [n.d.]. Improving Language Understanding by Generative Pre-Training. *Technical report, OpenAI* ([n. d.]). https://www.cs.ubc.ca/~amuham01/LING530/papers/radford2018improving.pdf

[22] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100, 000+ Questions for Machine Comprehension of Text. *CoRR* abs/1606.05250 (2016). arXiv:1606.05250 http://arxiv.org/abs/1606.05250

[23] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. [n.d.]. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. ([n. d.]). https://arxiv.org/abs/1910.01108

[24] M. Schuster and K. Nakajima. 2012. Japanese and Korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 5149–5152. https://doi.org/10.1109/ICASSP.2012.6289079

[25] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1715–1725. https://doi.org/10.18653/v1/P16-1162

[26] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. [n.d.]. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. ([n. d.]). https://arxiv.org/abs/1909.08053

[27] Michael Stonebraker, Daniel Bruckner, Ihab F. Ilyas, George Beskales, Mitch Cherniack, Stanley B. Zdonik, Alexander Pagan, and Shan Xu. 2013. Data Curation at Scale: The Data Tamer System. In *CIDR*.

[28] Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. 2019. Adaptive Attention Span in Transformers. *CoRR* abs/1905.07799 (2019). arXiv:1905.07799 http://arxiv.org/abs/1905.07799

[29] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3104–3112. http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 5998–6008. http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

[31] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. *CoRR* abs/1804.07461 (2018). arXiv:1804.07461 http://arxiv.org/abs/1804.07461

[32] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, and Jamie Brew. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *ArXiv* abs/1910.03771 (2019).

[33] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. *CoRR* abs/1906.08237 (2019). arXiv:1906.08237 http://arxiv.org/abs/1906.08237

[34] Chen Zhao and Yeye He. 2019. Auto-EM: End-to-end Fuzzy Entity-Matching Using Pre-trained Deep Models and Transfer Learning. In *The World Wide Web Conference (WWW '19)*. ACM, New York, NY, USA, 2413–2424. https://doi.org/10.1145/3308558.3313578

Gallery: A Machine Learning Model Management System at Uber

Chong Sun
chong@uber.com
Uber Technologies Inc.
San Francisco, California

Nader Azari
nazari@uber.com
Uber Technologies Inc.
San Francisco, California

Chintan Turakhia
chintan@uber.com
Uber Technologies Inc.
San Francisco, California

ABSTRACT

Machine learning is critical to the success of many products across application domains. At Uber, we have a variety of machine learning applications including matching, pricing, recommendation, and personalization. As a result, we have a large number of machine learning models to manage in production. Generally, building machine learning models is an iterative process and machine learning models span across a set of stages of a lifecycle. In this paper, we describe Gallery, a machine learning model lifecycle management system to save and serve models and metrics and automatically orchestrate the flow of models across different stages in the lifecycle. We then use the Uber Marketplace Forecasting and Simulation platforms as examples to show how Uber uses Gallery in production and the benefits we get by using Gallery.

1 INTRODUCTION

Machine learning is critical to the success of many products across application domains. Companies employ machine learning for recommendation, targeting, and personalization. Uber uses machine learning across product features including matching, pricing, personalization, ETA estimation, and Uber Eats recommendations. Recently, there have been various systems and frameworks [1, 12, 22, 26] designed and built to make machine learning easy-to-use and scalable in production systems. However, as the interaction of models with systems have become more complex, a growing technology need exists to manage machine learning models through their lifecycle to accelerate the process of getting a model from exploration to production and improve the model iteration velocity.

Building machine learning models is an iterative process [7]. Given a problem to solve, the common lifecycle of a model, as shown in Figure 1, starts with model exploration, during which we design and explore multiple models. When we find a model that beats a benchmark, we build the model into a production system. Getting a model into production starts at the model training, where we generate model instances. We refer to the trained models as the instances of a model. We evaluate the performance of trained model instances and deploy instances when the performance is above certain thresholds. Otherwise, we continue to improve the models. When models are deployed in production, monitoring performance is critical. If a performance degradation is detected or we have a new model, we will need to re-train the appropriate model, deprecate the old model instances, and deploy the new model instances.

Managing a handful of models is feasible for a production system. However, operational scale quickly becomes untenable

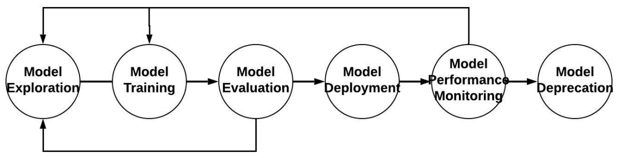


Figure 1: Machine Learning Model Lifecycle

with multiple machine learning problems, and more so when each problem has hundreds of model instances to manage. For example, when doing Marketplace-level forecasting at Uber, we forecast supply, demand, and other quantities in real time for hundreds of cities across the globe. We shard the problem spatially by city, training a model instance for each city-quantity combination because Uber is operating in many cities across the world, and different cities may pose different geospatial characteristics. Besides, the Uber business might be at different growth stages for different cities.

Though there are variances across applications and projects, many interesting questions about how to manage a large number of machine learning models in production are common. In this section, we list a sample of the questions which are raised between use cases: Where do we save and serve the models generated during model exploration or trained in production? How do we efficiently search for models and their experiment results? How can we confidently deploy a large-scale number of models and avoid regressions? How and when do we trigger model retraining due to model performance deterioration? In a complex system like the Uber Marketplace, the result of applying one model could be the input to another model. How can we manage the dependencies between multiple models?

How to address the above model management questions often depends on the experiences of machine learning engineers who work on these problems. Even within one company like Uber, different machine learning applications may use different approaches to solve these problems. For example, prior to Gallery there were over seven different storage solutions (e.g., MySQL, HDFS, Cassandra and Git repo) engineers used to save machine learning models. As a result, similar functionalities to manage machine learning models are scattered across a variety of production systems. This results in increased overhead to build and maintain individual systems, and causes a loss of visibility into the machine learning models across a production system. With the increasing number of machine learning solutions being built to solve business problems at Uber, we built Gallery, a model lifecycle management system to systematically and uniformly address these common questions to improve machine learning model velocity and productivity across Uber.

Gallery was started as a system to solve Uber Marketplace Forecasting model management problems and was later integrated as part of Michelangelo [6], Uber's ML Platform. It is a system designed to manage machine learning models by providing functions for model saving, searching, serving, performance monitoring, and orchestration across different stages of the model development lifecycle.

Overall, the major contributions in this paper are as follows:

- A comprehensive analysis of the problems we addressed at Uber in order to manage machine learning models in a large-scale, distributed, microservices-based system.
- We describe Gallery, a model lifecycle management system used in production at Uber.
- We use Uber Marketplace Forecasting and Simulation case studies to demonstrate how we utilize Gallery to manage machine learning models and the benefits we have gained with Gallery.

2 THE MODEL MANAGEMENT PROBLEM

We first share some machine learning model management context at Uber before we define the problem. At Uber, we employ numerous machine learning applications, such as request dispatching, pricing, user growth, and recommendations. Overall, the Uber platform is microservice-based. Application teams build their own services and have different requirements for cadence and latency, implying different patterns of running the models. For example, long-term forecasting predicts hourly trips for a city for weeks in the future, while real-time forecasting predicts sub-hour demand. As a result, different applications might use different languages, modeling techniques, and frameworks for building and serving models.

In addition to the variety of applications and models, Uber is operating in markets across the world that have unique conditions in terms of population, city layout, climate, and population density. As a result, it is common to see machine learning models trained separately for different markets. We also need to frequently retrain the models when we detect model performance degradation due to the changing market conditions, and we need to independently trigger the retraining of the models for a city. Often it is not efficient to blindly re-train the models for all the cities, e.g., the training data for real-time demand forecasts can easily go up as much as terabytes for one city. Instead, we would like to retrain the models periodically if performance evaluation shows the need.

To solve the model management issue across heterogenous use cases, a system to manage a variety of machine learning models across different frameworks, languages, and usage patterns, from model exploration to models deployed in production, is necessary. To be clear, we refer to a machine learning model as an abstract data transformation which we can use to solve a particular problem or business use case. A model contains the specification of the input, output, and transformation, e.g., linear regression or random forest classification and all the corresponding hyperparameters. A model instance consists of a set of coefficients that is a learned representation of a given model on a particular training data set. The terms "model" and "model instance" are commonly used interchangeably when there is no ambiguity. Accordingly, we define the model management problem as: how to consistently and scalably manage a large number of complex models and model instances across stages of a model lifecycle.

3 THE GALLERY SYSTEM

In the section, we describe Gallery, a model lifecycle management system, built at Uber to solve the aforementioned model management problems. We first introduce the principles that guide our design. Then, we discuss the overall system architecture, followed by the description of each major system component.

3.1 Design Principles

Immutable. Any machine learning model and model instance generated and managed in our system is immutable. Any update of a model or model instance will result in a new version in production. This is critical to guarantee no unexpected behavior in production, and ensures that all decisions can be traced back to a specific model version. This builds the foundation for model performance observability and debuggability.

Model Neutral. Each model is treated as a black box and the model management system does not interpret the models. In this way, we can have one system to provide management for the varying models built for each application, e.g., a deep learning model using TensorFlow or PyTorch, or linear regression models using scikit-learn. Users are not blocked from leveraging the model management system because of their modeling technology choices.

Framework Agnostic. Any framework for model training, evaluation, deployment and serving, e.g., model exploration with Python code manually on a local server or scheduled pipelines to train models in production, could be seamlessly integrated with the model management system. With this flexibility, we can manage models from all different machine learning projects at different development stages. This lowers the on-boarding cost for new users and provides model management support for a wide array of use cases.

Automation. With a large number of machine learning models and model instances, automatically moving models across different stages in the lifecycle is the key to high scalability and velocity. Achieving automation requires the management system to have an integrated holistic view of the model workflow including training, evaluation and deployment.

3.2 Overall Architecture

We show the overall view of the Gallery architecture in Figure 2. Advanced model management is a core component of a machine learning system as it orchestrates the flow of a model across different stages of a lifecycle. For the sake of completeness, we include a generic machine learning system that encompass the basic stages of a machine learning lifecycle and the data infrastructure we leverage in Gallery for the storage in the architecture. We describe the major components of the Gallery system separately in the rest of this section.

3.3 Data Model

To manage the lifecycle of a machine learning model, Gallery collects data of *models*, *model instances*, and *model performance*, and the corresponding metadata information. We present a simplified version of the basic Gallery data model in Figure 3.

3.3.1 Model. A machine learning model is generally a representation of a transformation from a given input to a given output. We use model metadata to store the basic model information including the model owner, model description (e.g., linear

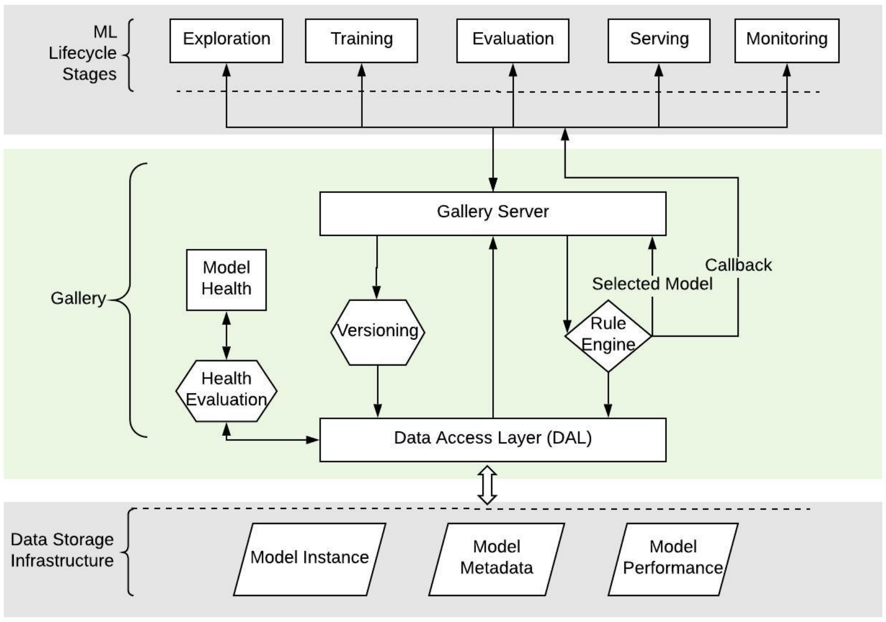


Figure 2: Gallery Overall Architecture

regression formula or neural network structure), features, hyperparameters, and also the information on how the model can be trained and served.

Building machine learning models is always an iterative process through which we generally start with a simple baseline model and subsequently improve the model performance by optimizing the model structure, tuning the hyperparameters, or updating the model features. Therefore, we keep track of the evolution of a model through next and previous pointers in the model record. In a complex production system, we often have one model depending on the output of other models. To get a holistic view of the application of machine learning models in such a system, we also keep track of the model dependency via upstream and downstream pointers.

3.3.2 Model Instance. A model instance is a realization of a model given a set of training data. It consists of the model parameters learnt from the training data and it is used to construct the model in serving for prediction. To achieve model neutrality, we treat model instances as uninterpreted binary blob data and any updates to the blob will be versioned as a new instance in Gallery. As a result, Gallery can not interpret any model and treats all the models the same. Depending on the types of models, the model instance sizes vary from a few KBs to 10s GBs. Model blob storage is abstracted from the users, and the blob is saved via Gallery in distributed data storage systems, e.g., S3 or HDFS.

We decouple the storage of the model instance blob with other model information. Each model instance has a field to record the model instance blob location, which could be a HDFS or S3 path.

For a model instance, we use metadata to keep track of the training data, training framework, and other configurations (e.g., seed for random number generator, number of epochs for training a deep learning model) we have set for the training to generate the model instance. Storing all the information about the models and model instances allow users of Gallery to closely reproduce their model instances on demand. Note that it is not always possible to generate exactly the same model instance due to the randomness introduced in training the models.

3.3.3 Model Performance. We track the performance of every model instance for offline model evaluation and online performance monitoring. When users measure their models either offline or online, they can write blobs of evaluation metrics that pertain to a specific mode instance. Each metric also has its own set of metadata to describe the nature of the evaluation. We store metrics as blobs in order to remain model neutral and framework agnostic. For different model evaluations, we can have different metrics, e.g., precision, recall, AUC for classification models and MSE (Mean Squared Error), MAPE (Mean Absolute Percentage Error) for regressions models. There are also a lot of customized metrics defined for application-specific evaluations. Gallery treats all the metrics the same and the metrics take the form of a structured blob with the basic format of "<metric>: <value>" pairs.

3.3.4 Metadata. As shown in the Figure 3, for Gallery models, model instances, and model performance, we keep a comprehensive set of metadata to identify model ownership, associate each model with its serving context, and link models to their training datasets. With metadata, we can improve the discoverability of models and instances by enabling search over key metadata fields. We record all necessary configurations, e.g., training code pointer, hyperparameters, and training data location and version, to make model instances reproducible. We provide a standard set of metadata fields and naming conventions to unify the characteristics of a model over a production system.

Note that none of the metadata about a model or a model instance is generated in Gallery. Users of Gallery need to save the information to Gallery via APIs within the Gallery server. An example of the usage of the Gallery APIs is presented in Section 4.1. Gallery simply manages the information and indexes the data

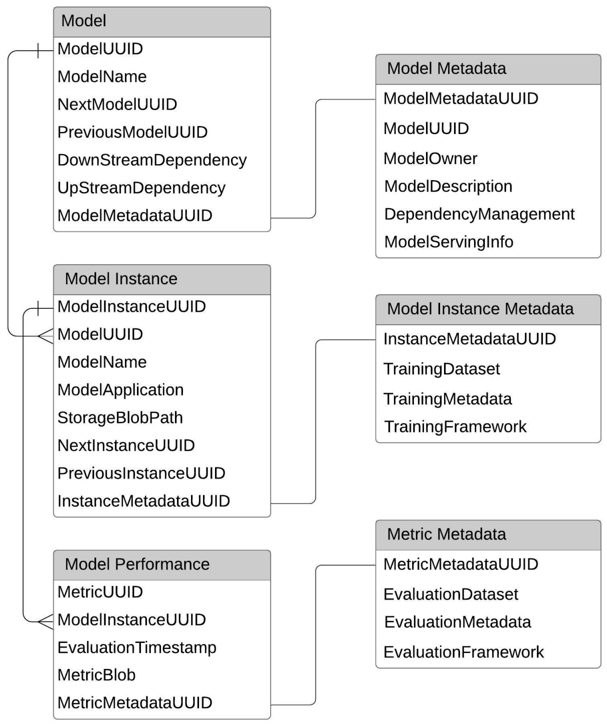


Figure 3: Gallery Basic Data Model

for querying. As a result, Gallery is model neutral and agnostic to any modeling framework.

3.4 Model Versioning

Model versioning is an approach to uniquely identify a model or model instance. It is the foundation for model immutability. With versioning, we never override an existing model or model instance. Any update to a model or model instance will introduce a new version. We keep track of the update history as the lineage of the model or model instance.

Versioning of code or artifacts is a basic requirement for any production system. While it is standard to use Git for code version control, there is no such standard for versioning models or model instances. As a result, many users derive their own versioning schemas, like Semantic Versioning [8] and timestamps, which lead to high maintenance costs due to lack of standandarization across users users and applications. Gallery abstracts model versioning away from users, analogous to what Git provides to code, and provides APIs for users to trace model lineages.

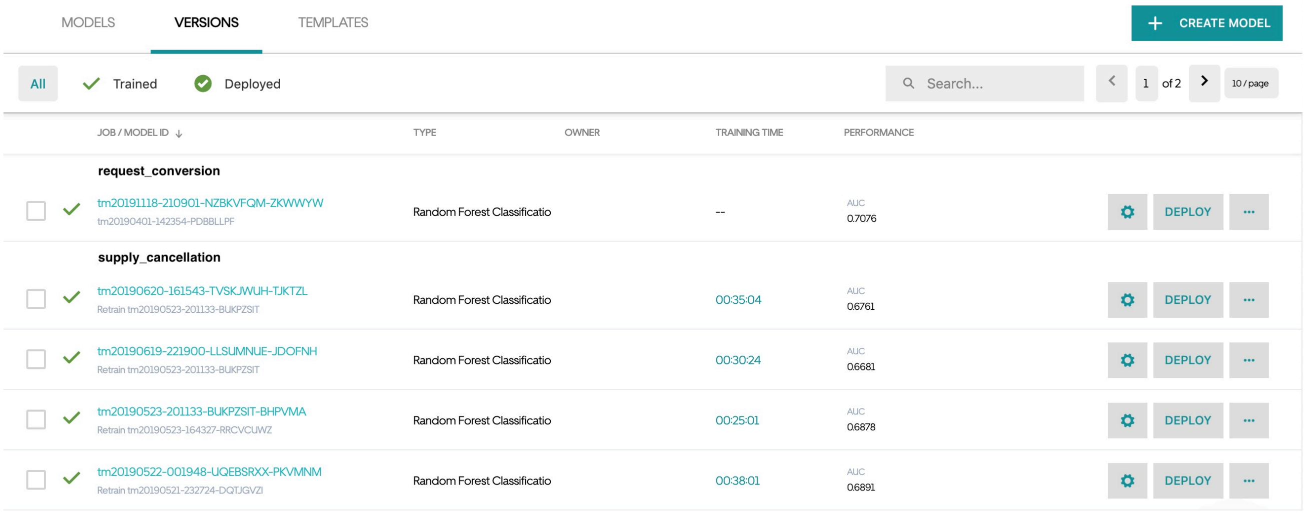
3.4.1 Model Instance Versioning. Each model can have one or multiple model instances. We not only version models, but also model instances. This is because both models and model instances have their own notions of change that need to be tracked. An update to a model represents some change to the underlying data transform such as feature and hyperparameter changes. Typically, these changes happen in response to new approaches for solving a problem and are usually less frequent than model instance updates. Model instance versions represent updates against an existing model with new training data. In production, periodic retraining is expected as new training data becomes available, and information about retraining is captured in the model instance versioning.

The versioning approach we took before Gallery is based on semantic versioning using the format of "<major>.<minor>.<patch>". A version example for a demand forecasting model instance is "1.3.10". We adhere to the following basic version updating rules: 1) update major versions when model architectures change, e.g., from linear regression to neural network; 2) update minor versions when features or hyper-parameters change, e.g., adding a new feature, and 3) update patch versions when the model instance is retrained. This approach works well when we have one simple forecasting model for a handful of cities. However, it is not manageable when we build and launch multiple forecasting model for hundreds of cities. As different models might perform better or worse for different cities and the forecasting model performance might degrade gradually due to the changes in the Uber business, we expect retraining models to improve model performance. However we do not want to retrain models for all the cities if one city performs poorly since that needlessly wastes computing resources. As a result, we very quickly end up with multiple model versions for different cites in the production system which becomes impossible to manually manage. The basic semantic versioning schema also loses meaning because cities are no longer aligned against the same versions.

In Gallery, instead of incorporating model semantics into the versions, we adopted a Git style versioning approach and assign a UUID for each model instance. We associate metadata to capture the model semantics and make it easy to search for. To be specific, when users create a new model, they declare a base version id for the model. The base version id is the top-level identifier that is linked to all its descendent model instances. Typically, a base version id represents some approach to solving a particular problem (e.g., demand forecasting). Each time a model instance is trained, a unique identifier is assigned to the trained model and its metadata tracks which base version id the instance was trained from. In this way, users can query for specific model instances, or traverse the evolution of their model by following all instances linked to a given base version id.

Figure 4 shows one example of a model and model instance. There are two base model version ids "demand_conversion" and "supply_cancellation" which represent models for the corresponding business problems. For example, "supply_cancellation" has evolved over four iterations with different model instances which are identified by four different UUIDs. The model instances are sorted by time and linked to the base model they were trained from.

3.4.2 Dependency Management With Versioning. Besides model and instance identification, versioning is at the core of model dependency tracking and management. As collaboration grows within a production system and models become more advanced, there are scenarios where models become dependent on one another. For example, the output of one demand forecasting model could be used as a feature for a pricing model. As systems become more complex, these types of relationships become very difficult to track. Tracking these relationships is an important prerequisite for understanding how a model impacts the entire production environment with a holistic view. Users need to be aware of the consequences of changes in their models, or need to be aware that changes in their model's behavior could be due to upstream dependencies. For example, the performance of Model A could improve even though the only change is on its upstream Model



	JOB / MODEL ID	TYPE	OWNER	TRAINING TIME	PERFORMANCE
	request_conversion				
✓	tm20191118-210901-NZBKVFQM-ZKWWYW tm20190401-142354-PDBBLLPF	Random Forest Classificatio		--	AUC 0.7076
	supply_cancellation				
✓	tm20190620-161543-TVSKJWUH-TJKTZL Retrain tm20190523-201133-BUKPZSIT	Random Forest Classificatio		00:35:04	AUC 0.6761
✓	tm20190619-221900-LLSUMNUE-JDOFNH Retrain tm20190523-201133-BUKPZSIT	Random Forest Classificatio		00:30:24	AUC 0.6681
✓	tm20190523-201133-BUKPZSIT-BHPVMA Retrain tm20190523-164327-RRCVCUWZ	Random Forest Classificatio		00:25:01	AUC 0.6878
✓	tm20190522-001948-UQEBSRXX-PKVMNM Retrain tm20190521-232724-DQTJGVZI	Random Forest Classificatio		00:38:01	AUC 0.6891

Figure 4: Model and Model Instance Versioning

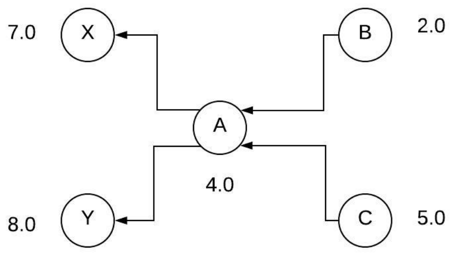


Figure 5: Model Dependency Graph

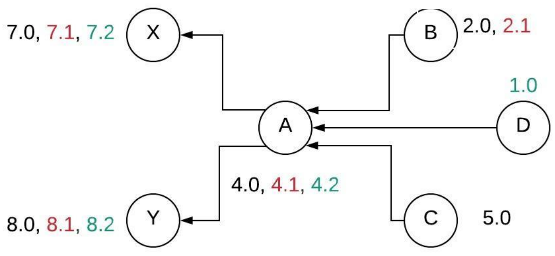


Figure 7: Adding New Model Dependency

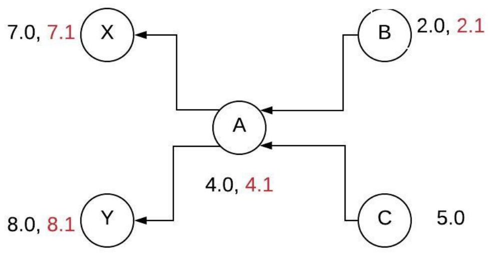


Figure 6: Model Dependency Update

B. Without tracking this dependency relationship, we would lose track of Model B's impact on the production system.

Here, we present one example to show how dependencies of 5 models are managed by Gallery. In Figure 5, we show a dependency graph of five models. Both Model X and Y depend on Model A and Model A depends on Model B and C. For readability, we use numbers instead of UUIDs to represent the model instance versions in this example. In Figure 6, we show the case of a model dependency update. When we update instance of Model B from version 2.0 to 2.1, this triggers the version updates for all Model B's downstream models including A, X and Y. Considering that there is no real change of Model A, X or Y, we automatically update the model instance version by adding a new model version to Gallery without changing the production versions. The owner of Model A can choose to upgrade to the new model version, if they want to include the updated Model B. But, models are not automatically updated because we would like users to be aware that their model dependencies have changed before their production environment is updated.

Figure 7 shows a use case when we add a new model dependency for an existing model. By adding the Model D as the dependency of Model A, we will automatically update model A's instance version to 4.2. Accordingly, the downstream Models X and Y will also be updated to instance version 7.2 and 8.2 separately.

Dependencies between models are established by the user when models are first registered in Gallery. When adding models, Gallery provides operations for the user to add dependent models by their uuids. There are also operations exposed for updating or removing dependencies. Once the dependencies are established, Gallery provides users with APIs to query their model's upstream or downstream dependencies and will track model updates across dependencies.

3.5 Model Storage

The Gallery model storage layer defines the interface through which model blobs, metadata, and metrics are stored and retrieved. We have the following model storage requirements: *searchability*, *agnostic*, *high availability*, and *low-latency*.

To satisfy these requirements, we build the Gallery model storage using a hybrid approach. Considering that model metadata and metrics are commonly structured data, we use a relational database, e.g., MySQL, for storing metadata and providing support of flexible queries. The MySQL service is supported by the Uber infrastructure team to guarantee high availability and deployed cross data centers. Considering the enormity of models and model instances, we would not be able to scale up to handle thousands of models and instances if we need to interpret each model. As a result, we treat each model equally and store each model blob as binary data. We leverage Uber's large data storage service built on top of S3 and HDFS to store the model blobs to achieve model neutral and framework agnostic design principles. We expect Gallery users to provide their models as serialized binaries, which are in turn stored in Uber's large data storage service. The storage locations are subsequently stored as part of the model metadata so that they can be retrieved at serving time. Another benefit of taking this approach to save model blobs is that it does not have data size constraints, which can be an issue for large deep learning models. To handle cases of inconsistent data due to system failures, e.g., MySQL or HDFS write fails, we always write model blobs first and only write the model metadata after the model blobs are successfully stored. If the model blob of a model instance is saved but the metadata fails to save, then the model instance will not be available in the system.

Model metadata searchability is critical for users managing a high volume of models. Users conducting experiments or managing production environments need the ability to easily search and query models based on key metadata like training dates, model type, and features. Model searchability allows easy tracking of all models in the wild and more efficient analysis and experimentation over the various models.

Briefly, model storage is accessed via a unified DAL (data access layer). The model performance metrics are saved and read from MySQL to support flexible queries for analysis and monitoring. When models instance blobs are queried, the request first goes to MySQL to get the location of the model blob, and then the model is directly accessed via the storage location. The cache is updated with the requested blob and then is subsequently returned to the user.

3.6 Model Performance and Health

When building and maintaining production-grade software systems, it is standard to define SLAs with consumers to establish accountability and trust. Typical SLAs for software systems include availability, latency, and throughput. For machine learning systems, we also would like to have SLAs on performance. We define model health as a set of metrics and standards for users to adhere to in order to guarantee some level of accountability of models in Gallery.

More specifically, we define two categories of metrics to measure the model health. One category of the metrics is on the completeness of model information, which consists of metadata for model reproducibility and model performance. Production models should contain enough metadata to reproduce the model and annotate the behavior leading to a decision. Different models may have different performance metrics. In Gallery, we ensure that the performance of each model is evaluated and stored for monitoring and analysis.

The second category of model health metrics provides a holistic view of model performance across model lifecycle stages including training, validation and production. All model performance is agnostic to the system and provided by the applications. Model training performance is generally available as a by-product of model training. Model validation performance is produced by validation processes or backtesting and is used to check for model overfitting or as a gauge of whether to deploy a model to production. Model production performance is measured against served predictions and reflects the online performance of a model. The evaluation criteria for each performance metric is entirely up to the user and is configurable, since different models and applications optimize for varying outcomes. We store an object of metrics in Gallery and define the above metrics as guidelines for users.

With model performance metrics, we can derive various insights about the models in Gallery. The insights can give model owners a signal on how their model behaves over time, information about their serving environment, and establishes a level of trust between model owners and model consumers. Here are two examples of insights that Gallery can provide: model drift and production skew.

Model Drift . Model drift refers to the case when the statistical properties of the target variable, which the model is trying to predict, change over time in unpredictable ways. With real-time platforms, data changes. Accordingly, if the data we use in production gradually evolve to have different patterns from the data we use in the training, we may see the model performance degrade over time. Considering Uber's rapid growth in many markets, this drift can occur and once detected, triggers model re-training via Gallery rule engine using the new training data.

Production Skew. Production skew is the difference between performance at training time and serving time. Multiple factors can result in production skew, such as bugs in training or serving implementation, or discrepancies between training data and data feeding to model serving. The ability to detect production skew is critical for model performance monitoring.

3.7 Orchestration Rule Engine

As the number of models and model instances grow in a production system, it becomes increasingly difficult to manually manage their various states. Therefore, we designed and built a rule engine in Gallery to orchestrate the model workflows. Based on the model metadata, such as deployment configuration and various model performance metrics in Gallery, users can define conditions and actions in rules to automatically move the models across the stages of the lifecycle, such as model deployment and serving, monitoring and retraining, and deprecation.

In the following section, we first show the basic automations we need in the model lifecycle stages where the rule engine can help. Then, we describe our design of the rule engine system and illustrate how it works with examples.

Model Deployment and Serving Automation. When we generate model instances through training, we decide whether to deploy a model instance to production and replace the existing instance. It is common to have multiple models and instances deployed in production and use rules to select the best performer for serving, based on performance metrics generated by evaluation systems. Normally, different applications will have their own evaluation systems to measure the performance of the models. For example, in real-time forecasting, we have a heuristic model which uses the mean value of last 5 minutes as the forecasts. The heuristic model is stable and consistent, but may not always produce the best performance. We also have complex forecasting models that take in more features, like historical data, into the prediction, which are generally better performing but may not perform well when there are unanticipated events not specifically considered in the modeling. We have a real-time system to evaluate the performance of the models. Therefore, we can combine the benefits of different models to achieve the overall best performance by using the model metrics in Gallery to make decisions. With a rule engine, we can define the model candidates to consider and the selection criteria for choosing a champion. At serving time, users will query Gallery for the champion model to serve based on the user-defined rules.

Model Monitoring and Retraining. After we train models and deploy model instances into production, we need to keep monitoring the performance and alert in cases like model drift, as discussed in the previous section. At the same time, model performance can degrade because the training data we used to fit the model no longer best represents the production data. Therefore, we can re-train the model with the latest training data. With a rule engine, we can set rules to automatically detect model drift, send out alerts, and trigger model training.

Model Deprecation. Models are not used forever, we may not always improve the performance of a model by retraining, and we keep experimenting with new models. When a model consistently performs worse than other models, we should deprecate it to save computational resources. Users can utilize the rule engine to define the deprecation criteria based on sustained underperformance, and training and serving costs, e.g., training takes too long or requires a large amount of computational resources. This precaution allows users to ensure that their production systems are not being negatively impacted by poorly performing models. When a model or model instance is deprecated, we would not delete them from the system, but rather flag them as deprecated. With the flag, we can skip them during model fetching or searching. Any application depending on these deprecated models or model instances can still use them until the application finishes migration to new models.

3.7.1 Rule Engine Design. To satisfy the needs of orchestrating models across lifecycle stages in production, we identify three requirements to design the rule engine: rules being *easy to understand*, *reliable*, and *agnostic to supporting services.*

Making rules easy to understand and safe to update is the first principle. We use the rules to control the production systems for model deployment and serving. We need to make sure Gallery users understand the rules and have confidence updating the rules to avoid outages due to unexpected rule usage or changes. At the same time, we need to make sure the system reliably applies the rules within a reasonable response time (SLA) when the rule is triggered. The rule engine needs also to be framework-agnostic so that any model training, monitoring, or serving components can leverage and integrate with the Gallery.

Based on frequent use cases, Gallery leverages two type of rules: *model selection rules* and *action rules.* Applying a model selection rule will return a model based on some selection criteria, e.g., returning the model that maximize AUC (area under curve). Applying an action rule will trigger some event, e.g., deploying a model instance. To make the rule engine agnostic to different frameworks within the machine learning workflow, we expect users to define callback functions that will be triggered by the rule engine. For example, to deploy a real-time forecasting model, we have one action that automatically makes a configuration change, via http request, that updates the version of the model served to consumers. There are also a default set of common actions that users can leverage or extend, like sending an email or alerting.

We use the classical *Given/When/Then* model to define rules. More concretely, for "Then" we define two templates: model selection and callback action. For example, the following rule is designed for the selection of a forecasting model.

Listing 1: Model Selection Rule Example

```
{
  "team": "forecasting",
  "uuid": "316b3ab4-2509-4ea7-8025-ca879dac61",
  "rule": {
      "GIVEN": modelName ==
        "linear_regression"
        AND model_domain == "UberX",
      "WHEN":  "metrics["mae"] <= 5",
      "ENVIRONMENT": "production",
      "MODEL_SELECTION":
      "a.created_time > b.created_time"
    }
}
```

With this rule, we select the latest trained linear regression model if the model performance is good, i.e., mae (mean absolute error) is less than 5. This rule allows the user to automatically fetch the freshest models and have confidence that the returned models are within their accepted error threshold.

The following is an example of an action rule which specifies: when we have a new model instance, and if the model performance is within a threshold, e.g., forecasting bias less than 0.1 and greater than -0.1, we can deploy the model in production.

Listing 2: Action Rule Example

```
{
  "team": "forecasting",
  "uuid": "43057544-92b0-4421-a1b0-d7d87f77a",
  "rule": {
      "GIVEN": "model_domain == "UberX"
           AND model_name == "Random Forest",
      "WHEN": "metrics.bias <= 0.1
           AND metrics.bias > -0.1",
      "CALLBACK_ACTIONS": [
        {
            "action":"forecasting_deployment"
        }
       ],
   "ENVIRONMENT": "production"
   }
}
```

3.7.2 Rule Engine Implementation. We show the overall architecture of the Gallery rule engine in Figure 8. For rule storage, we use a Git repository. To add or update a rule, users need to check it into Git within their allocated directory. The advantage of using Git is that we automatically have version control for the rules, which is critical for a production environment and enables us to set up a test framework to validate each rule before it can impact production. With Git, we can also easily enforce the peer review process for the rules to avoid production outages due to accidental updates of rules. We use Java Expression Language (JEXL) [2] to implement the rules.

The rule evaluation implementation is event based and we currently have two kinds of events to trigger the evaluation of a

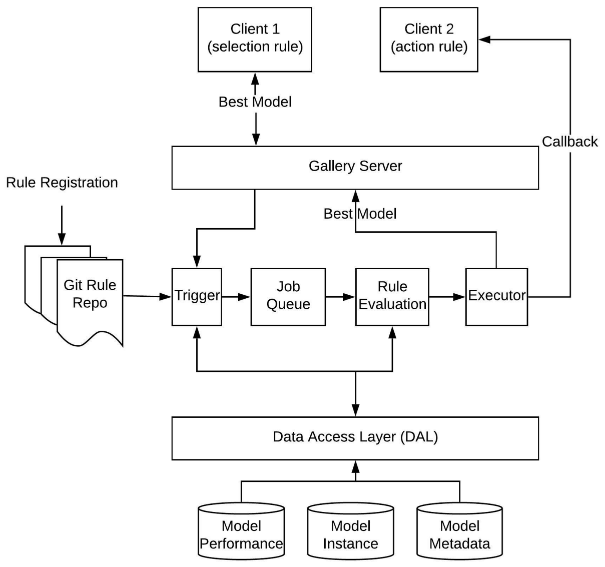


Figure 8: Gallery Rule Engine

rule: 1) directly sending a request to the rule trigger or 2) updating any metadata or metrics specific in a registered rule. Since we focus on two types of customized rules, we implement the rule evaluation ourselves instead of using other rule engines. In Figure 8, we demonstrate the workflow of the rules with two application clients. Client 1 has a model selection rule and sends the rule directly to the rule engine trigger via Gallery service. The rule is first placed in the job queue and goes through the evaluation during which the performance metrics of the related models are queried from the storage. Then the best model instance is fetched and returned to Client 1 via a Gallery service based on the condition in the rule.

Client 2 has an action rule registered in the Git rule repo. Whenever there is an update of the corresponding metadata or metric specified in the rule via a Gallery service, the evaluation of the rule will start and the job is put into evaluation job queue. If there are any action triggers, then the callback specified in the rule is executed, e.g., "when a model instance performance metric mae is less than 0.2, we deploy the model in production."

4 GALLERY IMPLEMENTATION AND USAGE CASE STUDIES

Gallery was built by the Uber Marketplace team for managing forecasting models to improve the Uber plaftform. We leverage the Uber storage infrastructure for saving the model related information and we built Gallery as a stateless microservice that includes versioning, dependency management and rule engine in Java. Gallery was designed and built to be horizontally scalable across different data centers.

Before Gallery, there was a lot of manual cost and overhead to manage our forecasting models. For about 100 models, engineers and data scientists spent 1-2 hours a day manipulating files on HDFS and Git, measuring performance and triggering model retraining. Now, Gallery has been in production for two years and is supported as part of Uber's Michelangelo ML platform. Under Michelangelo, Gallery is managing more than 1 million model instances for many machine learning applications, and engineers or data scientists no longer spend time managing files and training scripts, but instead are able to spend their energy on model iteration and experiments.

4.1 Gallery Interface

Gallery users interact with Gallery via a standard set of Thrift APIs with language-specific clients. By using Thrift, users can access the functionality of Gallery in their own modeling environment and language of their choice (e.g., Jupyter notebook, Spark application, services build in Java). With the APIs, Gallery users can manage their models cross the stages of a model lifecycle. In the following example, we use a Python application example to show one typical Gallery user workflow using the basic Gallery APIs.

```
'''
Train a ML model using SparkML and upload the
    model blob to Gallery with model instance
    metadata information.
'''
# Using a SparlML pipeline to fit a model
model_object = pipeline.fit(train_df)
# Model is serialized to a binary blob
model_content = serialize(model_object)

# Create and upload the trained model instance
    with metadata to Gallery.
model = createGalleryModel(project='example-
    project', base_version_id='supply_rejection'
    )

# Add model instance information
modelInstance = createModelInstance(model)
modelInstance.content = model_content
modelInstance.modelName = 'Random Forest'
modelInstance.city = 'New York City'
modelInstance.modelType = 'SparkML'
modelInstance.trainingDataSet = '...'
modelInstance.trainingDataMetadata = '...'
...

# Update model instance to Gallery
modelInstanceId = uploadModel(project='example-
    project', base_version_id='supply_rejection'
    , modelInstanceRecord=modelInstance
  )
```

Listing 3: Create and Save Gallery Model

The sample code in Listing 3 shows that we use SparkML to train a machine learning model and serialize the model into a binary blob. Then we add the related model instance metadata information and save the model instance to Gallery.

```
# Upload a model instance performance metric
modelPerformanceRecord = ModelEvaluationMetric(
  metricName='bias',
  scope='Validation',
  value=0.05
  )
insertModelInstanceMetric(project='example-
    project', modelInstanceId=modelInstanceId,
    modelPerformanceRecord=
    modelPerformanceRecord
  )
```

Listing 4: Save Model Performance Metric

With the model instance we have trained previously, we keep track of the model evaluation performance by saving the performance metrics in Gallery as shown in the sample code in Listing 4. At the same time, if we have one rule similar to what is shown in Listing 2 registered in the rule Git repository, then the corresponding model deployment process might be triggered based on a rule condition. How to automatically deploy model in production is different for different serving systems and we leave this part to Gallery users to define their own callback functions. For example, a realtime forecasting model is deployed in Uber by updating some configuration which is continuously monitored by the forecasting serving system in production.

```
# Model query with a given performance criteria
searchConstraint = [
    {'field':'projectId',   'operator':equal,
         'value':'example-project'},
    {'field':'modelName',   'operator':equal,
         'value':'random_forest'},
    {'field':'metricName',  'operator':equal,
         'value':'bias'},
    {'field':'metricValue', 'operator':
    smaller_than, 'value':0.25}
]
modelInstance = modelQuery(searchConstraint)
```

Listing 5: Model Search

We save the related metadata and performance metrics of the models and model instances in Gallery. Then, we could fetch the models we want with the conditions as shown in Listing 5.

4.2 Case 1: Marketplace Forecasting

At Uber, the Marketplace Forecasting team generates real-time and long-term forecasts for multiple applications, including driver suggestions and pricing [4]. Multiple supply and demand models have evolved through different model classes ranging from simple time series models, linear regression models, and now deep learning models. Each model class is trained and deployed per city Uber operates in. Each city faces different market dynamics, and classes of models perform differently based on certain spatial or temporal characteristics of the city. Therefore, the team needs a mechanism to track and train each model's performance on a per city basis, and a systematic way to determine which model class to serve at a given time. As a result, the Marketplace Forecasting team alone has thousands of model instances to maintain and decisions to make each minute about which model to serve. Gallery's model management solution with storage and automation via rule engine has reduced model deployment from two hours of engineering work per model to 0.

Another use case is dynamic model switching for forecasts when there are events e.g., holidays. Via action rules, Gallery is able to inform forecasting serving system about the performance of models that include holiday/event features versus those that do not, and subsequently switch to serve the appropriate models for the duration of the event. This improves the accuracy of the served predictions by more than 10% MAPE (Mean Absolute Percent Error) compared to a static served model. Furthermore, Model Health alerts continue to monitor the performance of such models and can alert engineers regarding issues with prediction accuracy. These alerts have proven useful in the case of unplanned events (e.g., public transit outages) that cause unexpected spikes in demand, and gives engineers or ops an opportunity to intervene and mitigate the performance degradation.

4.3 Case 2: Marketplace Simulation Platform

The Marketplace Simulation platform [5] hosts a simulated world with driver-partners and riders, mimicking scenarios in the real world. Leveraging an agent-based discrete event simulator, the platform allows Uber Marketplace engineers and data scientists to rapidly prototype and test new features and hypotheses in a risk-free environment.

Priori to leveraging Gallery, one issue the simulation platform had is model reusability. ML developers implemented models directly in the simulator and trained them on the fly as the simulator ran. As a result, the complexity of the system and the wide array of models being simulated degraded the performance of the platform.

Gallery became part of the solution by providing the simulation platform with metadata and model binary storage, which enabled the platform to decouple model training and serving. Offline processes can store reusable model instances into Gallery, and the simulation backend service can instantiate such models as they're needed. This decoupling allows model developers to iterate and evolve their models, independently of the simulator's backend, whereas before they need to wait for the entire end-to-end process each time they trained/updated a model. Once a model developer is satisfied with their model, they can store their model in Gallery, which will signal to the backend service the presence of a new model. Furthermore, decoupling model training workloads from the simulator, allowed the Simulation team to simplify the complexity of the simulator reduce maintenance costs, conserve hardware resources while improving system efficiency. The Gallery system has saved the simulation platform an estimated 8GB memory and one hour CPU time per simulation.

5 RELATED WORK

With the proliferation of Big Data and large-scale computing (e.g., MapReduce [16], Apache Spark[31]), several scalable machine learning platforms [32] have emerged in recent years with the focus of machine learning training on a large amount of data. Apache Spark is a general data processing framework. MLlib [20, 23], the machine learning library added to Apache Spark, makes Apache Spark broadly used for some simple large-scale machine learning model training. However, for complex machine learning tasks, especially deep learning [17], which requires state updates and iteration, parameter server architecture is used for enabling in-place updates for very large parameters, e.g., Parameter Server [18], PMLS [30], Google DistBelief [15]. TensorFlow [11] and MXNet [24] are advanced machine learning frameworks focusing more on the deep learning problem and can fully utilize different devices like CPU and GPU.

Managing a large number of machine learning models in production is challenging. There is an ever-increasing interest in the problem and several academic and industry efforts have been published. ModelDB [28] is an open source model management system which provides APIs for saving models and associated metadata, measuring performance, and querying models. It has a web-frontend for easy visualization and summary of the model information. ModelDB also provides clients with the ability to integrate its model management features with Apache Spark ML and scikit-learn [25]. However, there is no orchestration of model training, serving and deployment in ModelDB.

A lightweight system is proposed in [27] to extract, store, and manage machine learning model metadata. It tracks the provenance information of datasets, models, predictions, evaluations

Table 1: Model Management System Comparison (Y: Yes, N: No)

Systems	Saving	Loading	Metadata	Searching	Serving	Metrics	Orchestration
ModelDB [28]	Y	Y	Y	Y	N	Y	N
ModelHUB [21]	Y	Y	Y	Y	N	Y	N
Metadata Tracking [27]	N	N	Y	Y	N	Y	N
Velox [13]	Y	Y	Y	N	Y	Y	Y
Clipper [14]	Y	Y	N	N	Y	Y	Y
MLFlow [22]	Y	Y	Y	Y	Y	Y	N
TFX [12]	Y	Y	Y	N	Y	Y	Y
Azure ML [1]	Y	Y	N	N	Y	N	Y
SageMaker [26]	Y	Y	N	N	Y	N	Y
Gallery	Y	Y	Y	Y	N	Y	Y

and training runs in machine learning experiments. With model metadata and provenance, the system provides a basis for comparability and repeatability of machine learning experiments. Similar to ModelDB, there is no orchestration of machine learning stages in this system. This system is focusing on the metadata associated with model experiments instead of managing the machine learning models.

ModelHUB [21] was built to manage the lifecycle of deep learning models. Considering the huge space of potential deep learning models by tweaking neural network architectures and hyperparameters, ModelHUB compactly stores a large number of models and snapshots with fast query performance. It also keeps track of the model metadata including the model accuracy score. ModelHUB focuses on deep learning models and is not framework agnostic.

Velox [13] is a low-latency and large-scale model serving system. It focuses on making the model serving efficient by caching computation and scaling out model prediction. Velox leverages a cluster computation framework and incremental model updates to scale out the model training process. Velox also manages the model lifecycle by detecting model performance degradation to trigger model rollback or re-training. However, the project was deprecated [29].

Clipper [14], the follow-up project of Velox, is a general-purpose low-latency prediction serving system. It can incorporate multiple machine learning frameworks including TensorFlow [11], Apache Spark [31] and scikit-learn [25]. Similar to Velox, it uses caching, as well as batch and adaptive model selection to improve model prediction latency and performance. Clipper focuses on serving models with low latency across different frameworks.

MLflow [22] is an open source platform under active development for managing the machine learning lifecycle. There are three major components in MLflow: MLflow Tracking, which tracks the experiments results and parameters; MLflow Project, which packages the ML code to be easily reproducible; and MLFlow Model, which provides a standard format for packaging ML models across different libraries or framework. The same model could be packed with different flavors such that a model could be applied in different frameworks, e.g., a TensorFlow model can be used with TensorFlow flavor or python function flavor. With this MLflow Model format, the models can be used in a variety of downstream tools, e.g., real-time serving through a REST API or batch inference on Apache Spark. MLFlow also provides CLI to run the MLflow models for model deployment. However, there is no orchestration to coordinate the moving of models across different stages in a model lifecycle.

TFX [12] is a production-scale machine learning platform for TensorFlow and it consists of multiple components for machine learning, including data transformation, model training, model evaluation, and model serving. With the TensorFlow serving component [9], TensorFlow models can be deployed and served in production. However, TensorFlow serving is a not generic component for managing a variety of machine learning models using different frameworks. Kubeflow [3] is a project to make deploying ML workflows on Kubernetes simple, portable, and scalable. It started as an open source project from Google that highlighted how the company ran Tensorflow internally based on TFX. Now, Kubeflow has extended to be a multi-architecture, multi-cloud framework for running entire ML pipelines.

Azure ML [1] is a machine learning platform where we can process data, build models and publish and stage a predictive model as an Azure-based service. Similar to Azure ML [1], AWS SageMaker [26] also provides the functionality to build models, train models, and deploy models in production. Both Azure ML and AWS SageMaker are closed systems making single component integrations challenging. The model management in Azure ML and AWS SageMaker is really focused on training a model and deploying a model in its own system. They are not model neutral and framework agnostic. Kepler [19] and Taverna [10] are popular scientific workflow systems which manage complex data analytics pipelines including data access, data analysis and mining steps, and many other steps including computationally intensive jobs on high-performance cluster computers.

We present a feature comparison of different model management systems in Table 1.

6 LESSONS LEARNT IN BUILDING GALLERY

6.1 Common ML Tools

Machine learning is becoming the essential component of many Uber product features. Accordingly, more and more teams at Uber are using machine learning or beginning to use machine learning. Different teams might be at different maturity stages of applying machine learning depending on the team's experience, but all of them will go through solving the common issues of managing the models in a machine learning application. Without shared common ML infrastructure tools, each team might waste a lot of resources to "reinvent the wheel." When we built Gallery, many teams express the similar needs of Gallery to manage their models. As a result, we made Gallery part of Uber's standard ML infrastructure as part of Michelangelo so that it could benefit all the teams at Uber. It also shows the importance of building common ML tools so that all the product teams can boost their productivity by focusing on their own business problems and model iterations without worrying about how to train, manage and serve their models.

6.2 Model Reproducibility

Shortly after onboarding the first group of users of Gallery, we observed and learned that one of the more desired features was model reproducibility. Reproducibility was not one of the original use cases we had in mind when we built Gallery. Instead, we had focused directly on performance tracking and alerting. However, it became clear that a user's natural follow-up to an alert is to attempt to reproduce the problem, and it was apparent a model management system needs to support this functionality. The original Gallery data model did not include many of the metadata components included today that led to model reproducibility (e.g., training data information, training frameworks and features), but an important lesson learned is that reproducibility is central to supporting the ML model lifecycle. Users need the ability to recreate models or replay history in order to understand their production flows and debug performance. Gallery has proven to be a valuable tool for model builders at Uber in simplifying the model debugging process.

6.3 Tiered Service Offering

Another lesson we have learned during the development of Gallery is how to make user adoption of such tools easy. As discussed before, there is a wide variety of ML tooling being used at Uber and teams across the company are working against their own deadlines with different features in their tech stacks. Therefore, there would be no single opportunity to ask users to migrate their workflows and no central mechanism by which we could directly onboard them to Gallery. Instead, we opted to provide a tiered set of features and solutions that teams could leverage as they had the bandwidth and discovered the need. We wanted our features to be modular in that users at any point in their maturity could leverage the system, with the opportunity to add more complex functionalities in the future.

Gallery features are broken up into four groups that are built on top of one another: 1) model storage and retrieval; 2) metadata storage and search; 3) metric storage and search; and 4) rule engine automation. As teams start to use Gallery in their systems, they sequence their onboarding based on the features and complexity that they need to unlock. For teams exploring new modeling techniques or building a system from scratch, it is often the case that they only need feature set 1), and optionally 2). Teams doing experimentation have not yet thought about automation and only need a place to dump models to rapidly do more testing. However, once teams have built out a model and are trying to scale to meet business requirements, they then see the need for feature sets 3) and 4). By using the base functionality of data storage and retrieval in their experimentation, there is only an incremental additional effort required to unlock more complex Gallery features that help to automate entire workflows. This approach helped Gallery gain quick adoption among five teams with in its first six months.

7 CONCLUSIONS

In this paper, we describe the machine learning model management problem across the different stages of a model's lifecycle for a large number of models and model instances in production environments at Uber. We describe the model lifecycle management system, Gallery, a solution used in production to manage machine learning models across different services at Uber.

Design for system automation up front is critical to manage thousands of machine learning models in production. Developing and applying machine learning models involves multiple stages across a model's lifecycle. Manually managing the models and model instances in production is not scalable and is error-prone. Building generic systems to be able collect and keep track of model and instance information, as well as dependencies is critical for maintaining accurate production systems. On top of the raw information, we can produce intelligence. With the help of rules, we can orchestrate the whole modeling workflow, which dramatically boosts data scientists and engineers' productivity and also makes the machine learning systems more reliable and scalable.

Building an agnostic model management system is critical for adoption and user on-boarding. At Uber, there are a large number of existing machine learning applications, which often leverage different languages and frameworks for model development and serving. Building Gallery to be agnostic to machine learning frameworks has allowed the system to be adopted by many teams at Uber and has helped the company to align on a common infrastructure for the machine learning model management.

8 ACKNOWLEDGMENTS

We would like to sincerely recognize and acknowledge the following contributors who played a significant role in the design, development, and productionization of Gallery at Uber: Anne Holler, Yifan Ma, Luke Duncan, Mani Bhushan, Aparna Dhinakaran, and Zhenghui Wang. Additional contributors to the project that we would like to thank also include Andrii Iasynetskyi, Rashmi Venugopal, and Steven Wright. We would also like to thank Haoyang Chen, from the Simulation Platform, for being the first to adopt Gallery and providing extensive feedback to improve the design of the system.

REFERENCES

[1] Azure ML. https://studio.azureml.net.
[2] JEXL. https://commons.apache.org/proper/commons-jexl/.
[3] Kubeflow. https://www.kubeflow.org.
[4] Machine learning at uber. https://eng.uber.com/machine-learning.
[5] Marketplace Simulation. https://eng.uber.com/simulated-marketplace/.
[6] Michelangelo. https://eng.uber.com/michelangelo/.
[7] Rules of machine learning. https://developers.google.com/machine-learning/guides/rules-of-ml/.
[8] Semantic Versioning. https://semver.org/.
[9] TensorFlow Serving. https://www.tensorflow.org/serving.
[10] The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Res*, 41, 2013.
[11] 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. OSDI'16.
[12] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich. TFX: A TensorFlow-based production-scale machine learning platform. KDD '17.
[13] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with Velox. CIDR'15.
[14] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. NSDI'17.
[15] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large scale distributed deep networks. NIPS'12.
[16] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 2008.

[17] Y. LeCun, Y. Bengio, and G. E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.
[18] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. OSDI'14.
[19] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10), 2006.
[20] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine learning in apache spark. *CoRR*, 2015.
[21] H. Miao, A. Li, L. S. Davis, and A. Deshpande. ModelHub: Deep learning lifecycle management. ICDE'17.
[22] MLFlow. https://mlflow.org.
[23] MLlib. http://spark.apache.org/mllib.
[24] MXNet. https://mxnet.apache.org.
[25] 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.*, 2011.
[26] SageMaker. https://aws.amazon.com/sagemaker.
[27] S. Schelter, J.-H. Böse, J. Kirschnick, T. Klein, and S. Seufert. Automatically tracking metadata and provenance of machine learning experiments. In *NIPS Workshop ML Systems*, 2017.
[28] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. ModelDB: A system for machine learning model management. HILDA '16, 2016.
[29] Velox. https://github.com/amplab/velox-modelserver.
[30] E. P. Xing, Q. Ho, W. Dai, J.-K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. KDD '15.
[31] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. HotCloud'10.
[32] K. Zhang, S. Alqahtani, and M. Demirbas. A comparison of distributed machine learning platforms. ICCCN '17.

Diverse User Selection for Opinion Procurement

Yael Amsterdamer
Bar-Ilan University
first.last@biu.ac.il

Oded Goldreich
Bar-Ilan University
first.last@live.biu.ac.il

ABSTRACT

Many applications maintain a repository of user profiles with semantically rich information on each user. Such repositories have a potential of allowing active *opinion procurement*: reaching out to users to ask for their opinions on different topics. An important desideratum of the procurement process is that it targets a *diverse* set of users.

To realize this potential, we present `Podium`: a first framework, to our knowledge, that supports the selection of diverse representatives in presence of high-dimensional, semantically rich user profiles. We demonstrate that data dimensionality is a challenge for both defining and achieving diversification. We address these challenges by proposing a lightweight, flexible notion of diversity that in turn allows explanations and customization of diversification results. We show that the problem of finding an optimally diverse user subset is intractable, and provide a greedy algorithm that computes an approximate solution. We have implemented our solution in a system prototype and tested it on real-world crowdsourcing platform data. Our experimental results show that `Podium` is effective in selecting users with diverse properties, and in turn that the opinions of these users are diverse according to multiple metrics.

1 INTRODUCTION

Multiple applications involve active procurement of opinions from users. Consider, for example, a traveler planning a trip and looking for specific "tips" on some destination; an owner of a new restaurant wishing to perform a preliminary customer survey; or a website manager seeking usability feedback. A recurring desideratum in such applications is that procured opinions are *diverse*: the restaurant owner may seek users with diverse culinary preferences who live in a certain region, whereas the website manager may seek feedback from users with diverse activity history. Notably, diversity considerations may greatly differ between scenarios, even if users are selected from the same set.

Platforms such as Yelp[1] that have a large user base and high-dimensional, rich data on each user, provide an opportunity for procuring opinions from a diverse set of users. Yet, to our knowledge, there is no generic solution for selecting diverse representative users accounting for high-dimensional user profiles. In particular, users chosen for opinion procurement should ideally reflect the full range of user properties as observed in the source population – e.g., the full range of opinions on different topics, from positive to negative; the full range of user skills or activity levels, from low to high; etc. Hence, existing diversification solutions that target the overall accuracy of user answers/relevance of items and therefore operate by optimizing properties across multiple axes (e.g., selecting users' highest skills or activity levels) are inapplicable in this context, as explained in Section 2.

[1]Yelp website: https://www.yelp.com

To this end, we introduce `Podium`: a novel tool for the procurement of diverse opinions, utilizing multidimensional user profiles. We next overview our main contributions.

Model. Our model captures user profiles including both personal details provided by the users and their past interactions with the platform. These properties may be associated with a numeric score (reflecting, e.g., rating) and form high-dimensional data. We then provide a formal definition of the diverse user selection problem that is *coverage-based* [1–4]: i.e., the goal is selecting a user subset that in some sense represents or "covers" many of the different, possibly overlapping *groups* within a source population. This class of diversity notions fits typical scenarios of opinion procurement (e.g., surveys, market research), in contrast with *distance-based diversity*, which focuses on maximizing the differences between the members of the selected group [4–7]. As observable from Table 1, our diversity notion fulfills a unique combination of desiderata that arise at an opinion procurement scenario. We overview the desiderata and the compared solutions in Sections 2 and 9. We further propose an operative method for computing user groups from a repository of profiles, along with weight functions to prioritize the coverage of these groups, where the coverage of every group is impossible.

Analysis of the Basic Problem. Based on our model, we develop a solution to the diverse user selection problem. First, we show by a reduction from Set Cover that the decision problem corresponding to user selection in our context is NP-complete, and that finding a user subset of size approximately minimal that covers all the possible groups is also computationally hard. Moreover, in a high-dimensional setting, full coverage would typically require an unrealistically large number of procured opinions. Thus, instead of targeting full coverage and optimizing the subset size, we bound the size according to some budget and aim to select a user subset of that size that maximizes the *total coverage score*, to be defined in Section 3. Fortunately, a user subset whose coverage score is within a constant factor of the optimal can be found in PTIME. We show a simple greedy algorithm that achieves this bound, explain its data structures and optimizations, analyze its time complexity and demonstrate its operation on a sample user repository.

Customization and explanations. The required notion of diversity may vary based on the concrete application and depending on the multiple dimensions of user data, as exemplified above with respect to the different needs of a traveler versus restaurant owner versus website manager. We thus adopt a lightweight solution that facilitates interpretation of the results and in turn allows the clients to interact with the system to customize and fine-tune user selection. This is achieved through a formal definition of *explanations* for how the selected subset covers the population groups and the contribution of each selected user. We then formally define the semantics of a *user feedback* that allows an informed control over the user groups/data dimensions whose coverage is targeted. We extend our problem definition and analysis accordingly.

System	Type		Range	High-Dimension	Explanations	Customizable
Podium	Coverage-based	Intrinsic	✓	✓	✓	✓
Cohen & Yashinski [2]	Coverage-based	Intrinsic	✓*			✓
Stratified sampling (e.g., [8])	Coverage-based	Intrinsic			✓	✓
T-Model [4]	Coverage-based	Predicted	✓*			
APM [3], IA-Select [1]	Coverage-based	Predicted				
Yu et Al. [7]	Distance-based	Intrinsic		✓	**	
S-Model [4], DiRec [5]	Distance-based	Intrinsic	✓***	✓***		
DivRSci [6]	Distance-based	Predicted	✓***	✓***		

Table 1: Comparison of selected diversification solutions, according to the aspects discussed in Section 2. A diversity notion fulfills **Range** if it can diversify along a range of values (low to high) rather than just among categories, and **High-Dimension** if every candidate may be associated with a high number of properties. See Section 9 for more details on these solutions. Remarks: * Can support range coverage on a single dimension/property. ** Explanation for item relevance rather than subset diversity. *** Depends on the choice of distance function.

Implementation and experiments. We have implemented our solution in `Podium`, a prototype system including back-end implementation of our diverse user selection algorithm and a front-end that provides visualizations for our notions of explanations and user-friendly means of providing customization feedback (see Figure 1 for its architecture). We use this prototype to examine our approach over data from large-scale real-life user repositories. We first study the performance of our approximation algorithm, showing that it is effective in achieving diversity in terms of the selected user profiles according to the target function it approximates as well as multiple other diversity metrics. Next, we simulate opinion procurement using our algorithms (using ground truth user opinions), and test the diversity of procured opinions according to different metrics. Finally, scalability tests support the practicality of our algorithm for real-world data.

Paper Outline. In Section 2 we describe and motivate the desiderata from a diversification system in our context. Section 3 presents our model and basic problem definition, without customization, and in Section 4 we develop and analyze our solution for this basic problem. Next, we extend the basic solution to support the explanation and customization of the selection results in Sections 5 and 6 respectively. We describe our implementation of `Podium` in Section 7 and the experimental study conducted over it in Section 8. Section 9 discusses related work and we conclude in Section 10.

2 DESIDERATA

Diversification has been extensively studied in multiple contexts; we claim that diversification in the concrete context of opinion procurement has a *unique combination of traits*, which are not accounted for by previous work. We compare several representative previous solutions under the prism of these traits in Table 1. Next, we explain these features as well as the desiderata of diversification that follow; further detailed comparison with related work is given in Section 9.

Coverage vs. distance-based. A prominent approach for diversification is to quantify the (dis)similarity between items, and to then aim at finding items that optimize some aggregate function over the similarity scores, for instance, maximizing the minimal pairwise distance (e.g., [4–7]). Such an approach is valid in our setting, yet its sensitivity to skews in group sizes may yield less meaningful results for real-life datasets, as observed in our experimental results for the Yelp dataset in Section 8.

When it comes to gathering user opinions, a natural desideratum is that opinions are collected from users that in some sense faithfully represent the characteristics of the full population. Such representativeness is targeted by *coverage-based* approaches in different selection contexts – e.g., retrieving documents that cover the topics in a repository, or users that represent predefined groups within a source population (e.g., [1–4]). In contrast with distance-based approaches, coverage-based approaches can in particular be agnostic of the similarities within the selected subset.

We next define the *proportionate-allocation* user subset.

Definition 2.1. Let $\mathcal{G} \subseteq P(\mathcal{U})$ be a set of user groups. A user subset $U \subseteq \mathcal{U}$ is a *proportionate allocation* of $\mathcal{G}$ if for every $g \in \mathcal{G}$, it holds that

$$\frac{|g \cap U|}{|U|} = \frac{|g|}{|\mathcal{U}|}$$

A user subset for which this definition holds faithfully represents the source population in the sense that it has a number of selected representatives from each group that is proportionate to their number in the population. This trait is used by surveyors in *stratified sampling* to guarantee that certain inferences from the survey are statistically sound (e.g., [8, 9]). For that, surveyors and domain experts carefully define a small set of non-overlapping population groups to be represented (in particular, $|U| \geq |\mathcal{G}|$). See further discussion on surveys in Section 9.

However, in this work we consider user repositories that often form a huge number of highly overlapping user groups, *making proportionate allocation infeasible.* A user subset of size $|U| << |\mathcal{G}|$ with every group even roughly proportionally represented is unlikely to exist. We therefore develop, in the following sections, solutions for a relaxed problem formulation, in particular, aiming to avoid under-representation of groups but allowing over-representation and prioritizing the coverage of certain groups over others.

Intrinsic vs. predicted. Intrinsic diversity is computed based only on known properties (e.g., [2, 4, 5, 7]), whereas *predicted* diversity utilizes a function predicting unknown values for each selected item (e.g., a probabilistic distribution of the answer to some question) [1, 3, 4, 6]. Thus, predicted diversity notions typically optimize an expected target function (e.g., the expected number of different answers to be obtained).

In opinion procurement scenarios, the intrinsic approach, i.e., relying on user profiles rather than prediction of their opinions, is often preferable. First, the representation of different groups in the population may be the main client need, regardless of what their opinions are (e.g., having representatives for as many genders, age groups, nationalities, etc. as possible). Second, obtaining a reliable prediction of user opinions may be impractical – at least as hard as the original opinion procurement task. When this is the case, users with diverse profiles may provide a good alternative, since they are likely to provide relatively diverse opinions (as demonstrated in our experimental results in Section 7 and in [4]).

Diversification along a range of opinions. Diversification for opinion procurement is characterized by the need to diversify *along ranges of property values* – for example, one has to represent the full range of user opinions, from negative to positive; the full range of user activity or expertise levels, from low to high; users of all ages; etc. In contrast, diversification solutions that target the maximization of user skill or item relevance in diverse categories (as in, e.g., [1, 3]) are not applicable for capturing the full range of (skill/relevance) values in each category.

Support of high data dimensionality. In large-scale user repositories, each profile may consist of hundreds to thousands of properties (e.g., up to 2189 properties per user in the TripAdvisor dataset used in Section 8). Using such properties along with ranges of values associated with them (e.g., frequencies of some activity from lowest to highest), allows defining a huge number of meaningful population groups, larger by orders of magnitude from the number of selected representatives. A practical diversification solution should address this dimensionality problem either by significantly reducing the number of considered groups and/or by adopting a diversity notion and implementation that scale with the problem dimension.

Explanations and customization. Last, we have already noted (in the Introduction) that there is no one-size-fits-all solution for diversification and that different clients may have different diversification needs. To be able to fine-tune the diversification results, the clients must first be able to understand them - via some notion of *explanations* – and then have user-friendly *customization* mechanisms of modifying them according to their needs. The use of intricate optimization problems and/or interdependencies between selected items, which often makes sense the context of diversification, as well as the high scale and dimension make this desideratum nontrivial to achieve. Here, we address this challenge by adopting a simple diversification notion based on profile properties, which in turn are human-understandable, and then explanations and customization pertain to (modifying) how these properties are represented by the selected subset. (See Sections 5-6.)

In the following sections we describe our model and algorithmic solutions, achieving these desiderata.

3 MODEL

We next describe how user profiles are modeled in our framework. We then formally define the problem of diverse user selection with respect to this model.

Property	**Alice**	**Bob**	**Carol**	**David**	**Eve**
livesIn	Tokyo$^{(2)}$	NYC$^{(1)}$	Bali$^{(1)}$	Tokyo	Paris$^{(1)}$
ageGroup	50-64$^{(2)}$	–	50-64	–	–
avgRating Mexican	0.95$^{(3)}$	0.3$^{(1)}$	–	0.75	0.8
visitFreq Mexican	0.8$^{(1)}$	0.25$^{(1)}$	–	0.6$^{(2)}$	0.45
avgRating CheapEats	0.1$^{(1)}$	0.9$^{(1)}$	0.45$^{(2)}$	–	0.6
visitFreq CheapEats	0.6$^{(1)}$	0.85$^{(1)}$	0.2$^{(2)}$	–	0.3

Table 2: Example user profiles

3.1 User Profiles

Let $\mathcal{U}$ be a population of users and $\mathcal{P}$ be some domain of property labels. Following [10], we define the profile of a user $u \in \mathcal{U}$ as a tuple $D_u = \langle P_u, S_u \rangle$ where $P_u \subseteq \mathcal{P}$ includes all the properties known for u and $S_u : P_u \to [0, 1]$ maps each property to a score (normalized to $[0, 1]$). We use the notation $|p| = |\{u \in \mathcal{U} \mid p \in P_u\}|$, where $\mathcal{U}$ is assumed to be clear from the context. Property scores may have different interpretations depending on the type of property, e.g., true/false, user rating, and so on, and may be provided directly by u or automatically derived from u's activity in the website.

Example 3.1. Table 2 shows a few profiles from a travel website (for now, ignore the numbers in superscript). In the first two rows, `livesIn <city>` and `ageGroup <X-Y>` are true/false properties for relevant cities and age ranges. E.g., `livesIn Tokyo` is a property with score 1 (i.e., true) in Alice's profile. The third and fifth rows show scores that reflect the user average ratings for different types of restaurants, normalized to $[0, 1]$. Not every property is recorded for every user, e.g., Carol has never rated Mexican food. The fourth and sixth rows show scores reflecting the relative frequency that each of the users visits different types of restaurants.

In practice, user profiles may contain many properties – e.g., we have constructed from TripAdvisor[2] a user repository with up to 665 properties per user (Section 7). This is due to various activities of a user in the system (e.g., providing opinions about many destinations, each with many different features), due to various types of analysis performed over the data (e.g., one can compute the average rating, maximum rating...) and so on.

Using taxonomies to enrich profiles. To allow for an informed selection of users based on their profiles, these profiles should be as complete as possible. To this end, we perform a preprocessing step and apply *inference rules* on Boolean properties or on the raw data from which properties are derived. Such inference rules can be pre-specified as in RDF languages [11, 12] or derived via rule mining techniques [13]. A particularly useful type of inference rules is *generalization rules*, as exemplified next.

Example 3.2. The property `avgRating Mexican` in Table 2 is derived by averaging over the ratings given by each user to restaurants labelled as "Mexican Cuisine". On this raw data, we can apply a generalization rule if we know, e.g., by a cuisine taxonomy, that Mexican cuisine is a particular type of Latin cuisine. This will enable us to derive properties such as `avgRating Latin` for existing user profiles.

As another example, if `livesIn` is known to be a function, i.e., each person can only have one residence location in our repository, we can infer the falsehood of residence locations other

[2]TripAdvisor website: https://www.tripadvisor.com

than the one specified. Thus, by $S_{\text{Alice}}(\texttt{livesIn Tokyo}) = 1$ we can infer that $S_{\text{Alice}}(\texttt{livesIn X}) = 0$ for every X≠Tokyo.

Having inferred all possible properties, we consider all other properties by the *open world assumption*: missing properties may be either false or true. For instance, if no frequency of visiting Mexican restaurants is known for Carol, this does not mean she has not been to such restaurants.

3.2 Weight-based Diversification

We next define a generic, weight-based approach to coverage-based diversification. We exemplify different choices of weights and show their usefulness for capturing user selection strategies.

Definition 3.3. A *diversification instance* is a tuple $(\mathcal{G}, \text{wei}, \text{cov})$ where $\mathcal{G} \subseteq 2^{\mathcal{U}}$ is a set of (possibly overlapping) user groups of interest, $\text{wei} : \mathcal{G} \rightarrow \mathbb{R}^+$ captures the weight of each group, and $\text{cov} : \mathcal{G} \rightarrow \mathbb{N}$ captures the number of users required so that a group is said to be covered.

Given a diversification instance and a selected user set $U \subseteq \mathcal{U}$, we define the score of U as $\text{score}_{\mathcal{G}}(U) = \sum_{G \in \mathcal{G}} \text{wei}(G) \cdot \min\{|U \cap G|, \text{cov}(G)\}$.

Finally, given a diversification instance and a budget $B \in \mathbb{N}$, we define BASE-DIVERSITY as the problem of finding a subset $U \subseteq \mathcal{U}$ such that $|U| \leq B$ and $\text{score}_{\mathcal{G}}(U)$ is maximized.

Note that if groups in $\mathcal{G}$ are overlapping, each user may contribute multiple group weights to the total score. This definition accounts for diverse subset selection in the sense that the score increases as more groups in $\mathcal{G}$ have (more) representatives in U. *Excessive* representation ($|U \cap G| > \text{cov}(G)$) is not rewarded but also not penalized.

The problem is defined in a generic way with the diversification instance given as input. We next discuss and exemplify the three parts of this instance.

Groups. Our diversification solution can support any set of groups input by the client, including manually crafted groups as typically defined by surveyors [8, 9].

To support large-scale, high-dimensional user repositories we develop here a concrete group definition that is efficiently computable for such repositories on the one hand, and effective in identifying meaningful groups for diversification on the other hand. Recall that user profiles comprise of properties from $\mathcal{P}$ with scores in $[0, 1]$.

Definition 3.4. Let $p \in \mathcal{P}$ be a property and $b \subseteq [0, 1]$ be a (continuous) range of scores. A *simple user group* is the subset of users whose score for p falls in b, formally,

$$G_{p,b} := \{u \in \mathcal{U} \mid D_u = \langle P_u, S_u \rangle \wedge p \in P_u \wedge S_u(p) \in b\}$$

For the ranges of scores, we split the range of scores of each property $p \in \mathcal{P}$ into a set of *non-overlapping buckets* $\beta(p)$. The rationale is, e.g., to group Mexican food lovers and dislikers separately. The computation of $\beta(p)$ is done by partitioning the 1-d data into intervals (clusters). There are several methods for 1-d interval splitting that are more effective than general clustering since the data is ordered (e.g., Jenks optimization [14], K-means, Expectation Maximization and by kernel density).

Simple user groups can be used to define more complex ones as the intersection or union of a few simple groups.

We note that the simplicity of our group definition is key for allowing explanations (see Section 5). There are more complex alternatives to splitting ranges into groups, such as multidimensional clustering (in our case, over multiple properties); however, these generally do not facilitate explainability. For instance, multidimensional clusters have no intuitive "label" or meaning.

Example 3.5. Reconsider Table 2. Let p be the property `livesIn Tokyo` and $b = [1, 1]$; then $G_{p,b} = \{\texttt{Alice}, \texttt{David}\}$ (group of "Tokyo residents"). Let p' be the property `avgRating Mexican` and $b' = (0.65, 1]$; then $G_{p',b'} = \{\text{Alice, David, Eve}\}$ (group of "Mexican food lovers"). One can also define, e.g., $G_{p,b} \cap G_{p',b'} = \{\texttt{Alice}, \texttt{David}\}$ ("Tokyo Residents who are also Mexican food lovers").

Our default definition of $\mathcal{G}$ consists only of simple groups, and we examine its effectiveness in Section 8. In particular, we empirically show that this approach also implicitly accounts for more complex groups in the population (such as 'Tokyo Residents who are also Mexican food lovers" from the example above).

Group functions. Similarly to group definition, the group weights (wei) and cover sizes (cov) functions can in principle be manually tailored for a specific domain and diversification context. As a more practical alternative, we next propose a few general-purpose choices, which can be fine-tuned by clients via our customization mechanism (see Section 6).

Definition 3.6. Weights are used to prioritize groups. The following are three examples of $\text{wei}(G)$:

- *Identical Group Importance (Iden):* $\text{wei}(G) := 1$ (constant function).
- *Group Importance Linearly By Size (LBS):* $\text{wei}(G) := |G|$.
- *Group Importance Enforced By Size (EBS):* define $\text{ord}(\cdot)$ as an ordering of the groups from smallest to largest, [3] then define $\text{wei}(G) := (|U| + 1)^{\text{ord}(G)}$

Iden is the most "diverse" choice in the sense that it does not distinguish between groups, which by our problem definition will maximize the number of groups that are covered. However, in cases where only a small fraction of the groups can be represented/covered, one may choose to prioritize certain groups – e.g., large groups. Using LBS, the group importance is linear with its size, thus, e.g., the total weight of two groups of size X equals the weight of one group of size $2X$. This roughly corresponds to maximizing the number of groups represented *per user.* In EBS group importance by size is enforced, meaning that representing larger groups is always preferred over smaller ones. The latter requirement may apply to some diversification contexts, e.g., political surveys may aim to have at least one representative for each of the largest population groups.

Definition 3.7. The coverage function $\text{cov}(G)$ is used to guide how many users will be selected from each group. Examples include

- *Single Representative (Single):* $\text{cov}(G) := 1$ (constant function).
- *Proportional Representation (Prop):* $\text{cov}(G) := \max\{\lfloor |U| \cdot |G| / |\mathcal{U}| \rfloor, 1\}$ where $|U|$ is the size of the subset to be selected.

Here, Single is the most "diverse" definition in the sense that it requires only one representative from a group to consider it covered. In contrast, Prop rewards a representation that is proportional to the group size in the population.

We next exemplify the effect of using different functions on the resulting user choices.

Example 3.8. Reconsider the user profiles in Table 2 and assume that we define, for each property, three groups of users:

[3]Ties, i.e., groups of equal size, are broken arbitrarily.

those with scores in $[0.65, 1]$ ("high"), in $[0.4, 0.65)$ ("medium") and in $[0, 0.4)$ ("low"). The numbers in superscript at the table show the weights according to LBS – i.e., number of users – on the first user of each group. E.g., the only group with 3 users is `avgRating Mexican high`. The diverse user subset of size 2 that would be selected is {Alice, Eve} with total score 17. Single and Prop behave similarly here, and EBS would yield the same result with different scores. If instead we use Iden, then {Alice, Bob} will be selected with total score 11 (number of represented groups). This exemplifies the tendency of Iden to select more eccentric users, in this case Bob who is the only member of his groups, where LBS and EBS prioritize representatives of larger groups, in this case leading to a larger overlap (Alice and Eve are both Mexican food lovers).

Having defined our model, we next address the computational problem of BASE-DIVERSITY.

4 SOLVING BASE-DIVERSITY

We next consider the computation of a diverse subset of users according to Def. 3.3 of the BASE-DIVERSITY problem. We start by analyzing the complexity of the problem.

Unsurprisingly, we show that achieving an optimal solution is intractable in the subset size B unless P = NP, even for simple weight functions and even without customization. The decision problem DEC-DIVERSITY corresponding to BASE-DIVERSITY is that of the existence of a subset U with $|U| \leq B$ such that the sum of (customized) weights of covered groups exceeds a threshold T. We can then show:

PROPOSITION 4.1. DEC-DIVERSITY *is* NP*-complete in B.*

PROOF. Membership is immediate, since computing the total weight of a given user subset is in PTIME.

Hardness is proved by a reduction from Set Cover: Given a universe $\{1 \ldots N\}$, a set of subsets $\{S_1, \ldots, S_m\}$ and an integer k, we define $B = k$, $\mathcal{G} = \{G_1, \ldots, G_N\}$ and $\mathcal{U} = \{u_1, \ldots, u_m\}$, such that iff $i \in S_j$, then $u_j \in G_i$. Finally, we set $T = \sum_{G \in \mathcal{G}} \text{wei}(G) \cdot \min\{\text{cov}(G), B\}$, where wei$(G)$ can be any legal function and we set cov(G) as the constant function 1 (Single, as we need only one set to cover each element). Since T is the maximum total score achievable, by covering every group in $\mathcal{G}$, it will be achieved by and only by a user subset that corresponds to a Set Cover. □

Approximate solution. The reduction from Set Cover implies not only the intractability of an exact solution but also of a constant-factor approximation in terms of the size of the *covering group.* To formalize this, given an instance of BASE-DIVERSITY and a threshold score T, let opt(T) be the minimal size of a subset $U \subseteq \mathcal{U}$ whose score exceeds T. We then have, based on [15] inapproximability result for set cover:

PROPOSITION 4.2. *Assuming $P \neq NP$, there is no PTIME algorithm for* BASE-DIVERSITY *that given a threshold score T, finds a user subset U of size* $(1-O(1)) \cdot ln(|\mathcal{G}|) \cdot \text{opt}(T)$ *with* $\text{score}_{\mathcal{G}}(U) \geq T$.

Fortunately, this does not exclude the possibility of approximation in the second axis, namely achieving a near-optimal score while conforming to the given budget. Indeed, a simple greedy algorithm achieves a constant approximation ratio in this sense.

Algorithm 1 outlines this greedy selection. Its input is a repository of users, a bound B on the number of users and a diversification instance (groups, weight function and coverage function). The algorithm starts by initializing an empty U (line 1) and computing, for each user the value $\text{marg}_{u,U}$, which stands for the

Algorithm 1: Greedy User Selection

```
Input: $\mathcal{U}$, $B$, $\mathcal{G}$, $wei$, $cov$
Output: $U$ (a set of $\leq B$ users)
1  $U \leftarrow \emptyset$;
2  foreach $u \in \mathcal{U}$ do $\text{marg}_{u,U} \leftarrow \sum_{G \in \mathcal{G} | u \in G} \text{wei}(G)$ ;
3  for $i \in 1..B$ do
4      if $\mathcal{U}$ is empty then break;
5      maxUser $\leftarrow \arg\max_{u \in \mathcal{U}} \text{marg}_{u,U}$;
6      $U \leftarrow U \cup \{\text{maxUser}\}$, $\mathcal{U} \leftarrow \mathcal{U} - \{\text{maxUser}\}$;
7      foreach Group $G$ such that maxUser $\in G$ and $\text{cov}(G) > 0$
        do
8          $\text{cov}(G) \leftarrow \text{cov}(G) - 1$;
9          if $\text{cov}(G) = 0$ then
10             foreach $u \in G$ do $\text{marg}_{u,U} \leftarrow \text{marg}_{u,U} - \text{wei}(G)$;
11 return $U$
```

potential marginal contribution of u to the total score if added to U (line 2). The algorithm then iteratively selects B users. Unless $\mathcal{U}$ is empty (line 4), the user maxUser with the greatest marginal contribution is selected (line 5) and moved from $\mathcal{U}$ to U (line 6). For each group G covered by maxUser, its required coverage cov(G) decreases by 1 (line 8), and if no more representatives are required to cover G (cov$(G) = 0$) then G should have no effect on the selection of the following users. We thus, subtract wei(G) from the marginal contribution of its other members (line 10). After B iterations (or earlier, if $|\mathcal{U}| < B$) the algorithm returns U.

Data Structures. For efficiency, we represent both the groups and the users as lists, each group $G \in \mathcal{G}$ with its current wei(G) and cov(G) values, and each user $u \in \mathcal{U}$ with $\text{marg}_{u,U}$. We further keep links in both directions between the lists, from groups to their members and vice versa. Whenever we (re)compute $\text{marg}_{u,U}$ we can remove the links from the user to groups with weight 0 or coverage size 0, which are not (or no longer) relevant for the user selection, to improve the performance of subsequent computations.

Example 4.3. We next exemplify the execution of Algorithm 1 for the user selection scenario in Example 3.8, using LBS and Single. After executing line 2 the marginal contributions of Alice, Bob, Carol, David and Eve, namely, the sum of weights of their properties, are 10, 5, 7, 6 and 10 respectively. Assume that at the first iteration of the external loop Alice is chosen and removed from $\mathcal{U}$ to U (ties are arbitrarily broken; in this example, selecting Eve happens to lead to the same output). Then the coverage of each of Alice's groups is set to 0. For each such update, the marginal contribution of other members of the groups is reduced: first, the contribution of David is reduced by 2 due to the `livesIn Tokyo` group; next, the contributions of David and Eve are reduced by 3 due to the `avgRating Mexican high` group; and so on. At the end of the first iteration, the contributions of Carol, David and Eve are updated to 5, 2 and 7 respectively. Thus, Eve is chosen at the next iteration, and {Alice, Eve} would be the output, which in this case is also the optimal solution.

PROPOSITION 4.4. *Algorithm 1 computes a* $(1-1/e)$*-approximation of* BASE-DIVERSITY*, i.e. achieves a score that within a multiplicative factor of at least* $\geq (1 - 1/e)$ *of the optimal for the given budget, in time* $O(B \cdot \max_{G \in \mathcal{G}} |G| \cdot \max_{u \in \mathcal{U}} |\{G' \in \mathcal{G} \mid u \in G'\}|)$.

PROOF. The complexity of Algorithm 1 is $O(B \cdot |\mathcal{U}| \cdot |\mathcal{G}|)$ due to the updates of the marginal user contributions (line 10). This line is nested within three loops. The loop line 3 repeats O(B) times, the loop at line 7 repeats O($\max_{u \in \mathcal{U}} |\{G' \in \mathcal{G} \mid u \in G'\}|$) times, namely, bounded by the maximal number of groups per user, and the innermost loop (line 10) repeats at most O($\max_{G \in \mathcal{G}} |G|$) times, namely, bounded by the size of the largest group. We assume constant complexity for arithmetic computations and for getting the next group of a given user/next user of a given group (as links in both directions are maintained).

As for the approximation ratio, observe that the score function satisfy the following properties *regardless of the choice of wei, cov*:

- *Submodularity.* For any $U \subseteq U' \subseteq \mathcal{U}$ and $u \in \mathcal{U}$ we have $\text{score}_{\mathcal{G}}(U \cup \{u\}) - \text{score}_{\mathcal{G}}(U) \geq \text{score}_{\mathcal{G}}(U' \cup \{u\}) - \text{score}_{\mathcal{G}}(U')$.
- *Non-negativity.* $\text{score}_{\mathcal{G}}(\cdot) > 0$ since wei(G) and cov(G) are positive.
- *Monotonicity.* If $U \subseteq U'$ then $\text{score}_{\mathcal{G}}(U) \leq \text{score}_{\mathcal{G}}(U')$.
- *Bounded input.* The size of a selected subset is bounded by B.

For such functions, a greedy algorithm that iteratively adds one user u to the selected subset U so as to maximize $\text{score}_{\mathcal{G}}(U \cup \{u\})$ is known to guarantee the stated approximation ratio [16]. □

Clearly, $\max_{G \in \mathcal{G}} |G| = \text{O}(|\mathcal{U}|)$ and $\max_{u \in \mathcal{U}} |\{G' \in \mathcal{G} \mid u \in G'\}| = \text{O}(|\mathcal{G}|)$. If we use only simple groups, the complexity bound of Prop. 4.4 may be simply written as $\text{O}(B \cdot \max_{G \in \mathcal{G}} |G| \cdot \max_{u \in \mathcal{U}} |P_u|)$.

5 EXPLANATIONS

We have proposed a simple generic framework for diverse user selection. We next consider notions of *explanations* of the diversification results, allowing clients to understand why certain users were selected and how certain groups were covered. This, in turn will enable the clients to use customization (see the next section) to refine these results.

Recall first that we have defined user profiles based on support values with respect to properties. We will use the set of property names in $\mathcal{P}$ to define *labels*; in practice, this entails that we will keep them in a human-readable form, and their combination will be used in presented explanations.

We further introduce labeling to simple groups, as follows. Each bucket is given a label, e.g. "low scores", "medium scores" and "high scores". Then, the *label* $G_{p,b}$ of each group can be constructed from the property name p and the label corresponding to the bucket b, e.g., "high scores for Mexican cuisine (average rating)".

We then define the notion of explanation to be presented to the client alongside the computed user subsets. Such explanations may be practically shown to users by visual means (see Section 7).

Definition 5.1. We introduce three types of explanations.

- *Group explanations.* Let $G \in \mathcal{G}$ be a group labeled l_G, we define its explanation as $\text{expl}(g) = \langle l_G, \text{wei}(G), \text{cov}(G) \rangle$, namely the property and bucket that defines it, along with its weight and required coverage.
- *User explanation.* The explanation of a selected user $u \in U \subseteq \mathcal{U}$ is defined as $\text{expl}(u) = \{G \in \mathcal{G} \mid u \in G\}$, namely, the groups which u represents.
- *Subset-group explanation.* Let $U \subseteq \mathcal{U}$ and $G \in \mathcal{G}$ be a selected user subset and a group. The explanation of how U covers G is the pair $\langle \text{cov}(G), |U \cap G| \rangle$, which represents the required versus actual coverage.

These explanations are complementary in the sense that they provide intuition about different aspects of the diverse selection: respectively, of the group meaning and importance; of why a given user was selected; and on how the selected user subset, as a whole, covers a certain group.

Example 5.2. Reconsider the selection of {Alice, Eve} in Example 3.8 in our running example. Assume that each property is given a human readable label, and we are further given labels for the buckets of Boolean properties and properties with a score. Group explanations may then be ⟨"high average rating for Mexican Cuisine", 3, 1⟩, since the weight of this group reflects its size, 3, and we use Single – one user to cover each group. "High" is the label of the bucket in range $(0, 65, 1]$. Similarly, we may have ⟨"lives in Tokyo", 2, 1⟩, where the label of the bucket $[1, 1]$ is empty for Boolean properties, and "lives in Tokyo" is the property label. Next, an explanation for Alice would be the groups she represents, "high average rating for Mexican Cuisine", "lives in Tokyo" and so on. The explanation for {Alice, Eve} with respect to the former group would be ⟨1, 2⟩, meaning both selected users belong to this group, exceeding the required coverage.

6 CUSTOMIZATION

Given the user selection results and their explanations, clients may fine-tune the algorithms if the results do not fit their needs. Specifically, we introduce customization at the level of individual groups (which, in a sense, correspond to the granularity of explanations that are shown). This customization is applied "on top" of the high-level decisions of how weights are assigned, which would typically not be made at the group level.

Definition 6.1. A *customization feedback* of the user is composed of four subsets of $\mathcal{G}$.

- $\mathcal{G}_+$: "must have" groups, each selected user must belong to all of them.
- $\mathcal{G}_-$: "must not" groups, each selected user must belong to none of them.
- $\mathcal{G}_d$: "priority coverage" groups, whose coverage is prioritized over others.
- $\mathcal{G}_{d?}$: "standard coverage" groups, whose coverage is of a lower priority with respect to the priority coverage groups.

Intuitively, the first two types of feedback serve to filter the repository of users. To avoid contradictions, if $\mathcal{G}_+$ contains more than one bucket of some property p, users need only belong to one of them. By default, $\mathcal{G}_+ = \mathcal{G}_- = \emptyset$. The priority and standard coverage group definitions allow to prioritize the coverage of certain groups, or completely ignore them in terms of coverage (groups in $\mathcal{G} - (\mathcal{G}_d \cup \mathcal{G}_{d?})$). By default, $\mathcal{G}_d = \emptyset$ and $\mathcal{G}_{d?} = \mathcal{G}$.

Example 6.2. Assume that for a particular application, the client prefers users from diverse locations and who are familiar with Mexican food. This may be captured by the following customization feedback:

- The "must have" groups consists of the three buckets of `AvgRating Mexican`, thereby requiring that the selected users have provided some rating for some Mexican restaurant.
- The "priority coverage" groups $\mathcal{G}_d$ consists of the multiple `livesIn <city>` properties.
- Finally, $\mathcal{G}_- = \emptyset$ and $\mathcal{G}_{d?} = \mathcal{G} - \mathcal{G}_d$.

We will demonstrate below how these choices guide user selection.

The effect of a customization feedback on the chosen groups is formalized as follows.

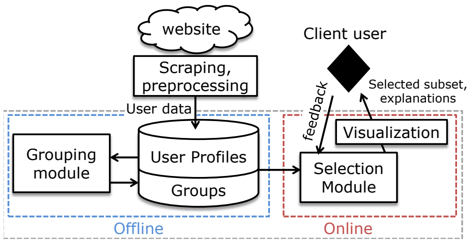


Figure 1: System Architecture

Definition 6.3. Given a customization feedback $\mathcal{G}_+, \mathcal{G}_-, \mathcal{G}_d$ and $\mathcal{G}_{d?}$, define the refined set of users as

$$\begin{aligned}\mathcal{U}' =& \{u \in \mathcal{U} \mid \forall G_{p,b} \in \mathcal{G}_+,\ \exists b' \in \beta(p):\ u \in G_{p,b'} \wedge G_{p,b'} \in \mathcal{G}_+\} \\ & \cap \{u \in \mathcal{U} \mid \forall G_{p,b} \in \mathcal{G}_-:\ u \notin G_{p,b}\}\end{aligned}$$

The customized diversity problem CUSTOM-DIVERSITY is then to select new subset $U \subseteq \mathcal{U}'$, of size $\leq B$, that maximizes $\text{score}_{\mathcal{G}_d}(U)$, namely, the sum of weights over covered groups from $\mathcal{G}_d$, breaking ties by $\text{score}_{\mathcal{G}_{d?}}(U)$.

Example 6.4. Reconsider the problem of selecting a user subset of size 2 from Example 3.8. We now incorporate the customization feedback of Example 6.2. The refined user set will exclude Carol who did not rate Mexican food. The best user subsets using Single and LBS functions is still {Alice, Eve}: first, it maximizes the sum of weights over `livesIn <city>` properties (to 3). Among other subsets that achieve this maximum (e.g., {Alice, Bob}), the selected subset further maximizes the sum of weights over other properties (to 14). Note that a different customization feedback would yield a different result; e.g., if we set $\mathcal{G}_{d?} = \emptyset$ then any subset maximizing the sum of weights over `livesIn <city>` properties may be selected.

Results revisited. CUSTOM-DIVERSITY is NP-complete, as an easy consequence of the NP-completeness of BASE-DIVERSITY. Further, the counterpart of Proposition 4.4 holds:

Proposition 6.5. CUSTOM-DIVERSITY *may be approximated within a multiplicative factor of at least* $(1 - 1/e)$ *in time* $O(B \cdot \max_{G \in \mathcal{G}} |G| \cdot \max_{u \in \mathcal{U}'} |\{G' \in \mathcal{G} \mid u \in G'\}|)$

Proof. The approximation algorithm is an adaptation of Algorithm 1 to account for customization feedback, as follows.

We first change the weights of the total score function to simulate a primary order by "priority coverage" groups and secondary order by "standard coverage" groups. The $\widetilde{\text{score}}(U) = \text{score}_{\mathcal{G}_d}(U) \cdot \text{MAX-SCORE} + \text{score}_{\mathcal{G}_{d?}}(U)$ where MAX-SCORE is a value greater than the maximum value of $\text{score}_{\mathcal{G}_{d?}}(U)$.

It now holds that:

Lemma 6.6. *The* $\widetilde{\text{score}}(U)$ *function is submodular, non-negative and monotone.*

We further refine the user repository to be $\mathcal{U}'$ of Definition 6.3, by filtering out user profiles that do not satisfy the conditions.

Last, we change Algorithm 1 so that instead of greedily selecting from $\mathcal{U}$ based on $\text{score}_{\mathcal{G}}(U)$, it selects from $\mathcal{U}'$ based on $\widetilde{\text{score}}(U)$. Following Lemma 6.6, the refined algorithm satisfies the approximation guarantees.

□

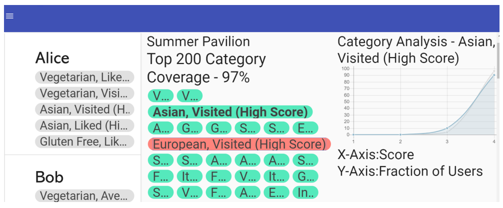

Alice
Vegetarian, Like...
Vegetarian, Visi...
Asian, Visited (H..
Asian, Liked (Hi...
Gluten Free, Lik...
Bob
Vegetarian, Ave...
Summer Pavilion
Top 200 Category
Coverage - 97%
V... V...
Asian, Visited (High Score)
A... G... G... S... S... E...
European, Visited (High Score)
S... S... A... A... A... S...
F... It... F... V... It... G...
S... V... F... A... E... In..
Category Analysis - Asian, Visited (High Score)
X-Axis:Score
Y-Axis:Fraction of Users

Figure 2: Screenshot of Podium UI: selection explanation

Explanations. The explanations defined in Section 5 can also be used for explaining customized results. The set of users and weights of groups may be different; in particular priority coverage groups will have a higher weight indicating a higher priority. Clients may not be able to interpret the values of weights, but they will be able to compare weights between groups to understand their relative importance.

7 IMPLEMENTATION

We developed Podium as a prototype system, implemented in Python using Flask[4]. Its architecture is depicted in Figure 1. The input to Podium is a set of user profiles, as explained in Section 3.1, in JSON format. Given a set of user profiles, the *Grouping Module* computes the bucketing of properties and the weights of groups in an offline process. Podium also allows an administrator to feed in an *initial set of diversification configurations* with associated textual descriptions.

The Graphical User Interface of Podium was created using AngularJS 1.6.4[5]. Given a user selection request, the *Selection Module* executes the user selection algorithm and returns the selected subset and its explanations to the client via the *Visualization* module. Figure 2 shows the explanation page for the initial configuration titled "Summer Pavilion", which only considers properties related to a restaurant in that name. The labels of the groups in this page are taken from the group explanations of Def. 5.1. The left pane displays the names of selected users, along with the top-weight groups that were covered by each (corresponding to user explanations of Def. 5.1). The middle pane uses the subset-group explanations of Def. 5.1 to show the percentage of top-weight relevant groups covered by the selected subset (in this case, 97%). The list of groups, ordered by decreasing weight, is displayed below with covered groups in green and the others in red.[6] When clicking any group, the right pane displays a graph comparing the score distribution for the relevant property between the entire population and the selected subset (in Figure 2 the distributions are almost identical). Users can browse the different groups and refine the selection by adding groups to $\mathcal{G}_+$ and $\mathcal{G}_-$ ("Selected users must / not have this property"); and to $\mathcal{G}_d$ and $\mathcal{G}_{d?}$ ("Do not / diversify on this property").

8 EXPERIMENTAL STUDY

We have examined the performance of our system, first, by evaluating the intrinsic diversity of the selected subset, i.e., how well it represents the source population (as explained in Section 2, proportional allocation is generally impossible in our setting). While an intrinsically diverse subset is sufficient in some user selection scenarios, in others one cares also for the eventual diversity of procured opinions. In order to examine this aspect, we

[4]Flask. http://flask.pocoo.org
[5]AngularJS. https://angularjs.org
[6]For space constraints, some group names in Figure 2 are truncated.

have selected datasets *with known ground truth*, i.e., where user opinions are already recorded. We have used these to simulate opinion procurement from the selected user subset and evaluate the diversity of collected opinions.

8.1 Datasets

The datasets used in our experimental study are real-world user repositories, focusing on the domain of restaurant reviews. The raw data is pre-processed to obtain aggregated scores for different categories based on user activity, as explained below.

The first dataset that we use consists of a sample of TripAdvisor [17] restaurant reviews data. This dataset contains data from 4475 users reviewing a total of 50K restaurants, and 11749 different groups. The raw data contains both user submitted data (e.g. age, residence) and user activity data (e.g. visited destinations), pre-processed and enriched as explained in Section 3.1, to generalize, e.g., Mexican cuisine to Latin cuisine.

The second dataset is the Yelp Open Dataset [18], which contains businesses, reviews, and user data for use in academic purposes. In our experiments we have used a subset of the data: for compatibility with the TripAdvisor dataset we used only restaurant-related data and took the 60K users with most reviews – reviewing a total of 52K restaurants and forming 8491 different groups. This limit was used in our *qualitative* experiments (see Section 8.4) due to memory limitations of some of the other baselines – recall that each user belongs to many groups. In comparison with the TripAdvisor dataset, the Yelp dataset has more users, but less groups due to its simpler semantics.

The datasets include two types of properties: ones that appeared explicitly in the original data, such as age and address; and ones that we have derived based on aggregation of user activities, as follows.

- *Average Rating.* The average rating given by a user to restaurants of a certain category (e.g. French cuisine), normalized by the overall average rating of that user.
- *Visit Frequency.* The fraction, among all the restaurants visited by a user, of restaurants from a certain category.
- *Enthusiasm Level.* A combination of rating and visit frequency, computed as the fraction of rating points given by the user to restaurants of a certain category.

8.2 Metrics

We next introduce metrics for algorithm performance, in three categories. *Intrinsic diversity metrics* are computed from the known properties of the selected user subset. *Opinion diversity metrics* are computed from opinions of the user subset, which are unknown to the user selection algorithms as explained in the beginning of this section. Finally, we evaluate the *scalability* of the algorithms.

Intrinsic diversity metrics. We consider a few complementary metrics, including our definition of total score – since our algorithm only approximates its optimal value – but also metrics of coverage that are not targeted directly by `Podium`.

- *Selection total Score.* According to Def. 3.3. We focus on the LBS weights and Single coverage functions, which our algorithm aims to approximate. This score can give us an intuition about alternative algorithms, since it reflects the number of groups and users within them that are represented by the subset.
- *Top-k groups coverage.* There are thousands of groups within the source population, which cannot be covered even by one representative in a small selected subset. We consider whether the top-k largest groups have selected representatives. In our experiments we have set $k = 200$.
- *Intersected-Property Coverage.* This metric is similar to the previous one, but now we consider intersections of simple groups that are at least as large as the k-th largest simple group.
- *Distribution Similarity.* This metric examines the similarity of user distribution between the source population and the selected subset, according to Def. 8.1 below.

The last metric aims at testing whether the number of representatives selected for groups is proportional to their number in the population, even if the coverage size is Single. Intuitively, our algorithm is likely to choose more representatives for larger groups without targeting it explicitly. However, standard distribution similarity metrics (such as Kolmogorov-Smirnov goodness of fit test) are not adequate for this purpose: to enhance coverage, small groups *must be over-represented.* We therefore define a distribution similarity metric that only taxes the selected user subset for under-representation of groups.

Definition 8.1. Let $B = b_1, \ldots, b_k$ be a discrete set of values. Let $f_{\text{subset}}, f_{\text{all}} : B \to [0, 1]$ be two functions over B, intuitively applied to the entire population and the selected subset respectively. We define the *coverage-oriented distribution similarity* (CD-sim, for short), as cd-sim$(f_{\text{subset}}, f_{\text{all}})$ =

$$1 - \frac{1}{k} \sum_{f_{\text{subset}}(b_i) < f_{\text{all}}(b_i)} \frac{(f_{\text{all}}(b_i) - f_{\text{subset}}(b_i))}{f_{\text{all}}(b_i)}$$

Note that this definition sums only over values of the domain for which the subset (f_{subset}) returns a lower result than the full population (f_{all}), corresponding to under-representation. Normalizing by the size of the full population guarantees that under-representations of larger groups are preferred, since the relative tax each missing user incurs is smaller.

For the group bucket distribution similarity, for a given property $p \in \mathcal{P}$, we set $B = \beta(p)$ (i.e., the set of buckets computed for p) and for $b \in \beta(p)$, we define $f_{\text{all}}(b) \mapsto \frac{\text{wei}(G_{p,b})}{\sum_{b' \in \beta(p)} \text{wei}\left(G_{p,b'}\right)}$ (the fraction of the weight that falls in the b bucket, which corresponds to the fraction of the users that belongs to this group). Similarly, we define $f_{\text{subset}}(b) \mapsto \frac{\text{wei}(G_{p,b} \cap U)}{\sum_{b' \in \beta(p)} \text{wei}\left(G_{p,b'} \cap U\right)}$ for a selected subset $U \subseteq \mathcal{U}$. For the overall distribution score, we average CD-sim for the top-20 largest groups.

Example 8.2. An example user distribution for the property "Mexican Food Average Rating" could be [0.23,0.4,0.37], meaning 23% of the population rate Mexican food poorly, etc. A selection distribution of [0.4,0.5,0.1] would receive a CD-sim score of 0.76, reflecting a penalty solely for the under-representation of the third sub-group, and not for the over-representation of the others.

Diverse opinion metrics. Thus far, the diversity metrics we considered were defined over user profiles. We next introduce metrics that consider the diversity of procured opinions. For that, we split the data into profiles used for selection, and data that simulates the procured opinions. For instance, we can select users from TripAdvisor based on their profiles excluding the data related to some destination, then evaluate diversity of the selected subset reviews on the excluded destination.

To measure diversity of opinions we have used complementary metrics that relate to the rating provided by the selected subset and their reviews' contents. Importantly, user opinions range not only over sentiment (positive or negative), but also over the facets that interest them with respect to the object in review.

- *Topic+Sentiment Coverage.* We measure content coverage using a list of prevalent topics extracted by TripAdvisor from each destination's reviews. We measure the fraction of topics that appear in the selected subset reviews. We also consider the review sentiment, such that 100% coverage means every topic appears in both a positive and a negative review.
- *Usefulness.* Available only for Yelp dataset, based on user feedback to reviews. A review is more useful when it is well-written, but also when a larger group of users agree or can relate to its contents. In this sense, the review is more likely to represent the opinions of large population groups, which is what we target in coverage-based diversity. We compute this metric by summing over individual reviews usefulness levels.
- *Rating Distribution Similarity.* Reusing our distribution similarity metric CD-sim, we measure the similarity in *rating distribution* between the selected subset and the entire population. For a given destination we set $B = \{1, \dots, k\}$ (i.e., the set of possible rating values) and for $i \in \{1, \dots, k\}$, let $R_i \subseteq \mathcal{U}$ be the set of users that gave this destination a rating of i. We define $f_{\text{all}}(i) \mapsto \frac{|R_i|}{\sum_{j=1}^{k} |R_j|}$. Similarly, for a selected subset $U \subseteq \mathcal{U}$ we define $f_{\text{subset}}(i) \mapsto \frac{|R_i \cap U|}{\sum_{j=1}^{k} |R_j \cap U|}$
- *Rating variance.* Variance of the rating given by the selected subset to a given destination.

All of the above metrics are defined per destination, to obtain an overall score we average over all destinations.

Scalability. We have tested the system execution times and scalability with respect to the number of users and profile size.

8.3 Baselines

We consider the following alternatives algorithms for diverse user selection.

- `Podium`. Our implementation as described above. By default, we use no customization feedback, LBS weights (Def. 3.6), the Single coverage function (Def. 3.7) and a budget $B = 8$, which also applies to the other baselines.
- *Random Selection.* An algorithm that selects a subset of the users uniformly at random. This method is a common practice in user selection for opinion procurement in the context of e.g. surveys, and under certain conditions there are reasons to assume the selected set of users is likely to be diverse. However, it has already been observed that explicitly managing diversity is often helpful in improving the results [4], which we will demonstrate in our setting.
- *Clustering.* Splitting the entire user repository into clusters, and choosing one representative from each – assuming each cluster represents a community. This approach has an inherent drawback as the clusters may have no intuitive explanation or customization; yet here we compare its performance to ours on other metrics. There are many options for clustering algorithms and representative choice. We have tested several options and show here one generally practical choice: computing B clusters using k-means (Scikit-Learn implementation[7]), then taking the near-mean user as the representative per cluster. k-means is particularly suitable to our settings: large, high-dimensional normally-distributed data, easy parametrization and is known to achieve comparatively high quality and low execution times (see, e.g., a comparison of clustering solutions in [19]).
- *Distance-based diversity.* While the distance-based approach for diversification has a different goal than coverage-based diversity (as explained in Section 2), it is still interesting to compare its performance to ours. As a representative distance-based baseline we use the S-Model of [4] via a greedy algorithm that maximizes the pairwise Jaccard distances between the properties of the selected subset.
- *Optimal Selection.* Naïve iteration over all user subsets of size B to obtain the optimal total score. This baseline is naturally applicable only for small values of B, and used to examine how good is the approximation achieved by our algorithm in practice, compared to the theoretical bound.

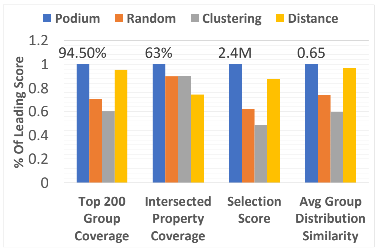
[7]Scikit-Learn. http://scikit-learn.org

8.4 Qualitative Results

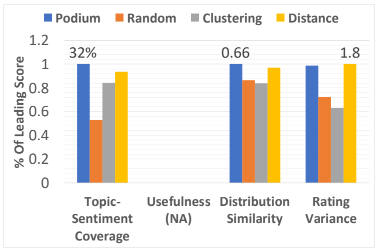
We next describe our experimental results regarding the achieved diversity. All experiments have been conducted on a Windows 10 machine powered by an Intel Core i7 7500U processor with a 16 GB of DDR4 memory.

Intrinsic diversity results. We depict the intrinsic diversity comparison between baselines for the TripAdvisor and Yelp datasets in Figures 3a and 3c, respectively. For showing different metrics on a similar scale, all scores are *normalized relative to the leading algorithm's score*; the value of the leading score is denoted on the relevant bar. Our main findings are summarized as follows.

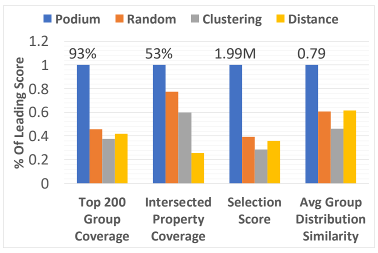
- `Podium` outperforms its alternatives in every tested diversity metric.
- Yelp is a more difficult dataset than TripAdvisor, since the former has less properties and less "room for maneuver"; for this dataset our results are better than the baselines by a significantly larger gap.
- Results for top-200 coverage and intersected property coverage indicate that our algorithm implicitly accounts for representing a high percentage of the largest groups, including complex ones – suggesting that selection based on simple groups may be sufficient for coverage purposes.
- The distance-based baseline performs poorly in covering complex groups, since it explicitly avoids intersections with overlapping properties between users.
- Surprisingly, our algorithm achieves a high similarity to the group distribution in the source population, although we do not optimize this directly.
- Our algorithm achieves the best total selection score by a large gap - this is expected, since our algorithm approximates the optimal value for this function.
- We were only able to test the optimal selection algorithm on a restricted source population and very small subset sizes due to the exponential runtime, hence it is omitted from the graphs. Generally, the total score achieved by `Podium` greatly exceeded the approximation bound and was near-optimal in all of our experiments. E.g., for selecting 5 out of 40 users `Podium` provided a .998 approximation ratio of the optimal.
- Since each user belongs to many groups, we can achieve high coverage even with a small B. As B increases, all the



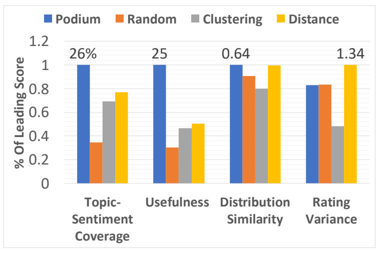
(a) TripAdvisor intrinsic diversity



(b) TripAdvisor opinion diversity



(c) Yelp intrinsic diversity



(d) Yelp opinion diversity

Figure 3: Quality experimental results

quality metric improve and the gaps between the baselines slightly decrease, but the general trends are preserved.

These results indicate that it is able to select good representatives of the sources population in different respects, covering most large groups and leaving few under-represented groups. Regarding the competitors, we observe that clustering is inferior in almost every metric; this indicates that the splitting of population into cluster is probably unable to identify meaningful groups, and is outperformed even by random sampling.

The results also indicate that distance-based selection is less able to represent groups not explicitly defined in the dataset. Generally, the main difference between the distance-based approach and ours is the pairwise intersection in user properties – e.g., 2 versus tens on average that we get for the Yelp dataset. Consequently, when there are a few prevalent categories that are shared by many users, the distance-based approach tends to seek the few users that do not have these categories, which comes at the expense of coverage and distribution similarity.

Opinion diversity results. We now consider whether indeed the selected user subset, by Podium and its alternatives, provides diverse opinions, according to the metrics defined in Section 8.2. Naturally, the considered groups in $\mathcal{G}$ may affect the opinion diversity for algorithms that rely on groups. In these experiments, we have chosen to consider groups that are defined from properties related to cuisine and location, as a client seeking opinions about a restaurant might have chosen.

For the TripAdvisor dataset (Figure 3b) we have examined 50 destinations with an average of 90 reviews per destination.

For the Yelp experiment (Figure 3d) we have considered 130 destinations with an average of 1730 reviews per destination.

Concluding both experiments, our main findings are:

- Podium achieves the best results in any tested metric for each dataset, with the exception of rating variance.
- Distance-based is the strongest competitor of Podium in this set of experiments; however, in the Yelp dataset we still see a significant gap w.r.t. Podium in topic coverage and usefulness.
- Podium achieves a good balance in the tradeoff between attaining dissimilar ratings/sentiments (as reflected in rating variance and distribution similarity) – which tends to the selection of "eccentric" users – and attaining representative opinions that cover prominent topics (as reflected in topic coverage, usefulness) – which tends to the selection of "mainstream" users.
- Random achieves a comparatively better performance in "dissimilarity" metrics (rating variance and distribution similarity), although still inferior to Podium and distance-based, and inferior results in "representativeness" metrics (topic-sentiment, usefulness), as expected.
- Clustering shows the opposite trends to those of Random, probably due to selection of near-mean users as representatives, which reduces the randomness of their selection but increases their representativeness.

These results reconfirm the assumption, proposed in previous work, that diverse users provide diverse opinions [4]. We have been able, by selecting a small user subset, to capture prominent topics and the ratings of the source population – even though Podium is not explicitly calibrated to predict opinions.

The effect of customization. We next consider the effect of customization on the selected user subset, with respect to the intrinsic quality metrics of the selected subset. We focus on the effect of "priority coverage" feedback from Def. 6.1. For that, we have selected from the Yelp dataset with 30K users, uniformly at random, four subsets $\mathcal{G}_{20} \subseteq \mathcal{G}_{40} \subseteq \mathcal{G}_{60} \subseteq \mathcal{G}_{80} \subseteq \mathcal{G}$ such that $|\mathcal{G}_i| = i$. Each subset was, in turn, fed into Podium as the set of priority coverage groups $\mathcal{G}_d$. Then, we have selected a user subset of size 8 in the customized setting. We have repeated this process 20 times and recorded the average for each metric.

The results are detailed in Figure 4, along with the intrinsic diversity metrics for the setting without customization, for comparison. Notably, all the quality metrics slightly decrease with

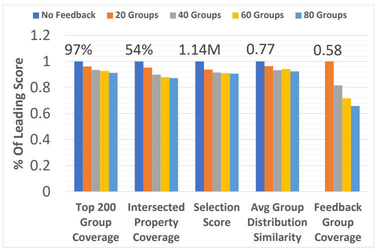


Figure 4: Yelp intrinsic diversity with customization

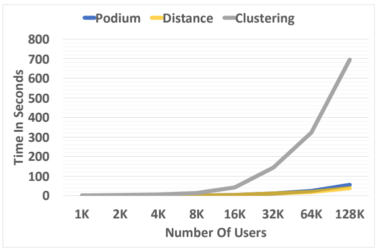


Figure 5: The effect of $|\mathcal{U}|$ on execution time.

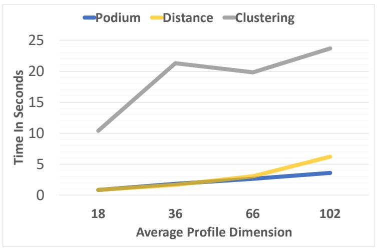


Figure 6: The effect of profile size on execution time.

every increase of the subset size, indicating that covering the priority groups restricts Podium's ability to cover standard priority groups – surprisingly, not by a significant gap. The newly-added *Feedback Group Coverage* metric measures the percentage of priority groups that were covered. Note that the groups are randomly selected with equal probability and are thus likely to be small and non-overlapping. Hence, there may not be 8 users who cover all of them. As expected, we can observe that the more priority groups are defined their coverage significantly decreases.

8.5 Scalability Results

We have examined the scalability of our algorithm w.r.t. the number of users and size of user profiles, which affect the number of groups. Here we only compare results with the clustering and distance-based baselines (random is immediate).

Scalability in number of users. In these experiments we have used user profiles with up to 200 properties. Following the complexity analysis in Section 4 we expect to witness a linear growth in the running time of the algorithm with accordance to the change in population size.

Scalability in profile size. The number of users has been set at 8K, and we varied the properties assembling the user profiles thus affecting their size. Again, we expect the running time to be linear to the average profile size.

Figures 5 and 6 depict the running times achieved by the algorithms. Our main findings are:

- Podium and distance-basedare ∼9 times faster than the clustering alternative.
- Execution time for Podium scales linearly in the size of the population as well as the number of properties.
- The Optimal baseline, due to its exponential complexity, demonstrated poor scalability. E.g., for $|\mathcal{U}| = 40$ and B=5 its execution time was 443 seconds, and for $|\mathcal{U}| = 100$ we have terminated its execution after an hour. It is therefore omitted from the graphs.

9 RELATED WORK

A comparison between diversification approaches is given in Table 1. We now elaborate more on these solutions and others.

Diversity in crowdsourcing. A few studies (e.g., [2–4]) have considered the selection of diverse users in the context of crowdsourcing, namely performing tasks with the collaborative help of Web users/workers. The work of [4] is the most relevant to ours since it also studies diverse opinion procurement. They present two approaches for diversification: S-Model is distance-based, where pairwise distance is assumed to be known; and T-Model is coverage-based on predicted data, i.e., targets the selection of a user subset with a certain opinion distribution, but only in a single category. Other studies consider the selection of diverse crowd workers in order to improve the overall accuracy. In [3] the authors study the selection of diverse users by modeling the dependence of error rates within access paths (corresponding to non-overlapping user groups), and optimizing the information gain by the selected subset. This, however, does not apply to opinion procurement where there are no errors and every opinion should be accounted for. The recent [2] resembles ours in considering coverage-based diversity and supporting customization. However, they consider only a single group per worker.

Diverse search results. Search results diversification has been extensively studied in the field of information retrieval (e.g., [1, 6, 20, 21]). Apart from solving query ambiguity, diversification is used to avoid over-personalization of search results [22]. The classification of diversity definitions as coverage-based versus distance-based is also considered in this context [23, 24]. In contrast with our approach, IR solutions generally target relevance and therefore are inadequate for diversifying along different axes and accounting for positive and negative opinions.

Diversity in recommender systems. Diversification has also been studied in the context of recommender systems. Diversity can be computed based on item properties [6] or collaborative filtering, namely, the ratings of similar users to similar items [5, 7]. Specifically, in [7] a notion of explanation-based diversity is presented, but is different than ours – certain item properties are identified as recommendation-relevant and these are used for diversification. In contrast, we do not assume that relevant properties are predefined but rather derive explanations from the actual diversification results. Moreover, to our knowledge, coverage-based approaches have not been considered in the context of recommender systems.

User sampling in survey research. The selection of people representing some population has been vastly studied in the context of surveys. While also concerned with opinion procurement, the focus of this research field is different. Specifically, as explained in Section 2, the dimensionality of user profiles in surveys is typically, by design, much lower than ours. This is because the goal of surveys is to ensure the statistical soundness of specific inferences from the participants' answers to larger populations [8, 9]. Statistical soundness may require the selected participants to be *proportionally allocated* (Def. 2.1), which, as explained in Section 2, is impossible in our high-dimensional setting due to the presence of many overlapping groups. Our approach involves a different problem formulation suitable for the high-dimensional setting. Also in contrast to surveys, which require a careful design and thereby a heavy load of manual curation, our solution applies to a given user repository as-is and may be easily executed multiple times, e.g., to incorporate data updates.

User selection. Various studies have considered the selection or filtering of users who undertake a task in crowdsourcing platforms or social networks. This includes assessment of crowd worker skill and filtering of low-skill workers [25]; filtering of low trust or spammer users [26]; filtering of slow or inefficient users [27]; expert finding [28–30]; and general-purpose declarative crowd selection [10, 31–33]. In general, these works are orthogonal to ours, since we can view the scores they derive as additional user properties that can be used for diversification.

A particular line of work considers *team formation* (or group formation), namely the selection of a set of workers that in some sense function as a team, by having e.g. complementary skills, similar properties, and/or better collaboration means [2, 34–37]. Among these, [2] is the most relevant to ours in targeting worker diversification, as discussed above. [37] uses coverage and diversity notions that our quite different than ours and thus render the problem and solution techniques quite different: diversity is considered between formed groups and is distance-based; and coverage is considered with respect to items rather than groups and does not support dimensionality.

10 CONCLUSION AND FUTURE WORK

In this work, we presented a framework for the selection of diverse user subsets for opinion procurement. We define a generic diversity notion that, while simple, satisfies a unique combination of desiderata that arise in presence of high-dimensional user profiles. In particular, as we showed, this notion admits efficient near-optimal computation and allows explanations and customization by the client. Our experimental study, on real user data, examines different metrics for diverse selection and shows that our algorithm outperforms a variety of baselines.

In future work, we plan to investigate further enhancement of the usability of our system, by methods of proposing relevant refinements for the user and by additional visualizations of the selection results. Another direction involves foundational study of the statistical properties of our algorithm: we have empirically shown that it performs well with respect to various measures other than our total score, e.g., distribution similarity and coverage of complex groups; the next step is formulating the guarantees for the algorithm performance in these metrics. The framework we have proposed is deterministic in choosing the (near-)optimal user subset by our definition, and is shown to outperform a fully random algorithm. Our implementation adds some randomness in randomly breaking ties, and we plan to further incorporation of randomness in our solution, e.g., adding noise to group weights, and its effect on the output diversity.

ACKNOWLEDGEMENTS

This work was funded in part by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Ministers Office, and by the Israel Science Foundation (grant No. 1157/16).

REFERENCES

[1] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong, "Diversifying search results," in *WSDM*, 2009.
[2] S. Cohen and M. Yashinski, "Crowdsourcing with diverse groups of users," in *WebDB*, 2017.
[3] B. Nushi, A. Singla, A. Gruenheid, E. Zamanian, A. Krause, and D. Kossmann, "Crowd access path optimization: Diversity matters," in *HCOMP*, 2015.
[4] T. Wu, L. Chen, P. Hui, C. J. Zhang, and W. Li, "Hear the whole story: Towards the diversity of opinion in crowdsourcing markets," *PVLDB*, vol. 8, no. 5, 2015.
[5] R. Boim, T. Milo, and S. Novgorodov, "Diversification and refinement in collaborative filtering recommender," in *CIKM*, 2011.
[6] M. Servajean, E. Pacitti, S. Amer-Yahia, and P. Neveu, "Profile diversity in search and recommendation," in *WWW*, 2013.
[7] C. Yu, L. V. S. Lakshmanan, and S. Amer-Yahia, "It takes variety to make a world: diversification in recommender systems," in *EDBT*, 2009.
[8] J. C. Helton and F. J. Davis, "Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems," *Rel. Eng. & Sys. Safety*, vol. 81, no. 1, 2003.
[9] P. H. Rossi, J. D. Wright, and A. B. Anderson, *Handbook of survey research.* Academic Press, 2013.
[10] Y. Amsterdamer, T. Milo, A. Somech, and B. Youngmann, "Declarative user selection with soft constraints," in *CIKM*, 2019, to appear.
[11] G. Klyne, J. J. Carroll, and B. McBride, "Resource description framework (RDF): Concepts and abstract syntax," *W3C rec.*, vol. 10, 2004.
[12] D. L. McGuinness, F. Van Harmelen *et al.*, "OWL web ontology language overview," *W3C rec.*, vol. 10, p. 10, 2004.
[13] L. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek, "Fast rule mining in ontological knowledge bases with AMIE+," *VLDB J.*, vol. 24, no. 6, 2015.
[14] G. F. Jenks, "The data model concept in statistical mapping," *International yearbook of cartography*, vol. 7, 1967.
[15] I. Dinur and D. Steurer, "Analytical approach to parallel repetition," in *STOC*, 2014.
[16] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functions—I," *Math. Prog.*, vol. 14, no. 1, 1978.
[17] "Tripadvisor website," 2018, https://tripadvisor.com/.
[18] "Yelp open dataset," 2018, https://yelp.com/dataset/.
[19] M. Z. Rodriguez, C. H. Comin, D. Casanova, O. M. Bruno, D. R. Amancio, F. A. Rodrigues, and L. da F. Costa, "Clustering algorithms: A comparative approach," *CoRR*, vol. abs/1612.08388, 2016.
[20] J. Carbonell and J. Goldstein, "The use of MMR, diversity-based reranking for reordering documents and producing summaries," in *SIGIR*, 1998.
[21] K. D. Onal, I. S. Altingovde, and P. Karagoz, "Utilizing word embeddings for result diversification in tweet search," in *AIRS*, 2015.
[22] F. Radlinski and S. T. Dumais, "Improving personalized web search using result diversification," in *SIGIR*, 2006.
[23] M. Drosou and E. Pitoura, "Search result diversification," *SIGMOD*, 2010.
[24] W. Zheng, X. Wang, H. Fang, and H. Cheng, "Coverage-based search result diversification," *Inf. Retr.*, vol. 15, no. 5, 2012.
[25] P. G. Ipeirotis, F. Provost, and J. Wang, "Quality management on amazon mechanical turk," in *HCOMP*, 2010.
[26] V. C. Raykar and S. Yu, "Eliminating spammers and ranking annotators for crowdsourced labeling tasks," *JMLR*, vol. 13, 2012.
[27] D. Haas, J. Wang, E. Wu, and M. J. Franklin, "Clamshell: Speeding up crowds for low-latency data labeling," *PVLDB*, vol. 9, no. 4, 2015.
[28] A. Bozzon, M. Brambilla, S. Ceri, M. Silvestri, and G. Vesci, "Choosing the right crowd: expert finding in social networks," in *EDBT*, 2013.
[29] J. Tang, J. Zhang, R. Jin, Z. Yang, K. Cai, L. Zhang, and Z. Su, "Topic level expertise search over heterogeneous networks," *Machine Learning*, 2011.
[30] J. Zhang, J. Tang, and J. Li, "Expert finding in a social network," in *DASFAA*, 2007.
[31] Y. Amsterdamer, T. Milo, A. Somech, and B. Youngmann, "December: A declarative tool for crowd member selection," *PVLDB*, vol. 9, no. 13, 2016.
[32] M. Martın, C. Gutierrez, and P. Wood, "SNQL: A social networks query and transformation language," in *AMW*, 2011.
[33] R. Ronen and O. Shmueli, "SoQL: A language for querying and creating data in social networks," in *ICDE*, 2009.
[34] M. Kargar, A. An, and M. Zihayat, "Efficient bi-objective team formation in social networks," in *PKDD*, 2012.
[35] T. Lappas, K. Liu, and E. Terzi, "Finding a team of experts in social networks," in *SIGKDD*, 2009.
[36] H. Rahman, S. B. Roy, S. Thirumuruganathan, S. Amer-Yahia, and G. Das, "Optimized group formation for solving collaborative tasks," *VLDB J.*, vol. 28, no. 1, 2019.
[37] B. Omidvar-Tehrani, S. Amer-Yahia, P. Dutot, and D. Trystram, "Multi-objective group discovery on the social web," in *PKDD*, 2016.

Elastic scaling in VectorH

Industrial Paper

Steffen Kläbe
TU Ilmenau, Germany
steffen.klaebe@tu-ilmenau.de

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

Stephan Baumann
Actian Germany GmbH
stephan.baumann@actian.com

Michael Rink
Actian Germany GmbH
michael.rink@actian.com

ABSTRACT

Cloud infrastructures allow to dynamically adapt to workload changes by provisioning additional resources or deprovisioning resources to reduce costs. This offers also opportunities for scalable distributed data management. However, elastic scaling in databases requires to migrate or even repartition data. In this work, we present an approach implemented in Actian's MPP solution VectorH that speeds up the elastic resizing process by minimizing partition reassignments while still achieving load balancing. Moreover, we describe a buffer matching and pre-filling technique to further increase performance after the resize step. The experimental evaluation shows that our solution significantly outperforms the non-elastic way of scaling using a system restart by a factor of 2 up to 4 and reduces downtimes during resizing to less than one minute.

1 INTRODUCTION

Cloud computing gained extraordinary importance over the past several years with the advent of enabling technologies like distributed systems, virtualization or fast network. While consumers moved their business applications to the cloud environment, important technology companies like Amazon, Google or Microsoft directed their focus towards cloud technologies, competing with each other for the leadership position in this promising new market. One of the decisive properties for the success of cloud computing is elasticity, providing users with flexibility to meet requirement changes in the fast moving technology world we live in. In the context of the cloud environment, elasticity describes a system's ability to adapt to changes in user demands and can be achieved in every system layer, e.g. storage, network or computing power. Therefore, developing software that is optimized for the cloud environment requires the software to support elastic changes in the underlying environment.

Actian Avalanche is the Software as a Service version of Actian VectorH [4], a massively parallel processing (MPP) solution for data analytics. The software is deployed in public cloud environments like Amazon web services (AWS) or Microsoft Azure, which both offer an elastic environment. As the VectorH MPP solution was originally designed to run on a static cluster, it does not exploit the opportunities provided by those environments. Furthermore, the currently implemented partition management is not designed to cope with non-static clusters and needs to be overhauled.

As the cloud environment offers elasticity, the main goal is to make VectorH able to utilize the provided elasticity of computing and storage resources. For this purpose, we focus on three major goals:

- **Develop an elastic resize feature:** Nodes need to be **efficiently** added or removed from a VectorH cluster during the system uptime, avoiding the drawbacks of a full restart. This is the key feature to enable elastic scaling.
- **Provide a partition strategy applicable for the cloud environment:** Partitioning is the key concept of VectorH to distribute work among nodes. As scaling the system becomes a frequent operation in the elastic environment, the partition strategy has to support elastic scaling by minimizing partition reassignments to nodes while providing load balancing.
- **Smoothen the performance after a cluster resize:** Although the partition strategy in intended to minimize partition reassignments, each cluster resize operation involves changes in data responsibilities. These changes should be transparent to the user by providing the full system performance once the scaling process finished.

The remainder of this paper is organized as follows. We give an overview over related work in Section 2, including a discussion of commercial products available on the market and how they provide elasticity. Section 3 provides an overview over the basic concepts of VectorH, before describing the design of the elastic scaling feature in Section 4. We compare different approaches for partition management and present the implemented approach in Section 5. Based on experiments, we describe the design of the buffer matching and filling mechanism in Section 6, which is an optimization on top of elastic scaling. Finally, we evaluate our solution in several experiments in Section 7, before concluding and giving an outlook in Section 8.

2 RELATED WORK

Achieving elasticity for cloud database systems is solved in different ways in the research community. ElasTraS [6] as an example is explicitly designed for the cloud environment and supports elasticity by the separation of storage and compute resources. The system distinguishes between high level transaction managers (HTM) and owning transaction managers (OTM). While HTMs handle user connections and execute queries using their local caches, OTMs are responsible for the actual data access. Depending on the current load, both HTM and OTM can be scaled independently and tasks are distributed among all running managers. As a different approach, Albatross [7] uses virtualization and live migration of databases to provide elasticity. The system aims at multi-tenant databases, which often face the problem of efficiently migrating single tenants instead of the whole system. To accelerate migration performance Albatross creates a database snapshot, which is placed in a network storage, and only migrates caches and active transaction states to a newly provisioned system. Afterwards, the system is switched using an atomic handover operation. In addition to elasticity, the system ensures serializability and correctness in failure cases. The experiments show that this approach does not abort any active transactions, harms query latency only in a negligible way and makes the database unavailable only for a small time window of about 300ms. A similar concept is used by ShuttleDB [1]. In contrast to Albatross, ShuttleDB can be seen as a middleware to make an arbitrary database management system elastic. It uses virtualization techniques to be transparent from the actual database instance and therefore works with common database systems without changes in their engine. On top of the live migration concept for single tenants, ShuttleDB offers automated elasticity. The system monitors query latencies of database tenants, automatically chooses tenants to scale and migrates them to another instance after deciding for a suitable migration strategy.

While ElasTraS, Albatross and ShuttleDB are mainly evolved from a cloud provider point of view, meaning maximizing utilization while fulfilling user demands, the Kingfisher system presented in [14] deals with elasticity from a customer point of view. Exploiting cloud pricing models, Kingfisher dynamically provisions virtual server capacity while being cost-aware. Using monitoring and forecasting of the query workload in combination with solving a linear optimization problem, the system minimizes infrastructure costs (e.g. cores, servers) and transition costs (time and costs to change environment).

In the field of commercial systems, Snowflake [5] was build from scratch for the cloud environment. It uses so called micro-partitions of several MB size to automatically partition and cluster data. Based on that, work distribution among nodes is realized using consistent hashing. In combination with work stealing, this approach automatically handles node failures as well as elastic scaling, while also minimizing the reassignment of micro-partitions as a property of the consistent hashing algorithm. As a second example, Amazon Redshift [9] divides compute nodes into the abstract concept of slices and assigns data to these slices. For scaling, the system offers two possibilities. While the "classic resize" deploys a new cluster in the background and sets the system in a read-only mode for several hours, the "elastic resize" reduces the downtime by saving a snapshot to the cloud file system, adding/removing nodes and reassigning work by reshuffling the abstract slices between nodes. This way, the downtime for the scaling process is reduced to several minutes. Nevertheless, user queries are on hold during the scaling. Third, Googles BigQuery relies on the concept of overpartitioning to avoid repartitioning in the elastic environment. In the case of scaling, tasks in the Dremel execution engine [13] are resized and data is read again from the storage layer, trusting in the speed of Google's Jupiter network technology and the Colossus storage system.

3 VECTORH OVERVIEW

Actian VectorH [4] is the scale-out version of the Vectorwise/X100 system [3] running on Hadoop clusters. It offers high and scalable query performance by exploiting opportunities of modern CPUs (e.g. SIMD, caching) with its vectorized execution engine. As a key for parallel processing of data the system uses hash partitioning and exploits co-located foreign-key joins for efficient node-local join processing, while partitions are assigned to nodes using a round-robin assignment. Furthermore, it reduces I/O costs by advanced compression methods and data skipping.

The Hadoop distributed file system (HDFS) is used as the storage layer and provides fault tolerance and scalability. Although HDFS is append-only, VectorH offers efficient updates by using Positional Delta Trees [11]. In order to efficiently read data from HDFS, the system is aware of data locality and replication. Nevertheless, processing data that is already in memory is another key to high query performance. Therefore, VectorH allocates a configurable bufferpool on system startup and maintains buffered blocks over different buffer policies [15].

For data exchange among cluster nodes, VectorH uses the Message Passing Interface (MPI) for implementing exchange operators described in the Volcano model [8]. The MPI library offers point-to-point communication as well as collective communication and is based on the concept of groups. Nodes within a group are identified using a rank starting at 0 and they are able to communicate with each other using a so called intra-communicator related to this group. In addition to that, so called inter-communicators allow communication among groups.

In order to scale a VectorH installation, the cluster configuration has to be changed and the system has to perform a restart, which has two major drawbacks. During the start process the system replays the write-ahead log, which might be a long-running operation depending on the log size. In addition to that, allocated memory is freed during the system shutdown. As a result, buffers are empty after a restart and data needs to be read again from storage, which impacts the performance of the first queries. Therefore, we need a solution to scale a running MPI application without performing a full restart. In addition to that, we want to avoid a performance degradation after the scaling operation by adapting the buffer management using the buffer matching and filling mechanism presented in Section 6 and by being aware of this issue when assigning work to nodes in a scaled environment.

Scaling can be invoked by the user to achieve one of the following goals. Either performance should be increased while keeping the data size fixed, or the system should be enabled to handle larger data sizes (while keeping performance constant). While the first goal refers to Amdahl's law, the second one is the use case for Gustafson's law (both in [10]). For VectorH, work distribution among cluster nodes is realized using partitioning, where the optimal number of partitions per table is approximately the total number of parallel threads the cluster offers. Assuming a homogeneous cluster, this is equal to the number of nodes multiplied with the number of physical cores per node. In order to support both mentioned scaling cases while avoiding an expensive repartitioning, we use the concept of overpartitioning, initially splitting tables into more partitions than necessary for the initial cluster configuration. The chosen number of partitions is crucial in terms of performance and the ability to scale the cluster size. Figure 1 shows the runtime of the TPC-H [2] query set on scale factor 1000 GB as a function of the number of partitions. The experiments were made on a cluster of 4 to 6 nodes with 24 cores each, providing a parallelism of 96 to 144 threads. First, the results show that increasing the cluster size while keeping the data size constant can lead to a performance benefit. Second, one can observe that increasing the number of partitions above the optimal partitioning of one partition per thread comes with an increasing performance penalty, which is caused by the introduction of an Union operator on top of table scans. Furthermore, increasing the number of partitions heavily impacts update performance, as update operations have to be performed on more fine-grained partitions. As a consequence, the number of partitions should not be chosen too large. Third, one can observe that VectorH is able to handle underpartitioning to a certain degree by performing node-local splits during table scans, achieving the assignment of one partition per thread. However, this resplitting harms node-local join processing and is therefore not a desired behaviour.

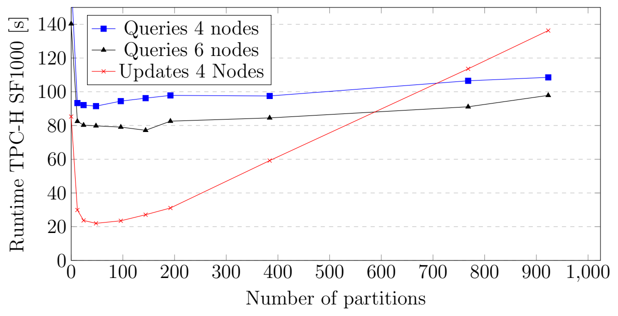


Figure 1: Dependency of Partitioning on TPC-H SF1000 runtime

4 ELASTIC SCALING

This section presents the design and implementation a dynamic cluster resize functionality for VectorH. With this feature we want to face the problem of scaling VectorH and solve the first goal stated in Section 1. Two new functions, *add_nodes* and *remove_nodes*, were implemented using the VectorH syscall functionality, allowing to issue commands against the system without forming SQL queries.

The new cluster resize functionality exploits the opportunities of the MPI group and communicator management. The basic approach of adding nodes using MPI routines is illustrated in Figure 2. Starting from an existing group of current nodes with an intra-communicator (A), a new group of processes is spawned by starting the application on the newly provisioned cluster nodes using the MPI_COMM_SPAWN routine. All nodes of the new process group are able to communicate with each other using their own intra-communicator (B) and with the old group of nodes using the inter-communicator (C) returned by the MPI routine. In order to abstract from these different types of communication in a second step, both groups form a new intra-communicator (D) using the MPI_INTERCOMM_MERGE routine and replace group communicators (A), (B) and (C). After new nodes are added in the *add_nodes* call, the master node has to broadcast the current partition mappings as described in Section 5 in order to provide the new nodes with the correct initial partition assignments. Afterwards the current nodes have to follow the new nodes' startup, as there are various synchronization points within the startup procedure. To simulate a collective start of all servers, the current servers have to perform all of these synchronizations to make the added servers finish their initialization, before performing the buffer matching mechanism described in Section 6. The cluster resize functionality has to be compatible with the following optimization made in VectorH. In order to reduce memory consumption, the creation of storage objects and minmax indexes is skipped for partitions a node is not responsible for. After adding nodes, this responsibility assignment changes as some partitions of the current nodes get assigned to the newly added ones. As a result, current nodes hold structures for partitions they are not responsible for anymore. Each node checks for these kind of unused structures by iterating over all tables in their catalogs, dropping information and freeing memory whenever possible. This could also be done in a lazy way by checking for unused information within a partition responsibility check during query execution. But as these checks are very frequent operations and are called multiple times within each query, the decision was made to cleanup the unused structures directly as part of the *add_nodes* operation.

Removing nodes reverses the presented mechanism. In the first step the nodes are divided into two distinct groups S and R with corresponding intra-communicators using a MPI_COMM_SPLIT routine. While nodes of group S perform a collective shutdown as a second step, the nodes of group R form the new cluster. Passing a list of hostnames to the function call, each node checks whether it is included in S and should terminate or not. It is currently not allowed to terminate the master node, so *remove_nodes* returns an error in this case. The remaining nodes (group R) now update their partition mappings as described in Section 5. Removing nodes assigns more partitions to the remaining nodes, and, as the nodes were not responsible for these partitions before, the creation of storage structures and minmax indexes was skipped. In order to reconstruct the missing information, an adapted replay of the write ahead log is performed. Within this replay, all log actions are skipped except those related to storage and minmax indexes for tables the node is now responsible for and was not responsible for before. This information is provided by the partition manager.

With the design of the elastic cluster resize feature we achieved the possibility to scale the VectorH cluster without restarting the existing nodes. We therefore ensured that the scaled system state is similar to the state before the scaling in terms of communication, metadata and catalog state. Furthermore, we automatically achieve support for updates that are resident in in-memory PDT structures

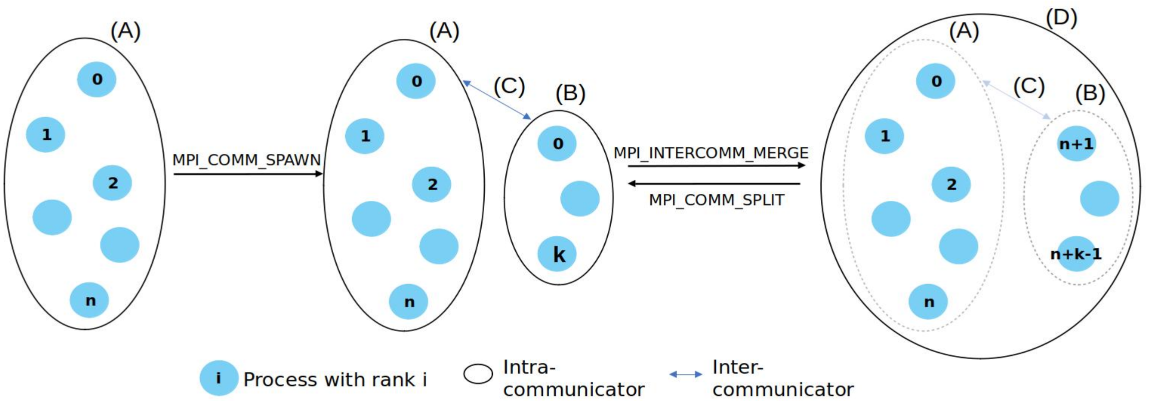


Figure 2: Schematic overview of cluster resize

mentioned in Section 3, as the log replay that is included in both *add_nodes* and *remove_nodes* also covers PDT log actions. As a result, the process reconstructs the update information needed to ensure data consistency.

Our implementation relies on the following assumption. For the *add_nodes* functionality binaries, configuration files and user data are accessible from the new nodes in the same way (especially using the same directory paths) as for the already running nodes. Triggering the operation over a front-end, e.g., a web-based management tool, would invoke starting the new nodes (dependent on the cloud service provider) and synchronizing necessary data before the *add_nodes* operation is started. A functionality for synchronizing the VectorH installation directory already exists and is used for cluster deployment, so this assumption is no restriction.

Besides assumptions the presented implementation comes with some limitations. First, the resize functionality must be issued in a separate session with no concurrent sessions. The system is able to block/hold incoming session requests when a scaling request is made. Second, the hosts to be removed are restricted to be the ones with the highest ranks. As all nodes except the master node are treated equally and the user does not call this function directly but through a frontend only providing the cluster size he wants to reach, the frontend can choose the nodes to remove as the ones that where added most recently. This ensures that only the highest ranks are selected. In case this is considered as too restrictive, one could extend the algorithm such that it first rearranges the nodes before the resize operation and assigns ranks in a way that fulfills the restriction. Third, it is currently not allowed to remove the master node as worker nodes are not able to replace a missing master node (which also holds for master node failure).

5 PARTITION MANAGEMENT

In this section, we present a partition management approach that is suitable for the elastic cloud environment, which corresponds to the second goal stated in Section 1.

5.1 Partition assignment approaches

We start by comparing basic approaches for partition assignment. The comparison is based on the following requirements, prioritized from most important to least important:

(1) **Load balancing:** Assigning an equal number of partitions to each node is crucial to achieve an optimal query performance. As partitions are already built using the partitioning method, the assignment strategy can only affect the number of partitions per node, not the size of each partition.
(2) **Lookup time:** The mapping *partition* $\rightarrow$ *node* is evaluated numerous times within each query and should therefore be an efficient operation.
(3) **Update time:** As resizing the cluster changes partition assignments, the structures of the partition assignment should be updatable in an efficient way.

Besides these main objectives, the partition assignment strategies must fulfill the following side conditions for performance reasons:

- Keep co-locality of foreign-key related tables
- Minimize the number of reassigned partitions on cluster resize

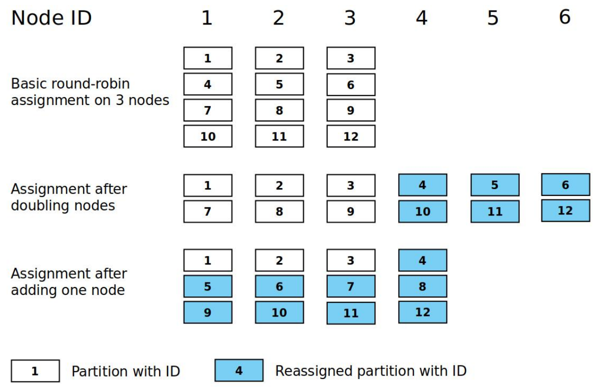
Keeping the co-locality of related tables is a key for achieving optimal query performance by exploiting node-local joins and is therefore an important demand. Reassigning partitions has several effects and should therefore be minimized. First, it leads to storage access for reading the partition, as the data is not present in the node's buffer. Second, nodes have to update their catalog information when becoming responsible or loosing the responsibility for a new partition, as described in Section 4.

We can state a lower bound for the minimum number of partitions that have to be reassigned on cluster resize. Let d be the total number of partitions for an arbitrary table and we assume that partition assignment is balanced before a resize operation. When adding n nodes to an existing cluster of n_{old} nodes with $n_{new} = n_{old} + n$, it is clear that every node has to be responsible for $\frac{d}{n_{new}}$ partitions after resizing to achieve load balancing. We assume that n_{new} is a divider of d and if not, every node gets one additional partition until the remaining partitions are assigned. As every new node get's $\frac{d}{n_{new}}$ partitions, the minimum total number of reassigned partitions is $\frac{d}{n_{new}} \cdot n$. For removing n nodes from an existing cluster of n_{old} nodes it can be easily seen that $\frac{d}{n_{old}} \cdot n$ partitions have to be reassigned, as every node was responsible for $\frac{d}{n_{old}}$ partitions before resizing. Overall we can state the lower bound of

$$reassigned\_partitions \geq \frac{d}{n_{max}} \cdot n$$

with $n_{max} = max\{n_{new}, n_{old}\}$ for adding/removing n nodes.

Round-robin assignment: Round-robin assignment is the currently used assignment strategy in VectorH. It is clear that this strategy provides load balancing and as it can be evaluated using an arithmetic operation, has a very fast lookup time while not needing additional data structures. Nevertheless, with respect to the additional cluster resize functionality, round-robin is one of the worst choices as it reassigns nearly all partitions when not resizing the cluster with a factor that is a power of two, as shown in Figure 3. Doubling the number of nodes leads to a reassignment of half of the partitions, which is basically the minimum for achieving load balancing. But increasing the number of nodes by a factor being an arbitrary number and not being a power of two leads to a reassignment of nearly all partitions. Therefore, this strategy is not applicable for the cloud environment.



Node ID
1
2
3
4
5
6
Basic round-robin assignment on 3 nodes
Assignment after doubling nodes
Assignment after adding one node
Partition with ID
Reassigned partition with ID

Figure 3: Round-robin partition assignment on cluster resize

Consistent hashing: The requirement of minimizing the number of reassignments on cluster resize directly leads to the method of consistent hashing, which places elements and buckets on a logical ring representing the set of hash values produced by the hash function. In order to improve load balancing, buckets can be replicated on the logical ring. It is shown that removing/adding one hash bucket leads to a reassignment of $\frac{k}{n}$ keys, with k being the total number of keys and n being the number of hash buckets [12]. With keys being partitions and hash buckets being nodes, this is the lower bound of reassigned partitions we stated before. If a node is removed, only the partitions assigned to the removed node have to be reassigned and adding a node leads to a reassignment of all partitions between the added node and its last predecessor on the logical ring. Consistent hashing can be implemented by holding a sorted array or list of nodes, so a lookup operation to find a node responsible for a partition takes time $O(\log(n \cdot r))$ with r being the replication factor and using binary search on that list. Updating the number of nodes leads to resorting the list with a complexity of $O(n \cdot \log(n))$, for example using insertion sort for adding a few nodes or merge sort for adding a sorted list of nodes. Holding the list consumes memory in the size of $O(n \cdot r)$, which is independent from the number of partitions.

Explicitly storing and maintaining the assignment mapping: This approach tries to provide a minimal lookup time and a best possible load balancing by explicitly maintaining and storing the mapping $[d] \rightarrow [n]$ from the set of partitions to the set of nodes for each possible partitioning using a partition manager structure. The mapping can be stored as an array with the size d, providing lookup time $O(1)$. Tables with equal number of partitions d_i build an equivalence class $i \in Eq$, so as a side effect, this guarantees co-location of foreign-key related tables when assuming them to have equal partition numbers (otherwise node-local joins would not be possible anyway). Maintaining the mapping explicitly ensures that the best possible load balancing is achieved. As a drawback, this approach has a quite high memory consumption of $O(\sum_{i \in Eq} d_i)$, which is especially not independent from the number of partitions and increases with the number of distinct numbers of partitions. Nevertheless, the number of different partitionings and therefore the number of equivalent classes is typically small in user scenarios.

Comparison: Table 1 compares the approaches of consistent hashing and partition manager. The partition manager approach outperforms consistent hashing in the most important categories load balancing and lookup time, while also minimizing partition reassignment. Assuming that the number of different partitionings is quite small and hence the number of equivalence classes is small, the time for updating the structure and the memory consumption is justifiable. As an example, a database consisting of 1000 tables sharing the same partitioning schema of 1000 partitions would lead to a memory consumption of around 5KB for 1000 4-byte-integer values and 1000 boolean values regarding the partition manager design shown in Section 5.2. Even for 1000 different partitioning schemas with a maximum of 2000 partitions each we would get a few MB of memory consumption. Therefore, the decision has been made towards the partition manager approach.

5.2 Partition manager design

The partition manager structure, illustrated in Figure 4 maintains *partition mapping* objects for each equivalent class, which consist of the number of *partitions*, a *mapping* array and an *is_moved* array, both of the size of the specific number of partitions. Each position i in *mapping* holds the node ID of the node responsible for partition i. The mappings are adapted during each cluster resize operation to maintain load balancing. In addition to that, the boolean *is_moved* value at position i indicates, whether the partition was moved during the last cluster resize operation, which is important to determine partitions to replay from the log when removing nodes or to delete the storage objects from when adding nodes. On top of the *partition mapping* objects, the partition manager maintains a hash table of *partition mapping* pointers to efficiently find the mapping for a given number of partitions.

For implementing a partition assignment, two assumptions are stated. First, it is assumed that co-local partitions have the same partition ID. This assumption is fulfilled by the hash partitioning method, as tuples with same keys

	Consistent hashing	**Partition manager**
Load balancing	Good, not guaranteed	Best possible
Partition reassignment	Minimized	Minimized
Lookup time	$O(\log(\mathit{nodes} \cdot \mathit{replication}))$	$O(1)$
Update	Re-sort array	Adapt every mapping $O(\mathit{equi\_classes} \cdot \mathit{partitions})$
Memory consumption	$O(\mathit{nodes} \cdot \mathit{replication})$	$O(\mathit{equi\_classes} \cdot \mathit{partitions})$

Table 1: Comparison of partition assignment strategies

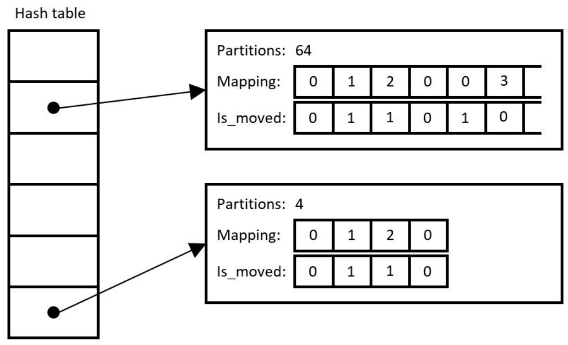


Figure 4: Partition manager overview

(either primary or foreign keys) are mapped to the same hash value, which is used as partition ID. Second, it is assumed that tables with a foreign key relationship have the same number of partitions specified. Having an unequal number of partitions while using the currently implemented hash partitioning violates the requirement of co-locality. Especially, it is not ensured by the current hash function that having a table T with k times the number of partitions than it's join partner S results in a partitioning that maps one partition of S to exactly k partitions of table T.

Initialization: During server startup, the partition manager structure is initialized to a global variable by creating an empty hash table. When the partition manager gets queried for a partition mapping that is not present in its hash table, a partition mapping object for the queried number of partitions is created and inserted into the hash table using the number of partitions d as key. The *mapping* is initialized using a round-robin strategy, so for partition ID i we get $\mathit{mapping}[i] = i \bmod d$. The choice of this initial strategy is arbitrary and could be replaced by any other strategy that provides a balanced assignment for n nodes. The *is_moved* array is initialized with *FALSE* at every position. The described process has time complexity $O(d)$ to initialize a single partition mapping.

Lookup: Knowing the structure of the partition manager, the lookup implementation is straight forward by a single hash table access and a single array access. Due to the design, the lookup operation has complexity $O(1)$.

Update: Whenever adding or removing nodes, all partition mappings have to be adapted. Therefore, we iterate over all entries in the partition manager's hash table and adapt every mapping using an algorithm divided into the following steps:

(1) Compute the optimal load balancing for the new cluster state by computing *partitions_per_node* and a *remainder* if the number of nodes is not a divider of the number of partitions.
(2) Compute a *diffs* array, with *diffs*[i] indicating wether node i has to get additional partitions (positive entry) or get partitions removed (negative entry) to achieve the computed load balancing. The sum over all entries in the *diffs* array is 0, as the total number of partitions does not change.
(3) Adapt the actual partition mapping by iterating over the *mapping* array. If we find a partition whose responsible node has a negative *diffs* entry, we move the partition to a node with a positive *diffs* entry.

With step three being the dominant step in terms of runtime, the algorithm runs in $O(d)$. As we adapt the mapping of every equivalence class, the update operation has a total runtime of $O(\sum_{i \in Eq} d_i)$.

Exchange: After new nodes are added to the system, they check during startup whether they are added to an existing cluster and if so, they prepare to receive the current partition assignments. The current nodes trigger this exchange functionality within the execution of the *add_nodes* query. The exchange operation is implemented using MPI_BCAST (broadcast), so it has to be performed collectively. After broadcasting the number of mappings, the number of partitions and the *mapping* array are broadcasted for each partition mapping. This way it is ensured that all nodes call the broadcast the same number of times.

As a result, the chosen approach minimizes the reassignment of partitions in the case of scaling while providing load balancing and efficient lookup and update functions.

6 BUFFER MATCHING AND FILLING

We now motivate the problem of decreased performance after a cluster resize operation and present the design and the implementation of the buffer matching and filling mechanism, solving the performance problem and therefore being a solution to the third goal stated in Section 1.

For a brief intermediate evaluation of the cluster resize functionality and the partition manager described in Sections 4 and 5, the TPC-H benchmark was run on scale factor 300 using 8 of the 16 cluster nodes described in Section 7. Afterwards, the remaining 8 nodes were added and the benchmark was run again. Overall, it was observed that queries are slower after adding nodes than before. As an example, we pick query 1 of the benchmark, which is a selection and aggregation query on the lineitem table. The runtimes are shown in Figure 5. The query was run on 8 nodes with filled buffers (run 1) before adding 8 additional nodes and running the query several times again (runs 2, 3, and 4). Doubling the number of resources, we would expect a speedup up to factor 2, but the results clearly show an increase in runtime for the first run after the resize operation. However, the performance increases as expected with more consecutive query runs.

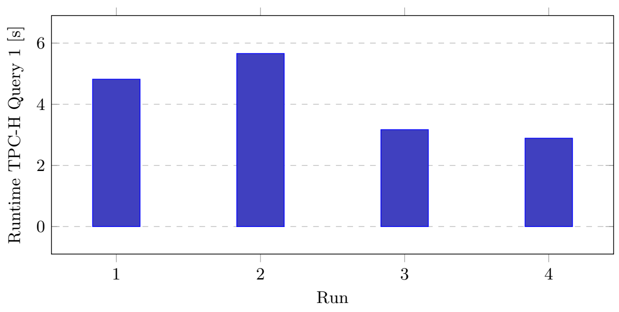


Figure 5: Runtime of TPC-H SF300 Query 1 in consecutive runs with run 1 on 8 nodes and runs 2,3 and 4 on 16 nodes after a cluster resize

From a user perspective, this behavior is not satisfactory, as he pays for extra resources without getting any immediate benefit. The observed performance is a direct consequence of the buffer management. Figure 6 shows the qualitative performance (runtime and output cardinality) of the scan operation of TPC-H query 1 for all threads on all nodes. The right half of the plot, which covers the 8 nodes added by the cluster resize, shows a runtime that is up to 5 times slower with respect to the left half of the plot, which covers the 8 nodes that were already running before the cluster resize. This is caused by filled buffers of the old nodes, while the added nodes start with empty buffers. In order to smoothen this runtime, we present the buffer matching mechanism to fill the buffers of added nodes during the cluster resize step. Doing so, we move the overhead of filling the buffer from user queries to the cluster resize operation, which is more justifiable to the user, as the elapsed time between the user toggling a cluster resize until finishing the resize also involves demanding new cluster nodes from the cloud service provider, which is also a potentially long-running operation and dependent on the actual provider. General approaches for buffer pre-filling must answer the following questions:

- Which data should be chosen to fill into the buffers?
- How is data brought to the buffers?

Especially the second question is important for the cloud setup. As cloud storage systems offer lower bandwidth and higher latencies than node-local storages, sending buffered data between nodes is also worth considering next to reading data from storage.

The chosen solution for this problem is our buffer matching mechanism. In order to discuss the mechanism from an abstract point of view, we identify a sender side and a receiver side, dividing the set of nodes into two distinct sets. Nodes of the sender side are characterized by losing the responsibility of partitions and having blocks in their buffer they are not responsible for anymore, while nodes of the receiver side become responsible for new partitions and do not have any buffered data for them. Note that because of our partition manager design in Section 5, a node is either a sender or a receiver. During data exchange, each node of the sender side can possibly have a connection to each node of the receiver side.

Block selection: The first step of buffer matching is to identify for each node the set of blocks that needs to be sent to other nodes, as well as each block's specific destination. First we get a list of all blocks currently resident in the buffer memory sorted by importance. The importance of a block is determined by the actually used buffer replacement policy. In addition to that, we identify the destination of the blocks by querying the partition manager described in Section 5 to get the responsible node. Blocks that belong to a partition for which the node remains responsible are not sent and therefore dropped from the list. All other blocks are appended to a list of blocks per receiver, so as the result of the block selection step, each sender node holds a (potentially empty) list of blocks for each receiver node. One special case needs to be handled. Due to data distribution or due to buffering blocks of only a few partitions caused by selection predicates, it might occur that receiver nodes are intended to receive more blocks than they can actually fit into their buffers. The calculated cardinality difference is balanced between all senders to this receiver node and each sender is informed about the number of blocks to send before starting to send data. As the block lists are sorted by importance, the sender just drops the end of the list in this case.

Data exchange: We now want to answer the question how buffer data is brought to nodes. As reading data from cloud storage might be slow compared to usual local disks or network transfer, the decision was made towards explicitly sending data to nodes over the network. The implemented data exchange mechanism follows three basic steps:

(1) Exchange the number of blocks to transfer between each sender and receiver node.
(2) Exchange block metadata.
(3) Exchange buffer content.

The first step is important to establish synchronization between senders and receivers. Each node of the receiver side has to know about the number of blocks to receive from each node of the sender side. After the block selection step, each sender node holds a list of blocks per receiver. The length of these lists is shared with the respective receivers using MPI_GATHER routines, called within a loop over all added nodes. A receiver node with rank i becomes the receiver of the MPI_GATHER call in exactly one loop iteration. In this round, all other nodes send the length of their list i, indicating the number of blocks to send to node rank i. As a result, node i has the complete information about the number of blocks to receive after success of loop iteration i. In the second step, we transfer the blocks metadata (e.g., used bytes, columnID or the ID of the commit creating the block) to the receiver nodes. It is important to note that metadata and actual data of a block cannot be transferred all-in-one using a single MPI call by default, as metadata and actual data are not placed in consecutive memory areas due to the VectorH buffer management, which preallocates the whole buffer memory during startup. Constructing an additional structure holding both metadata and buffer data would lead to copying major parts of the buffer, which is not desirable. Sending

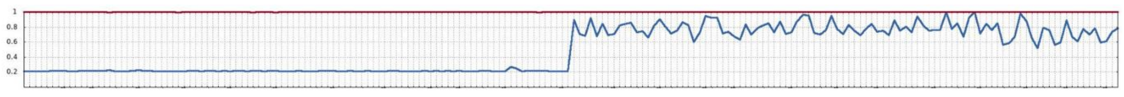


Figure 6: Qualitative TPC-H scan performance of query 1 for run 2

the metadata is important for two reasons. First, the respective block can be searched in the receiver's catalog. VectorH uses a replicated catalog, so a each block is already created on the receiver side. Second, the metadata contain information about the buffer content, like the actual data size or a flag to indicate wether data has been changed. This information is created during data load, so it needs to be sent as we do not load data from storage. Sending the metadata arrays is realized using non-blocking MPI point-to-point communication. This eliminates the need for explicit synchronization in this step. After receiving the metadata, each receiver performs catalog lookups to get pointers to the block structures and demands memory in the buffer memory for each block, before the blocks are inserted into the buffer replacement policy. In the third step, we transfer the actual buffer data. As the data for blocks is not placed in consecutive memory areas, they have to be sent one-by-one. Using non-blocking MPI communication like in the second step would therefore lead to k communication calls per sender and receiver with k being the number of blocks to transfer between sender and receiver, making it difficult to handle for the MPI environment when scaling the problem up. Therefore, we use synchronous, blocking communication for this step. To avoid deadlocks and reduce waiting time, we handle the communication in a multi-threaded way. For each point-to-point connection between a sender and a receiver node, having a non-zero number of blocks to transfer, both sender and receiver node open a separate thread running their side of the communication, resulting in a communication network. Each thread blocks until the respective counterpart of the communication is called. Upon receiving buffer data for one block, the data is copied to the buffer memory of the receiver node.

Fault tolerance: The buffer matching mechanism is an addition to the cluster resize functionality. The success of the buffer matching step is not indispensable for the success of the whole cluster resize operation, but should not lead to a undefined state on the occurrence of errors. Therefore, the mechanism is designed to be fault tolerant. We distinguish between different times of error occurrence. If an error is detected before the block metadata in the catalog of a receiver is changed, we can simply perform a collective abort, as no durable changes were done yet. This is realized using a synchronization point between the second and the third step. If an error occurs during data exchange within third step, we have to ensure that the system handles the block's buffer content in the right way. After receiving a single block's data and copying it to the buffer memory, we verify the correctness of the data using an already existing magic number in the block's data. This magic number is a fixed constant which is used to discover transmission failures. This mechanism could be further improved using a checksum. If the verification succeeds, the block is flagged to be in memory. If the verification fails or an error occurs during communication, the current block is flagged as "LOAD", leading the system to not use the buffer content before an IO-thread loads the data from storage (and adjusts the metadata again). Furthermore, all pending blocks that have not been transferred yet are also flagged as "LOAD". All other communication threads are not affected and may succeed.

Integration: The described mechanism is integrated into the *add_nodes* and the *remove_nodes* call. For adding nodes, the sender side is formed by the current nodes, as they lose responsibilities for partitions and may have buffered data for them, while all newly added nodes form the receiver side. During the system startup of the added nodes, buffer matching is integrated after the partition mapping exchange and the log replay, but before the server is able to handle user connections. This way the server already has the full catalog information. The current nodes perform buffer matching after following the server startup communication of the new nodes. For scaling the cluster size down, the removed nodes form the sender side of the buffer matching mechanism, while the remaining nodes form the receiver side. In order to enable removed nodes to determine the target of their buffered blocks, they update their partition mappings according to Section 5. Afterwards, they perform buffer matching while the remaining nodes perform it during execution of the *remove_nodes* call. In order to isolate the buffer matching communication from all other communication, a separate MPI communicator is build, being only valid during the buffer matching step. Finishing the buffer matching step, this communicator is destroyed. To provide the user the possibility to toggle the buffer matching mechanism on/off, an additional parameter is introduced into the VectorH configuration API.

Optimizations: After describing the basic ideas behind the buffer matching mechanism, we want to introduce two additional optimizations to the concept: the deletion of unused blocks at the sender side and the use of an alternative data exchange implementation. The strategies of the buffer policies are designed to keep the most important elements in the buffer by using priority queues. After performing buffer matching, blocks a sender node is not responsible for anymore may remain in its buffer queues. Due to the behavior of the strategies, these blocks are displaced at some time in the future. Nevertheless, a block may remain a long time in the queues once it is categorized as very important, depending on the actual displacement strategy. As a consequence, this buffer page is useless for a long time, blocking possibly important blocks from finding their way into the buffer. Therefore, we explicitly drop sent blocks from the buffer replacement policy on the sender side. The third step of the buffer matching mechanism uses blocking MPI calls to send the block's data one-by-one, as the buffered data of multiple blocks are not placed in consecutive memory areas (in this case, one call pointing to the start of the memory area would suffice). The MPI environment can be configured to use several communication protocols and uses the Transmission Control Protocol (TCP) in the VectorH integration. Therefore, each call to send/receive a data block invokes communication setup, as well as the common TCP slow start phase, which is unnecessary overhead. As an optimization, we introduce a second, socket-based data exchange implementation. Similar to the described mechanism in the third step of the data exchange step, each sender-receiver pair with non-zero number of blocks to be transferred opens a thread on each side. Instead of starting MPI communication, the nodes establish a TCP stream socket connection. The receiver node creates a socket, sends the socket address information to the sender node using MPI and listens for an incoming connection. The sender node connects to the socket and sends data over the socket. As this single connection keeps alive until all data is sent, the overhead of communication establishment and slow start phase is reduced compared to the MPI implementation. In order to provide the same level of fault tolerance, each side of the socket checks the socket status using *select* before sending/receiving data. Furthermore, data blocks can be send in chunks, and only after a full block is received, the receiver verifies the block. Similarly to the fault tolerant behavior, blocks are flagged on connection or communication errors.

7 EVALUATION

During the evaluation, we want to prove the superiority of the implemented cluster resize feature over the inelastic way of scaling. Furthermore, we want to show that the usage of the buffer matching and filling mechanism improves query performance after a cluster resize operation and is, therefore, a useful extension. Due to expenditure reasons, the evaluation was done on a private cluster of 16 nodes, each with the following configuration:

- AMD Opteron Processor 3380 @2600MHz with 4 modules of 2 cores each
- 32 GB DDR3 RAM
- 3.5 TB disk space, distributed among 4 HDDs
- CentOS-7 64 Bit

The nodes are connected over a 1GBit/s ethernet connection and Hadoop 2.7.1 is installed on the cluster. Comparing the hardware to resources available on Amazon Web Services (AWS), this setup should be slower than all available EC2 instances. Therefore, the measured runtimes in the experiments can be seen as an upper bound and we expect our implementation to perform better on any AWS clusters with equal number of nodes. In addition, the TPC-H benchmark [2] on scale factor 1000 GB provided test data and test queries. This benchmark covers a well-understood synthetical workload in order to evaluate and compare data warehouse solutions with a dataset inspired by real world applications. The large tables of the benchmark are partitioned into 192 partitions, which is an overpartitioning for the cluster of 16 nodes with 8 cores each.

Scaling performance: The first experiment evaluates the performance of the implemented cluster resize feature. For the investigation of the upscaling process we start with a cluster of 4 nodes with filled buffers and vary the number of added nodes, while we start with 16 nodes and vary the number of removed nodes for the downscaling process. For these experiments, we keep the size of the bufferpool at 10 GB and the block size at 1 MB. Figure 7 illustrates the results of the experiment. One can observe that the runtime for adding nodes without using buffer matching increases in a linear way with the number of added nodes. The main reason for this behavior is the collective startup of the nodes. Starting more nodes at the same time increases the impact of the various synchronization points within the startup process. The buffer matching mechanism adds a nearly constant overhead to the measured *add_nodes* runtime. In a more detailed consideration one can observe that the buffer matching mechanism shows its minimum runtime when adding 8 nodes. Adding more nodes also increases the buffer matching data exchange parallelism (the number of receiver nodes per sender node), so the minimum runtime is expected to be at the number of physical cores, which is 8 in the used hardware setup. The downscaling runtime shows a slight increase when removing more nodes, which is caused by the adapted log replay the remaining nodes have to perform. Within this step, removing more nodes leads to more log entries that have to be replayed in the remaining nodes. The buffer matching mechanism was not applied for the downscaling process, as buffers where totally filled before scaling, so there was no buffer space left in the remaining nodes to receive blocks from removed nodes. Overall we can state that downscaling takes significantly less time than upscaling, because the synchronization effort for downscaling is lower. Once the nodes are split into two groups within the *remove_nodes* process, the group of removed nodes can simply perform a shutdown.

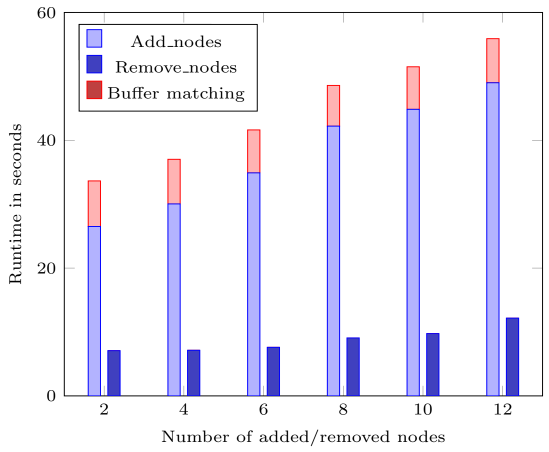


Figure 7: Scaling performance for adding and removing nodes

Scaled query performance: This experiment investigates the impact of the cluster resize operations on query performance. For this we repeatedly run a query while changing the cluster size between runs. As the main impact is expected to be within the scan operators, we use a query scanning two columns of the lineitem table. In order to fit the data into the buffer of the smallest cluster configuration, we limit the number of tuples to 500 million using a selection predicate, while setting the buffer memory to 20 GB per node. This way we avoid I/O access that would create an unintended bias in the measurements. Moreover we minimize the network traffic by applying an aggregation on each column, which is executed locally on each partition and results in a single tuple that needs to be send to the master node. Figures 8 and 9 show the measured query runtimes. Regarding the case of upscaling, we can observe that adding nodes accelerates the query runtime as expected. However, the behavior varies between different buffer matching configurations. With activated buffer matching, query runtime drops and stays on the same level for the respective query runs after cluster resize, because the buffers already contain the needed data. On the contrary, not using buffer matching leads to significantly slower queries, especially for the first run after a cluster resize. The reason for this behavior is that added nodes have to read data from storage. In the following runs the query performance improves as buffers of added nodes fill. For the case of downscaling, we have to distinguish between two use cases. On the one hand, the reason a user triggers a downscale operation can be that the system is in an overprovisioning state, so the system underutilizes the provided hardware. In this case, removing server capacity should not have an impact on the systems performance. For VectorH we neglect this case as we aim to have more partitions than nodes at every point in time. If this condition is violated, a repartition operation is triggered by the system. On the other hand, the user could invoke the downscaling to save costs while accepting slower system performance, e.g., when the expected load becomes less during specific times of a day. This case is expressed by Figure 9. When scaling down the cluster, we can observe a performance degradation, again varying between buffer matching configurations. Similar to the upscaling case, the runtime of the first query run after cluster resize is significantly slower when not using buffer matching, as remaining nodes become responsible for data of removed nodes and have to read it from storage. With activated buffer matching, data is sent to the remaining nodes, leading to an immediately fast runtime after resize. Overall, this experiment proves that using the buffer matching mechanism during cluster resize perceptibly increases query performance after resizing.

Buffer matching performance: In this experiment, we want to evaluate the buffer matching performance for both implemented data exchange mechanisms, using MPI and using data streams over sockets. The two main parameters that have an impact on the buffer matching performance are buffer size and block size. While the buffer size affects the amount of data that is shipped during the buffer matching data exchange, the block size affects the granularity of shipped blocks and as a result also the communication overhead. Trying to touch as much data as possible, we use query 9 of the TPC-H benchmark for this experiment, as it touches five of the 8 tables in the benchmark and scans about seven billion tuples for the used scale factor of 1000 GB.

The plot in Figure 10 shows the runtime of the buffer matching mechanism as a function of the buffer size per node for both data exchange implementations, using a constant block size of 1 MB. For these measurements, we used the up-scale step from 8 to 16 nodes. First of all, the results show that the data exchange takes the major part (about

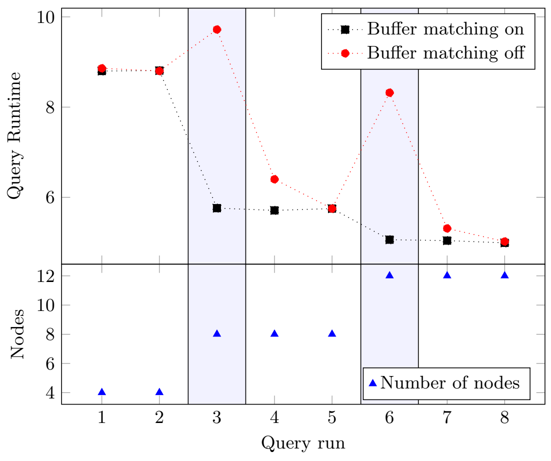


Figure 8: Scaled query performance after adding nodes (buffer matching impact highlighted)

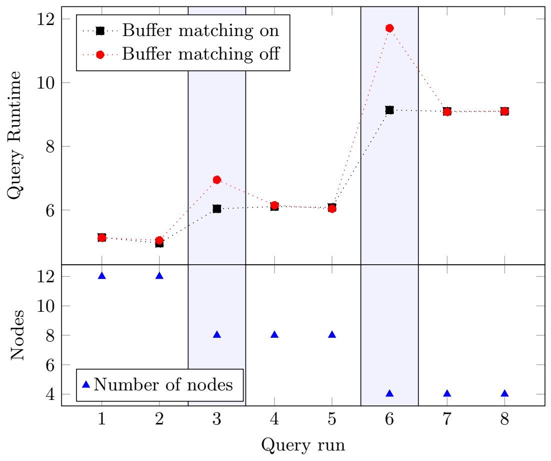


Figure 9: Scaled query performance after removing nodes (buffer matching impact highlighted)

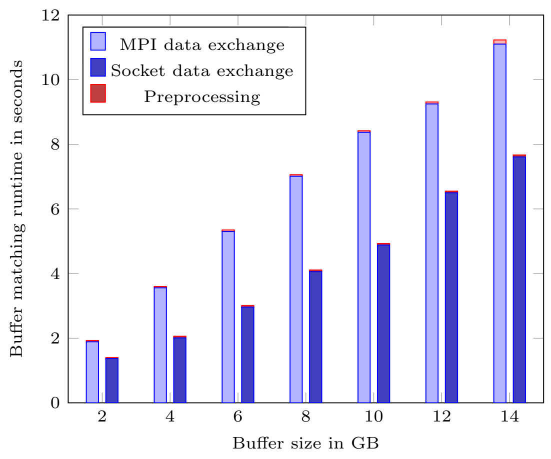


Figure 10: Buffer matching runtime for preprocessing and actual data exchange as a function of the buffer size for upscaling from 8 to 16 nodes with a constant block size of 1 MB

98%) of the whole buffer matching step. Furthermore, the plot illustrates the expected runtime increase when increasing the buffer size and shows that they correlate in a linear way. Comparing both data exchange implementations, we can observe that the socket implementation is faster in terms of runtime and also shows a smaller grow in runtime when increasing the buffer size. As discussed in Section 6, this behavior is presumably caused by the fact that the MPI implementation has to re-initiate the communication for each block, as the blocks may be randomly placed within the buffer memory space. The socket implementation on the other hand initiates the communication once before sending a data stream and therefore reduces the communication overhead.

In a further experiment we evaluate the impact of varying block sizes on the buffer matching mechanism. For this we keep the buffer size constant at 10 GB and we only consider the pure data exchange runtime, as we have seen in Figure 10 that preprocessing only takes a minor part of the overall runtime. Increasing the block size implies an increase of the necessary memory to allocate blocks. Therefore, we had to switch to scale factor 300 GB for this experiment as the overall available memory did not suffice for the largest tested block size of 8 MB and scale factor 1000 GB. Figure 11 shows the results of this experiment. For both implementations we can observe a slight increase in runtime when increasing the block size. This is caused by increased data volume that has to be exchanged, as larger block sizes come along with larger unused space or padding. Besides that, the socket implementation is not heavily impacted by varying block sizes, as data is simply written to stream sockets not considering any block boundaries. On the contrary, the MPI implementation profits from larger block sizes, as the communication overhead shrinks with the decreasing number of blocks to be sent. As a result, the difference in runtime between both implementations also shrinks with larger block sizes. Nevertheless, the choice of the block size also impacts other parts of the system, so this choice is usually fixed around a value of 1MB and can not be changed after database creation. For these block sizes, the socket implementation is surely the better choice compared to the MPI implementation.

Cluster resize usability: In the last experiment, we compare the implemented cluster resize functionality with the "inelastic" scaling, which involves the following steps:

(1) Shutdown of the system
(2) Adjustment of a list that holds the VectorH node names
(3) Restart of the system
(4) Run a query

These step are encapsulated in a script to reliably measure the runtime. We define the start state of the experiment as a running VectorH system with filled buffers. Furthermore, we define the end state as the moment we get a query result from a scaled VectorH instance. The runtime between start and end state is measured for the cluster resize feature with and without activated buffer matching, as well as for the inelastic scaling process. As query workload we choose query 1 and query 9 of the TPC-H benchmark. The buffer size is set to 10 GB per node and we investigate the cases of scaling from 8 to 16 nodes and vice versa.

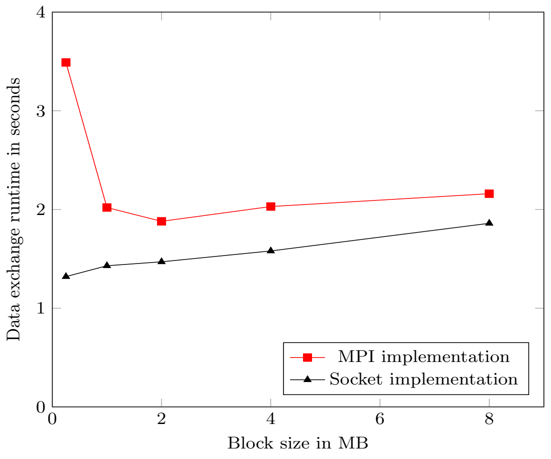


Figure 11: Buffer matching data exchange runtime as a function of the block size with constant buffer size of 10GB when upscaling from 8 to 16 nodes

	add_nodes		remove_nodes	
	Q1	Q9	Q1	Q9
BM on	57.74 s	77.55 s	32.02 s	51.06 s
BM off	58.70 s	87.95 s	32.56 s	52.04 s
Inelastic	167.68 s	223.30 s	123.20 s	147.94 s

Table 2: Runtime for scaling the system using the inelastic scaling process or the cluster resize feature with and without buffer matching (BM)

Table 2 shows the measured runtime results of the experiment. For all cases, the implemented cluster resize feature outperforms the inelastic scaling up to a factor 4. In addition to that, activated buffer matching shows a benefit in runtime for the cases of scaling the system up, caused by the buffer pre-filling. The amount of benefit is dependent on the actual query for this experiment, so query 9 shows a better speedup than query 1 using buffer matching. For the case of removing nodes, buffer matching has a negligible impact, as buffers of the remaining nodes are already filled. Therefore, a buffer merging strategy could be a possible optimization in the future. Moreover, we can state that adding nodes to the system is slower than removing nodes, both for inelastic scaling and the cluster resize feature. The reason for this is that started servers perform a collective startup with several synchronization points and do a full log replay. Increasing the number of servers, this startup time also increases, leading also the inelastic scale-up to be slower than the scale-down. In the contrary, removed nodes can shutdown independently after the remaining servers form a new communicator (see Section 4), not influencing the further query processing. This experiment is highly dependent on the cluster configuration (e.g. network speed) as well as the size of the write-ahead log that has to be replayed. Therefore, this experiment should not be used for a quantitative comparison, but is intended to show a qualitative difference between the scaling methods.

Overall the evaluation proves that the implemented cluster resize feature outperforms the "inelastic" scaling method using a restart up to a factor of 4. The buffer matching mechanism shows to add a minor runtime overhead to the scaling step, but proves to have a major impact on the query performance. The first queries after scaling show a significant performance gain when using buffer matching, which is caused by the pre-filling of buffers. As a result, the user gets an immediate performance boost when deciding to scale the VectorH installation up, which is the behavior he expects when increasing his service cost. Furthermore, the experiments prove that the socket implementation improves the buffer matching data exchange step compared to the MPI implementation, which was the expectation this optimization was based on. As we evaluated our implementation using a private cluster, the time for acquiring resources from a cloud service provider is not included in our results.

8 CONCLUSION

In this paper, we presented our approach to adapt Actian VectorH for the elastic cloud environment. As the first goal, we implemented an elastic cluster resize feature for VectorH, enabling adding and removing nodes during system uptime and therefore avoiding the drawbacks of a full system restart, e.g. full log replay and empty buffers. For the implementation of the feature we utilized group and communicator management offered by the Message Passing Interface (MPI), which is used for node-to-node communication within VectorH. As a second contribution, we designed a partition manager that is suitable for the cloud environment. By using overpartitioning and explicitly managing partition-to-node mappings for equivalence classes of partitionings, the implemented solution minimizes partition reassignments, keeps partition co-locations, balances load on partition level and provides efficient lookup and update functions. The partition manager replaces the round-robin partition assignment in VectorH, which showed to be not suitable for the elastic cloud environment. While evaluating the cluster resize feature in combination with the implemented partition manager, queries did not show the expected speedup immediately after the resize, which was caused by empty buffers. As an optimization, we introduced the buffer matching and filling mechanism into the cluster resize feature. After changing partition mappings, nodes scan their buffers and send buffered data to other nodes in order to pre-fill their buffers, leading to immediate speedup after cluster resize. For the buffer matching data exchange, we implemented two different approaches using MPI communication and using data streams over sockets and evaluated them against each other.

The experiments showed that the elastic cluster resize feature significantly outperforms the inelastic scaling process using a system restart. Activating the buffer matching mechanism further increases the performance after the cluster resize, enabling the user to immediately profit from additional resources. Evaluating both buffer matching data exchange mechanisms, the socket implementation showed to be the better choice for all tested cases.

Future work includes the investigation on online scaling allowing concurrent read and/or write transactions during the scaling process, as well as query driven scaling, optimizing a query for a given goal (e.g., cost, runtime) by scaling the system automatically. In addition to that, further data exchange mechanisms, such as RDMA (remote direct memory access) based data exchange, will be evaluated to accelerate the buffer matching data exchange step. The buffer matching mechanism could also be further extended with a predictive buffer filling strategy for adding nodes or a buffer merging strategy for removing nodes.

REFERENCES

[1] Sean Barker, Yun Chi, Hakan Hacigümüs, Prashant Shenoy, and Emmanuel Cecchet. 2014. ShuttleDB: Database-Aware Elasticity in the Cloud. In *11th International Conference on Autonomic Computing (ICAC 14)*. USENIX Association, Philadelphia, PA, 33–43. https://www.usenix.org/conference/icac14/technical-sessions/presentation/barker

[2] Peter Boncz, Thomas Neumann, and Orri Erling. 2014. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking*, Raghunath Nambiar and Meikel Poess (Eds.). Springer International Publishing, Cham, 61–76.

[3] Peter Boncz, Marcin Zukowski, and Niels Nes. [n.d.]. MonetDB/X100: Hyper-Pipelining Query Execution. ([n. d.]).

[4] Andrei Costea, Adrian Ionescu, Bogdan Răducanu, MichałSwitakowski, Cristian Bârca, Juliusz Sompolski, Alicja Luszczak, MichałSzafrański, Giel de Nijs, and Peter Boncz. 2016. VectorH: Taking SQL-on-Hadoop to the Next Level. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1105–1117. https://doi.org/10.1145/2882903.2903742

[5] Benoit Dageville, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, Philipp Unterbrunner, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, and Martin Hentschel. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*. ACM Press, San Francisco, California, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[6] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2013. ElasTraS: An Elastic, Scalable, and Self-managing Transactional Database for the Cloud. *ACM Trans. Database Syst.* 38, 1, Article 5 (April 2013), 45 pages. https://doi.org/10.1145/2445583.2445588

[7] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment* 4, 8 (May 2011), 494–505. https://doi.org/10.14778/2002974.2002977

[8] G. Graefe. 1994. Volcano-an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering* 6, 1 (Feb. 1994), 120–135. https://doi.org/10.1109/69.273032

[9] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*. ACM Press, Melbourne, Victoria, Australia, 1917–1923. https://doi.org/10.1145/2723372.2742795

[10] John L. Gustafson. 1988. Reevaluating Amdahl's Law. *Commun. ACM* 31, 5 (May 1988), 532–533. https://doi.org/10.1145/42411.42415

[11] Sandor Heman, Niels Nes, Marcin Zukowski, and Peter Boncz. [n.d.]. Positional Delta Trees to reconcile updates with read-optimized data storage. ([n. d.]), 11.

[12] Wolfgang Lehner and Kai-Uwe Sattler. 2013. *Web-Scale Data Management for the Cloud.* Springer New York, New York, NY. https://doi.org/10.1007/978-1-4614-6856-1

[13] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-scale Datasets. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 330–339. https://doi.org/10.14778/1920841.1920886

[14] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. 2011. A Cost-Aware Elasticity Provisioning System for the Cloud. In *2011 31st International Conference on Distributed Computing Systems*. 559–570. https://doi.org/10.1109/ICDCS.2011.59

[15] Michał Switakowski, Peter Boncz, and Marcin Zukowski. 2012. From cooperative scans to predictive buffer management. *Proceedings of the VLDB Endowment* 5, 12 (Aug. 2012), 1759–1770. https://doi.org/10.14778/2367502.2367515

Fairness in Online Jobs: A Case Study on TaskRabbit and Google

“Applications” paper

Sihem Amer-Yahia
CNRS, Univ. Grenoble Alpes, France
sihem.amer-yahia@univ-grenoble-alpes.fr

Shady Elbassuoni
American University of Beirut, Lebanon
se58@aub.edu.lb

Ahmad Ghizzawi
American University of Beirut, Lebanon
ahg05@mail.aub.edu

Ria Mae Borromeo
UP Open University, Philippines
rhborromeo@up.edu.ph

Emilie Hoareau
IAE, Univ. Grenoble Alpes, France
emilie.hoareau@univ-grenoble-alpes.fr

Philippe Mulhem
CNRS, Univ. Grenoble Alpes, France
philippe.mulhem@univ-grenoble-alpes.fr

ABSTRACT

Online job marketplaces are becoming very popular. Either jobs or people are ranked by algorithms. For example, Google and Facebook job search return a ranked list of jobs given a search query. TaskRabbit and Fiverr, on the other hand, produce rankings of workers for a given query. Qapa, an online marketplace, can be used to rank both workers and jobs. In this paper, we develop a unified framework for fairness to study ranking workers and jobs. We case study two particular sites: Google job search and TaskRabbit. Our framework addresses group fairness where groups are obtained with any combination of protected attributes. We define a measure for unfairness for a given group, query and location. We also define two generic fairness problems that we address in our framework: *quantification*, such as finding the k groups (resp., queries, locations) for which the site is most or least unfair, and *comparison*, such as finding the locations at which fairness between two groups differs from all locations, or finding the queries for which fairness at two locations differ from all queries. Since the number of groups, queries and locations can be arbitrarily large, we adapt Fagin top-k algorithms to address our fairness problems. To evaluate our framework, we run extensive experiments on two datasets crawled from TaskRabbit and Google job search.

1 INTRODUCTION

Online job search is gaining popularity as it allows to find people to hire for jobs or to find jobs to apply for. Many online job search sites exist nowadays such as Facebook job search[1] and Google job search[2]. On those sites, users can find jobs that match their skills in nearby businesses. On the other hand, freelancing platforms such as TaskRabbit[3] and Fiverr[4] are examples of online job marketplaces that provide access to a pool of temporary employees in the physical world (e.g., looking for a plumber), or employees to complete virtual "micro-gigs" such as designing a logo.

[1]https://www.facebook.com/jobs/
[2]https://jobs.google.com/about/
[3]https://www.taskrabbit.com/
[4]https://www.fiverr.com/

In online job search, either jobs are ranked for people or people are ranked for jobs. For instance, on Google and Facebook job search, a potential employee sees a ranked list of jobs while on TaskRabbit, an employer sees a ranked list of potential employees. This ranking of jobs or individuals naturally poses the question of fairness. For instance, consider two different users searching for a software development job in San Francisco using Google job search. If the users are shown different jobs based on their search and browsing history, which could correlate with their demographics such as race or gender, this may be considered unfair. Similarly, a ranking of job seekers in NYC might be unfair if it is biased towards certain groups of people, say where White Males are consistently ranked above Black Males or White Females. This can commonly happen since such rankings might depend on the ratings of individuals and the number of jobs they completed, both of which can perpetuate bias against certain groups of individuals.

In this paper, we propose to quantify unfairness in ranking when looking for jobs online. We develop a unified framework to address group unfairness, which is defined as the unequal treatment of individuals based on their protected attributes such as gender, race, ethnicity, neighborhood, income, etc. [11]. To quantify unfairness for a group, we measure the difference in rankings between that group and its *comparable* groups, i.e., those groups which share at least one protected attribute value with the given group. For instance, consider the group “Black Females”, comparable groups would be “*Black* Males”, “White *Females*” and “Asian *Females*”.

The difference in ranking naturally depends on what is being ranked, jobs or people, and we formalize various measures of unfairness on different types of sites (job search sites and online job marketplaces). Figures 1 and 2 illustrate examples of job ranking on Google job search and people ranking on TaskRabbit, respectively. For a given query on Google job search, “Home Cleaning" in location "San Francisco” in Figure 1, we quantify unfairness in ranking for a given demographic group, “Black Females”, using Kendall Tau (we also use Jaccard Coefficient in our data model), between the search results of black females and all other users in comparable groups, as is done in [12]. To quantify unfairness for “Black Females” on TaskRabbit for the query "Cleaning Services" in location "New York City", we compute the average Earth Mover’s Distance [20] between the distribution of rankings of Black Females and all comparable groups, as in [11]. In our framework, we also compute the difference of exposure of workers from this demographic group and their relevance in contrast to comparable groups and then use this as a measure of unfairness for this group, as in [2, 22].

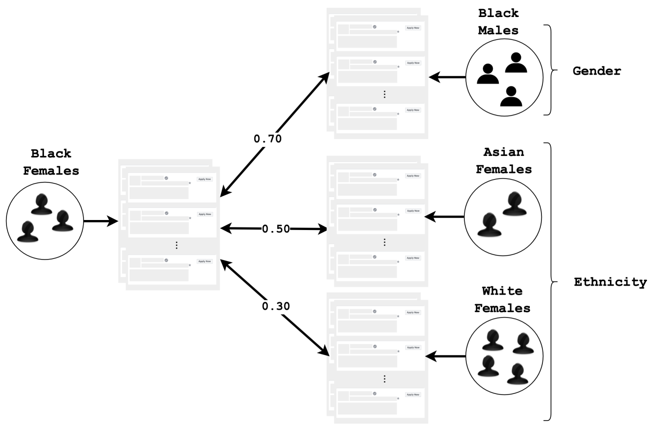


Figure 1: The unfairness for "Black Females" for the Google job search query "Home Cleaning" in location "San Francisco" using Kendall Tau between the search results of Black Females and all other users in comparable groups is $\frac{0.70+0.50+0.30}{3} = 0.50$.

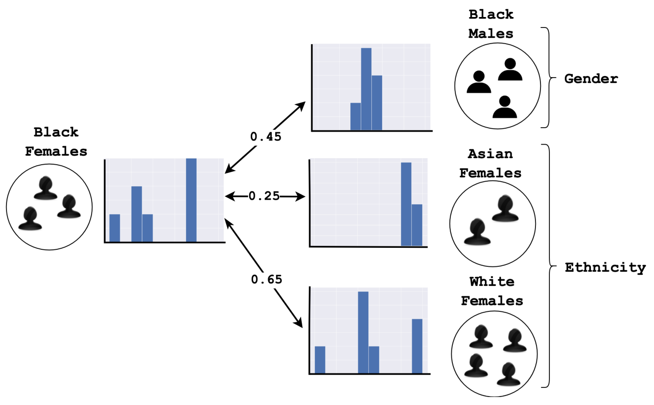


Figure 2: The unfairness for "Black Females" for the query "Cleaning Services" in location "New York City" on TaskRabbit using Earth Mover's Distance between ranking distributions of Black Females and its comparable groups is $\frac{0.45+0.25+0.65}{3} = 0.45$.

Various fairness questions can be formulated either to quantify how well a site treats groups for different jobs and at different locations, or to compare groups, queries or locations. Our framework allows us to define two generic fairness problems: *quantification*, such as finding the k groups (resp., queries, locations) for which the site is most or least unfair, and *comparison*, such as finding the locations at which fairness between two groups differs from all locations, or finding the queries for which fairness at two locations differ from all queries. Examples of *quantification* questions are: *what are the five groups for which Google job search is most unfair? what are the five fairest queries for women?* and *at which locations do Asians have the highest chance to be hired for a given job?*. Examples of *comparison* questions are: *how differently does TaskRabbit treat men and women and for which queries is the treatment different? at which locations is it easiest to be hired as a house cleaner than as a gardener?* and *which jobs are the most likely to accept hiring asian females over black females?*.

We develop efficient Fagin top-k algorithms to solve our problems. Our algorithms make use of three types of indices: *group-based*, *query-based*, and *location-based*, that pre-compute unfairness values for combinations of groups, queries and locations, for faster processing.

To evaluate our framework, we run extensive experiments on two datasets crawled from Google job search and TaskRabbit. The choice of these two platforms is justified by our goal to show the applicability of our framework to two different treatments of online employment, namely ranking jobs and ranking workers. We ran 5,361 queries on TaskRabbit and extracted for each query, the rank of each tasker, their profile pictures, and demographics, where the number of taskers returned per query was limited to 50. We processed the results and recorded unfairness values. We then derived user groups of interest and equivalent Google search terms from data crawled from TaskRabbit. This resulted in 20 queries (the top 10 and bottom 10 frequently searched queries) and their corresponding locations from data crawled in TaskRabbit. We setup 60 user studies on Prolific Academic[5] and recruited participants, who belong to chosen groups. To control for noise in search, we asked those participants to use a Google Chrome extension we developed that automatically executes on Google the search queries in 10 locations. We processed the results and recorded unfairness values.

Our results are organized into the two problems we solve: fairness quantification and fairness comparison. On TaskRabbit, we found that *Asian Females and Asian Males are the ones most discriminated against.* We also found that *Handyman and Yard Work are the most unfair jobs and that Furniture Assembly and Delivery, are the fairest* and that *Birmingham, UK and Oklahoma City, OK are the least fair while Chicago and San Francisco are the fairest locations across all jobs.* We also quantified the fairest/unfairest locations for some jobs and the fairest/unfairest jobs for some locations. Our TaskRabbit results demonstratethe flexibility and expressiveness of fairness quantification, and provided the ability to generate *hypotheses to be tested on Google job search.*

On Google job search, we found that *Washington, DC is deemed the fairest.* On the other hand, *London, UK is deemed the unfairest location.* For queries, we found that *Yard Work jobs are deemed the most unfair whereas Furniture Assembly jobs are deemed the most fair.*

While fairness quantification resulted in largely known results, our fairness comparison experiment on both platforms revealed new results. For instance, *on TaskRabbit, in Chicago, Nashville and San Francisco, Females are treated more fairly than Males, which differs from the overall comparison.* Most results are consistent between EMD and Exposure. Similarly for Google job search, most results are consistent between Jaccard and Kendall Tau. This is quite encouraging and and merits further investigation in future work.

The paper is organized as follows. We review related work in Section 2. In Section 3, we present our data model. In Section 4 we describe our unfairness problems and the algorithms we use to solve these problems. Section 5 describes our case study on two sites, Google job search and TaskRabbit. Finally, we conclude and present future work in Section 6.

[5]https://prolific.co

2 RELATED WORK

To the best of our knowledge, our work is the first to formalize group-fairness, query-fairness, location-fairness, and fairness comparisons, and conduct an extensive evaluation of job search on a virtual marketplace and a job search site. Further statistical and manual investigations are necessary for causality and explainability. Our goal is to reduce initial manual effort by providing necessary tools to assess fairness.

Fairness has been trending in research for the last few years as we increasingly rely on algorithms for decision making. Bias has been identified as a major risk in algorithmic decision making [4, 11, 16, 23, 27]. One algorithmic solution is based on the formalization in [16] to quantify unfairness. To detect unfairness in algorithms, a framework [24] for "unwarranted associations" was designed to identify associations between a protected attribute, such as a person's race, and the algorithmic output using the FairTest tool. In [11], the notion of unfairness was defined as a disparity in treatment between different groups of people based on their protected attributes (i.e., what is commonly referred to as *group unfairness*). In this context, to assess unfairness mathematically, one needs to compare distributions of decisions across different groups of people. In our work, we adapt the definition of unfairness in [11]. However, rather than trying to fix it, the goal of our work is to just *reveal* any unfairness by the ranking process, which in some cases might be *positive* discrimination [19] where certain disadvantaged individuals are favored based on their protected attributes.

There is a wealth of work on addressing fairness of ranking in general (for example [6, 16, 22, 24–26]). Unlike our work, the majority of these works that focus on group fairness either assume the presence of pre-defined groups based on protected attributes of users, or the presence of ranking constraints that bound the number of users per protected attribute value in the top-k ranking. On the other hand, the work in [2] focuses on addressing amortized individual fairness in a series of rankings. In [15], the authors introduce *subgroup fairness* and formalize the problem of auditing and learning classifiers for a rich class of subgroups. Our work differs in many ways: we are interested in ranking individuals and not classifying them, as well as ranking jobs and we seek to quantify the fairness of jobs, locations and groups and compare fairness across different dimensions.

In [1], the authors develop a system that helps users inspect how assigning different weights to ranking criteria affects ranking. Each ranking function can be expressed as a point in a multidimensional space. For a broad range of fairness criteria, including proportionality, they show how to efficiently identify groups (defined as a combination of multiple protected attributes). Their system tells users whether their proposed ranking function satisfies the desired fairness criteria and, if it does not, suggests the smallest modification that does.

In [9], the authors studied fairness of ranking in online job marketplaces. To do this, they defined an optimization problem to find a partitioning of the individuals being ranked based on their protected attributes that exhibits the highest unfairness by a given scoring function. They used the Earth Mover's Distance between score distributions as a measure of unfairness. Unlike other related work, we did not assume a pre-defined partitioning of individuals and instead developed two different fairness problems, one aiming at quantifying fairness and the other at comparing it.

There is a wealth of work that empirically assessed fairness in online markets such as crowdsourcing or freelancing platforms [8, 13, 17, 17, 21]. For instance, the authors in [17] analyze ten categories of design and policy choices through which platforms may make themselves more or less conducive to discrimination by users. In [13], the authors found evidence of bias in two prominent online freelance marketplace, TaskRabbit and Fiverr. Precisely, in both marketplaces, they found that gender and race are significantly correlated with worker evaluations, which could harm the employment opportunities afforded to the workers on these platforms. The work in [21] studies the Uber platform to explore how bias may creep into evaluations of drivers through consumer-sourced rating systems. They concluded that while companies like Uber are legally prohibited from making employment decisions based on protected characteristics of workers, their reliance on potentially biased consumer ratings to make material determinations may nonetheless lead to a disparate impact in employment outcomes. Finally, discrimination in Airbnb was studied in [8] and high evidence of discrimination against African American guests was reported.

In [7], the authors study ethics in crowd work in general. They analyze recent crowdsourcing literature and extract ethical issues by following the PAPA (privacy, accuracy, property, accessibility of information) concept, a well-established approach in information systems. The review focuses on the individual perspective of crowd workers, which addresses their working conditions and benefits.

Several discrimination scenarios in task qualification and algorithmic task assignment were defined in [3]. That includes only accounting for requester preferences without quantifying how that affects workers, and vice versa. Another discriminatory scenario in [3] is related to worker's compensation since a requester can reject work and not pay the worker or a worker can be under-payed. Discrimination in crowdsourcing can be defined for different processes.

In [18], the authors study how to reduce unfairness in virtual marketplaces. Two principles must be adapted: 1) platforms should track the composition of their population to shed light on groups being discriminated against; and 2) platforms should experiment on their algorithms and data-sets in a timely manner to check for discrimination. In this same paper, the authors define four design strategies to help reduce discrimination, a platform manager should first answer these questions: 1) are we providing too much information? 2) can we automate the transaction process further? 3) can we remind the user of discriminatory consequences when they are making a decision? 4) should the algorithm be discrimination-aware? In question 1), they address the issue of transparency. Discrimination and transparency might be highly correlated but their correlation has yet to be studied profoundly. In [3], transparency plug-ins are reviewed. Those plug-ins disclose computed information, from worker's performance to requester's ratings such as TurkBench [14], and Crowd-Workers [5]. Such plug-ins might be helpful in a more detailed study of the effect of transparency on fairness.

3 FRAMEWORK

3.1 Unfairness Model

On any given site, we consider a set of groups $\mathcal{G}$, a set of job-related queries $\mathcal{Q}$, and a set of locations $\mathcal{L}$. We associate to each group g a label $label(g)$ in the form of a conjunction of predicates $a = val$. We use $A(g)$ to refer to all attributes used in $label(g)$. For example, if *label*(g) is *(gender = male)* ∧ *(ethnicity = black)*, we have: $A(g)$ is *{gender, ethnicity}*. We define $variants(g, a)$ where $a \in A(g)$ as all groups whose label differs from g on the value of a. For instance, $variants(g, gender)$ contains a single group whose label is *(gender = female)* ∧ *(ethnicity = black)*, $variants(g, ethnicity)$ contains two groups whose labels are *(gender = male)* ∧ *(ethnicity = asian)* and *(gender = male)* ∧ *(ethnicity = white)*, respectively.

We define the set of *comparable* groups for a group g as $\{g' \in \cup_{a \in A(g)} variants(g, a)\}$. In our example, it is $variants(g, gender) \cup variants(g, ethnicity)$. This notion of comparable groups can be more easily leveraged for explanations. To consider other notions, we believe we would need to extend only our fairness model, and not the full framework.

Each query $q \in Q$ contains a set of *keywords* such as "Home Cleaning" or "Logo Design". The same query can be asked at different geographic locations $l \in \mathcal{L}$. In some applications such as TaskRabbit, a query will be used to refer to a set of jobs in the same category such as Handyman, Furniture Assembly and Delivery services.

We denote by $d_{<g,q,l>}$ the unfairness value of the triple $< g, q, l >$. We discuss next how this unfairness value is computed for different types of sites.

3.2 Unfairness Measure for Search Engines

In a search engine such as Google Search, each user $u \in g$ is associated with a ranked list of search results $E^l_q(u)$. We compute unfairness of g as:

$$d_{<g,q,l>} = \text{avg}_{g'}\ DIST(g, g')\ \forall g' \in \cup_{a \in A(g)} variants(g, a) \quad (1)$$

A common way to compare search results is to use measures such as Jaccard Index or Kendall Tau [12]. Hence, we define $DIST(g, g')$ as one of the following two:

- $\underset{u,u'}{\text{avg}}\ \tau(E^l_q(u), E^l_q(u')), \forall u \in g, \forall u' \in g'$, where $\tau(E^l_q(g), E^l_q(g'))$ is the Kendall Tau between the ranked lists $E^l_q(u)$ and $E^l_q(u')$.
- $\underset{u,u'}{\text{avg}}\ JACCARD(E^l_q(u), E^l_q(u'))$, $\forall u \in g, \forall u' \in g'$, where $JACCARD(E^l_q(u), E^l_q(u'))$ is the Jaccard Index between the ranked lists $E^l_q(u)$ and $E^l_q(u')$.

In Table 1, we display a toy example of the top-3 results for 10 users on a search engine for the query "Home Cleaning" in location "San Francisco". Figure 3 shows how the unfairness value for the group "Black Females" is computed using Jaccard index. In the figure, the Jaccard index between every Black Female user and Asian Female user is computed and then average of the Jaccard index is used to measure unfairness value between the two groups "Black Females" and "Asian Females". To compute the overall unfairness value for the group "Black Females", the same computation must be done between Black Females and all other comparable groups, namely "Black Males" and "White Females" and then the average of the individual unfairness values between groups is taken.

3.3 Unfairness Measure for Online Job Marketplaces

In online marketplaces such as TaskRabbit, we are given a set of workers $\mathcal{W}$, and a scoring function $f^l_q : \mathcal{W} \rightarrow [0, 1]$. Each worker $w \in \mathcal{W}$ is ranked based on her score $f^l_q(w)$. To measure $d_{<g,q,l>}$, we can use one of two methods: *Earth Mover's Distance (EMD) [20] and Exposure [2, 22].*

Table 1: Top-3 results for 10 users for the query "Home Cleaning" in location "San Francisco" on a search engine.

Worker	Top-3
w1	b, d, e
w2	d, b, e
w3	a, b, c
w4	b, a, c
w5	a, b, c
w6	d, a, b
w7	a, b, d
w8	d, a, b
w9	a, b, c
w10	a, b, c

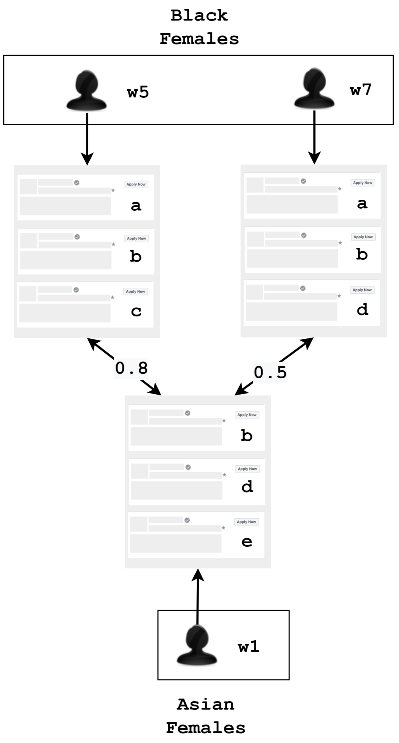


Figure 3: The partial unfairness in a search engine for "Black Females" in Table 2 with respect to one of its comparable groups, "Asian Females", using Jaccard Index is $\frac{0.8+0.5}{2} = 0.65$.

3.3.1 EMD Unfairness. In the EMD notion of unfairness, the unfairness for a group g for query q at location l is computed as the distance between the score distributions of workers in group g and all its comparable groups $g' \in \cup_{a \in A(g)} variants(g, a)$ as follows:

$$d_{<g,q,l>} = \text{avg}_{g'}\ DIST(g, g')\ \forall g' \in \cup_{a \in A(g)} variants(g, a) \quad (2)$$

where

$$DIST(g, g') = EMD(h(g, f^l_q), h(g', f^l_q))$$

where $h(g, f^l_q)$ is a histogram of the scores of workers in g using f^l_q.

In Table 2, we show a toy example consisting of 10 workers looking for a "Home Cleaning" job in San Francisco and their protected attributes. The ranking of these workers is shown in Table 3. Figure 4 illustrates how the EMD unfairness of Black Females, g, is calculated. Since $A(g)$ is *Gender* and *Ethnicity*, the comparable groups in the toy example are *Black* Males, Asian *Females* and White *Females.*

Table 2: Example of 10 workers looking for a "Home Cleaning" job in San Francisco and their protected attributes

Worker	Gender	Nationality	Ethnicity
w1*	Female	America	Asian
w2	Male	America	White
w3*	Female	America	White
w4	Male	Other	Asian
w5	**Female**	**Other**	**Black**
w6*	Male	America	Black
w7	**Female**	**America**	**Black**
w8*	Male	Other	Black
w9	Male	Other	White
w10*	Female	America	White

Table 3: Ranking of the 10 workers for the query "Home Cleaning" in San Francisco on an online job marketplace

Ranking	Worker	$f_q^l(w)$
1	w3	0.9
2	w8	0.8
3	w6	0.7
4	w2	0.6
5	w1	0.5
6	w4	0.4
7	w7	0.3
8	w5	0.2
9	w9	0.1
10	w10	0

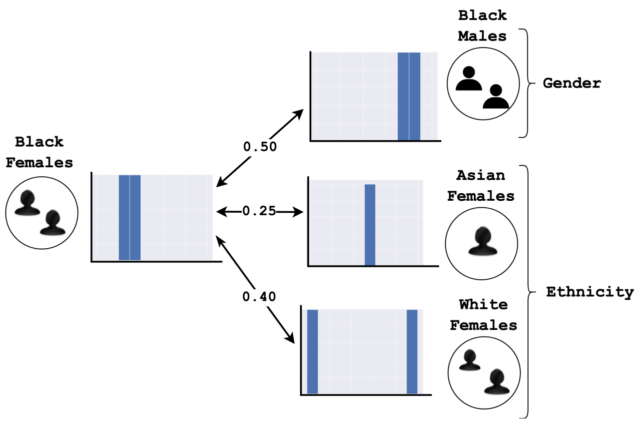


Figure 4: The unfairness of "Black Females" based on the ranking in Table 3 using EMD is $\frac{0.70+0.50+0.30}{3}$ = 0.50.

Since the actual scores of each worker for a query and location, $f_q^l(w)$ is not always available (no job marketplace makes that score available), we rely on the rank of workers $rank(w, q, l)$ to compute their relevance for a query and location. The rank of workers for a pair (q, l) is available since it can be observed in the results of running q at l. We can hence compute $rel_q^l(w)$, the relevance score of a worker as follows:

$$rel_q^l(w) = 1 - \frac{rank(w, q, l)}{N}$$

where $rank(w, q, l)$ denotes the rank of worker w for query q at location l as shown in Table 3, and N is the number of workers in the resultset, here set to 10. The relevance scores generated for all workers in our example are reported in Table 3.

To compute the EMD unfairness of Black Females for this query at this location, we generate a histogram for Black Females and each of the comparable groups based on the relevance scores $rel_q^l(w)$ computed for workers. We then compute the average EMD between the histogram of Black Females and each of the comparable groups' histograms.

3.3.2 *Exposure Unfairness.*

In the exposure notion of fairness, the intuition is that higher ranked workers receive more exposure as people tend to only examine top-ranked results. Thus, each worker receives an exposure inversely proportional to her rank $d_{<g,q,l>}$ as follows. First, for every $w \in g$, we compute her exposure as:

$$exp_q^l(w) = \frac{1}{log(1 + rank(w, q, l))}$$

We also compute the relevance of worker $w \in g$ as $rel_q^l(w)$ as defined above. Now, the exposure of a group of workers g is set to:

$$exp_q^l(g) = \frac{\sum_{w \in g} exp_q^l(w)}{\sum_{g' \in g \cup_{a \in A(g)} variants(g,a)} \sum_{w \in g'} exp_q^l(w)}$$

Similarly, we define the relevance of a group g as:

$$rel_q^l(g) = \frac{\sum_{w \in g} rel_q^l(w)}{\sum_{g' \in g \cup_{a \in A(g)} variants(g,a)} \sum_{w \in g'} rel_q^l(w)}$$

Next, we assume that each group g should receive exposure proportional to its relevance. We thus measure deviation from the ideal exposure using the L1-norm as the unfairness of a group g: $d_{<g,q,l>} = |exp_q^l(g) - rel_q^l(g)|$.

Figure 5 illustrates how the exposure unfairness of Black Females, g, is calculated. To compute the exposure unfairness of Black Females for this query in the given location, we compute the exposure and relevance of all Black Female workers (bold in Table 2) and the workers belonging to their comparable groups (* in Table 2) using $f_q^l(w)$ and ranking shown in 3. We then sum up the exposure and relevance values for all Black Females workers and the comparable groups separately.

3.4 Notation Generalization

We have used $d_{<g,q,l>}$ to refer to the unfairness for group g for the job-related query q at location l. This value is obtained by contrasting the ranking for group g with the ranking of all its comparable groups. Unfairness can also be computed for several job-related queries and at multiple locations. For a set of queries $Q \subseteq \mathcal{Q}$ and a set of locations $L \subseteq \mathcal{L}$, we can compute the unfairness for group g as follows:

$$d_{<g,Q,L>} = \text{avg}_{q \in Q, l \in L}\, d_{<g,q,l>}$$

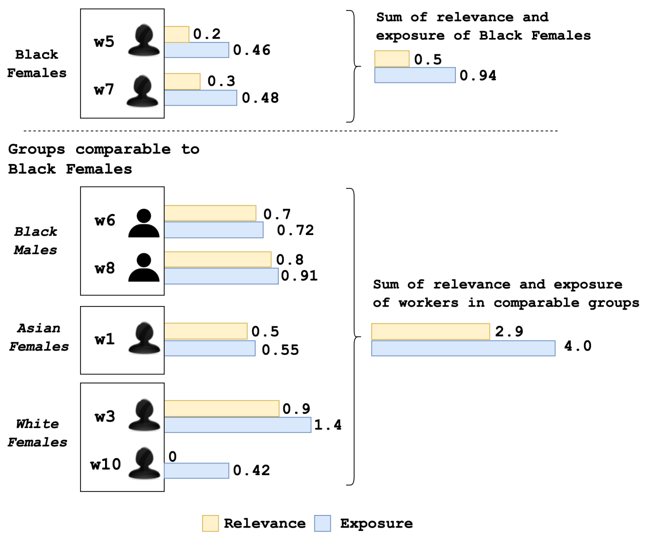


Figure 5: Computing the unfairness for "Black Females" based on the ranking in Table 3. The exposure of Black Females is $\frac{0.94}{0.94+4.0} = 0.19$**. Its relevance is** $\frac{0.5}{0.5+2.9} = 0.15$**. Its unfairness is** $0.19 - 0.15 = 0.04$**.**

Similarly, we could compute the unfairness for a set of groups $G \subseteq \mathcal{G}$ at a location $l \in \mathcal{L}$ for all queries in $Q \subseteq \mathcal{Q}$ as follows:

$$d_{<G,Q,l>} = \text{avg}_{g \in G, q \in Q}\, d_{<g,q,l>}$$

Finally, we could also compute the unfairness for a set of groups $G \subseteq \mathcal{G}$ for a given query $q \in \mathcal{Q}$ at all locations $L \subseteq \mathcal{L}$ as follows:

$$d_{<G,q,L>} = \text{avg}_{g \in G, l \in L}\, d_{<g,q,l>}$$

4 PROBLEMS AND ALGORITHMS

In this section, we first provide two generic problem formulations that capture the variety of group fairness questions we may ask (Section 4.1). We then describe the algorithms we designed to solve those problems (Section 4.2).

4.1 Problem Variants

To formulate a generic problem, we will use the term *dimension* to refer to one of group, query or location. Our first problem aims to quantify how well a site treats groups for different queries and at different locations. The problem returns instances of a chosen dimension, e.g., groups, and aggregates their unfairness values along the two others, e.g., queries and locations.

Problem 1 (Fairness Quantification). *Given R a dimension to be returned and two other dimensions AGG1 and AGG2 to be aggregated, return the k results in R for which the site is most/least unfair, where the unfairness for each result $r \in R$, $d_{<AGG1,AGG2,r>}$, is computed as:* $\text{avg}_{agg1 \in AGG1,\, agg2 \in AGG2}\, d_{<agg1,agg2,r>}$

There are 3 instances of this problem: one where R is a set of groups, one where it is a set of queries, and the third one where it is a set of locations.

When R is a set of groups, the problem, referred to as *group-fairness*, returns k groups for which the site is most/least unfair. For instance, it could be used to find *the 5 groups for which the site is least unfair with respect to all queries at all locations* or to answer the question: *Out of Black Males, Asian Males, Asian Females, and White Females, what are the 2 groups for which the site, say Google job search, is the most unfair?*

When R is a set of queries, the problem, referred to as *query-fairness* returns k queries which are the most/least unfair. This instance of the problem can address questions such as *what are the 5 least unfair queries at all locations?* or *which 2 queries are black males most likely to get in the West Coast?*

Finally, when R refers to locations, the problem, referred to as *location-fairness* addresses questions such as *Which 3 locations are the easiest to find a job at?* or *out of NYC, Boston and Washington DC, what is the least unfair location for women looking for an event staffing job on a given site, say TaskRabbit?*

Our second problem formulation aims to capture comparisons between two dimensions. It admits two dimensions to compare, e.g. males and females, or NYC and San Francisco, or cleaning services and event staffing, and it returns a breakdown of comparison dimensions into sub-dimensions whose fairness comparison differs from the comparison of the input dimensions.

Problem 2 (Fairness Comparison). *Given two comparison dimensions r_1 and r_2, and a breakdown dimension B, return all $b \in B$ s.t.* $d_{<r_1,b>} >= d_{<r_2,b>} \wedge d_{<r_1,B>} <= d_{<r_2,B>}$ $\vee d_{<r_1,b>} <= d_{<r_2,b>} \wedge d_{<r_1,B>} >= d_{<r_2,B>}$

The first instance of our comparison problem is referred to as *group-comparison* in which r_1 and r_2 are demographic groups. For example, when r_1 refers to Males, r_2 to Females, and B to locations, fairness comparison returns all locations where the comparison between males and females differs from that of all males and females. Table 4 shows an example. In this case, our problem returns the unfairness values of males and females at those two locations that compare differently from all locations.

Table 4: Comparison between Male and Female workers in Oklahoma City and Salt Lake City differ from the overall

Group-comparison	Males	Females
All	0.48	0.74
Oklahoma City, OK	0.853	0.732
Salt Lake City, UT	0.933	0.553

The second instance of our comparison problem is referred to as *query-comparison*. For example, if r_1 is lawn mowing and r_2 furniture mounting and B is ethnicity, fairness comparison returns all ethnicities for which the comparison between lawn mowing and furniture mounting differs from the whole population. For instance, our problem finds that ethnicity Black must be returned because the unfairness values between lawn mowing and furniture mounting for blacks compare differently from all ethnicities.

The third instance of our comparison problem is referred to as *location-comparison*. For example when r_1 is California, and r_2 is Arizona, and B is outdoor home services, fairness comparison returns all queries related to outdoor home services (e.g., lawn mowing, garage cleaning, patio painting, etc), for which the comparison in California and Arizona differs from all outdoor home services. Our problem returns the jobs garage cleaning and patio painting because the unfairness values between California and

Arizona for those two jobs are different from all outdoor home services.

4.2 Algorithms

The computational complexity of our problems calls for designing scalable solutions. In this section, we propose adaptations of Fagin's algorithms to solve our problems. We first describe the indices we generate: *group-based*, *query-based*, and *location-based*.

The *group-based* indices associate to every (q, l) pair an inverted index where groups are sorted in descending order based on $d_{<g,q,l>}$.

The *query-based* indices associate to every (g, l) pair an inverted index where queries q are sorted in descending order based on $d_{<g,q,l>}$.

The *location-based* indices associate to every (g, q) pair an inverted index where locations l are sorted in descending order based on $d_{<g,q,l>}$. Table 5 shows an illustration of the three types of indices.

Table 5: Group-based, query-based, location-based indices

$I_{(q,l)}$

.	.
g_j	$d(g_j, q, l)$
.	.

$I_{(g,l)}$

.	.
q_j	$d(q_j, g, l)$
.	.

$I_{(g,q)}$

.	.
l_j	$d(g, q, l_j)$
.	.

Algorithm 1 is an adaption of Fagin's Threshold Algorithm [10] for the *group-fairness* instance of our problem. It finds the k groups for which the site is most unfair. The algorithm takes as input a set of groups G, a set of queries Q and a set of locations L, and returns k groups. It makes use of the group-based indices (Table 5).

All other instances of Problem 1 including query-fairness, location-fairness and their bottom k versions, are adaptations of Algorithm 1.

Algorithm 2 solves our second problem (Problem 2) for the *group-comparison* instance of our problem. It takes as input 2 groups g_1 and g_2 and a breakdown dimension L. It first calls Algorithm 3 to compute the fairness values of g_1 and g_2 for all values of L and all queries Q. It then calls the query-based index to sum up all the values for all the queries by scanning the index for each location and for each of the two groups. Finally, it returns only those locations for which the order on unfairness values for the two groups is reversed. All other instances of Problem 2 including query-comparison and location-comparison are adaptations of Algorithm 2.

Algorithm 3 computes the fairness for a group g for all queries in Q and all locations in L. It takes as input a group g, a set of queries Q and a set of locations L, and returns the average unfairness value for g over all queries and locations.

5 EXPERIMENTS

Our experiments use real data collected from TaskRabbit and Google Search and were conducted from June to August 2019. We first describe the overall setup for each platform and then report the results.

5.1 Experimental setup

5.1.1 TaskRabbit setup. TaskRabbit is an online marketplace that matches freelance labor with local demand, allowing consumers to find immediate help with everyday tasks.

Algorithm 1 findTopKGroups(G: a set of groups, Q: a set of queries, L: a set of locations, k: an integer)

```
1: $topk \leftarrow createMinHeap()$
2: Initialize $|Q| * |L|$ cursors to 0
3: $\tau \leftarrow +\infty$
4: while $topk.minValue() < \tau$ or $topk.size() < k$ do
5:     $\tau \leftarrow 0$
6:     for $q \in Q$ do
7:         for $l \in L$ do
8:             $(g, d_{<g,q,l>}) \leftarrow I_{(q,l)}.find(cur_{(q,l)})$   ▷ Read entry in $I_{(q,l)}$ pointed to by cursor $cur_{(q,l)}$
9:             $d_{<g,Q,L>} \leftarrow d_{<g,q,l>}$
10:            $\tau \leftarrow \tau + d_{<g,q,l>}$
11:            for $q' \in Q$ do
12:                for $l' \in L$ do
13:                    if $q' \neq q$ or $l' \neq l$ then
14:                        $d_{<g,q',l'>} \leftarrow I_{(q',l')}.find(g)$   ▷ Perform a random access on $I_{(q',l')}$ to retrieve the unfairness value of $g$ for the pair $(q', l')$
15:                        $d_{<g,Q,L>} \leftarrow d_{<g,Q,L>} + d_{<g,q',l'>}$
16:                    end if
17:                end for
18:            end for
19:            $d_{<g,Q,L>} \leftarrow d_{<g,Q,L>}/(|Q| * |L|)$
20:            if $topk.size() < k$ then
21:                $topk.insert(g, d_{<g,Q,L>})$
22:            else
23:                if $topk.minValue() < d_{<g,Q,L>}$ then
24:                    $topk.pop()$
25:                    $topk.insert(g, d_{<g,Q,L>})$
26:                end if
27:            end if
28:            $cur_{(q,l)} \leftarrow cur_{(q,l)} + 1$
29:        end for
30:    end for
31:    $\tau \leftarrow \tau/(|Q| * |L|)$
32: end while
33: return $topk$
```

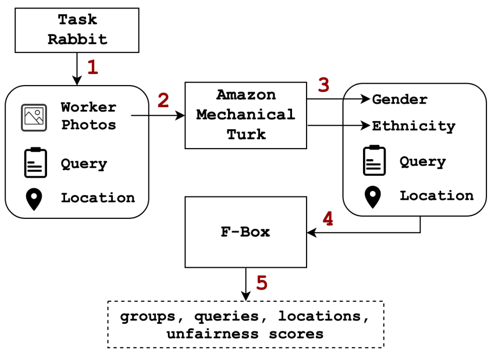


Figure 6: Flow of TaskRabbit Experiments

TaskRabbit is supported in 56 different cities mostly in the US. For each location, we retrieved all jobs offered in that location. We thus generated a total of 5,361 job-related queries, where each query is a combination of a job and a location, e.g., *Home Cleaning* in *New York*.

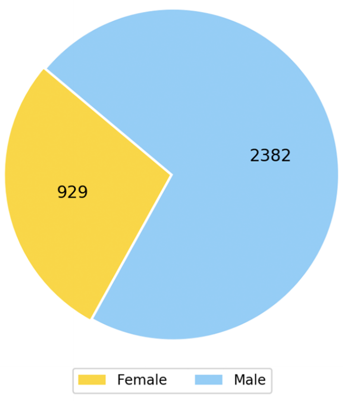


Figure 7: Gender breakdown

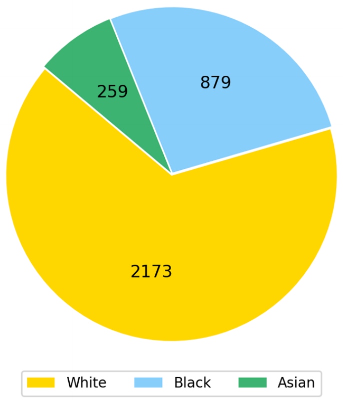


Figure 8: Ethnic breakdown

Algorithm 2 CompareGroups(Groups: g_1, g_2, L: a set of locations as breakdown, Q: a set of queries)

```
1:  $loc \leftarrow \emptyset$
2:  $d_{<g_1,Q,L>} \leftarrow ComputeGroupUnfairness(g_1, Q, L)$
3:  $d_{(<g_2,Q,L>} \leftarrow ComputeGroupUnfairness(g_2, Q, L)$
4:  for $l \in L$ do
5:      $sum_1 \leftarrow 0$
6:      $sum_2 \leftarrow 0$
7:      $cur_1 \leftarrow 0$
8:      $cur_2 \leftarrow 0$
9:      for $q \in Q$ do
10:         $sum_1 + = I_{(g_1,l)}.find(cur_1)$
11:         $sum_2 + = I_{(g_2,l)}.find(cur_2)$
12:         $cur_1 \leftarrow cur_1 + 1$
13:         $cur_2 \leftarrow cur_2 + 1$
14:     end for
15:     if $reversed(sum_1, sum_2, d_{<g_1,Q,L>}, d_{<g_2,Q,L>})$ then
16:         $loc + = l$
17:     end if
18: end for
19: return $loc$
```

Algorithm 3 ComputeGroupUnfairness(g: a group, Q: a set of queries, L: a set of locations)

```
1: $sum \leftarrow 0$
2: for $q \in |Q|$ do
3:     for $l \in |L|$ do
4:         $sum \leftarrow sum + I_{(q,l)}.find(g)$        ▷ Perform a random
   access on $I_{(q,l)}$ to retrieve the unfairness value of $g$ for the
   pair $(q, l)$
5:     end for
6: end for
7: return $sum/(|Q| * |L|)$
```

Figure 6 summarizes the flow of the TaskRabbit experiment. Our algorithms are encapsulated in the F-Box. For each one of the 5,361 queries, we extracted the rank of each tasker, their badges, reviews, profile pictures, and hourly rates, where the number of taskers returned per query was limited to 50. Since the demographics of the taskers were not readily available on the platform, we asked workers on Amazon Mechanical Turk (AMT)[6] to indicate the gender and ethnicity of the TaskRabbit taskers based on their profile pictures. The taskers were given pre-defined categories for gender = {Male, Female} and ethnicity = {Asian, Black, White}. Each profile picture was labeled by three different contributors on AMT and a majority vote determined the final label.

The gender and ethnic breakdowns of the taskers in our dataset are shown in Figures 7 and 8. Overall, we had a total of 3,311 unique taskers in our crawled dataset, the majority of which were male ($\approx$ 72%) and white ($\approx$ 66%).

5.1.2 Google Search setup. Google Search personalizes queries based on a user's profile which includes user data, activity, and saved preferences. While personalization can be beneficial to users, it may introduce the possibility of unfairness, which we aim to observe.

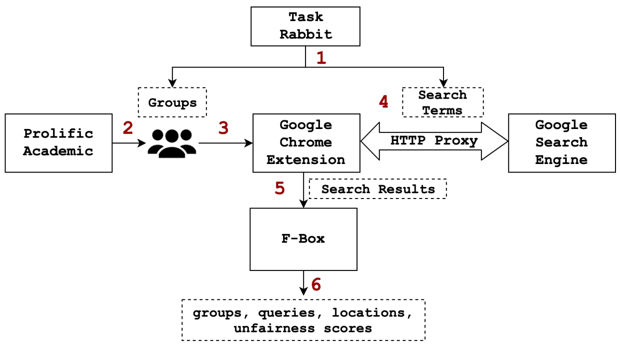


Figure 9: Flow of Google job search Experiments

We designed the experiments to ensure that variations in the search results are largely based on differences in profiles rather than other known noise sources identified in related work such as carry-over-effect, geolocation, distributed infrastructure, and A/B testing [12].

The flow of the Google Search experiment is summarized in Figure 9. We first derived user groups of interest and equivalent Google search terms from data crawled from TaskRabbit. We then setup user studies on Prolific Academic[7] and recruited participants, who belong to those groups. We asked those participants

[6]https://mturk.com
[7]https://prolific.co

Table 6: Sample TaskRabbit queries and equivalent Google search terms

TaskRabbit Query	Location	Equivalent Google Search Terms
run errand	London, UK	*run errand jobs near London UK, errand service jobs near London UK, errand runner jobs near London, UK, errands and odd jobs near London, UK, jobs running errands for seniors near London, UK*
yard work	New York City, NY	*yard work jobs near New York City, NY, yard worker near New York City, NY, lawn work needed near New York City, NY, yard help needed near New York City, NY, yard work help wanted near New York City, NY*

to use our Google Chrome extension that automatically executes on Google the search queries derived. Finally, we processed the results and provided them as input to the F-Box and recorded unfairness values.

Search Queries. For our Google Search experiments, we selected 20 queries (the top 10 and bottom 10 frequently searched queries) and their corresponding locations from data crawled in TaskRabbit. From this list, we chose those from 10 unique locations. We then generated equivalent search terms using Google Keyword Planner, a tool that outputs a list of search terms similar or related to a given search string and a location. We shortlisted 50 formulations for each query, manually examined them, then chose 5 search terms whose results are similar to the original term. Table 6 shows sample queries from TaskRabbit and their equivalent Google search terms.

Groups. The combination of pre-defined categories for gender = {Male, Female} and ethnicity = {Asian, Black, White} results in six groups: Asian Male, Asian Female, Black Male, Black Female, White Male, and White Female.

We recruited an average of 3 participants per study through Prolific Academic, a crowdsourcing platform that allows researchers to recruit participants who have been categorized through the platform's screening mechanism.

User Study. Given the search terms and the groups, we have a total of 60 studies. Each study is composed of two tasks. In the first task, a participant is asked to set her browsing language to English and install our Google Chrome extension that runs the search terms. Participants who are able to successfully complete the first task are invited to do a second task where they are asked whether they think the instructions of the first task were clear and whether the reward is fair. The reward for each task is 0.50 GBP.

Given the distribution of workers on Prolific Academic, we ended up with 10 locations, namely London, UK, New York City, NY, Los Angeles, CA, Boston, MA, Bristol, UK, Charlotte, NC, Pittsburg, PA, Birmingham, UK, Manchester, UK and Detroit, MI. For those 10 locations, we have five categories of jobs: yard work, general cleaning, event staffing, moving job and run errand. Table 7 shows the number of locations per job that we collected search results for.

Google Chrome Extension and noise handling. We developed a Google Chrome extension that automatically executes the Google search terms. The extension runs the five search terms every 12 minutes to minimize noise due to the carry-over effect. Meanwhile, every search term is executed at least twice to account for noise caused by A/B testing. The extension also sets the browser's location to a fixed location and uses a proxy so that all queries originate from the same location thus minimizing noise caused by distributed infrastructure and different geolocations. The search results are then inserted to a Google Sheets document. We emphasized to the participants that we store no identifying information about them.

Table 7: Number of locations per job

Job	Location
yard work	4
general cleaning	3
event staffing	1
moving job	1
run errand	1

5.2 Fairness quantification

5.2.1 TaskRabbit fairness quantification. We report the results of solving our fairness quantification problem (Problem 1 in Section 4.1) for groups, queries and locations using both EMD and exposure to measure unfairness (see Sections 3.3.1 and 3.3.2 for their formal definitions).

Table 8 reports all groups in TaskRabbit ranked by their decreasing unfairness values (both EMD and exposure). We can see that the two measures agree on the top 7 groups for whom TaskRabbit is the most unfair: *Asian Females and Asian Males are the ones most discriminated against.*

Table 9 reports all job types in TaskRabbit ranked by their decreasing unfairness values (both EMD and exposure). The two measures largely agree on the ranking showing that *Handyman and Yard Work are the most unfair jobs and that Furniture Assembly and Delivery, are the fairest.*

Since the number of locations is large, we report the top and bottom 10 locations in Tables 10 and 11 respectively. The results show that *Birmingham, UK and Oklahoma City, OK are the least fair while Chicago and San Francisco are the fairest locations across all jobs.*

We also report the fairest/unfairest locations for some jobs and the fairest/unfairest jobs for some locations. *For Handyman and Run Errands, the fairest location is San Francisco Bay Area, CA for both when using EMD and, when using exposure, it is Boston, MA for Handyman, and San Francisco Bay Area, CA for Run Errands. The unfairest location for both jobs is Birmingham, UK when using EMD.*

For Birmingham, Detroit, and Nashville, the fairest jobs are Delivery and Furniture Assembly for all, and the unfairest are Yard Work, General Cleaning, and General Cleaning, respectively. For Philadelphia, San Diego and Chicago, the fairest jobs are Delivery, Furniture Assembly, and Delivery, respectively, and the unfairest is Yard Work for Birmingham, Detroit, and Run Errands for Nashville.

Table 8: EMD and Exposure of all groups in TaskRabbit, *ranked from the unfairest to the fairest.*

Group	EMD	Group	Exposure
Asian Female	0.876	Asian Female	0.821
Asian Male	0.755	Asian Male	0.662
Black Female	0.726	Black Female	0.615
Asian	0.694	Asian	0.594
Black Male	0.578	Black Male	0.413
White Female	0.542	White Female	0.359
Black	0.498	Black	0.341
Male	0.468	Female	0.299
Female	0.468	White Male	0.154
White	0.448	Male	0.117
White Male	0.421	White	0.104

Table 9: EMD and Exposure for all jobs in TaskRabbit, *ranked from the unfairest to the fairest.*

Job	EMD	Job	Exposure
Handyman	0.692	Handyman	0.515
Event Staffing	0.639	Event Staffing	0.504
General Cleaning	0.611	General Cleaning	0.456
Yard Work	0.672	Yard Work	0.5
Moving	0.604	Moving	0.418
Delivery	0.499	Furniture Assembly	0.383
Furniture Assembly	0.541	Delivery	0.331
Run Errands	0.519	Run Errands	0.352

Table 10: 10 unfairest locations using EMD and Exposure, *ranked from the unfairest to the fairest.*

City	EMD	City	Exposure
Birmingham, UK	1	Birmingham, UK	0.926
Oklahoma City, OK	0.998	Oklahoma City, OK	0.819
Bristol, UK	0.91	Bristol, UK	0.761
Manchester, UK	0.851	Manchester, UK	0.739
New Haven, CT	0.838	New Haven, CT	0.67
Milwaukee, WI	0.824	Memphis, TN	0.668
Indianapolis, IN	0.815	Milwaukee, WI	0.668
Nashville, TN	0.808	Charlotte, NC	0.643
Detroit, MI	0.806	Nashville, TN	0.637

Table 11: 10 fairest locations using EMD and Exposure, *ranked from the fairest to the unfairest.*

City	EMD	City	Exposure
Chicago, IL	0.274	Chicago, IL	0.107
San Francisco, CA	0.286	San Francisco, CA	0.12
Washington, DC	0.329	Boston, MA	0.169
Los Angeles, CA	0.33	Washington, DC	0.174
Boston, MA	0.353	Los Angeles, CA	0.189
Atlanta, GA	0.4	Houston, TX	0.217
Houston, TX	0.417	Atlanta, GA	0.234
Orlando, FL	0.431	San Diego, CA	0.241
Philadelphia, PA	0.45	Orlando, FL	0.242
San Diego, CA	0.454	Philadelphia, PA	0.273

In summary, our results demonstrate the flexibility and expressiveness provided by solving the fairness quantification problem for groups, queries and locations. They also provide the ability to generate hypotheses to be tested across platforms, in our case from TaskRabbit to Google job search.

5.2.2 Google fairness quantification. We ran our unfairness quantification algorithm (Algorithm 1) on the data crawled from Google Search. Our algorithm found that regardless of the metrics we use, Kendall Tau or Jaccard Index, *the most discriminated against group is White Females and the least is Black Males.* This indicates that search results between White Females were the most different, whereas those for Black Males were the most similar.

When quantifying unfairness for locations, we found that *Washington, DC is deemed the fairest* indicating no difference in search results between users at this location using both Jaccard Index and Kendall Tau. On the other hand, *London, UK is deemed the unfairest location.*

Finally, for queries, we found that using both metrics, *Yard Work jobs are deemed the most unfair whereas Furniture Assembly jobs are deemed the most fair.*

5.3 Fairness comparison

5.3.1 TaskRabbit fairness comparison. We report the results of solving our fairness comparison problem (Problem 2 in Section 4.1) in Tables 12, 13, 14 and 15. The tables only report *the locations, demographics, and jobs that differ from the overall comparison.*

Table 12: Comparison between Male and Female workers after including locations using Exposure. *The listed locations are the ones for which Females are treated more fairly than Males, which differs from the overall comparison.*

Group-comparison	Males	Females
All	**0.117**	**0.299**
Charlotte, NC	0.399	0.345
Chicago, IL	0.062	0.062
Nashville, TN	0.330	0.309
Norfolk, VA	0.331	0.168
San Francisco Bay Area, CA	0.084	0.084
St. Louis, MO	0.255	0.190

Table 13: Comparison between Lawn Mowing and Event Decorating workers after including Ethnicity using EMD. *Caucasians are the ones for which the comparison between Lawn Mowing jobs and Event Decorating jobs is different from the whole population, showing that Lawn Mowing jobs are fairer than Event Decorating for Caucasians.*

Job-comparison	Lawn Mowing	Event Decorating
All	**0.674**	**0.613**
White	0.552	0.569

In summary, we can conclude that overall, EMD and Exposure yield the same observations when solving the fairness comparison problem on TaskRabbit.

Table 14: Comparison between Lawn Mowing and Event Decorating jobs after including Ethnicity using Exposure. ***Unlike Table 13, in this case blacks are the ones for whom Lawn Mowing jobs are fairer than Event Decorating. This warrants further investigation in the future.***

Job-comparison	Lawn Mowing	Event Decorating
All	**0.500**	**0.442**
Black	0.445	0.453

Table 15: Comparison between San Francisco Bay Area and Chicago after including General Cleaning jobs using EMD. ***San Francisco is shown to be fairer for all jobs but the trend is inverted for the listed jobs.***

Location-comparison	San Francisco Bay Area, CA	Chicago, IL
All	**0.213**	**0.233**
Back To Organized	0.198	0.135
Organize & Declutter	0.224	0.191
Organize Closet	0.174	0.153

5.3.2 Google fairness comparison. Similarly to TaskRabbit, we report the results of solving our fairness comparison problem (Problem 2 in Section 4.1) in Tables 16, 17, 18, 19, 20, and 21. *The tables show the cases that differ from the overall comparison.*

Table 16: Comparison between Male and Female workers after including locations using Kendall Tau. ***The listed locations are the ones for which Females are treated more fairly than Males, which differs from the overall comparison.***

Group-comparison	Males	Females
All	**0.537**	**0.552**
Birmingham, UK	0.906	0.901
Bristol, UK	0.921	0.918
Detroit, MI	0.928	0.901
New York City, NY	0.913	0.906

Table 17: Comparison between Male and Female workers after including locations using Jaccard. ***The results differ from the ones in Table 16 because the overall results differ. This warrants further investigation in the future.***

Group-comparison	Males	Females
All	**0.395**	**0.393**
Boston, MA	0.894	0.896
Charlotte, NC	0.893	0.901
London, UK	0.776	0.785
Los Angeles, CA	0.875	0.878
Manchester, UK	0.869	0.875
Pittsburgh, PA	0.877	0.88

In summary, we observed that Kendall Tau and Jaccard report mostly similar results when solving the fairness comparison problem on Google job search. This is quite encouraging and merits further investigation in future work.

Table 18: Comparison between Running Errands jobs and General Cleaning jobs after including Ethnicity using Kendall Tau.

Job-comparison	Running Errands	General Cleaning
All	**0.927**	**0.926**
Black	0.927	0.950
Asian	0.925	0.938

Table 19: Comparison between Running Errands jobs and General cleaning jobs after including Ethnicity using Jaccard. ***The results differ from those reported in Table 18. This warrants further investigation in the future.***

Job-comparison	Running Errands	General Cleaning
All	**0.902**	**0.887**
Black	0.903	0.94

Table 20: Comparison between Boston, MA and Bristol, UK after including General Cleaning jobs using Kendall Tau. This result is similar to the one reported in Table 21.

Group Comparison	Boston, MA	Bristol, UK
All	**0.641**	**0.689**
office cleaning jobs	0.735	0.627
private cleaning jobs	0.572	0.398

Table 21: Comparison between Boston, MA and Bristol, UK after including General Cleaning jobs using Jaccard. This result is similar to the one reported in Table 20.

Group Comparison	Boston, MA	Bristol, UK
All	**0.447**	**0.603**
private cleaning jobs	0.403	0.364

6 CONCLUSION

We develop a framework to study fairness in job search and a detailed empirical evaluation of two sites: Google job search and TaskRabbit. We formulate two generic problems. Our first problem returns the k least/most unfair dimensions, i.e., the k groups for which a site is most/least unfair, the k least/most unfair jobs (queries), or the k least/most unfair locations. Our second problem captures comparisons between two dimensions. It admits two dimensions to compare, e.g. males and females, or NYC and San Francisco, or cleaning services and event staffing, and it returns a breakdown of those dimensions that exhibits different unfairness values (for instance, on TaskRabbit, while females are discriminated against when compared to males, this trend is inverted in California). We apply threshold-based algorithms to solve our problems. We report the results of extensive experiments on real datasets from TaskRabbit and Google job search.

Our framework can be used to generate hypotheses and verify them across sites. That is what we did from TaskRabbit to Google job search. It can also be used to verify hypotheses by solving the comparison problem. As a result, one could use it in iterative scenarios where the purpose is to explore and compare fairness. We are currently designing such exploratory scenarios.

ACKNOWLEDGMENTS

This work is partially supported by the American University of Beirut Research Board (URB)

REFERENCES

[1] Abolfazl Asudeh, H. V. Jagadish, Julia Stoyanovich, and Gautam Das. 2019. Designing Fair Ranking Schemes. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* 1259–1276.

[2] Asia J Biega, Krishna P Gummadi, and Gerhard Weikum. 2018. Equity of Attention: Amortizing Individual Fairness in Rankings. *arXiv preprint arXiv:1805.01788* (2018).

[3] Ria Mae Borromeo, Thomas Laurent, Motomichi Toyama, and Sihem Amer-Yahia. 2017. Fairness and Transparency in Crowdsourcing. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.* 466–469. https://doi.org/10.5441/002/edbt.2017.46

[4] Toon Calders and Sicco Verwer. 2010. Three naive Bayes approaches for discrimination-free classification. *Data Mining and Knowledge Discovery* 21, 2 (01 Sep 2010), 277–292. https://doi.org/10.1007/s10618-010-0190-x

[5] Chris Callison-Burch. 2014. Crowd-Workers: Aggregating Information Across Turkers To Help Them Find Higher Paying Work. In *The Second AAAI Conference on Human Computation and Crowdsourcing (HCOMP-2014).* http://cis.upenn.edu/~ccb/publications/crowd-workers.pdf

[6] L Elisa Celis, Damian Straszak, and Nisheeth K Vishnoi. 2017. Ranking with fairness constraints. *arXiv preprint arXiv:1704.06840* (2017).

[7] David Durward, Ivo Blohm, and Jan Marco Leimeister. 2016. Is There PAPA in Crowd Work?: A Literature Review on Ethical Dimensions in Crowdsourcing. In *Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), 2016 Intl IEEE Conferences.* IEEE, 823–832.

[8] Benjamin Edelman, Michael Luca, and Dan Svirsky. 2017. Racial discrimination in the sharing economy: Evidence from a field experiment. *American Economic Journal: Applied Economics* 9, 2 (2017), 1–22.

[9] Shady Elbassuoni, Sihem Amer-Yahia, Christine El Atie, Ahmad Ghizzawi, and Bilel Oualha. 2019. Exploring Fairness of Ranking in Online Job Marketplaces. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019.* 646–649.

[10] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences* 66, 4 (2003), 614–656.

[11] Sorelle A. Friedler, Carlos Scheidegger, and Suresh Venkatasubramanian. 2016. On the (im)possibility of fairness. *CoRR* abs/1609.07236 (2016). http://arxiv.org/abs/1609.07236

[12] Aniko Hannak, Piotr Sapiezynski, Arash Molavi Kakhki, Balachander Krishnamurthy, David Lazer, Alan Mislove, and Christo Wilson. 2013. Measuring personalization of web search. In *Proceedings of the 22nd international conference on World Wide Web.* ACM, 527–538.

[13] Aniko Hannak, Claudia Wagner, David Garcia, Alan Mislove, Markus Strohmaier, and Christo Wilson. 2017. Bias in Online Freelance Marketplaces: Evidence from TaskRabbit and Fiverr. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing, CSCW 2017, Portland, OR, USA, February 25 - March 1, 2017.* 1914–1933.

[14] Benjamin V. Hanrahan, Jutta K. Willamowski, Saiganesh Swaminathan, and David B. Martin. 2015. TurkBench: Rendering the Market for Turkers.. In *CHI*, Bo Begole, Jinwoo Kim, Kori Inkpen, and Woontack Woo (Eds.). ACM, 1613–1616. http://dblp.uni-trier.de/db/conf/chi/chi2015.html#HanrahanWSM15

[15] Michael J. Kearns, Seth Neel, Aaron Roth, and Zhiwei Steven Wu. 2018. Preventing Fairness Gerrymandering: Auditing and Learning for Subgroup Fairness. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018.* 2569–2577.

[16] Keith Kirkpatrick. 2016. Battling algorithmic bias: how do we ensure algorithms treat us fairly? *Commun. ACM* 59 (2016), 16–17.

[17] Karen Levy and Solon Barocas. 2017. Designing against discrimination in online markets. *Berkeley Tech. LJ* 32 (2017), 1183.

[18] Michael Luca and Rayl Fisman. 2016. Fixing Discrimination in Online Marketplaces. *Harvard Business Review* (Dec 2016). https://hbr.org/product/fixing-discrimination-in-online-marketplaces/R1612G-PDF-ENG

[19] Mike Noon. 2010. The shackled runner: time to rethink positive discrimination? *Work, Employment and Society* 24, 4 (2010), 728–739.

[20] Ofir Pele and Michael Werman. 2009. Fast and robust earth mover's distances. In *2009 IEEE 12th International Conference on Computer Vision.* IEEE, 460–467.

[21] Alex Rosenblat, Karen EC Levy, Solon Barocas, and Tim Hwang. 2017. Discriminating Tastes: Uber's Customer Ratings as Vehicles for Workplace Discrimination. *Policy & Internet* 9, 3 (2017), 256–279.

[22] Ashudeep Singh and Thorsten Joachims. 2018. Fairness of Exposure in Rankings. *arXiv preprint arXiv:1802.07281* (2018).

[23] Latanya Sweeney. 2013. Discrimination in Online Ad Delivery. *CoRR* abs/1301.6822 (2013). http://arxiv.org/abs/1301.6822

[24] Florian Tramèr, Vaggelis Atlidakis, Roxana Geambasu, Daniel J. Hsu, Jean-Pierre Hubaux, Mathias Humbert, Ari Juels, and Huang Lin. 2015. Discovering Unwarranted Associations in Data-Driven Applications with the FairTest Testing Toolkit. *CoRR* abs/1510.02377 (2015). http://arxiv.org/abs/1510.02377

[25] Ke Yang and Julia Stoyanovich. 2017. Measuring fairness in ranked outputs. In *SSDM.* 22.

[26] Meike Zehlike, Francesco Bonchi, Carlos Castillo, Sara Hajian, Mohamed Megahed, and Ricardo Baeza-Yates. 2017. Fa* ir: A fair top-k ranking algorithm. In *CIKM.* 1569–1578.

[27] Indre Zliobaite. 2015. A survey on measuring indirect discrimination in machine learning. *CoRR* abs/1511.00148 (2015). http://arxiv.org/abs/1511.00148

Inventory Reduction via Maximal Coverage in E-Commerce

Shay Gershtein
Tel Aviv University
shayg1@mail.tau.ac.il

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

Slava Novgorodov
eBay Research
snovgorodov@ebay.com

ABSTRACT

Many e-commerce platforms serve as an intermediary between companies and consumers, receiving a commission per purchase. To increase sales, these platforms tend to offer as many items as possible. However, in many situations a reduced subset of the items should be offered for sale, e.g., when opening an express delivery branch, starting operations in a new region, or disposing of redundant items to improve data quality and decrease maintenance costs. In all these cases it is imperative to select a reduced inventory which maximally covers consumer needs. A naïve, yet popular, solution is to focus on the top selling items. This however ignores the hidden relations between items, and in particular the tendency of shoppers to buy, in the absence of an item they are looking for, a satisfying alternative.

In this paper we introduce the *Preference Cover problem*, and investigate its application to practical inventory reduction. Given a large set of items, a bound on the number of items that can be retained and consumer preferences in terms of items popularity and suitability as alternatives, the goal is to select a reduced inventory which maximizes the likelihood of a purchase. We first model the problem via a dedicated weighted directed graph which captures the relevant information, then study two problem variants, which differ in their interpretation of the probabilistic dependencies between consumer preferences. We prove both variants are NP-hard, and characterize their approximation hardness. Since in the practical application the overall number of items and the bound on the reduced item set are very large - in the order of magnitude of millions - we propose a highly parallelizable and scalable algorithm along with approximation guarantees. Finally, we present an end-to-end solution that fits the real-world e-commerce application, and provide an extensive set of experiments demonstrating the efficiency and effectiveness of our solution.

1 INTRODUCTION

Due to the rapid growth of the e-commerce industry, online selling has become one of the most trending businesses of today. Many e-commerce platforms serve as an intermediary between companies and consumers, receiving a commission per purchase. To increase the number of sales, such sites tend to offer a large number of items[1]. Nonetheless, they often pursue complementary objectives where selecting and offering a reduced inventory is required. Common such scenarios, reported by our industry collaborators, are the following:

Launching an express delivery store. When large companies provide express delivery services (alongside existing services) offering items for same-day-delivery, these items should be available in different warehouses for immediate shipping. It is not feasible to ensure immediate availability, in terms of storage space and maintenance overhead, except for a significantly reduced inventory, as seen for example in the Amazon Prime same-day-delivery catalog, which offers a small percentage of the entire inventory.

Opening a branch overseas. When e-commerce platforms start operations in new regions, it is often done gradually, initially offering a small backlog of items. This is in part because one needs to require the vendors to ship abroad, with regulations often restricting the number of products that are allowed to be distributed. A notable example is AliExpress (consumer facing branch of Alibaba), which due to regulations restricts the number of items offered for shipping abroad.

Reducing maintenance costs. Maintaining large inventories incurs substantial data maintenance overhead (e.g., in data cleaning, validation, entity resolution and semantic enhancement [5]). Hence, companies periodically dispose of some small percentage of items deemed to be least valuable.

These examples demonstrate the need to select a reduced inventory that minimizes the loss in the number of sales. A naïve, yet popular, solution is to focus on the top selling items. This approach however entirely ignores the *hidden* relations between items. In particular, studies show that consumers are flexible, and when searching for a specific item are often willing to buy in its absence what they consider to be a reasonable alternative [34]. For example, in the absence of a specific 19" LG TV a customer may be willing to settle for a slightly bigger LG TV or for the same size TV from Samsung. Retaining a set of items which are not only popular in-and-of-themselves, but are also likely to "cover" the inventory and serve as suitable alternatives for omitted items, can significantly improve the overall satisfaction of the customers.

To model consumer preferences and item alternatives we use a *preference graph* - a directed graph with weights on both the nodes and the edges. The nodes correspond to the items, and the node weights reflect the items' purchase popularity (% out of total sold items). A (directed) edge from item A to item B indicates that, in A's absence, consumers consider B as a possible alternative[2]. The edge weight reflects the probability that a consumer is willing to buy B as an alternative to A, if A is missing. (We discuss how edge weights are derived via customary techniques from click-stream data, commonly available to e-commerce companies, in the Experiments Section.)

We use the *preference graph* to devise effective algorithms for the selection of items. However, before presenting our results, let us first illustrate through a simple example how the information provided in the graph is employed.

Example 1.1. Consider the preference graph depicted in Figure 1. Assume that of the five available items we wish to choose two. We can see that A is the best selling item (purchased by 33% of customers) while D is the least sold (6%). We can also see that consumers interested in E are likely to settle in its absence for D, but will not transitively buy C. Such behavior is common, for instance, when D (resp. C) is a one-step upgrade of E (D): people are often flexible and willing to add a small amount of

[1]Note that "item" here refers to a specific item type from a specific seller, e.g., Silver iPhone Xs 256GB by Best Buy, rather than to individual instances of this item.

[2]We assume that all transitive relationships, when/if exists, are directly represented in the graph by single edges.

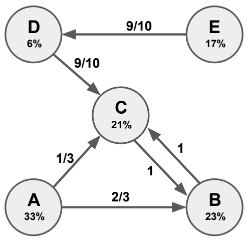


Figure 1: Sample graph of items

money in return for an upgrade, but a two-steps delta may be overly expensive. We can also see that consumers interested in C (B) will settle in its absence for B (C), and that B is a more likely replacement for A than C. Selecting the two best-sold items, A and B, is likely to satisfy about 77% of the customers (those interested in C, who in its absence are likely to purchase B, and those interested in A and B). Interestingly, a more careful analysis (which we only describe here intuitively and will formalize later in the paper) shows that in fact retaining B and D (the least sold item!) is the optimal solution, covering 87.3% of requests. Intuitively this is because B covers most requests for A, B, and C, whereas D covers itself and most of E. In this simple example the sets of requests served by the two retained items are disjoint, but a similar analysis can be applied to general overlapping cases.

The probability of a purchase is contingent on the dependencies between choosing different alternatives. Such dependencies can be extremely complicated, hence a practical model should simplify them in a manner which approximates well real life settings. We define in this paper two variants of the Preference Cover problem, which we have observed to be most prevalent in real-world e-commerce applications (see technical discussion in Section 5.2), the *Independent* and the *Normalized* variants. These variants differ by the semantics of edge dependencies. The Independent variant assumes independence between all alternatives. Whereas, the Normalized variant assumes that each consumer considers at most one item as an alternative she will actually buy, and thus the sum of weights of outgoing edges from any item is bounded by 1 (hence the name Normalized). In the experiments section, we show that both variants capture real-world consumer behavior. We will discuss the similarity and differences between these two problem variants (and when each is suitable) in depth in the following sections and propose effective algorithms to solve both.

To get an idea on what one may hope to achieve in terms of efficiency, we first study the computational complexity of the two problem variants. We prove that both are NP-hard but have different approximation bounds. Nevertheless, we devise a single greedy scheme that, with only minor adaptations, is able to support both variants, providing in both cases approximation guarantees. For the Independent variant we also prove this to be the optimal approximation ratio (matching the inapproximability bound we proved). For the Normalized variant, we show (by proving equivalence to the Max Vertex Cover problem) that while tighter theoretical upper bounds do exist, their corresponding algorithms are not scalable and thus impractical for our setting. Indeed, in the e-commerce application that we study here, both the overall number of available items and the number of items to be selected are very large - in the order of magnitude of millions. Thus, scalability is critical. Our solution consequently trades off the tightness of approximation guarantee in return for improved performance.

The simple greedy scheme which we use to solve both problem variants is highly parallelizable and scalable. It further has the added value of allowing to directly solve the complementary minimization problem, where, instead of an upper bound on the number of items to retain, one is given a lower bound on the percentage of item requests that should be covered, and the goal is to identify the smallest set of items that achieves the required coverage. Note that a naive solution for this complementary problem can be obtained via binary search on the target set size, by running any algorithm for the original problem. But this incurs a $O(\log n)$ factor overhead, n being the number of available items. Our direct greedy approach allows to avoid this overhead.

Our contributions can be summarized as follows.

(1) We formulate the Preference Cover problem and propose a graph-based model to capture it, with two natural possible variants.
(2) We study the theoretical computational complexity of the two problem variants and prove their NP-hardness and approximation hardness.
(3) We propose efficient algorithms to solve both variants, and prove their approximation guarantees.
(4) To demonstrate the practicality of our approach we describe the process of creating the preference graph from data available to actual e-commerce companies, based on well agreed upon inference methods in e-commerce research.
(5) We present an extensive experimental study, based on real-world data from a large e-commerce company, for both variants of the problem, demonstrating the effectiveness and efficiency of our algorithms.

Finally, we note that in the problem setting that we study here (which is common to many intermediary platforms [1]) the commission-per-purchase (or the perceived gain per purchase) is considered fixed and the intermediary platform is indifferent to the items' cost/revenue or required physical storage. Extending our work to support varying revenues and storage considerations, in a scalable manner, is an intriguing future work. We overview, in the Related Work, results in the field of operational research that incorporate such factors but in more complex models that do not lend to practical big data solutions. Other lines of work that bear resemblance to ours are recommendation systems and query results diversification. However, as we explain in the Related Work, the optimization problems they study differ from ours, yielding different complexity results and algorithms.

A first prototype of our system was implemented with help of our industry collaborators, and demonstrated at CIKM'19 [15]. The short paper accompanying the demonstration gives only a brief overview of the system architecture, whereas the present work provides a comprehensive description of the underlying model, algorithms and applications.

Paper outline. Section 2 provides the necessary definitions and formalism behind our problem. Sections 3 and 4 each study one of the two variants of our problem. Implementation and experimental studies are presented in Section 5. Finally, the related work appears in Section 6, and we conclude in Section 7.

2 PRELIMINARIES

We introduce here the *Preference Cover* problem. We start by formally describing the general model along with two concrete variants - the Independent and the Normalized variants.

Recall that the main motivation for our problem is an e-commerce setting, where consumers are interested in specific items but, if not available, may be willing to settle for some alternatives. Given a bound on the number of different item types a store may offer, our goal is to retain those which maximize the likelihood of purchase.

Formally, we represent consumer preferences via a *Preference Graph* which, along with an integer k, serve as input for the Preference Cover problem. A preference graph $G = (V, E, W_V, W_E)$ is a directed graph with weighted vertices and edges. The vertex set V corresponds to n items. For each vertex $v \in V$, its weight, $W_V(v) \in [0, 1]$, is defined as the probability of v being requested by a consumer. The sum of all node weights is therefore 1. We say that u is a *neighbor* of v if there exists an outgoing edge from v into u. The neighbors of a node v are all the items that are considered by consumers as possible alternatives. For each edge from v into a neighbor node u, its weight, $W_E(v, u) \in (0, 1]$, implies the probability of u matching a request for v as an alternative. We explain later how these edge weights are computed and interpreted. For simplicity we omit in the sequel subscripts of weighting functions when clear from context.

Given a number k, our goal is to choose a subset $S \subseteq V$ of k items, marking them as *retained*. Given a request for item v, if it is retained, the request is considered *matched*. Otherwise, if v is not retained, a request has some probability of being matched by another retained neighboring item of v, as indicated by the weights of edges outgoing from v into its retained neighbors. We define a target function $C : 2^V \rightarrow [0, 1]$, s.t. assuming S is the retained set of items, $C(S)$ is the probability a request drawn from the distribution indicated by the node weights is matched. The *Preference Cover problem* aims to compute $\arg\max_{S, |S|=k} C(S)$. We are ultimately only interested in whether or not a request is matched, and it makes no difference theoretically which item matches it. This corresponds to a real-life setting where intermediary platforms value the selling of each item as equally beneficial, and accordingly seek to maximize the number of sales.

We term $C(\cdot)$ as the *Cover function*, and say that the value $C(S)$ is the *cover of* S. Similarly, we call the probability a request for v is matched by a retained set S as the cover of v by S.

An explicit formula for computing $C(\cdot)$ is contingent on the dependencies, if such exist, between the probabilities indicated by the edges. In this paper we study two variants of the problem which, as our analysis of real data indicates, approximate well common real life scenarios: the *Independent* variant assumes that the probabilities modeled by edges are independent, while the *Normalized* variant assumes that each consumer considers at most one item as a most suitable alternative. In both cases, the goal is to retain the set of items which covers most of the predicted requests as implied by the preferences model.

In both variants when considering alternatives for a request for item v, we only take into account v's immediate neighbors. This is because, as mentioned earlier, the possible transitive processes of considering an alternative followed by an alternative to that alternative and so on is already taken into account when constructing the graph and assigning edge weights, which means, intuitively, that the preference graph is the transitive closure of a graph modeling the probabilities to correspond to such replacement paths.

We now formally describe the two variants we study in this paper. In the presentation below, given a retained set S we denote the retained neighbors of node v by $R_v(S) = \{u | (v, u) \in E, u \in S\}$

2.1 Independent variant

In the *Independent* variant we assume complete independence between the edges. Namely, the probability a given alternative matches a request is not affected by whether or not a different alternative matches it. Thus, the probability of the event of **not** matching a request for a non-retained item v, which occurs when no retained alternative is suitable, is, due to independence, the product of all such probabilities, $\prod_{u \in R_v(S)}(1 - W(v, u))$. The probability of the complement of this event, which is matching the request, is $1 - \prod_{u \in R_v(S)}(1 - W(v, u))$. We next formally define the Independent variant of the Preference Cover problem.

Definition 2.1 (IPC_k). Given a preference graph G and an integer k, the Independent Preference Cover problem (IPC_k) is computing $\arg\max_{S, |S|=k} C(S)$, where

$$C(S) = \sum_{v \in S} W(v) + \sum_{v \in V \setminus S} \left[W(v) \cdot (1 - \prod_{u \in R_v(S)} (1 - W(v, u))) \right]$$

The first addend is due to requests for retained items being matched with probability 1. The second addend corresponds to summing over all items not in S, for each such item v adding the probability it is both requested ($W(v)$) and covered by S ($(1 - \prod_{u \in R_v(S)}(1 - W(v, u)))$). Recall that we are computing the purchasing probability of a single consumer session, which in the hypothetical case where all items are available would have resulted in a purchase. The overall expected number of sales, given only S is retained, is thus the number of such sessions times $C(S)$. Cases where a consumer is looking to purchase several items, or several copies of the same items, are modeled as separate sessions.

2.2 Normalized Variant

In the *Normalized* variant we assume that each consumer considers at most one item as a suitable alternative (i.e. the item she will actually buy). Neighbors are therefore dependent, in the sense that for any requested item v, a retained neighbor matching the request implies that all other neighbors do not. It follows that the sum of the weights of all edges outgoing from any given node is at most 1, and given a request for a non-retained item v, the probability it is matched is $\sum_{u \in R_v(S)} W(v, u)$. We next formally define the Normalized variant of the Preference Cover problem.

Definition 2.2 (NPC_k). Given a preference graph G and an integer k, the Normalized Preference Cover problem (NPC_k) is computing $\arg\max_{S, |S|=k} C(S)$, where

$$C(S) = \sum_{v \in S} W(v) + \sum_{v \in V \setminus S} \left[W(v) \cdot \sum_{u \in R_v(S)} W(v, u) \right]$$

Here again the first addend, $\sum_{v \in S} W(v)$, is due to requests for retained items being matched with probability 1. The second addend corresponds to summing over all items not in S, and for each such item v adding the probability it is both requested ($W(v)$) and covered by S ($\sum_{u \in R_v(S)} W(v, u)$).

We discuss the choice of these particular variants and which real-life settings they correspond to in Section 5. Intuitively, the Independent variant asserts that the opinion on the suitability of a given alternative is not demonstrated to be strongly dependent for most consumers on their opinion of other alternatives. The dependencies are either insignificant overall or tend to cancel out when summed over the entire user base. In contrast, the Normalized variant is suited for domains where it is demonstrated that consumer requests are often very specific in nature, and the number of suitable alternatives is very small, which the Normalized variant models as a single alternative at most per request (though this can be a different alternative per each request for the same item). Finally, note that the weight of a node does not necessarily represent the probability of a premeditated and explicit request for the item, rather, more generally, the probability, given all items are available and a purchase is made, of this specific item being the one purchased.

2.3 Set functions

We next present some general definitions and results pertaining to set functions ($f : 2^V \to \mathbb{R}$, given universe V), which will be useful in the following sections for formally characterizing $C(\cdot)$.

Definition 2.3. f is **nonnegative** if $\forall S \subseteq V$: $f(S) \geq 0$.

Definition 2.4. f is **monotone** if $\forall S \subseteq V, \forall v \in V$: $f(S \cup \{v\}) \geq f(S)$.

Definition 2.5. f is **submodular** if $\forall S \subseteq T \subseteq V, \forall v \in V$: $f(S \cup \{v\}) - f(S) \geq f(T \cup \{v\}) - f(T)$.

The following is a key result in submodular optimization.

LEMMA 2.6 ([22]). *Given universe V, $k \leq |V|$, and a nonnegative, monotone submodular function $f : 2^V \to \mathbb{R}$, there exists a polynomial algorithm achieving an approximation ratio of at least $(1 - \frac{1}{e})$ in maximizing $f(\cdot)$ over subsets of size k. This algorithm, at each of its k iterations, selects an element maximizing the marginal gain to $f(\cdot)$.*

2.4 Related problems

Finally, we present definitions and results pertaining to related problems, which will also serve us in the formal analysis.

Definition 2.7. In the *Directed Max Dominating Set Problem (DS_k)* the input is a directed graph and a number k, and the goal is to find a subset of the vertices of size k such that the number of vertices adjacent to this subset is maximized.

Definition 2.8. In the *Max Vertex Cover Problem (VC_k)* the input is a number k and an undirected graph (self edges allowed) with positive weights assigned to its edges, and the goal is to select a subset of the vertices of size k such that the weight of edges incident to this subset is maximized.

We now provide hardness results pertaining to the above problems. The following theorem was proven in [21] for undirected graphs, but trivially extends to directed graphs, as undirected graphs are a special case where for each edge there exists a parallel edge in the opposite direction.

THEOREM 2.9. *[21, 25] The DS_k problem is NP-hard. It has no polynomial approximation algorithm of a factor higher than (1-1/e), unless $P = NP$. The problem is NP-hard even when the maximal degree in the graph is bounded by a constant.*

THEOREM 2.10. *[14, 27] The VC_k problem is NP-hard, and is hard to approximate to within $(1 - \delta)$ factor for some (small) $\delta > 0$, unless $P = NP$. Moreover, VC_k is NP-hard even for graphs of degree at most 3.*

Tighter approximation hardness bounds for VC_k are not known, and existing algorithms are discussed in Section 3.2. We also note that the NP-hardness of the bounded degree cases in both of the above theorems was proven by [14] and [25] for the minimization versions of the Vertex Cover (VC) and Dominating Set (DS) problems, resp. In the minimization versions the goal is to find the smallest set covering the entire graph. However, the hardness extends to our top-k versions, as VC (resp. DS) can be solved by solving VC_k (resp. DS_k) at most n times for varying values of k.

In each of the following two sections we study in detail one of the two variants of the Preference Cover problem. We analyze the Normalized variant first, as it is more complicated technically. Moreover, part of the discussion of the Normalized variant (in particular the proposed algorithm) applies, with minor adaptations, to the Independent variant as well.

Table 1: Approximation ratios of the greedy algorithm and best known polynomial algorithms for VC_k

Range of k/n	Greedy Algorithm	Best Known
$o(1)$	$(1 - 1/e)$	0.75+ϵ (SDP) [11]
$\Theta(1)$, [0, ≈0.39)	$(1 - 1/e)$	0.92 (SDP) [19]
(≈0.39, ≈0.72)	$(1 - (1 - \frac{k}{n})^2)$	0.92 (SDP) [19]
(≈0.72, 0.74)	$(1 - (1 - \frac{k}{n})^2)$	≈0.93 (SDP) [17]
[0.74, 1]	$(1 - (1 - \frac{k}{n})^2)$	$(1 - (1 - \frac{k}{n})^2)$ [11]

3 NORMALIZED VARIANT

3.1 Theoretical Analysis

We begin this section by studying the complexity and the approximation hardness of NPC_k, the Normalized variant of the Preference Cover problem. In both these respects we prove an equivalence to the VC_k problem, which has been studied extensively in the literature, implying in particular that both upper and lower bounds on the approximation of VC_k apply for NPC_k as well. We then discuss algorithms for solving NPC_k. Here again we utilize the equivalence to VC_k to adapt its known algorithms to our setting. Concretely, we present the currently best known approaches in terms of the approximation ratio guarantee, which vary for different ranges of k, and discuss their performance w.r.t. scalability. We note the trade-off between performance and approximation guarantees, and as our goal is to provide a scalable solution for big data settings, we focus on a fast algorithm, for which we provide an efficient parallelizable implementation, and demonstrate it to be highly scalable in our experiments. Moreover, its approximation factor is the best known for high values of k, and is not far off for lower ranges. We further argue that all algorithms known to provide a better approximation guarantee for lower ranges are not scalable at all. We discuss additional advantages of our approach in Section 3.2.

THEOREM 3.1. *The NPC_k problem is hard to approximate to within a $(1 - \delta)$ factor for some (small) $\delta > 0$, unless $P = NP$. The problem is NP-hard, even when the maximal degree (disregarding edge orientation) is 3. Furthermore, any α-approximation algorithm for VC_k implies an α-approximation algorithm for NPC_k, and vice versa.*

PROOF. We first reduce NPC_k to VC_k (Definition 2.8), and show that any α-approximation algorithm for VC_k implies an algorithm for NPC_k with the same factor. Given an instance $I = \langle G, k \rangle$ of NPC_k, we add a self edge to every node whose sum of outgoing edge weights is less than 1, and assign to it the weight which completes the total outgoing weight to 1. This added weight intuitively represents the relative part of requests for this item which cannot be covered by any alternative. Observe that this change has no bearing on the cover function $C(\cdot)$, as when a node is retained, its weight is covered entirely all the same. Next we reduce this instance I to an instance $I' = \langle G', k \rangle$ of VC_k, such that G' has the same nodes as G only without weights, the same edges only without orientation, and the weight of every edge (v, u) changes from $W(v, u)$ to $W(v) \cdot W(v, u)$ (multiplied by the weight of its origin node). Note that G' may have pairs of nodes connected by 2 parallel edges, if edges in both directions connected these nodes in G. This is equivalent, w.r.t. VC_k, to replacing both edges with a single edge whose weight is the combined weight of the original two. However, we do not combine parallel edges, as we analyze their weight contributions separately.

We now argue that for any choice of a node set $S \subseteq V$, its cover weight in G' is equal to the cover $C(S)$ in G. Indeed, let E_S denote all edges that are **outgoing** from a node in S in G, then the sum of weights of the edges in G' originating from E_S is exactly $\sum_{v \in S} W(v)$, which corresponds to the first addend in the formula in Definition 2.2 (each node in S contributes its weight to $C(S)$); considering all remaining edges adjacent to S in G' which are not in E_S: the sum of weights of all such edges originating in any given node $v \notin S$ is $W(v) \cdot \sum_{u \in R_v(S)} W(v, u)$ (recall that $R_v(S) = \{u|(v, u) \in E, u \in S\}$), which, when summing over all such $v \notin S$, is exactly the remaining addend in the formula for $C(S)$, thus proving the equivalence.

As for the other direction, which is proving that NPC_k is just as hard to approximate as VC_k, from which the NP-hardness and hardness of approximation follow, assume we are given an instance $I' = \langle G', k \rangle$ of VC_k. We reduce it to an instance $I = \langle G, k \rangle$ of NPC_k such that all nodes in G are the same as in G', and the edges are also the same with the orientation chosen arbitrarily. Now for any node v let M_v denote the sum of weights of all its outgoing edges at this point, then we set $W(v) = M_v$, and for every outgoing edge e from v, we change its weight from its original weight in G', denoted by $W'(e)$ to $W(e) = \frac{W'(e)}{M_v}$. It follows that the sum of weights of all outgoing edges from any given node, which has at least one such edge, is 1. We set the weight of any node without any outgoing edges adjacent to it to 0.

Following this reduction the sum of weights of all nodes, denoted by N, is not necessarily 1. This requirement over the sum of node weights is due to the semantics of the problem, and is computationally insignificant. Indeed, we can normalize, and divide all node weights by N, and denote the resulting graph $\hat{G}$. It follows that the cover of any solution S, including the optimal solution, changes after this normalization from $C(S)$ in G to $\frac{C(S)}{N}$ in $\hat{G}$, and thus the approximation ratio is not changed. Given an α-approximation algorithm for NPC_k, we run it over the normalized graph $\hat{G}$, and it follows that this solution has the same ratio for the non-normalized instance I of NPC_k. Finally, observe that if we reduce I to an instance of VC_k, as we described in the first direction of the proof, we once again get I' (multiplying edge weights by the original node weight cancels out their normalization by the same factor), implying, by the same logic as before, that for any set S its cover weight in G' equals $C(S)$ in G. Therefore, the same node set S guarantees an α-approximation of the original VC_k instance. Note that as the reduction preserves the maximal degree, it follows that NPC_k is NP-hard for maximal degree 3. □

Due to the equivalence to VC_k, tight inapproximability bounds for NPC_k are also not known. We discuss below existing approximation algorithms, as VC_k algorithms can be adapted to NPC_k maintaining the same approximation factors. For a recent review of VC_k results see [19].

3.2 Algorithms

In this section we discuss algorithms for NPC_k, and focus on the implementation of a greedy algorithm, which we argue to be by far the most scalable option, on top of having high approximation guarantees.

The equivalence of NPC_k to VC_k combined with the linear approximation-preserving reductions described in the proof of Theorem 3.1, imply that when looking for an algorithm for NPC_k, one should examine the algorithms known for VC_k, an extensively studied problem. The algorithms providing the best known approximation guarantees vary for different ranges of k. Detailed results are presented in Table 1 (the second column pertains to a greedy algorithm on which we focus in the sequel). Nevertheless, for $k < 0.74n$, all top algorithms are based on the same core technique of semidefinite programming (SDP). SDP is a much more general extension of linear programming (LP). An algorithm based on LP also exists [2], with an approximation factor of 0.75. These SDP (and LP) based algorithms are important mainly for the theoretical analysis of the problem, in particular finding out the best approximation ratios. On the other hand, these solutions are not scalable, especially for big data settings, as they are known for having an impractical running time, even for medium sized programs [7, 37] (for example, the number of constraints in the program, as devised in [11], is of order $O(n^3)$).

Algorithm 1: Greedy Algorithm

```
Input: G, k
1 $S = \emptyset$
2 foreach $1 \le i \le k$ do
3     foreach $v \in V \setminus S$ do
4         $C(S \cup v) = \text{Gain}(S, v)$
5     $\hat{v} = \arg\max_v C(S \cup v)$
6     $S, C(S) \leftarrow \text{AddNode}(S, \hat{v}, C(S))$
7 return $S, C(S)$
```

Another approach is a greedy algorithm, introduced in [16], and analyzed by [11] to have an approximation factor of $max\{(1 - 1/e), (1-(1-\frac{k}{n})^2)\}$. This factor positively correlates with k, and for $k \ge 0.74n$ it is the best known guarantee, exceeding a 0.93 factor. To the best of our knowledge, any algorithm which surpasses the approximation guarantee of the greedy algorithm, for any range of k, is based on SDP or LP. The greedy algorithm, unlike these alternatives, lends itself to an efficient implementation. As our goal is to provide a practical solution, we focus on the greedy algorithm, which we implement and evaluate. The implementation we provide is adapted directly into our setting without a reduction to VC_k. It is parallelizable, and as we prove in the experiments (Section 5), highly scalable even for graphs containing millions of nodes. Additional advantages of this approach are discussed towards the end of this section. As mentioned, Table 1 depicts, for various ranges of k, the best known approximation factor, alongside the factor of the greedy algorithm.

Greedy Algorithm The greedy algorithm (Algorithm 1) incidentally applies schematically for both the Normalized and the Independent variants (with latent distinctions, described in the next section, devoted to IPC_k, the Independent variant). We use an array I of size n, with an entry $I[v]$ for each $v \in V$, eventually set to the probability of v being both requested and matched by the produced solution S (the product of $W(v)$ and the cover of v by S). The summation of all entries in I, as indicated in Definition 2.2, equals $C(S)$. For simplicity, we assume the preference graph G and the array I are *global variables*, with I initialized to zeros.

Algorithm 1 maintains an initially empty solution set S (line 1). At each of its k iterations (line 2), it goes over all nodes currently not in S (line 3), and for each such node it computes the gain to $C(S)$ obtained by adding it to S (line 5). The node that maximized this gain is then chosen (line 6) and is added to S, with $C(S)$ updated accordingly (line 7). Finally, after completing k iterations, S and $C(S)$ are returned (line 8).

Algorithm 2: Gain - Normalized

```
Input: S, v
Global: G, I
1 $g = W(v) - I[v]$
2 foreach $u \in V \setminus S$ s.t. $W(u, v) \in E$ do
3     $g$ += $W(u) \cdot W(u, v)$
4 return $g$
```

Algorithm 3: AddNode - Normalized

```
Input: S, v, C(S)
Global: G, I
1 $S \leftarrow S \cup \{v\}$
2 $C(S)$ += $W(v) - I[v]$
3 $I[v] = W(v)$
4 foreach $u \in V \setminus S$ s.t. $W(u, v) \in E$ do
5     $C(S)$ += $W(u) \cdot W(u, v)$
6     $I[u]$ += $W(u) \cdot W(u, v)$
7 return $S, C(S)$
```

The procedures *Gain* (Algorithm 2) and *AddNode* (Algorithm 3) are conceptually similar as both compute the marginal effects of adding a given node, with the main distinction being that *AddNode* also updates accordingly I and $C(S)$.

In Algorithm 3, line 1 adds to S the node v which was chosen in Algorithm 1 as maximizing the marginal gain. Line 2 adds to $C(S)$ the gain in the cover of itself and line 3 updates $I[v]$ to $W(v)$ as the newly added node covers itself completely. Next we iterate over all nodes outside of S with an edge into v, and for each such node u, we compute the marginal gain to its cover by v, and add it to $I[u]$ and $C(S)$ (lines 5 and 6, resp.). We can see that after each call to Algorithm 3 the array I is updated with each entry set to the contribution of covering the corresponding node by S to $C(S)$. As we mentioned, Algorithm 2 is the same as Algorithm 3, only focusing solely on the marginal gain, without updating I and $C(S)$.

Although we adapted the greedy algorithm directly to preference graphs, without reducing to VC_k, it is easy to show along the same lines as the proof of Theorem 3.1, that a reduction to VC_k would have resulted in choosing the same nodes. Hence, the approximation factor of $max\{(1 - 1/e), (1 - (1 - \frac{k}{n})^2)\}$, proven for VC_k in [11], holds here as well.

To illustrate the operation of Algorithm 1, consider the following example.

Example 3.2. Recall the preference graph depicted in Figure 1 and assume $k = 2$. The algorithm first computes the gain obtained by selecting each node and retains the most beneficial, which is B (66%, covering $W(B)$, $W(C)$ and 2/3 of $W(A)$). After B is retained, the algorithm proceeds to the second and final iteration, to choose the next node with the highest marginal gain, which is D. D itself is requested only by 6% of consumers, while A and C are 22% and 33%, resp. However, B being selected in the previous step reduces A's and C's marginal gain to 11% (the 1/3 of $W(A)$ corresponding to consumers not accepting B as an alternative to A) and 0% (all consumers who wanted C are happy to get B instead), resp. On other hand, D covers 6% (itself) and 15.3% (9/10 of $W(E)$ - consumers that wanted E, but also agree to have D as alternative), which gives a total of 21.3%. Finally, the retained items, B and D, cover 87.3% of consumer preferences (which in this case is also the optimal possible pair).

Performance Analysis We now analyze the complexity of the greedy algorithm. Let $d(v)$ denote the **incoming** degree of a node v, and let D denote the maximum incoming degree over all nodes. Observe that in both Algorithms 2 and 3 the number of operations performed is $\Theta(d(v)) = O(D)$ (v is the node whose marginal gain is evaluated). For each of the k iterations, Algorithm 2 is called $O(n)$ times, hence the overall time complexity is $O(nkD)$.

Furthermore, the algorithm is highly parallelizable. When Algorithm 1 iterates in line 3 over $O(n)$ nodes to compute their marginal gain, computations for each node are independent, and can be performed in parallel. Moreover, in each such call to Algorithm 2 (or 3) the iteration in line 2 over the $O(D)$ nodes adjacent to the added node can also be done in parallel. Concretely, for $N < nD$ threads, the complexity of each of the k iterations becomes $O(\frac{nD}{N})$ resulting in an overall $O(k + \frac{nkD}{N})$. The space complexity of the algorithm (excluding the input, which we treat as read-only), is $O(|V|)$ for storing I. We can also reduce the space complexity to $O(k)$ by doing away with I, at the expense of computing the value $I[v]$ from scratch in line 1 of Algorithm 2 (line 2 in Algorithm 3), which takes $O(D)$ operations, and does not affect the overall complexity.

Additional Advantages Finally, we identify several additional benefits of our greedy algorithm. First, we can return I as part of the output. This enables to efficiently compute, for each non retained item u, its cover by S, which equals $\frac{I[u]}{W(u)}$. This provides important information about which item requests are affected by reducing the inventory and to what extent. Moreover, the incremental nature of the greedy approach, when the retained set is produced in the order in which the nodes were added to it, can provide solutions to related instances and problems. Namely, an ordered solution S of size k, also produces the solution for any $k' < k$, which is the first k' nodes in S (the same solution that running the algorithm with k' would have produced). Therefore, solving for $k = n$ provides at once the solutions for all k values, and, moreover, directly provides (an approximated) solution for the related problem where the goal is to retain the smallest set, such that the cover exceeds a given threshold.

4 INDEPENDENT VARIANT

4.1 Theoretical Analysis

We now study the theoretical properties of IPC_k, the Independent variant, and our proposed algorithm. We first prove our main result, stating that IPC_k is NP-hard (even given a constant bound on node degrees), and has a tight approximation factor of $(1 - 1/e)$ by a polynomial time algorithm. In fact, this factor is achieved by the same greedy approach as the one discussed in Section 3 for NPC_k, with small adjustments. The adaptations that need to be performed to previously presented algorithms and analysis of the adapted algorithms are then discussed in detail in Section 4.2.

THEOREM 4.1. *The IPC_k problem has a tight approximation bound of $(1 - 1/e)$ in polynomial time, unless P=NP. Moreover, it is NP-hard, even given a constant bound on the maximal node degree (disregarding edge orientation).*

PROOF. To prove hardness of approximation (and thereby the NP-hardness), we assume an α-approximation algorithm for IPC_k, and reduce an instance $I' = \langle G', k\rangle$ of DS_k to an instance $I = \langle G, k\rangle$ of IPC_k, such that an α-approximation solution is implied

Algorithm 4: Gain - Independent

```
Input: S, v
Global: G, I
1 $g = W(v) - I[v]$
2 foreach $u \in V \setminus S$ s.t. $W(u,v) \in E$ do
3     $g \mathrel{+}= W(u,v) \cdot (W(u) - I[u])$
4 return $g$
```

for DS_k. The hardness results then follow from Theorem 2.9. The reduction preserves the maximal degree, implying NP-hardness even given a constant bound on it. Concretely, G has the same nodes and edges as G', except all edge orientations are **reversed**. All edges are assigned the weight 1, and all nodes are assigned the weight $\frac{1}{n}$.

We argue that for any solution $S \subseteq V$, the number of vertices it dominates[3] in G' is $n \cdot C(S)$ ($C(S)$ is the cover of S in G), proving the same approximation ratio. To see this, observe that all $|S|$ nodes in S dominate themselves in G', which corresponds to the first addend in the formula in Definition 2.1 (the sum of weights of nodes in S) which equals $\frac{|S|}{n}$. The remaining nodes dominated by S in G' form the set of all nodes outside of S that have an incoming edge from S, denoted by T. This corresponds to the remaining addend in the formula for $C(S)$, which (as the edges in G are reversed w.r.t. G') is the cover of the set of nodes outside of S with an outgoing edge into S in G, which equals $\sum_{v \in T} \frac{1}{n} = \frac{|T|}{n}$. Overall, we see that there is a fixed ratio of $\frac{1}{n}$ between the values of the target functions of the two problems for any S, proving the equivalence of the approximation. This proves a $(1-1/e)$ hardness bound on the approximation of IPC_k.

To show this bound is tight, we prove that all conditions specified in Lemma 2.6 (the function is nonnegative, monotone and submodular) hold for $C(\cdot)$, implying that a greedy incremental algorithm maximizing the marginal gain at each of its k iterations guarantees a $(1-1/e)$ approximation. Indeed, these properties are evident from the formula in Definition 2.1. $C(\cdot)$ is by definition nonnegative. The addition of any node to the solution maximally covers itself, and can only increase the cover of any other node (it decreases the product $\prod_{u \in R_v(S)}(1 - W(v,u))$ in the formula for $C(S)$, increasing the overall value), which proves monotonicity. Finally, $C(\cdot)$ is submodular: given two sets $S \subset T \subseteq V$ and a node u', we show that $f(S \cup \{u'\}) - f(S) \geq f(T \cup \{u'\}) - f(T)$. If u' belongs to any of these two sets then this follows trivially. Otherwise, as T covered u' at least as much as S (due to monotonicity), the complete covering of u' after its addition is at most the same for T compared to S. As for any other node v (which has an edge into u', as otherwise adding u' can have no effect on it): if $v \in T$ and $v \notin S$, then T already covered it completely, and the addition of u' to T adds nothing. Otherwise ($v \notin T$), for S and T, resp., the effect is that both $\prod_{u \in R_v(S)}(1 - W(v,u))$ and $\prod_{u \in R_v(T)}(1 - W(v,u))$ are multiplied by the $(1 - W(v,u'))$. As the second product (with T) is not bigger than the first (with S), then the additive difference after multiplying it by $(1 - W(v,u'))$ is also not bigger, leading to an overall smaller (or equal) increase in the cover (when added to T). As the increase in the cover of every node is not bigger when adding to T, submodularity follows.

□

[3]A vertex is dominated by a solution S, if it is either in S or has an edge incoming from some node in S.

Algorithm 5: AddNode - Independent

```
Input: S, v, C(S)
Global: G, I
1 $S \leftarrow S \cup \{v\}$
2 $C(S) \mathrel{+}= W(v) - I[v]$
3 $I[v] = W(v)$
4 foreach $u \in V \setminus S$ s.t. $W(u,v) \in E$ do
5     $C(S) \mathrel{+}= W(u,v) \cdot (W(u) - I[u])$
6     $I[u] \mathrel{+}= W(u,v) \cdot (W(u) - I[u])$
7 return $S, C(S)$
```

4.2 Greedy Algorithm

We now discuss our proposed algorithm for IPC_k. Here again we opt for the same greedy approach as presented in Section 3.2 for NPC_k, which for IPC_k guarantees an optimal $(1 - 1/e)$ approximation factor, following Theorems 4.1 and 2.9. Moreover, the algorithm scheme is also depicted in Algorithm 1, with the distinction that the procedures *Gain* and *AddNode*, called, resp., in lines 4 and 6, have different implementations (resp., Algorithm 4 instead of Algorithm 2 and Algorithm 5 instead of Algorithm 3). These procedures, however, remain analogous to their counterparts for NPC_k (Algorithms 2 and 3), as the adjustments only reflect the technical difference between the formulas for $C(\cdot)$ in Definitions 2.1 and 2.2. Therefore, we do not discuss the algorithm in detail, as this is largely covered in Section 3.2, but rather focus on the distinctions from the NPC_k implementation.

Recall that I is an array, initialized with zeros, with an entry for each $v \in V$, denoted by $I[v]$, eventually set to the probability of v being both requested and matched by the produced solution S, and hence the summation of all entries in I equals $C(S)$. We once again set it to be a global variable whose value is saved between calls. In Algorithm 4 line 1 remains the same as in the analogous Algorithm 2, as in both variants it holds that a retained node is completely covered by itself. Line 3, however, is different. It pertains to the marginal gain obtained by adding the node v in the cover of node u, which has v as a neighbor. Let S and S' denote the solution set before and after resp., adding v, and let $I_S[u]$ and $I_{S'}[u]$ denote the correct value of $I[u]$ for solutions S and S', resp. The marginal gain by v over u is $I_{S'}[u] - I_S[u]$, which after doing the algebra is simplified into $W(u,v) \cdot (W(u) - I[u])$. This is the gain appearing in Line 3 of Algorithm 4. The correctness of this expression follows a straightforward computation, which we omit to avoid convoluted notation, and instead provide the intuition. Note that the computation of the probability u **not** being covered by S is the product of the probabilities of all its retained neighbours not being suitable alternatives. This probability can be easily computed from $I_S[u]$ (see the second sum in the formula in 2.1). The only change in S' is that this product is now multiplied by the probability v is also not a suitable alternative, which is $(1 - W(u,v))$. Therefore, when computing the product pertaining to S' (which implies $I_{S'}[u]$), we can reuse the computed product for S (implied by $I_S[u]$), and reduce the number of operations in computing the marginal gain to $O(1)$ per each such u. Moreover, the similarity of the computations for $I_S[u]$ and $I_{S'}[u]$ allows the simplification of the expression for the marginal gain. The adjustments made in Algorithm 5 are completely analogous.

Performance Analysis Following exactly the same considerations as in the NPC_k implementation (Section 3.2), both Algorithms 4 and 5 have the same time complexity, hence the overall

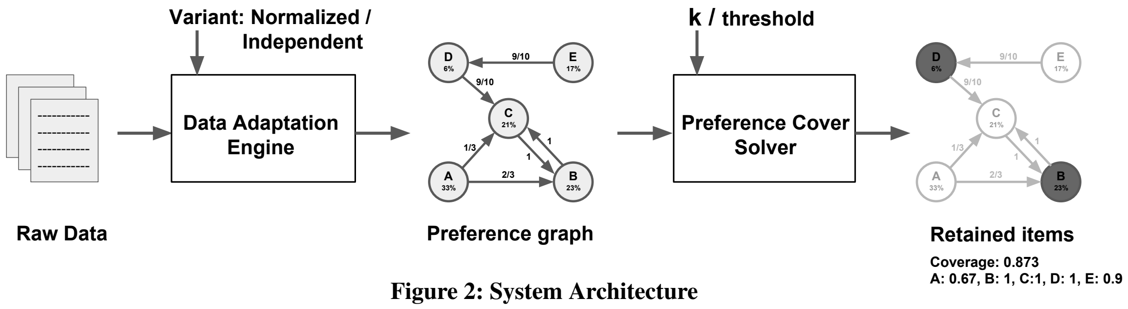


Figure 2: System Architecture

complexity of the greedy algorithm is $O(nkD)$ here as well (recall that D denotes the maximal incoming degree of a node in G). The potential parallelization is also the same, thus the complexity for $N < nD$ threads becomes $O(k + \frac{nkD}{N})$, same as in Section 3.2. As for the space complexity (excluding the input), while it is also $O(n)$, due to storing I, in the IPC_k case it is no longer true that we can do away with I to reduce the space to $O(k)$, without sacrificing efficiency (here the computation of the marginal gain is more reliant on previous computations). Finally, the additional benefits pertaining to the incremental nature of the greedy approach remain the same as for NPC_k, as described at the end of Section 3.2.

5 IMPLEMENTATION AND EXPERIMENTS

We start this section by describing the system architecture, focusing on the flow from raw data to a suggested list of k retained items. Afterwards, we discuss how real-life e-commerce data can be adapted to construct the preference graph. Then, we describe the experimental setup, where we introduce the real-life datasets we used for the experiments, and describe concretely which adaptations we made to these datasets to fit our models. Finally, we present the evaluation results.

5.1 System Architecture

The system architecture, depicted in Figure 2, demonstrates the end-to-end flow. The system consists of two main modules: the *Data Adaptation Engine* and the *Preference Cover Solver*. The *Data Adaptation Engine* takes as input the raw e-commerce data and the variant (Normalized or Independent), and builds the corresponding preference graph. Detailed discussion about what type of raw data is necessary, which of the two variants to choose in a given situation, and how the adaptation is actually performed is presented in the following section (Section 5.2). The constructed preference graph is then passed as input to the Preference Cover Solver, along with k, the desired number of retained items. The solver runs Algorithm 1, adapted to the specific variant (calling Algorithms 2 and 3 for the Normalized variant, and Algorithms 4 and 5, resp., for the Independent variant). The solver produces a list S of retained items (in the order in which the items were added by the algorithm), accompanied with metadata, such as $C(S)$ (the cover achieved by S), and the coverage percentage of every item (implied by the array I, used in our algorithms), i.e. how well the item is covered by the retained alternatives in S (the coverage of retained items is obviously 100%). For example, the rightmost part of Figure 2 highlights the produced set of retained items, B and D (for the input of $k = 2$ and the preference graph depicted in the center of Figure 2, originally introduced in Figure 1 as part of Example 1.1). The coverage of the non-retained item C is also 100%, since it is completely covered by B. The coverage of items A and E is 67% and 90% since they are covered by B and D, resp.

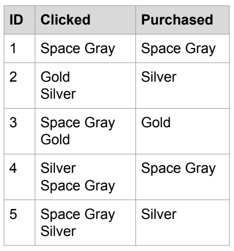
Note that, as explained at the end of in Section 3.2, the same architecture can be used to also solve the complementary problem where the goal is to retain the smallest set of items, such that the cover exceeds a given threshold.

5.2 Adaptation of Raw Data

As mentioned in previous sections, adapting e-commerce data into our graph model is an essential phase, which is also incorporated in our architecture. E-commerce platforms collect tracking data from browsing sessions to learn consumer patterns and preferences to improve the shopping experience and increase revenue. The data is often stored in the format of a clickstream, consisting of the history of events performed by consumers during browsing, grouped by sessions. The information included in the clickstream usually contains the session id, date and time, search query, search page results, clicks, add-to-cart events, purchases and consumer related information, such as username, IP address, geolocation, etc. To ensure wide applicability, we assume that the available clickstreams include only minimal basic information: clicks and purchases grouped by sessions (which is true for most existing platforms and datasets [3])[4].

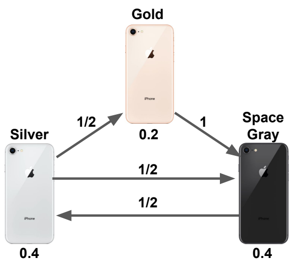
Graph construction process We now discuss the process of constructing a preference graph from the raw data. Recall that our model captures a session by identifying the “desired” item, which, if available, is the item the consumer would buy, and otherwise outgoing edges indicate her willingness to purchase concrete alternatives. Therefore, ideally, we would like to have for each session information identifying the desired item (for example, a search query specifying it explicitly), as well as a sufficient number of sessions where the specified item is not available, so as to accurately capture the suitability of alternatives, implied by the items purchased instead. However, in real life, when considering the main store which offers the product catalog in its entirety (which is the source for inferring the what-if probabilities necessary for curating the store with the limited inventory), it is overwhelmingly the case that all relevant items in a user session are in-stock. While this allows to identify the desired item as the one purchased[5], and derive an accurate estimation of each item’s relative popularity, it is, nevertheless, harder to approximate user preferences pertaining to alternatives. In light of this limitation, we can use clicks on each item to estimate its suitability as an alternative to the purchased item. We note that assuming strong positive correlation between clicks and an intention to buy is a common practice employed by analysts in many e-commerce companies, when modeling consumer preferences[6], and it is also suggested in relevant studies [26, 32]. When viewing each click as an intention to buy (as an alternative), it is possible to overestimate this actual willingness to make a purchase, likely resulting in a diminished probability assigned to the event where no alternative is suitable. This can be

[4]One may also use semantic similarity between items to approximate edge weights, however we do not pursue this direction here.
[5]Sessions with no purchase, as all items are assumed to be available, are not driven by an intention to buy, and hence do not affect our modeling.
[6]Based on private conversations with analysts in multiple companies.



ID	Clicked	Purchased
1	Space Gray	Space Gray
2	Gold Silver	Silver
3	Space Gray Gold	Gold
4	Silver Space Gray	Space Gray
5	Space Gray Silver	Silver

a) Sample of sessions



b) Corresponding preference graph

Figure 3: Preference graph construction process

addressed by a more refined learning process where more variables are taken into account, subsequently normalizing the edge weights by a corrective factor. For example, by considering the amount of time spent viewing each item [32]. However, discussing such methods in detail is beyond the scope of this paper.

In view of the discussion above, we construct the preference graph, given clickstream data containing clicks and purchases per session, in the following way. The nodes in the graph correspond to the items. The node weights are assigned the percentage of purchases of an item out of total number of purchases. An edge from A to B exists only if the data contains a session where A was purchased, and B was clicked. The weight assigned to this edge is the fraction out of all sessions where A was purchased, in which B was also clicked.

Note that, it may seem at first logical to learn the edges in the opposite direction, i.e. to assign the weight of and edge from A to B based on sessions where A was clicked and later B was purchased, such that the edge direction “matches" the order of the operations. However, this does not fit the semantics of our model. Namely, we assume a “requested item" is bought with probability 1 when in stock, and an edge from A to B refers to sessions where A is a requested item which is out of stock, and B is an alternative. Since in most cases (and in the available data in particular) the items are in stock (examining only data where a desired item is out of stock will reduce the size of the relevant sessions to several thousands), clicking on A when available and then choosing another item implies that A is not the requested item. The other direction, which we opt for, is arguably more logical, given sessions where all relevant items are in stock, as the purchased item is almost certainly the most preferred item, and clicks on other items serve as considering these items as alternatives. Moreover, when estimating the weights of edges between A and B, we purposely avoid taking into account sessions where both A and B were clicked but neither was purchased. This is because the edges do not represent browsing probabilities (i.e. the probability B is clicked on next, when currently A is view, or vice versa), rather purchasing probabilities. Thus, our graph can intuitively be viewed as a transitive closure of a graph modeling “browsing" probabilities, with the purchased item viewed last (see discussion in Section 2). Observe that for items rarely clicked, the low number of corresponding sessions allows for more noise and the derived correlations to alternatives are less reliable. However, rarely clicked items have also (by definition) low weights, and hence these "noisier" items correspondingly have negligible influence over solution, as it focuses on more popular items.

How to choose the variant Finally, we explain when each of our two variants is a suitable choice given the data. We note that we focus in this work on these two models for edge dependencies, as in our inspection of clickstreams, we observed that in almost all cases one of the variants fits the data (in an approximated manner described next). Nevertheless, it is of course theoretically possible for other dependencies to exist, fitting neither of our models. We leave the adaptation of our techniques to such cases for future work, as here we focus on dependency schemes we have observed to be particularly prevalent.

The Normalized variant models the data well when at most one item, apart from the one purchased, was clicked. In practice it is unlikely any data perfectly adheres to this rule. However, when exceptions are rare we consider this to be a good approximation. In our experiments we applied the Normalized variant when in at least 90% of the sessions at most one alternative was clicked. In such datasets, when processing sessions where $t > 1$ other items were clicked, we “normalized” by counting each as a $\frac{1}{t}$-fraction of a click.

As for the Independent variant, it fits well when for any 2 alternatives (w.r.t. a given desired item) the fraction of sessions in which one was clicked remains the same when conditioned on clicking on the other (counting the fraction only out of sessions where the other was clicked). Once again, expecting any data to demonstrate such complete independence is not realistic. We, therefore, consider the Independent variant to be a reasonable approximation, when the following condition holds. We use a common measure for dependence of random variables, known as *Normalized mutual information* [31], which produces a value in $[0, 1]$, such that 1 indicates total dependence and 0 total independence. For any given item, we compute this measure for all pairs of alternatives, and take the average. Finally, we take the weighted average of these averages over all desired items, corresponding to the node weights (such that the average is not skewed by rarely purchased items)[7]. If this measure is below 0.1, analogously to the 90% cutoff in the Normalized variant, then we consider the Independent variant to be a a fitting choice of model.

To illustrate the process described above, consider Figure 3a, depicting a tiny sample of sessions taken from a real-life clickstream of users that purchased an iPhone 8 256GB. This smartphone comes in 3 different colors: Silver, Gold and Space Gray. The clickstream consists of these 3 items and 5 sessions, each ending in a purchase. The corresponding preference graph is depicted in Figure 3b. There are 2 purchases of the Space Gray iPhone, 2 of Silver and 1 of Gold. Hence, the node weights are 0.4, 0.4 and 0.2, resp. Out of the 2 times the Silver iPhone was purchased, each of the other 2 phones was clicked exactly once. Hence, the edge weights from Silver to Gold and Space Gray are $1/2$. Whereas, from the 2 sessions where Space Gray phone was purchased, one had no other clicks, and the other had 1 click on Silver. Hence, there is an edge from Space Gray to Silver with weight $1/2$. Finally, the Gold iPhone was purchased once, and in that session the Space Gray phone was clicked as well. Hence, there is a single edge of weight 1 from Gold to Space Gray. As for the problem variant, it is clear that the Normalized variant is a good fit, since no session implies more than one alternative.

Note that given a more detailed clickstream, an e-commerce platform can construct a more precise graph. For example, one can analyze the clickstream and combine with the information about out-of-stock items to find which items were purchased instead. Another idea for improvement is using the search query text and filter out items from the clickstream that are not matching the user’s intent. However, as mentioned before, such information is

[7]Of course, other thresholds and statistical distances can be applied just as well.

Table 2: The datasets used in the experiments

DS	Sessions	Purchases	Items	Edges
PE	10,782,918	10,782,918	1,921,701	9,250,131
PF	8,630,541	8,630,541	1,681,625	7,182,318
PM	8,154,160	8,154,160	1,396,674	5,826,429
YC	9,249,729	259,579	52,739	249,008

not always available in sufficient volume, hence it can be used as an enhancement on top of the proposed solution to adjust the weights. In general, more sophisticated data can be collected, with more resources invested in its analysis, resulting in a refined module for constructing the graph, which comes in place of our Data Adaptation Engine, with the rest of the architecture remaining the same. The methods we focus on here are chosen to fit actual information currently available to most e-commerce platforms.

5.3 Experimental Setup

We implemented our system using Python, and ran the experiments on a server with 128GB RAM and 32 cores. To evaluate our solution, we have performed a set of extensive experiments, both on a publicly available real-world dataset and on a bigger (private) dataset provided by a large e-commerce company[8]. We compare our approach to 4 baselines, using 4 evaluation methods.

Datasets. The first dataset, provided by a large e-commerce company, contains 5M items and 27M sessions, all ending with a single item purchase (we specifically requested such sessions). The dataset is private and comes as three independent parts, divided by domains - Electronics (PE), Fashion (PF) and Motors (PM). Due to its size, this dataset is particularly useful for scalability tests. The second dataset, marked as YC, is a public dataset, which was provided by the company YooChoose for the RecSys 2015 Challenge [3]. This dataset contains a clickstream with approximately 260K sessions ending in a single item purchase, covering a 6 month period in 2014 (from April 1st to September 30th). We have included this publicly available data, to allow the reader to reproduce the results.

The summary of these datasets is presented in Table 2. It includes the number of sessions, purchases, items and edges. Recall our discussion at the end of Section 5.2 regarding the conditions we set for each variant to fit a given dataset. It follows that the YC, PE and PF datasets fit the Independent variant, as in all three datasets our proposed independence measure is below 0.1. The PM dataset (whose items are parts and accessories for automobiles), however, is better captured by the Normalized variant, as in its sessions few alternatives were considered prior to purchasing. In particular, the percentage of sessions implying no more than a single alternative is above 90%.

Algorithms. We compare 5 different algorithms, over the same inputs (each input consists of a preference graph and a size bound k). The experiments are performed separately for both variants of the problem, and hence the following algorithms have in fact two versions, each with minor adaptations in accordance with the difference in the computation of the coverage function. We refer to the Normalized and Independent versions of the corresponding algorithm as NAME-**N** and NAME-**I**, resp. We next describe our selected algorithms.

[8]Company name omitted due to privacy considerations.

- **Greedy** - Our proposed greedy algorithm (Algorithm 1).
- **BF** - A brute-force algorithm that evaluates all subsets of size k, and returns a set with the highest coverage. We use this baseline as it is the only one which guarantees the optimal solution, implying the exact approximation ratios achieved by our algorithm.
- **TopK-W** - An algorithm returning the top-k items by weight. This is the naïve baseline that considers each item individually without taking alternatives into account.
- **TopK-C** - An algorithm that returns the top-k items with the highest Coverage. This is a refined version of the previous baseline, which takes alternatives into account, however not from a global viewpoint as in our solution.
- **Random** - An algorithm that returns k items in a random manner. This is the simplest baseline.

Note that we did not include any SDP or LP based approximation algorithms, as they are not at all scalable (see discussion in section 3.2).

Evaluation Methods. We performed 4 complementary types of experiments to evaluate our end-to-end solution.

First, to examine the actual approximation factors Greedy attains in practice, we compare its coverage to that of BF, which is of course optimal. Greedy has a theoretical approximation guarantee in each variant, however it refers to the worst case, and in practice may achieve much better results. Moreover, we show in these experiments that approximation is necessary, as BF has impractical running times even on tiny inputs.

To show the quality of our algorithm in terms of the coverage it obtains on real-life high-scale data, we compare it to all other baselines, except BF, as it cannot scale to handle real-life data.

To demonstrate the scalability and parallelizability of our solution, in our third set of experiments we run it both on a single thread, varying the number of items (nodes), as well as on graphs of fixed size, varying the number of cores.

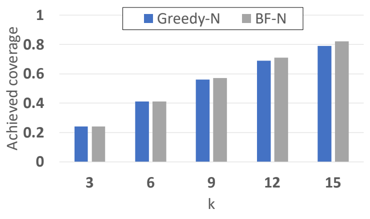
The fourth set of experiments was performed for the complementary minimization problem, where we set different thresholds and aim to find the smallest retained set whose cover exceeds the given threshold. We compare our adapted algorithm to analogous adaptations of the other algorithms, to demonstrate its effectiveness in terms of the size of the retained set.

5.4 Evaluation Results

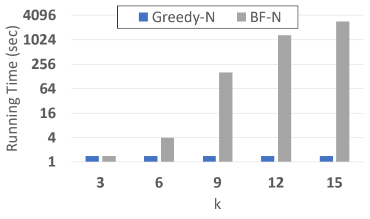
We present the results for the experiments detailed above.

Comparison to Brute Force. The brute-force algorithm does not scale to big graphs, since even for $n = 30$ and $k = 15$, there are $155M$ possible solutions. We show here a representative comparison of the algorithms on a subset of the YC dataset, reduced to 30 products (similar results were obtained for small subsets of all datasets). Figure 4a depicts the coverage achieved by Greedy compared to the optimal coverage achieved by BF. Figure 4b depicts (in log scale) the running times in seconds, over the Normalized variant (the Independent variant showed similar trends, hence omitted). We can see that the coverage of Greedy is very close to optimal, while having significantly better running times.

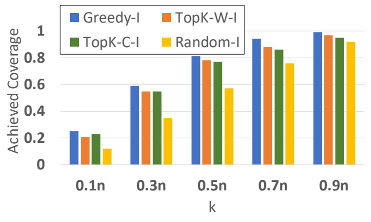
Coverage Quality. We compared all algorithms in terms of the achieved coverage quality, over all datasets. Figure 4c depicts the results on the YC dataset (Independent variant) for $k \in \{0.1n, 0.3n, ..., 0.9n\}$. The results over all other datasets (which include the Normalized variant) demonstrate a similar trend, hence omitted. BF cannot scale beyond small networks, and is excluded



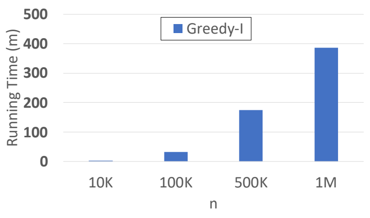
(a) Coverage of Greedy vs BF (Norm.)



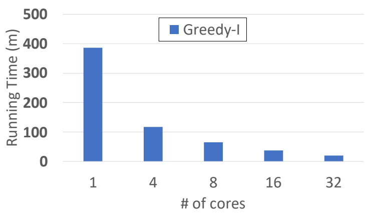
(b) Running time of Greedy vs BF (Norm.)



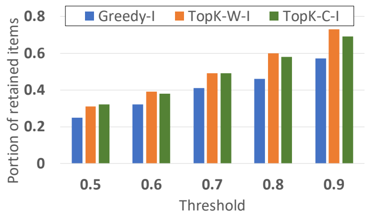
(c) Coverage Quality of all competitors



(d) Scalability of Greedy



(e) Parallelizability of Greedy



(f) Complementary problem

Figure 4: Experimental results

from this experiment. As expected, Greedy is the top performing algorithm, while TopK-W and TopK-C lag behind, since they do not take into account, resp., alternatives, or overlaps in covers by different items. Random also achieves bad results (taking the best across 10 executions) as it makes no use of information pertaining to the popularity of items or their alternatives.

Scalability and Parallelizability. We performed the scalability tests on graphs of various sizes over all datasets. Note that we present only the running times of the algorithm, as the graph construction is considered to be an offline phase, hence not included in the measurements. Figure 4d depicts the running time of Greedy for $n \in \{10K, 100K, 500K, 1M\}$ and $k = 5K$, performed on subsets of the largest dataset (PE), while over other datasets similar trends were demonstrated, hence not shown here. The parallelizability analysis of Greedy, depicted in Figure 4e, was performed over the same dataset and an input graph of fixed size for varying the number of cores: 1, 4, 8, 16 and 32. The results show almost perfect parallelization, which scales well as the number of cores grows. For example, the execution of Greedy on 1 core compared to 32 cores runs 20x times faster.

Complementary problem. We conclude this section by evaluating our approach when adapted to the complementary minimization problem (as explained at the end of Section 3.2). The goal is to find the smallest set whose coverage exceeds a given threshold. Figure 4f depicts the results, in terms of the size of the produced set, obtained by our algorithm, when executed over the YC dataset (Independent variant), for thresholds $\in \{0.5, 0.6, 0.7, 0.8, 0.9\}$, compared to the results obtained by TopK-W and TopK-C. These algorithms were also adapted to perform a binary search over a sorted list of nodes (by the relevant metric - weight or coverage, resp.), choosing the smallest prefix to exceed the threshold. The results demonstrate that the superiority of our approach carries over into this version of the problem, as it outperforms other baselines, producing a much smaller set. The results over the other datasets (and the Normalized variant) are similar.

6 RELATED WORK

E-commerce related problems attracted the interest of many researchers in recent years. Some extensively studied problems are product classification [9, 33], product ranking in search results [18] and automatic product content generation [10, 24]. Of such problems close to us in spirit are works on *diversity* [36], producing the top k most collectively dissimilar elements, typically out of a result set to a given search query. This relates to a similar concept in our problem, where we aim to avoid retaining items that match the same requests. The resemblance is even greater when elements are weighted by importance and there is a requirement, such as in [8], that each non-selected element is covered by a similar selected item, relating to our aim of choosing popular items and items covering as alternatives non-retained items. Nevertheless, there are many differences between their models and ours. Importantly, unlike in diversification problems, we do not aim to maximize diversity, rather it is a feature which is, to some extent, typical of good solutions to our problem, yet not at all necessary. Moreover, even when diversity is a constraint and the items are weighted [8], the goal is to maximize the total weight of the selection, in contrast to our model, where one must also (partially) count weights of adjacent items, which is a crucial property. Furthermore, as our edges represent choice probabilities (rather than item dissimilarity), we can support this original concept of covering neighboring items to a concrete and partial extent, which along with the aforementioned distinctions lead to vast differences in the algorithmic solutions and concrete computations.

Another line of work similar to ours is recommendations [28], as it also deals with selecting a subset of items to increase purchasing probability. However, there are important qualitative and quantitative differences. Primarily, recommendations are typically personalized w.r.t. a given user (and often a given product as well), and deal with a far smaller k. Some works on recommendations that derive product alternatives [20] may potentially serve as basis for another method of computing edge weights in preference graphs. We intend to investigate this approach in future work.

Other existing works in e-commerce that deal with finding top-k beneficial products to offer [35] focus on setting prices such that the predicted revenue (including costs) is maximized. However, in contrast to our models, they do not take into account consumers opting for alternatives.

Closest to our work is a subfield of Operations Research called Assortment Optimization. These works typically employ more complex models such as the Markov chain choice model [4, 23], multinomial logit model [29] and nested logit model [12]. The considered Markov chain model bears some resemblance to our Normalized variant, but is more complicated due to the consideration of varying item revenues and/or multiple-step graph paths (which are directly captured in our model by transitive edges). Consequently, their algorithms are also more complex and the work is geared towards theoretical analysis of the model rather than practical evaluation. The experiments, when exist, consider small scale item sets (order of 1000 items) [4], and the results are not scalable to big data. Furthermore, they mostly use synthetic datasets, and the process of model derivation from real-life data (which our end-to-end solution includes), is not considered there.

Our model is inspired by research in behavioral economics. In particular, [30] observed that consumers experience increased anxiety and are less likely to take action when faced with too wide of a selection. Additionally, [34] demonstrated that consumers, when searching for a specific item, are often willing to buy in its absence what they consider to be a reasonably satisfying alternative.

Our work draws on results in classical Top-k cover problems in graphs [13, 16]. Most notable is the Max Vertex Cover problem (VC_k) [11, 19], which was discussed in detail in Section 3. An existing direction in the study VC_k, whose practical adaptation to our setting would be an intriguing future work, is devising algorithms for graphs with bounded degree [13], as this special case arises in practice in our model. Similar problems include Max dominating set and Max edge domination [21]. All these problems can be viewed as special cases of the more abstract Maximum Coverage problem [6]. Moreover, each of these problems is strongly related to its more extensively researched variant, such as the Vertex Cover problem [25], where k is unspecified, and the goal is to find the smallest subset such that the entire graph is covered. Theoretical bounds and algorithms can often be adapted from one variant to the other, which is also the case for our problem as well, as discussed in Sections 3 and 4.

7 CONCLUSION

This paper introduces the *Preference Cover problem*, which aims to select a reduced inventory maximizing the likelihood of a purchase. We model consumer preferences via a *preference graph* and study two problem variants, *Normalized* and *Independent*, which differ in their interpretation of the probabilistic dependencies between the suitability of different alternatives. We study their approximation hardness, and since the overall number of items, and the bound on the retained set, tend to be very large in this context (in the order of magnitude of millions), we propose highly parallelizable and scalable algorithms that come with approximation guarantees. Finally, we present an end-to-end solution that maps real-world data into our model, and provide an extensive set of experiments on multiple datasets, demonstrating the efficiency and effectiveness of our approach.

In the problem setting studied here (which is common to intermediary e-commerce platforms [1]) the commission-per-purchase is considered fixed and the goal is to maximize the number of sales. Extending our work to support varying per-item revenues and storage considerations is an intriguing future work. Another interesting direction we are currently pursuing is incremental maintenance in response to changes over time.

Acknowledgements This work has been partially funded by the Israel Innovation Authority, the Israel Science Foundation, the Binational US-Israel Science foundation, Len Blavatnik and the Blavatnik Family foundation.

REFERENCES

[1] Marketplace Pricing Model. https://marketplace.webkul.com/marketplace-pricing-model-subscription-vs-commission/.

[2] A. A. Ageev and M. I. Sviridenko. Approximation algorithms for maximum coverage and max cut with given sizes of parts. *IPCO*, 1999.

[3] D. Ben-Shimon, A. Tsikinovsky, M. Friedmann, B. Shapira, L. Rokach, and J. Hoerle. Recsys challenge 2015 and the yoochoose dataset.

[4] J. Blanchet, G. Gallego, and V. Goyal. A markov chain approximation to choice modeling. *Operations Research*, 64(4):886–905, 2016.

[5] M. Cao, Q. Zhang, and J. Seydel. B2c e-commerce web site quality: an empirical examination. *IMDS*, 105(5):645–661, 2005.

[6] R. Cohen and L. Katzir. The generalized maximum coverage problem. *Information Processing Letters*, 108(1):15–22, 2008.

[7] A. C. Doherty, P. A. Parrilo, and F. M. Spedalieri. Complete family of separability criteria. *Physical Review A*, 69(2):022308, 2004.

[8] M. Drosou and E. Pitoura. Multiple radii disc diversity: Result diversification based on dissimilarity and coverage. *TODS*, 2015.

[9] E. Dushkin, S. Gershtein, T. Milo, and S. Novgorodov. Query driven data labeling with experts: Why pay twice? In *EDBT*, 2019.

[10] G. Elad, I. Guy, S. Novgorodov, B. Kimelfeld, and K. Radinsky. Learning to generate personalized product descriptions. In *Proc. of CIKM*, 2019.

[11] U. Feige and M. Langberg. Approximation algorithms for maximization problems arising in graph partitioning. *Journal of Algorithms*, 41(2):174–211, 2001.

[12] J. B. Feldman and H. Topaloglu. Capacity constraints across nests in assortment optimization under the nested logit model. *Operations Research*, 63(4):812–822, 2015.

[13] R. Gandhi, S. Khuller, and A. Srinivasan. Approximation algorithms for partial covering problems. *Journal of Algorithms*, 53(1):55–84, 2004.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[15] S. Gershtein, T. Milo, and S. Novgorodov. Reduce-comm: Effective inventory reduction system for e-commerce. In *Proc. of CIKM*, 2019.

[16] D. S. Hochbaum. *Approximation algorithms for NP-hard problems*. PWS Publishing Co., 1996.

[17] G. Jäger and A. Srivastav. Improved approximation algorithms for maximum graph partitioning problems. *Journal of combinatorial optimization*, 10(2):133–167, 2005.

[18] S. K. Karmaker Santu, P. Sondhi, and C. Zhai. On application of learning to rank for e-commerce search. In *SIGIR*, 2017.

[19] P. Manurangsi. A note on max k-vertex cover: Faster fpt-as, smaller approximate kernel and improved approximation. In *SOSA 2019*.

[20] J. McAuley, R. Pandey, and J. Leskovec. Inferring networks of substitutable and complementary products. In *KDD*, 2015.

[21] E. Miyano and H. Ono. Maximum domination problem. In *The Australasian Theory Symposium*, pages 55–62, 2011.

[22] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions—i. *Mathematical Programming*, 14(1):265–294, Dec 1978.

[23] K. Nip, Z. Wang, and Z. Wang. Assortment optimization under a single transition model. 2017.

[24] S. Novgorodov, I. Guy, G. Elad, and K. Radinsky. Generating product descriptions from user reviews. In *Proc. of WWW*, 2019.

[25] C. H. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *JCSS*, 43(3):425–440, 1991.

[26] C. Park, D. H. Kim, M. Yang, J. Lee, and H. Yu. Your click knows it: Predicting user purchase through improved user-item pairwise relationship. *CoRR*, abs/1706.06716, 2017.

[27] E. Petrank. The hardness of approximation: Gap location. *Computational Complexity*, 4(2):133–157, 1994.

[28] L. Qi, X. Xu, X. Zhang, W. Dou, C. Hu, Y. Zhou, and J. Yu. Structural balance theory-based e-commerce recommendation over big rating data. *IEEE Transactions on Big Data*, 4(3):301–312, 2018.

[29] P. Rusmevichientong, D. Shmoys, and H. Topaloglu. Assortment optimization with mixtures of logits. Technical report, 2010.

[30] B. Schwartz. The paradox of choice: Why more is less. 2004.

[31] A. Strehl and J. Ghosh. Cluster ensembles—a knowledge reuse framework for combining multiple partitions. *JMLR*, 3(Dec):583–617, 2002.

[32] Q. Su and L. Chen. A method for discovering clusters of e-commerce interest patterns using click-stream data. *ECRA*, 14(1):1–13, 2015.

[33] C. Sun, N. Rampalli, F. Yang, and A. Doan. Chimera: Large-scale classification using machine learning, rules, and crowdsourcing. *PVLDB*, 7(13):1529–1540, 2014.

[34] W. Verbeke, P. Farris, and R. Thurik. Consumer response to the preferred brand out-of-stock situation. *EJM*, 32(11/12):1008–1028, 1998.

[35] Q. Wan, R. C.-W. Wong, and Y. Peng. Finding top-k profitable products. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1055–1066. IEEE, 2011.

[36] Y. Wang, A. Meliou, and G. Miklau. Rc-index: Diversifying answers to range queries. *PVLDB*, 11(7):773–786, 2018.

[37] H. Wolkowicz. *Simple efficient solutions for semidefinite programming*. 2001.

Cost Estimation Across Heterogeneous SQL-Based Big Data Infrastructures in Teradata IntelliSphere®

Kassem Awada
Teradata Labs, CA, USA
kassem.awada@teradata.com

Mohamed Y. Eltabakh*
Teradata Labs, CA, USA
mohamed.eltabakh@teradata.com

Conrad Tang
Teradata Labs, CA, USA
conrad.tang@teradata.com

Mohammed Al-Kateb
Teradata Labs, CA, USA
mohammed.al-kateb@teradata.com

Sanjay Nair
Teradata Labs, CA, USA
sanjay.nair@teradata.com

Grace Au
Teradata Labs, CA, USA
grace.au@teradata.com

ABSTRACT

In big data ecosystems, it is becoming inevitable to query data that span multiple heterogeneous data sources (remote systems) to build meaningful querying and analytical workflows. Existing work that aims at unifying heterogeneous systems into a single architecture lacks the fundamental aspect of efficient cost estimation of SQL-based operators over remote systems. The problem is fundamental because all modern optimizers are cost-based, and without accurate cost estimation for each query operator, the generated plans can be way off the optimal plan. Nevertheless, the problem is mostly overlooked by existing systems because the focus is either on homogeneous distributed RDBMSs in which cost estimation is already extensively studied, or on fully heterogeneous engines in which SQL querying and SQL query optimization are not applicable (or at least are not the core problem). In this paper, we propose a comprehensive remote-system cost estimation module for SQL operators, which is a core module within the Teradata IntelliSphere architecture. The proposed module encompasses three costing approaches, namely *logical-operator*, *sub-operator*, and *hybrid* approaches, which are suitable for black box, open box, and a mix of black and open box systems, respectively. The cost estimation module leverages analytical and deep learning models with novel techniques for efficient extrapolation when needed. The techniques presented in this paper are modular and can be adopted by other systems. Extensive experimental evaluation shows the practicality and efficiency of the proposed system.

1 INTRODUCTION

There has been an increasing necessity, especially in big data applications, for managing and querying data that span multiple heterogeneous data sources (remote systems) [12, 13, 31]. The number of the remote system types is increasing dramatically, each system has unique inherent characteristics and processing capabilities, some systems are *openbox* with well-known internal details while others are *blackbox* with very little knowledge about their internals—and many levels in between, and each system offers different levels of sophistication w.r.t. query planning and optimization. Although such interconnectivity and interoperability create unprecedented opportunities for advanced analytics and data sciences, the unification of such diverse systems in a single architecture and the orchestration of the overall processing among them represent a classical challenging problem of many facets.

*Dr. Eltabakh is a faculty in the Computer Science Department at Worcester Polytechnic Institute, MA, USA (meltabakh@wpi.edu).

Several architectures have been proposed to address different aspects of the problem including *federated systems* [8, 11, 24], *polystore systems* [13], and *data integration and warehousing systems* [10, 12, 28, 31]. A big bulk of federated systems' research has focused on distributed relational database systems where distributed transaction processing, concurrency control, recovery control, and replica management have been extensively studied [9, 14, 20, 24]. Other research focuses on heterogeneous federated systems, where schema mapping, query translation, conflict management, and mediation design are the core addressed issues [10, 12, 28, 31]. More recent polystore systems, e.g., the BigDAWG system [13], target transparent unification and access across multiple backend systems of different data models, e.g., array, graph, streaming, and relational models. Although query optimization is a core component of BigDAWG, building an advanced cost estimation module is not the current focus as reported in [13]. Finally, the data integration and warehousing systems focus on offline data integration issues in contrast to online ad-hoc querying and query optimization.

"Teradata IntelliSphere" [4] is a project that shares a common theme with the aforementioned systems, i.e., accessing data across multiple heterogeneous data sources. In the *IntelliSphere* architecture (See Figure 1), Teradata is the master engine and the communication point with the end-users. The other underlying sources (called *remote systems*) are heterogeneous, but they are all assumed to have SQL-like interface (even if the internal execution is not SQL). This covers a wide spectrum of systems such as Hive [25, 26], SparkSQL [7], Presto [27], Impala [22], and other RDBMSs [1, 2, 23]. Therefore, *IntelliSphere*'s query language is SQL, and Teradata is responsible for building a SQL query plan and deciding where each SQL operator, e.g., join or aggregation, will execute on one of the *IntelliSphere*'s systems (either Teradata or a remote system).

In this paper, we focus on one fundamental aspect of *IntelliSphere*, which is the *cost estimation of a given SQL operator over remote systems*. The *"cost"* in our context is basically the elapsed execution time of a SQL operator on the remote system. The problem is fundamental because all modern optimizers (including Teradata's optimizer) are cost-based, and without accurate cost estimation for each query operator, the generated plans can be way off the optimal plan. Evidently, in the popular pay-as-you-go cloud model, bad execution plans can have unacceptable time and monetary overheads. Despite the importance of the problem, it is briefly touched by existing systems because as highlighted above, and will be elaborated on further in Section 6, each of the existing systems focuses on other aspects of the big problem.

Accurately estimating a remote operator cost is a challenging problem because: 1) Some remote systems are *openbox* where experts can inject a lot of details about them into *IntelliSphere* while other systems are *blackbox* with very little knowledge on how they execute. 2) Two remote systems, e.g., Hive and Impala, may offer entirely different set of algorithms to physically implement a given operator, e.g., joining two tables, and thus whatever learned from one system does not necessarily apply to another system. 3) Within a single remote system, it is not trivial for *IntelliSphere* to predict which physical algorithm, possibly from several candidates, will be used for a given operation. And 4) Putting the simplistic assumption that all remote systems are blackboxes and the only way to learn their behavior is by submitting many queries as in [13] is not a practical scalable solution. This is because, as we will show in the paper, such approach of learning is very expensive and should be used as a last resort instead of the default and only solution.

In this paper, we propose a comprehensive remote-system cost estimation module for SQL operators that addresses the challenges highlighted above. To be specific, the costing metric that we try to measure in this project is the *elapsed execution time within the remote system*. This time encapsulates and reflects other detailed costs, e.g., query compilation, scheduling, I/O and CPU costs within the remote system. We assume that the network costs, e.g., establishing a connection and data transfer back and forth, are learned through some other mechanisms, which are outside the scope of this paper. Ultimately, the Teradata optimizer will combine multiple costs together to come up with a final cost for the SQL operator, and based on that it decides where to execute the operator. The techniques presented in this paper focus only on estimating the elapsed execution time, which is a major factor in the cost equation.

We propose three costing approaches, namely *logical-operator*, *sub-operator*, and *hybrid* approaches, which are suitable for *blackbox*, *openbox*, and a mix of black and open box systems, respectively. The cost estimation module leverages analytical models as well as deep learning models within the different approaches. We show that although the deep learning models are good in capturing non-linear cost estimation, they fall short in providing accurate estimations for un-seen (un-trained) ranges. To overcome this limitation, we propose *online remedy* and *offline tuning* phases to enhance the estimation quality.

The key contributions of the paper are summarized as follows:

• Proposing a comprehensive remote-system cost estimation module for SQL operators, that encompasses three approaches, namely *logical-operator* (*logical-op*), *sub-operator* (*sub-op*), and *hybrid* approaches. Each of the *logical-op* and *sub-op* approaches has pros, cons, and applicability cases. The *hybrid* approach combines their advantages.

• Leveraging both analytical cost models and deep learning models within the different costing approaches. The deep learning models are empowered with online remedy and offline tuning phases to ensure high quality estimations even for un-trained ranges.

• The proposed cost estimation module is modular, and due to its applicability to openbox and blackbox systems, it can be easily adopted by and integrated within other systems such as polystore systems.

• Evaluating the proposed cost estimation module empirically in the context of Teradata and Hive as a proof of concept. Extensions to other systems such as SparkSQL, Presto, and Impala follow the same methodology. The results show the effectiveness of the proposed module compared to the state-of-art approaches.

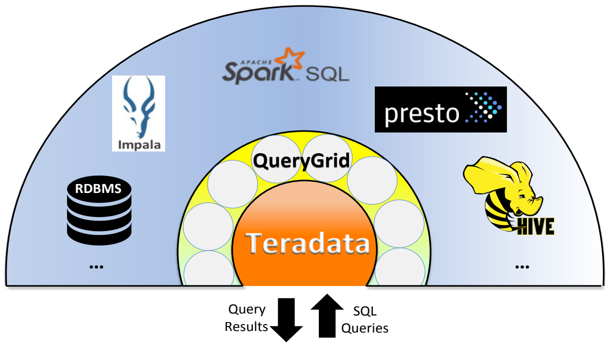


Figure 1: Teradata IntelliSphere Architecture.

The rest of the paper is organized as follows. In Section 2, we present the architecture of the *IntelliSphere* system and introduce the problem definition. In Sections 3, 4, and 5, we describe the details of the three costing approaches: *logical-op*, *sub-op*, and *hybrid*, respectively. In Section 6, we overview the related work, and in Section 7, we present the experimental evaluation of the system. Finally, Section 8 contains the conclusion remarks.

2 TERADATA INTELLISPHERE

In this section, we overview a simplified architecture of the *Teradata IntelliSphere* system [4] and the basic workflow components related to this paper[1]. *Teradata IntelliSphere* is designed to be a cost-effective and scalable analytical ecosystem that offers numerous software solutions to ingest, access, and manage big data across multiple heterogeneous data sources. For the purpose of this paper, we focus on the following basic components of the architecture (See Figure 1):

Teradata: The master engine in the entire architecture is the Teradata Database [2]. It also represents the communication point with the end-users. It receives a user's query in the form of a SQL query, generates a cost-based efficient query plan where each SQL operator is scheduled for execution on one of the *IntelliSphere*'s systems, combines the results, and passes the final answer back to the user.

Remote Systems: The underlying heterogeneous data sources are referred to as *remote systems*. They are all assumed to have SQL-like interface where they can receive a SQL operation such as a *join*, *aggregation*, *filter*, and *projection*, perform the computations of that operation and return the results back to Teradata. It is possible that the internal execution of a remote system is different from the relational DBMS model, e.g, Hive's internal execution is map-reduce. And it is also possible that a remote system may not support some of the SQL operations, e.g., a remote system may not have the capability to perform a join operation.

Remote System Profile: Each remote system registers in the *IntelliSphere* architecture through a profile. This profile describes the remote system setup, e.g., a cluster configuration, and the capabilities of the remote system, e.g., what operations it can or cannot support. The profile is constructed during the registration step, and can be modified afterwards as needed. We will use the profile extensively to store all metadata information related to the cost estimation module as will be described in the following sections.

QueryGrid: It is the communication layer that facilitates the transfer of data across the involved systems [3]. Several QueryGrid

[1]*Teradata IntelliSphere* is a more comprehensive architecture with features and functionalities beyond what is presented in this paper. We only highlight the aspects related to our paper.

connectors are built to enable queries to access tables stored in remote systems. The differentiating factor between Teradata's QueryGrid technology and other connectors is that it works in conjunction with the query processing engines to optimize the overall execution. For example, simple predicates—in a well-defined language—can be passed to QueryGrid for execution on-the-fly while the data is being transferred from one system to another. This capability can save unnecessary scanning of a local data, writing back to the file system after evaluating the predicate, and then passing the results to the QueryGrid for transfer.

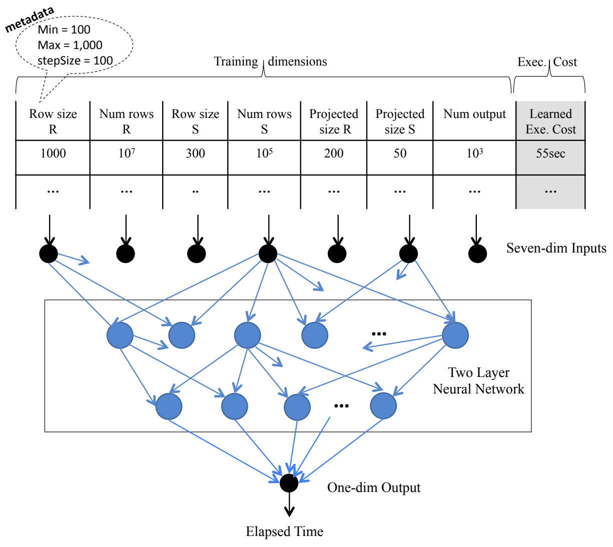
Data Storage, Statistics, and Transfer: A given dataset consists of a set of tables $\{T_1, T_2, ..., T_k\}$, where each table is stored on one of the *IntelliSphere*'s systems (Teradata or a remote system). Any remote table is registered inside Teradata as a *foreign table*—and thus Teradata knows its schema and location. As a result, a single SQL query can seamlessly reference multiple foreign tables across several remote systems. We assume that Teradata can collect basic statistics on remote tables, e.g., the number of rows, average row size, the number of distinct values in each column, etc. Such information is either already available on the remote system or Teradata can estimate them by submitting some queries over the data. Regarding the transfer of data, the data cannot be transferred directly between two remote systems, instead it can be only transferred between a remote system and Teradata.

Query Plans: As in standard RDBMSs, Teradata generates many equivalent SQL query plans during the optimization phase, and part of that is deciding on where each operator will execute—which clearly implies different costs depending on the host system. To limit the search space, *IntelliSphere* considers scheduling an operator only on a remote system that owns the input data (or part of it) or the Teradata system. For example, assume joining two relations R and S, where R is stored in Hive and S is stored in Presto. Then, there are three possibilities for placing the join operator, either on Hive (and S will be passed to Teradata and then to Hive), on Presto (and R will be passed to Teradata and then to Presto), or on Teradata (and both R and S will be passed to Teradata). The results computed on a remote system do not have to be immediately transferred to Teradata, instead they may remain on that remote system for further computations, and then at some point in the query, the results will be transferred to Teradata.

Problem at Hand and Design Assumptions: Given the setup described above, *IntelliSphere* leverages the full-fledged capabilities of Teradata's mature optimizer in generating efficient cost-based query plans. The only missing piece is estimating an operator's cost were it to be executed on a remote system. As highlighted in Section 1, this cost involves several factors, we only focus on estimating the wall-clock elapsed execution time within the remote system. Therefore, while estimating the execution cost, we assume that the needed data is already on the remote system—and thus the network communication and data transfer costs are out of the picture [2].

Supervised ecosystem: The learning and model building step for a given remote system is performed only once when the remote system is added to the *IntelliSphere* ecosystem. Therefore, the learned models are for specific cluster configuration, access methods, physical data layout, etc. The *IntelliSphere* ecosystem is supervised in the sense that changes to a remote system, e.g., adding or removing nodes, creating or dropping indexes, re-partitioning

[2]Teradata can estimate the amount of data that need to be sent to the remote system as well as the output size that will be sent back to Teradata. Based on these estimates, other costs such as the the network cost and data transfer are estimated.



Row size R	Num rows R	Row size S	Num rows S	Projected size R	Projected size S	Num output	Learned Exe. Cost
1000	10^7	300	10^5	200	50	10^3	55sec
...	...	..	...	...	...	...	...

Figure 2: Logical-Operator Costing for Join Operator.

the data, etc., are known to Teradata. Such changes, would require re-doing the learning phase from scratch.

Stable workload: Another underlying assumption is to have a roughly consistent workload on the remote system. That is, the workload during the training and model construction phase should roughly remain the same while executing users' queries later. In our experiments, we assume the remote system is dedicated to the submitted queries and no other workloads are running. Supervised ecosystem and stable workloads are the same assumptions used in almost all other related work [15, 21, 30], otherwise it is impossible to predict the remote system behavior.

Integration in bigger query plans: In Teradata, the cost of a SQL query operator includes several low-level factors such as the I/O costs, e.g., index scans, disk page accesses for data, and CPU costs, e.g., hash table creation, hash table lookup, records merge or sort, etc. Ultimately, these costs are translated to an estimated execution time cost per operator. As such, the estimated execution time for the remote operators fit directly in bigger plans.

3 LOGICAL-OPERATOR COSTING

One approach for estimating a remote operator cost is the *Logical-Operator Costing* (*logical-op costing* for short). In this approach, the training and learning phase is performed at the logical operator level, e.g., join and aggregation operators. This is the approach used in other systems, e.g., BigDAWG [13]. The main idea of the logical-op costing is to build a relatively large set of training queries, execute them on the remote system, and build a model for the target operator. The key advantage is that it requires no knowledge about the internal execution of the remote system, e.g., it does not need to know which physical join algorithm is used to perform the join. In other words, the remote system is treated as a blackbox. However, the main drawback is that to build a reasonably accurate model for a given operator, it would require a large number of queries to cover a wide range of possible configurations. This would certainly require a prolonged training phase and potentially consume valuable resources. In the following, we describe in detail the phases involved in this costing approach.

Building a training dataset: In general, the more complex the logical operator and the more variations in physically implementing it, the more training queries are needed to build its corresponding model. We created logical-op training models for the join and aggregation operators. For the join operator, the training model has seven dimensions, which include the *row size* and the *number of rows* in each of the two tables, the *sum of the projected attribute sizes* from the each table, and the *number of output rows* (See Figure 2). For the aggregation operator, the model has four dimensions, which include the *number of input rows*, *input row size*, *number of output rows*, and *output row size*.

Coming up with appropriate training dimensions is crucial and requires some level of expertise. On one hand, we need to minimize the number of dimensions because the number of queries grows exponentially when adding more dimensions. On the other hand, we need to capture enough parameters in order to model the targeted operator accurately. Based on our team's experience with the Teradata query optimizer, we selected the highlighted dimensions as the representative dimensions for the join and aggregation operators.

The next step is to assign for each dimension a domain reflecting the possible training values that this dimension may take. In some applications, there can be samples of existing data or workloads to help selecting the appropriate domain for each dimension. Otherwise, we start with reasonable assignments, and then a continuous learning phase will help to gradually expand the domains as the system observes and executes more queries (as will be explained later).

Assume dimension i has a domain d_i of size $|d_i|$, then the total number of configurations in the training set for one operator is computed as $\prod_{i=1}^{k} |d_i|$, where k is the number of dimensions. For example, as illustrated in Figure 2, each row represents one configuration, which maps to a single query over the remote system. After executing the queries, each configuration will be labeled with the observed execution cost. This step of executing the queries over the remote system can be very expensive, e.g., it may take days if the number of queries is large.

Building a costing model: The next step in the logical-op costing is to build a model from the observed costs. For that purpose, regression or neural network models can be used. We experimented with both, and we found out that linear regression models introduce more errors as will be demonstrated in the experiment section (around three times larger w.r.t the root-mean-square error RMSE). This is primarily because the number of data points can be large, e.g., in thousands, the number of input dimensions can be also large, and the relationship between the inputs and outputs might not be linear—especially for complex operators like join and aggregation. Simple light-weight neural networks tend to be more accurate under such complex modeling. For that reason, we opt for the neural network model in the rest of the paper.

There is no rule of thumb for deciding on the optimal neural network structure. Typically, two or three hidden layers are enough for not highly-complex problems [18]. Therefore, we fix the number of layers to two for both the join and aggregation operators. And then we use a cross-validation technique to determine the number of nodes (neurons) in each layer [16]. More specifically, we vary the number of nodes in the 1^{st} layer between the number of inputs (7 for join, and 4 for aggregation) and the double of that number, and vary the number of nodes in the 2^{nd} layer between three and half the number of the 1^{st} layer's nodes. Then, for each topology, we use a cross validation test involving 70% of data as training and 30% as a test to measure the accuracy of the network.

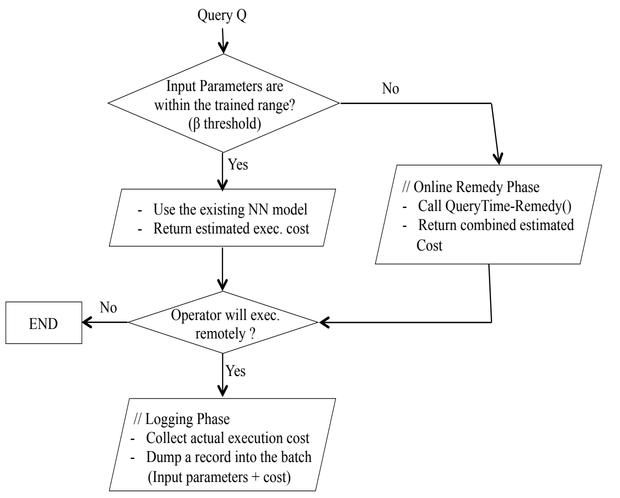

Query Q
Input Parameters are within the trained range? (β threshold)
No
Yes
- Use the existing NN model
- Return estimated exec. cost
// Online Remedy Phase
- Call QueryTime-Remedy()
- Return combined estimated Cost
END
No
Operator will exec. remotely ?
Yes
// Logging Phase
- Collect actual execution cost
- Dump a record into the batch (Input parameters + cost)

Figure 3: Logical-Operator Costing: cost-estimation flow chart at query time.

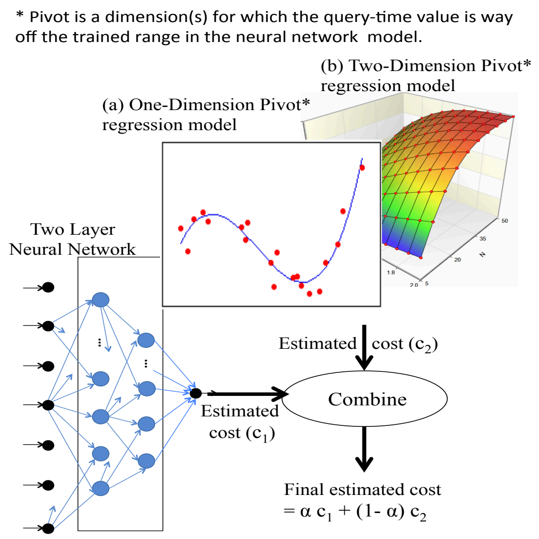
Finally, we select the topology that introduces the least root-mean-square error (RMSE). Figure 2 shows the neural network model over the seven-dimension inputs for the join operator.

Usage and model expansion: In the typical scenarios, the constructed model is directly used at query time to estimate the cost of a remote operator. Given an operator that is candidate for execution on a remote system, e.g., a join operator where one of the input relations is on that remote system, the system calculates and/or estimates the input parameters for the operator's model. For example, the seven input parameters illustrated in Figure 2 need to be estimated for the join operator. These parameters are then fed to the neural network model to predict the output value, which represents the estimated cost (See the flowchart in Figure 3).

The estimation process is straightforward as long as all the input parameters fall within (or in the proximity of) the ranges on which the neural network is trained. However, in the cases where one or more parameters are way off the trained ranges, the model may not provide accurate estimation. This is because neural networks are good in capturing complex relationships but not good in extrapolating out-of-range values.

In real deployment systems like Teradata *IntelliSphere* these cases need to be adequately handled. Therefore, we propose a two-phase solution that consists of an *online query-time remedy phase* and an *offline batch tuning phase*. The online remedy phase provides an immediate best-guess estimation to the operator at hand to continue the query optimization and execution. Whereas, the offline batch tuning phase provides a mechanism for readjusting and tuning the neural network from the actual logged executions. Both phases are described in detail below.

Online Remedy Phase: The main idea of the online remedy phase is presented in Figures 3 and 4. Initially, the system maintains metadata information for each input dimension in the training set of a given operator. This metadata includes the covered range using *min* and *max* boundaries and a *stepSize*. For example, Figure 2 shows the metadata of the *Row size* dimension, which indicates the training covers the range from 100 to $1,000$ bytes and the step size is 100. Now, if the current query at hand involves a join where the estimated row size is $10,000$ bytes, the system will detect that this parameter is way off the trained range, and will not get the estimate by relying only on the neural



* Pivot is a dimension(s) for which the query-time value is way off the trained range in the neural network model.
(b) Two-Dimension Pivot* regression model
(a) One-Dimension Pivot* regression model
Two Layer Neural Network
Estimated cost (c_2)
Combine
Estimated cost (c_1)
Final estimated cost = α c_1 + (1- α) c_2

Figure 4: Online Remedy Phase: Combining Neural Network and On-the-Fly Regression Model for Cost Estimation. (a) Regression model for one-dim pivot, (b) Regression model for two-dim pivot.

network model, but instead it will trigger the execution of the `QueryTime-Remedy()` procedure (See the top diamond box in Figure 3). More specifically, if the value of a given dimension is outside the *[min, max]* range by more than $\beta * stepSize$, where $\beta > 1$ is a configuration parameter, then that dimension is considered way off the trained range. The procedure on the fly builds a regression model and combine it with the neural network model to come up with the final estimate as illustrated in Figure 4.

Lets illustrate the construction of the regression model using a simple scenario. Assume a join query Q that involves only one dimension, say *Row size of R*, where its estimated value is way off the trained range in the neural network. We refer to that dimension as the *Pivot* dimension. All other dimensions (refer to them as $D_{inRange}$) are within the trained range. The `QueryTime-Remedy()` procedure extracts a set of training records of size k, where k is a system parameter, having the following properties: (1) their values in the $D_{inRange}$ dimensions are matching (or very close) to the corresponding values in Q, and (2) their values in the *Pivot* dimension are the immediate successors and/or predecessors of the corresponding value in Q. This set should represent the closest possible training points to the query point. The pivot values in this set are then extracted and used to build a regression model. The algorithm can be extended to handle more than one pivot dimension as illustrated in Figure 4.

The `QueryTime-Remedy()` procedure uses the constructed regression model to extrapolate on the pivot dimension(s) and predict the cost. This cost is then combined with the estimated cost from the neural network model to come up with the final cost (See Figure 4). The reason we combine the two costs is that they capture different and complementary factors. The neural network captures the complex relationship between the input parameters and the output but cannot extrapolate. In contrast, the regression model can extrapolate but oblivious to the other dimensions. The costs are combined using a weighted factor α $(0 < \alpha < 1)$ as illustrated in Figure 4. Initially, α is set to 0.5, and as the system executes more queries, α gets automatically adjusted to narrow the gap between the estimated and actual execution times.

Offline Tuning Phase: Whenever *IntelliSphere* executes a remote operator on an external system (depending on the optimizer's decision), it captures the actual execution cost and pushes this information to a log (See the bottom diamond box in Figure 3). Periodically, this log is fed to the neural network model to tune its structure with the new observed data.

One interesting detail to highlight here is the mechanism for updating the metadata information of the training dataset at the end of the tuning phase. Recall that a metadata information is maintained for each dimension in a training set including the *min, max, stepSize* values. When a log gets executed to update the neural network model, the metadata gets also updated. More specifically, the *[min, max]* range gets expanded if the log has entries with out-of-range values. However, this expansion takes place only if a continuity in the training points is maintained. For example, referring to the metadata in Figure 2, if the log has some entries with out-of-range values or the 1^{st} dimension like $8,000$ and $10,000$ bytes, then the current range will remain intact because there are many missing points between that range and the new values, i.e., continuity will be broken. Instead, more information is added to the metadata to indicate that training dataset of $8,000$ and $10,000$ bytes

The implication of this expansion strategy is that when a new join query comes and it includes an out-of-range value for the 1^{st} dimension, say $6,000$ bytes, the system will still trigger the *online remedy phase* highlighted in Figure 4 to come up with the final estimated cost instead of relying only on the neural network model. The positive thing is that the prediction from both the neural network and the regression models are getting better because they take into account the previous log records even if the [*min, max*] range has not been modified.

4 SUB-OPERATOR COSTING

Another approach for remote operator costing is what we refer to as *sub-operator costing* (or *sub-op costing* for short). In this approach, the learning and training is performed at the granularity of small building block operators, e.g., scan, shuffle, sort, read, and write operations. And then, the higher-level query operators, e.g., join and aggregation, are expressed as formulas on top of the sub-ops. The main advantage is that learning the cost of each sub-op is relatively straightforward and fast because: (1) The number of dimensions in a training set for each sub-op is typically very small (only two or three), (2) As a result of the low-dimensionality, the number of needed training queries is very small—which saves training time and cost, and (3) The logic and behavior of each sub op is relatively simple and thus linear regression is typically enough to model most of these sub ops.

On the other hand, the main disadvantage of the sub-op approach is that it requires a great deal of knowledge about the remote system, which may not be available in some cases. For example, it requires identifying a set of the building block operators (the sub ops) that is sufficient to accurately model the query operators. It also requires understanding the different algorithms of the physical implementations for the different operators, e.g., a join operator can have four or five different physical algorithms such as broadcast join, re-distribution hash join, etc., and defining a formula to express each algorithm in terms of the sub ops. Evidently, if such level of knowledge is not already available, then it takes a long time to collect with these details.

Identifying sub operators and costing formulas: The first step in this approach is to identify the key sub operators of the

Basic (Mandatory)	Read (DFS) [1]	r_D	Reading a record from dist. file system
	Write (DFS) [2]	w_D	Writing a record to dist. file system
	Read (Local)	r_L	Reading a record from local file system
	Write (Local) [3]	w_L	Writing a record to local file system
	Shuffle	f	Shuffle a record between machine
	Broadcast [4]	b	Broadcast a record to all machines
	Sort	o	Main memory sort cost per record
	Scan	c	Main memory scan cost per record
Specific (Optional)	HashTable Build [5]	h_I	Inserting a record into hash table
	HashTable Probe	h_P	Probing a hash table
	Rec Merge	m	Merging two records

[1] Query that reads from HDFS and does not produce any output.
[2] Query that reads from HDFS and writes back to HDFS. Subtract r_D from the measured values
[3] Query that reads from HDFS and writes content to local file. And then subtract r_D from the measured values
[4] Query that reads from HDFS, produces no output, and broadcasts a file (distributed cache) to all nodes (without reading it). Subtract r_D from the measured values
[5] Query that reads from HDFS, builds a hash table for each HDFS block, and does not produce any output. Subtract r_D from the measured values

Figure 5: List of Common Sub Operators in Remote Systems. Additional sub ops can be defined specifically for certain remote systems.

remote system, which may differ from one system to another. However, in the majority of the modern distributed systems, which have shared-nothing architecture in common, these sub operators typically include: *reading from disk, writing to disk, shuffling across machines, in-memory sorting,* and *scanning a memory block*. Other more specific sub operators include *insertion into a hash table, probing a hash table,* and *merging two records*.

In Figure 5, we highlight a list of the key and common sub operators and categorize them into two categories, namely *Basic* and *Specific*. The sub operators under the *Basic* category are kind of mandatory to learn, otherwise it would not make sense for the corresponding remote system to be costed using this approach. The other sub operators are good to have, but missing them is not a hinder to this approach because either they are specific to few query operators, they are not dominating factors in the cost formulas in which they participate, or *IntelliSphere* can provide rough default values for them. We will provide more details and examples in this section for these sub operators.

It is worth to highlight that Teradata costing mechanism is based on the *sub-op costing* approach. It is highly reliable, efficient to use for estimation, and easy to calibrate and extrapolate whenever needed. Given that all engine details are known, Teradata optimizer maintains a long and detailed list of sub operators. In contrast, for remote systems, it is more practical to assume limited knowledge about them. That is why we try to capture a minimal, but sufficient, set of sub ops as highlighted in Figure 5.

After defining the sub operators, each query operator for which a costing model need to be built, e.g., join and aggregation, need to be expressed as a composition of the sub operators. Since each of these query operators can have multiple physical implementations carrying significantly different costs, it is important for a technical expert to know the list of physical algorithms that are supported by the remote system for a given query operator. For example, Hive supports five types of join algorithms, which are: *Shuffle Join*, *Broadcast Join*, *Bucket Map Join*, *Sort Merge Bucket Join*, and *Skew Join* [19]. Similarly, Spark supports five join algorithms, which are: *Broadcast Hash Join*, *Shuffle Hash Join*, *SortMerge Join*, *Broadcast NestedLoop Join*, and *Cartesian Product Join*. Each of these algorithms need to be expressed in terms of the defined sub operators.

In Figure 6, we provide a detailed example using the *Broadcast Join* algorithm between two relations *R* and *S*, where *S* is assumed to be the small relation. The top part of the figure shows the algorithm workflow while the bottom part shows the corresponding cost formula. The algorithm starts by reading the small relation *S* from the distributed file system, e.g., HDFS, and broadcasting it to all workers, and it gets stored locally on each machine. Then each task—in Hadoop terminology, it is called a *map task*—executes the loop illustrated in Figure 6. Basically, each task reads relation *S* and builds a main memory hash table, and then reads one block from the big relation *R* and for each record in that block, it probes *S*'s hash table for possible joins. The read block from *R* is assumed to be on the local disk because most distributed systems try to achieve data locality by putting the computational task on the machine storing the data. Previous studies have shown that although data locality is a best effort mechanism, it is achieved more than 90% of times. The last step in the workflow is for the task to write its output back to the distributed file system[3].

The costing formula in Figure 6 has almost one-to-one mapping to the steps in the workflow. We use the notation | | to indicate the cardinality (number of records) of an input. The term `NumTaskWaves` represents the number of cascaded tasks executed on a single machine. It is computed as total number of tasks in the join job divided by the total number of parallelism in the system, i.e., the total number of cores. Notice that the values for factors such as `NumTaskWaves`, `|Block(R)|`, and `|TaskOutput|` are calculated and/or estimated by another module in the *IntelliSphere* system different from the costing module and that module is outside the scope of this paper.

Building a training dataset: The upfront effort put in specifying the sub ops and the cost formulas will pay of by simplifying the subsequent phases of the sub-op costing approach. For the training dataset, what is needed is to build a set of queries for each of the sub ops to learn its cost in the remote system. Since each sub op is *primitive*, the number of dimensions in its training dataset is very small. In fact, we found it enough for almost all sub operators under the *Basic* category (Refer to Figure 5) to have only two dimensions in the training dataset, which are the *number of records* and *record size*. The only exception is the *Broadcast* sub operator, which requires a third dimension, which is the *number of machines*.

Since the number of dimensions is small, and additionally the number of values assigned to each dimension is also small (because the sub op models are easy to extrapolate as we will discuss later), the number of records in the training set becomes very small. In fact, it is between one to two orders of magnitudes smaller than that of the logical-operator approach, which introduces a significant reduction in the training time and cost over a remote system.

For measuring the cost (execution time) for each sub-op, we avoided instrumenting and injecting special code inside the remote system since such instrumentation may not be feasible for some remote systems. Instead, we submitted primitive queries that execute specific type of operations, and from that we extracted the values of the individual sub-ops. In Figure 5, we show examples

[3]Cost formulas for other join algorithms can be derived in the same manner. We omitted them from the paper due to space limitations.

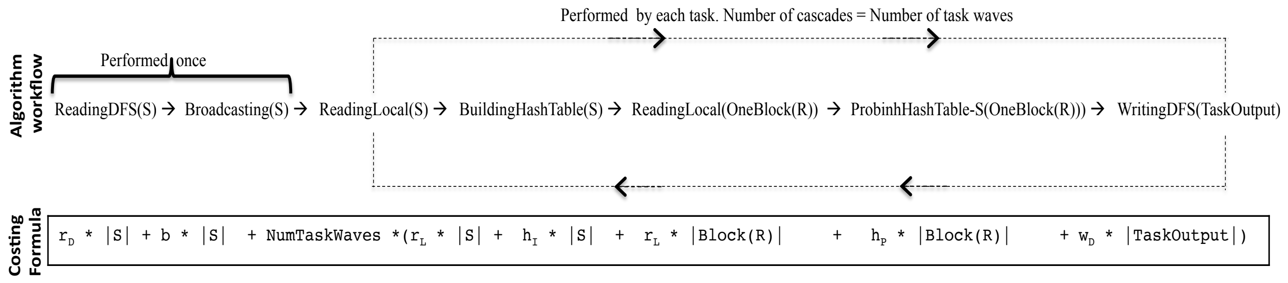

Performed by each task. Number of cascades = Number of task waves
Performed once
Algorithm workflow
ReadingDFS(S) → Broadcasting(S) → ReadingLocal(S) → BuildingHashTable(S) → ReadingLocal(OneBlock(R)) → ProbinhHashTable-S(OneBlock(R))) → WritingDFS(TaskOutput)
Costing Formula
r_D * |S| + b * |S| + NumTaskWaves *(r_L * |S| + h_I * |S| + r_L * |Block(R)| + h_P * |Block(R)| + w_D * |TaskOutput|)

Figure 6: Broadcast Join (R, S) in Hive & Spark (Broadcast Hash Join). The Algorithm workflow and the costing formula in terms of the sub ops. Relation S is the small relation to be broadcasted.

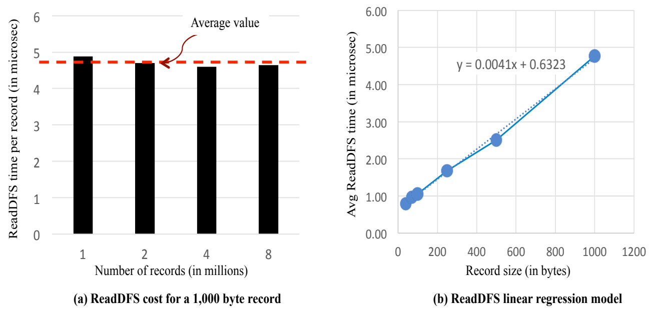


Figure 7: Sub-Op Costing Model for ReadDFS Operator.

of these queries and how they can be used to measure specific sub-ops.

Building a costing model: In this step, a cost model is built for each sub operator. For simplicity, we will focus our discussion on the majority of the sub ops, which involve two dimensions in the training set, i.e., *number of records* and *record size*. It is possible to consider these two dimensions as separate (orthogonal) dimensions while building the model. However, we experimentally observed that the model can be further simplified because for a given record size, say s, the measurements across the other dimension (the records' number) are very similar to each other. Therefore, it is practical to group the measurements by the record size, and compute the average across the varying number of records. In Figure 7(a), we illustrate this observation. The experiment is measuring the *ReadDFS* (Reading from distributed file system) cost for a record size of 1,000 bytes under varying number of records. The dotted line shows the average value. Similar findings are observed for other record sizes and other sub operators.

Based on this observation, a simple linear regression costing model can be built as depicted in Figure 7(b) for the *ReadDFS* operator. As can be noticed a big advantage of the sub-op costing approach is that most sub-ops have simple and tight linear regression models that can be easily learned from small training dataset (more results will be presented in Section 7). Moreover, these models are easy to extrapolate for un-seen values, which is not the case for the more complex neural network models presented in Section 3.

Usage: At query time, lets say a join query between R and S, the first thing to be done by the *IntelliSphere* cardinality estimation module is to provide the required cardinalities and statistics, e.g., the cardinality of each relation, the number of distinct values in the join keys, the average number of records per key, etc. Then, if the operator at hand has only one physical implementation in the remote system, then *IntelliSphere* uses the corresponding cost formula to estimate the cost. Otherwise, if there are multiple possibilities, which is the case for the join operator (Refer to Figures 6), then *IntelliSphere* needs to predict which algorithm the remote system will use.

Predicting the remote system choice is tricky, especially for complex systems such as other relational databases, e.g., DB2, SQL Server, or Oracle. Yet, it is more straightforward for systems like Hive and Spark. Lets take the join operator, which has the most algorithmic variations, as an example. Although it has five algorithms in Hive and Spark, most of the choices can be easily eliminated based on some observations. For example, if the relation in Teradata side, say S, which will be sent to the remote system is not partitioned by the join key—which *IntelliSphere* should know—then the choices of *Bucket Map Join* and *Sort Merge Bucket Join* in the case of Hive can be eliminated. Even if S is partitioned on the join key, but there is no way to tell the remote system such property after the data transfer, then still the two choices above can be safely eliminated. If the join is not Cartesian product, then the choices of *Broadcast NestedLoop Join* and *Cartesian Product Join* in the case of Spark can be eliminated. If both join relations are quite large, then the choices of *Broadcast Join* either in Hive or Spark can be eliminated.

These observations, or what we refer to them as *"Applicability Rules"*, are defined by the technical experts while defining the cost formula for each possible algorithm. *IntelliSphere* uses them at query time to eliminate inapplicable choices based on the cardinalities and statistics at hand. Finally, if there are still multiple possible choices, then the system can either take the highest cost (assuming the worst case scenario), the average cost, or the *"in-house comparable"* cost. The *in-house comparable* cost is applicable when the remote system is another relational database system. In this case, *IntelliSphere* assumes that the remote system will pick the algorithm that Teradata would have picked were the data in-house.

5 HYBRID-OPERATOR COSTING

As highlighted in Sections 3 and 4, each of the sub-op and logical-op approaches has pros and cons. Such tradeoff between the two approaches and the diverse remote systems available nowadays in the Big Data ecosystem call for a hybrid approach that can combine the advantages of both worlds.

In Figure 8, we provide a summary comparison between the two approaches. In general, the sub-op costing model can be significantly superior w.r.t the training cost, training time, and the ease of extrapolation given that a detailed knowledge on the remote systems is already available. Otherwise, the logical-op model would be the favorite.

	Sub-Op Costing	Logical-Op Costing
Modeled Operators	Low-level building block operators such as read, write, scan, and re-distribute	Logical query operators such as join and aggregation
Parameter Space (# dimensions in the training dataset per operator)	The parameter space is small. Most sub-ops need only two dimensions in their training dataset ***Example:*** "read", "write", and "re-distribute" each has two dimensions, i.e., (1) number of records, and (2) record size	The parameter space tend to be large and the number of dimensions is high. ***Example:*** "join" has at least seven dimensions including: (1) record size in R, (2) Number of records in R, (3) record size in S, (4) Number of records in S, (5) projected output record size from R, (6) projected output record size from S, and (7) number of output records
Size of training dataset (# of training queries per operator)	Small, because the parameter space is small	Can be very large because the parameter space is usually large
Training Time	Shorter	Longer
Ability to Extrapolate	Easier	Harder
Remote System Assumption	Open box	Black box
Remote System Prerequisites (Knowledge)	- Knowledge on how logical operators, e.g., join or aggregation, get physically implemented - Knowledge o what types of sub-ops operators to model - Knowledge on how to express logical operators in terms of the sub operators	None. No internal knowledge of the remote system is needed
Model Continuous Tuning (especially for un-seen values)	Less critical because extrapolation is straightforward	More critical because for complex models, extrapolation is not straightforward
Maintenance under change or addition of algorithms in remote system (E.g., adding a new join algorithm)	- Need to change or add a cost formula for the modified or added algorithm - Add the applicability rules indicating the cases under which the new algorithm is applicable	- Need to partially re-run queries from the training set that (hopefully) trigger the modified or new algorithm to learn its execution pattern

Figure 8: Comparison between Sub-Op and Logical-Op Costing Approaches.

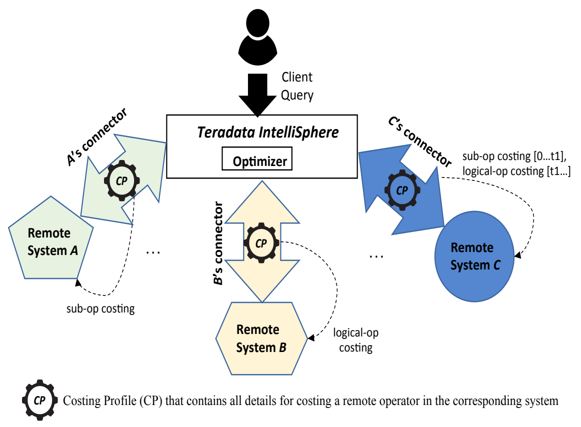


Figure 9: Overview on the Hybrid Costing Model.

The main idea of the hybrid approach is depicted in Figure 9. Basically, the Teradata *IntelliSphere* architecture will connect and communicate with different remote systems using one of the two costing approaches. The choice depends on several factors including whether or not there is enough knowledge about the remote system, and whether or not the resources allow for a prolonged training phases—which can be days in the case of logical-ops .

For example, referring to Figure 9, remote system *A* can be a well-known openbox system, e.g., Hive or Spark, and in this case the sub-op costing can be the model of choice. In contrast, remote system *B* is a blackbox and its workload and resources allow for a prolonged training phase, in this case the logical-op costing model is a good choice. On the other hand, remote system *C* has little knowledge known about it and its workload and resources do not allow for a dedicated several-days training phase (for logical-op training). In this case, an approximate sub-op costing can be applied to *C*—even if not highly accurate—until the more extensive training for the logical-op costing is performed, which may span weeks, and then *C* switches from the sub-op costing to the logical-op costing. The *IntelliSphere* architecture provides such flexibility.

As highlighted in Figure 9, each remote system has a *costing profile (CP)* containing all needed details based on its costing model. For example, for the sub-op costing, it includes a list of the sub-ops, a list of the physical algorithms for each logical operator, the costing formula of each algorithm, and the applicability rules for each algorithm. For the logical-op costing, it includes the neural network model for each operator, the metadata information of the training dataset, plus other information. Updating the costing profile information instantaneously reflects on the remote table costing. Although not currently supported in *IntelliSphere*, the hybrid approach is also applicable within a single system. That is, some operators, e.g., selection and aggregation, can be trained using the logical-op approach, while other higher-dimensional operators such as joins can be trained using the sub-op approach. The CP profile is flexible enough to store different costing models for different operators. We plan to explore this extension in the near future.

6 RELATED WORK

Accessing and querying datasets that span multiple heterogeneous data sources is a complex problem, and several systems and architectures have been proposed to address certain aspects of the problem. In this section, we overview these related systems and emphasize the differences to the *IntelliSphere* system.

Federated Systems: Federated systems provide a virtual layer of a unified access and management over a collection of data sources [8, 11, 24]. The federation can be over a collection of homogeneous relational databases, e.g., distributed DBMSs (Category I), and most of the research in this category focuses on distributed transaction processing, replica management, recovery control, and concurrency control [14, 20, 24].

Systems such as [30] belong to Category I, and they address the cost model issue across multi relational databases by dividing the workload into multiple query classes, then sample a subset of queries from each class and submit them to the remote database(s). The objective is to learn the corresponding unknown coefficients of the cost equation using statistical regression models. This approach is similar to our proposed logical-op learning, however in their work they did not consider the sub-op costing, which some times has clear advantage of the logical-op costing especially when dealing with heterogeneous systems.

The federation can also be over a collection of heterogeneous data sources (Category II), and the focus of this category is on building unified data and representation models, query translation, mediation design, data extraction, schema matching and coalescing, and conflict and resolution management [10, 12, 28, 31]. *IntelliSphere* is fundamentally different from these systems because IntelliSphere's focus is on efficient query plan generation and remote operator cost estimation.

Some work under Category II such as that proposed in [21] addresses the costing in such heterogeneous data sources. However, their assumed sources are not limited to SQL-like operators, e.g., the sources can be web search engines, image processing systems, CAD systems, etc. In this setting, the authors proposed wrappers around each source that acts as a *mini-optimizer* and feeds a global optimizer with the estimated costs for a given operation. The developers of the remote systems need to code these wrappers and augment in them optimizer-like logic to derive the cost of the different possible operations on these remote systems. IntelliSphere is fundamentally different from that work because our focus is only on the costing of SQL operators, e.g., selection, projection, join, aggregation, etc. For that, there are no strong justifications for the complexity of adding a wrapper's layer and the non-trivial task of coding a mini-optimizer for each remote system.

Polystore Systems: The key characteristic of the polystore systems, e.g., the BigDAWG system [13], is that they provide transparent access across multiple engines of different data models, e.g., relational, graph, NoSQL, array, and steaming engines. In BigDAWG, the underlying sources are grouped into islands by their data model type, and then each source has a *"shim"* which acts as the source's communication wrapper. BigDAWG addresses issues including location transparency, casting among the different data models, unified query language, and query planning and optimization across the islands.

The *IntelliSphere* system is distinct from the polystore systems in the following: (1) *IntelliSphere* is not a polystore system because it assumes a common relational-like data model for all underlying data sources with a SQL-like interface. Therefore, *IntelliSphere* does not focus on issues such as casting among the different data models and building a unified query language. (2) Although cost estimation is a fundamental issue in BigDAWG, it is briefly touched and the system is currently using primitive approaches as a first step [13]. In contrast, *IntelliSphere* introduces a comprehensive cost estimation module for efficient query plan generation across the underlying systems. The innovations presented in this project can be certainly leveraged by other systems such as the BigDAWG system.

Table naming convention: T_{x_y} (in total 120 tables)

- x (number of records): {k x 10^4, k x 10^5, k x 10^6, k x 10^7}, where $k \in \{1, 2, 4, 6, 8\}$
  Total configurations: 20
- y (record size): {40, 70, 100, 250, 500, 1000}
  Total configurations: 6

Table Schema: (a_1 , a_2, a_5, a_{10}, a_{20}, a_{50}, a_{100}, z, dummy)
- Each column a_i is of type Integer
- Duplication rate of column a_i is *i* (e.g., each value in a_5 is duplicated 5 times)
- Column *z* is of type Integer, where all values are zeros
- Column *dummy* is of type Character, and is used to reach a specific record size

Aggregation Queries:
- The aggregation factor (shrinking factor in the number of records) is achieved by aggregating over a specific column a_i to get a factor of *i*
- The number of aggregate functions computed varies from 1 to 5. All are of type SUM()

Join Queries:
- The join condition between R and S is fixed to R.a_1 = S.a_1 (which are unique-value columns)
- The output cardinality of the join is thus the cardinality of the smaller table.
  (The values in the smaller table are subset of the values in the larger table)
- To provide more flexibility on the output cardinality, an extra condition is added in the form of (R.a_1 + S.z < threshold). Since S.z is always zero, we can precisely control the selectivity of this predicate before producing the output. Combined with the join condition, the output selectivity is controlled to be 100%, 50%, 25%, or 1% of the smaller table cardinality.

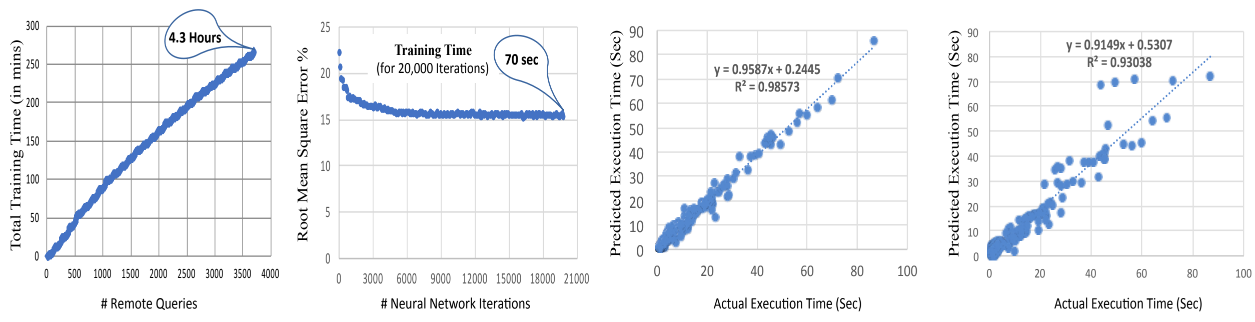
Figure 10: Experimental Setup and Synthetic Dataset Description.

Advanced Learning in Query Optimization: Learning-based models have been studied for both static and dynamic query workloads at coarse-grained plan-level models to fine-grained operator-level models [6]. Machine learning techniques have been also used in the context of query optimizers [17, 29]. The LEO project [17] uses model-based techniques to create self-tuning query optimizer by producing better cost estimates. The work in [29] uses regression techniques to create cost models for XML operators. And the work in [5] proposes building analytical models for query mix interaction to determine good execution schedules. The *IntelliSphere* system combines both the analytical models and machine learning techniques into its cost estimation module.

7 EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the various techniques of the *IntelliSphere*'s cost estimation module. As a proof of concept, we study the learning of one remote system, which is Hive/Hadoop. We focus on evaluating the *aggregation* and *join* operators since they are the most expensive operators in the relational model.

Cluster and Dataset Description: The Hive VM cluster has a total of four nodes, one master node and three data nodes. The total HDFS size is 445GBs divided equally across the data nodes. Each node has 8GBs of memory and two CPU cores model Intel(R) E5-2697@2.7GHz. We used synthetic datasets in which we generated 120 tables. The details of the generated tables are summarized in Figure 10. As presented in the figure, we created 20 different configurations for the number of records, and 6 different configurations for the record size. All tables have the same schema as indicated in the figure. The schema is designed such that the



(a) Query Training Cost (b) Neural Network Conversion Error (c) Neural Network Model Accuracy (d) Linear Regression Model Accuracy

Figure 11: Aggregation Logical-Operator: Training Costing & Accuracy over the remote system.

different columns will have different duplication factor, which facilitates the design of the aggregation and join queries to produce specific output cardinalities. Overall, the generated dataset occupies around 45% of the total HDFS capacity (including the default three-fold replication).

Logical-Op Evaluation: In Figures 11 and 12, we present the logical-op evaluation for the aggregation and join operators, respectively. Recall that the aggregation operator has four parameters (four dimensions) training dataset, which are the *number of input rows*, *input row size*, *number of output rows*, and *output row size*. We created a total of approximately 3,700 aggregation queries by varying the target table (from the 120 available ones), and the shrinking factor and the number of computed aggregates as highlighted in Figure 10. Figure 11(a) shows the cumulative training time needed to execute the queries over the remote system ($\sim$ 4.3 Hours).

The collected cost values are then fed to train a neural network model. As discussed in Section 3, the topology of the network has two layers, and the number of nodes in each layer is decided using a cross-validation technique. We omit such details from this section since it is not part of our core contributions. The neural network is trained using 70% of the data points, and then the accuracy is measured using the remaining 30% of the data points. Figure 11(b) illustrates the convergence of the model. It reaches a steady state after 7,000 to 9,000 iterations. The figures shows a total of 20,000 iterations (x-axis), and the y-axis represents the error percentage, which is measured as ($e \times 100/v$), where e is the root mean square error (REMS), and v is the average execution time over all queries. The entire network training takes negligible time ($\sim$ 70 Seconds).

After building the model, the test dataset (30%) is used to test the neural network model accuracy, which is presented in Figure 11(c). The figure shows very high agreement between the actual (x-axis) and estimated (y-axis) execution times. This indicates that the four-parameter model is a good model for the aggregation operator, and that the neural network model can capture the relationship between the inputs and outputs with high precision. In Figure 11(d), we illustrate the model accuracy under a linear regression model instead of the neural network model. For the aggregation operation, the linear regression model shows a reasonable accuracy, although it is still lower than the neural network model.

Figure 12 illustrates the training cost and accuracy of the join logical operator. The operator has seven dimensions training set (refer to Figure 2). We created a training set of 4,000 queries by varying the possibilities in each dimension according to the procedure highlighted in Figure 10. Figure 12(a) shows that the training time is really high ($\sim$ 26 Hours). It is worth highlighting that our testing cluster is small, and with bigger clusters, more training configurations need to be covered. Hence, the training time shown in Figures 11(a) and 12(a) can easily grow by an order of magnitude.

In Figure 12(b), we show the convergence and error percentage of the trained neural network model over the training dataset. And Figure 12(c) shows how well the model can learn the execution pattern. We tested the accuracy using the test dataset (30% of the entire data), and the model shows good linear correlation. In Figure 12(d), we illustrate the model accuracy under a linear regression model instead of the neural network model. Unlike the aggregation query type in which the linear regression performed relatively well, in the case of the join queries, the regression model performed poorly and could not capture the execution pattern. Therefore, we believe that for logical operators, it is more accurate and stable to use the neural network model.

Sub-Op Evaluation: For the sub-operator costing approach, the training of each sub-op needs only few number of queries, e.g., in the range of few 10s of queries. As mentioned in Section 4, we did not instrument the remote system to measure the execution time of the sub-op, but instead used primitive queries as presented in Figure 5. Figure 13(a) shows the training time for a number of queries ranging from 6 to 32, which is few minutes. The results from those queries are then used to construct a linear regression model for each sub-op. Figures 13(c), 13(d), and 13(e) illustrate the model of the *WriteDFS*, *Shuffle*, and *Rec Merge* sub-ops, respectively.

As we discussed in Section 4 while presenting the *ReadDFS* sub-op (Figure 7), we do not construct a separate sub-op model under different dataset sizes (number of rows). Instead, for each record size, say k bytes, we perform four experiments with varying number of rows (1, 2, 4, and 8 millions), and then use the average value to construct a single linear regression model for each sub-op. This average value is shown to be a good-enough representation across datasets as confirmed by the results in Figure 13(b) (for the *WriteDFS* sub-op as an example), and earlier in Figure 7(a) for the *ReadDFS* sub-op.

For the *Hash Build* sub-op an interesting behavior is observed, which is that the results actually resemble two distinct models (See Figure 13(f)). This is because the sub-op is sensitive to whether the hash table fits in memory or not. We experimented with both cases and constructed a model for each case. Recall that the *Hash Build* sub-op is primarily used in the hash join algorithm, where the smaller of the two joined relations is broadcasted to all machines,

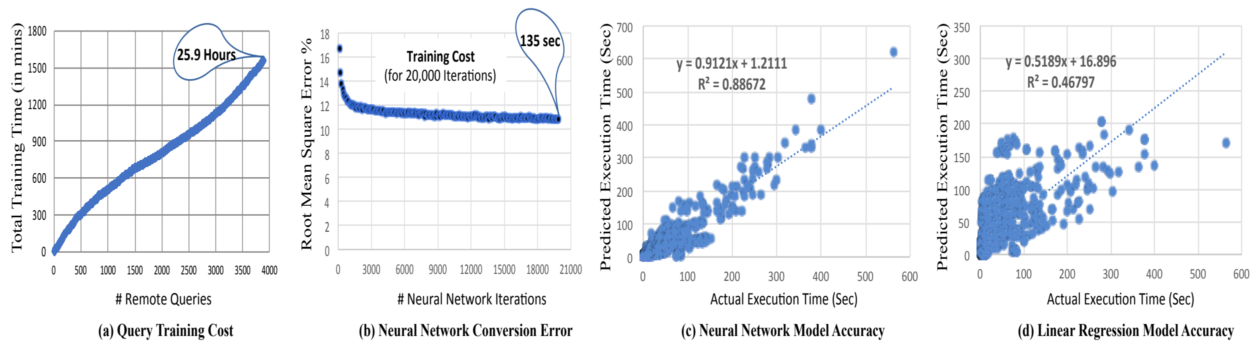


Figure 12: Join Logical-Operator: Training Costing & Accuracy over the remote system.

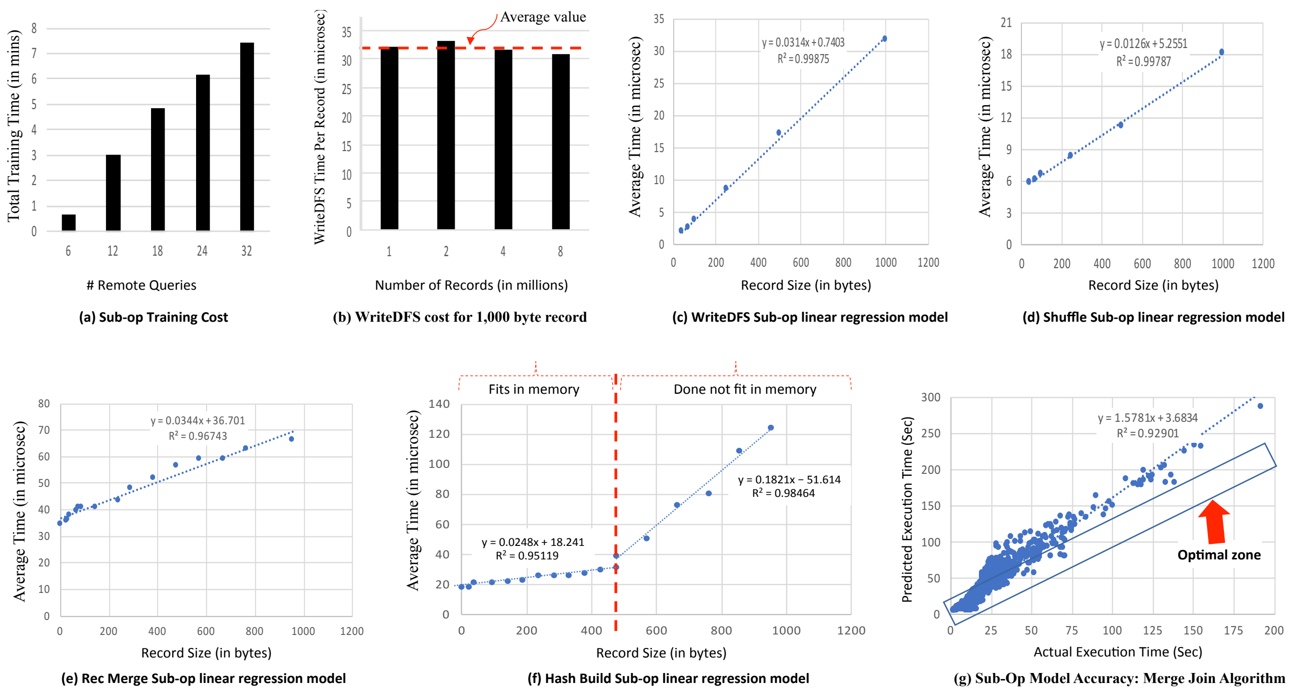


Figure 13: Sub-Op Model: Training Costing & Accuracy over the remote system.

and each machine will build a hash table for this smaller relation. Therefore, given a specific cluster configuration, if the broadcasted relation fits in memory, i.e., falls in the L.H.S area of the vertical dotted line in Figure 13(f), then the corresponding model is used. Otherwise, the system can predict that the broadcasted relation will not fit in memory, and hence the other model is used.

Finally, Figure 13(g) shows the results from combining multiple sub-ops in an analytical formula to estimate the merge join algorithm. Recall that such formula is provided by the domain expert and stored in the remote system profile (Refer to Figures 6). As the results show, the sub-op costing approach provides very good estimation. We found that the sub-op approach slightly tends to overestimate the cost (and similar trend is observed for other algorithms as well), which is a typical trend even within RDBMSs.

Estimation for Out-of-Range Inputs:

In Figure 14, we study the accuracy of the different costing approaches when estimating out-of-range values. This is a typical scenario because an initial training dataset—even if large in size—cannot cover every possible scenario. In this experiment, we studied the merge join algorithm. Both the sub-op and logical-op approaches are trained using datasets of up-to 8x10^6 records

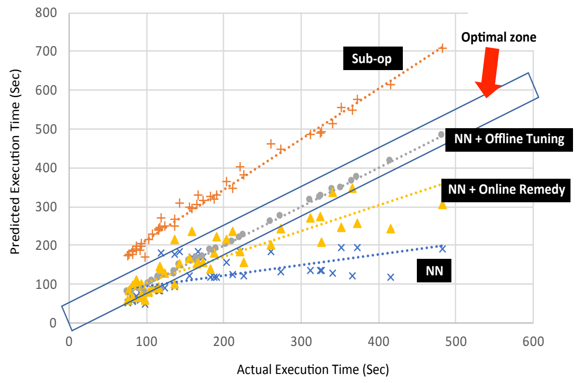


Figure 14: Evaluation of Out-of-Range Prediction Models: Merge Join Algorithm (Fixed α = 0.5).

with different record sizes. Then, the models are constructed from this training dataset. The figure shows the estimation accuracy for a set of new queries, where the number of input records is 20x10^6, while the record sizes are within the trained ranges. We generated 45 queries with different configurations, e.g., in some configurations only one of the join table is out-of-range and in

Table 1: Online Remedy Technique: Automatically Adjusting the Cost-Combining Factor α.

	Batch 1	Batch 2	Batch 3	Batch 4	Batch 5
α	0.5	0.62	0.66	0.57	0.71
RMES%	16.32%	12.6%	12.2%	10.87%	9.1%

other configurations both tables are out-of-range. We compared the estimation accuracy of the *sub-op* approach with that of the *logical-op* approach (the neural network "NN" model).

The results show that the sub-op approach is relatively consistent and can easily extrapolate its trained range to cover out-of-range values. However, due to the non-linearity in the neural network model (the "NN" line), its accuracy degrades and cannot extrapolate well. Interestingly, with the *Online Remedy* technique (Introduced in Figure 4), the accuracy of the estimation improves significantly as depicted in the figure. In this experiment, we fix α (the cost-combining factor) to 0.5.

We also measured the accuracy of the offline tuning phase as follows. We randomly divided the new out-of-range queries (45 in total) into two batches of sizes 70% and 30% roughly. The observed execution times from the 70% batch are added to the neural network model before executing the remaining 30%. And then, the accuracy of remaining 30% is measured. As Figure 14 shows the model adjusts its weights and nicely learns to provide accurate estimations for the new ranges.

Finally, to measure how well the system can adjust the cost-combining factor α in the *Online Remedy* technique (Refer to Figure 4), and its effect on the performance, we performed the following experiment. We initially set $\alpha = 0.5$, and then we randomly divide the 45 out-of-range queries into 5 batches each of size 9. After the execution of each batch, the system adjusts α to minimize the root-mean-square error percentage (RMSE%) of the previously executed batches. The RMSE% is computed as $(e \times 100/v)$, where e is the root-mean-square error (REMS) of a given batch, and v is the average execution time over all queries within that batch. The new value of α is then used for the cost estimation of the subsequent batch. In Table 1, we present the changes of the α values across batches along with the RMSE% for each batch. The results show a trend towards putting a higher weight on the cost factor produced from the neural network, but still the cost produced from the linear regression extrapolation contributes to the final cost by a 30% to 40%.

In summary, as Figure 14 shows, combining the two costs seems effective during the online estimation until the systems collects enough points and applies the offline tuning phase over the neural network model.

8 CONCLUSION

We presented a comprehensive cost estimation module, which is part of the *Teradata IntelliSphere* project. This work addresses a fundamental problem in the modern big data ecosystems, which is the need to efficiently access and query data across multiple heterogenous sources (remote systems). In order to generate efficient execution plans, accurate cost estimation on the remote systems is an essential building block step. We proposed three costing approaches, namely *logical-op*, *sub-op*, and *hybrid* approaches. They cover the spectrum of blackbox, openbox, and a mix of such systems. We demonstrated that none of the *logical-op* or *sub-op* approaches is superior (or practical) in all cases, and thus a hybrid approach should be deployed. We also presented the pros, cons, and applicability cases of each approach. Given the complexity of the problem, we integrated deep learning and analytical models within the proposed cost estimation module. Moreover, we proposed techniques for enhancing the estimation quality for the out-of-range (un-seen) values. As part of future work, we plan to study more types of remote systems such as SparkSQL and Impala.

REFERENCES

[1] MySQL. http://www.mysql.com.

[2] Teradata. *http://www.teradata.com.*

[3] Teradata Query Grid. *Teradata User Group, September 2014.*

[4] A unified software portfolio for a unified analytical ecosystem. Teradata intellisphere. http://www.teradata.com/products-and-services/IntelliSphere.

[5] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala. Qshuffler: Getting the query mix right. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 1415–1417. IEEE, 2008.

[6] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 390–401. IEEE, 2012.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394, 2015.

[8] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM computing surveys (CSUR)*, 18(4):323–364, 1986.

[9] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[10] P. A. Bernstein, J. Madhavan, and E. Rahm. Generic schema matching, ten years later. *PVLDB*, 4(11):695–701, 2011.

[11] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The tsimmis project: Integration of heterogenous information sources. 1994.

[12] H.-H. Do, S. Melnik, and E. Rahm. *Comparison of Schema Matching Evaluations*, pages 221–237. 2003.

[13] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. The bigdawg polystore system. *SIGMOD Rec.*, 44(2):11–16, Aug. 2015.

[14] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[15] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, Dec. 2000.

[16] A. Krogh and J. Vedelsby. Neural network ensembles, cross validation and active learning. In *Proceedings of the 7th International Conference on Neural Information Processing Systems*, NIPS'94, pages 231–238, 1994.

[17] V. Markl, G. M. Lohman, and V. Raman. Leo: An autonomic query optimizer for db2. *IBM Systems Journal*, 42(1):98–106, 2003.

[18] R. Miikkulainen. *Topology of a Neural Network*, pages 988–989. Springer US, Boston, MA, 2010.

[19] M. Mofidpoor, N. Shiri, and T. Radhakrishnan. Index-based join operations in hive. In *2013 IEEE International Conference on Big Data*, pages 26–33, 2013.

[20] S. Mullender, editor. *Distributed systems (2nd Ed.).* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[21] M. Roth, F. Özcan, and L. M. Haas. Cost models do matter: Providing cost information for diverse data sources in a federated system. In *VLDB*, 1999.

[22] J. Russell. Couldera-Impala. *O'Reilly Media*, 2013.

[23] M. Stonebraker. The design of the postgres storage system. In *VLDB*, pages 289–300, 1987.

[24] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: a wide-area distributed database system. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 5(1):048–063, 1996.

[25] A. Thusoo, R. Murthy, J. S. Sarma, Z. Shao, N. Jain, P. Chakka, S. Anthony, H. Liu, and N. Zhang. Hive - a petabyte scale data warehousing using hadoop. In *ICDE*, 2010.

[26] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[27] M. Traverso. Presto: Interacting with petabytes of data at Facebook. 2013.

[28] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, 1992.

[29] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical learning techniques for costing xml queries. In *Proceedings of the 31st international conference on Very large data bases*, pages 289–300. VLDB Endowment, 2005.

[30] Q. Zhu and P. . Larson. Building regression cost models for multidatabase systems. In *Fourth International Conference on Parallel and Distributed Information Systems*, pages 220–231, 1996.

[31] P. Ziegler and K. R. Dittrich. *Data Integration — Problems, Approaches, and Perspectives*, pages 39–58. Springer Berlin Heidelberg, 2007.

Fast Entropy Maximization for Selectivity Estimation of Conjunctive Predicates on CPUs and GPUs

Diego Havenstein
SAP SE
diego.havenstein@sap.com

Peter Lysakovski
SAP SE
peter.lysakovski@sap.com

Norman May
SAP SE
norman.may@sap.com

Guido Moerkotte
University of Mannheim
moerkotte@uni-mannheim.de

Gabriele Steidl
University of Kaiserslautern
steidl@mathematik.uni-kl.de

ABSTRACT

Entropy maximization is the only principled approach to combine several (partial) selectivity estimates to an estimate for a full conjunction. However, this approach has no appearance in database management systems. We conjecture that the main reason is a lack of implementations with good performance. Indeed, the originally proposed iterative scaling algorithm has a slow convergence rate and high complexity in each iteration. As an alternative, we propose to use a method based on Newton's algorithm to solve the entropy maximization problem. Further, we show how this general approach can be implemented very efficiently for both CPUs and GPUs. Our experiments show that our CPU and GPU implementation is more than 4 orders of magnitude faster than the state-of-the-art method for the most complex problem it could handle. For even more complex problems our new GPU implementation outperforms our CPU implementation by more than 43x. In a few milliseconds it is now possible to compute all partial selectivities for complex conjunctive predicates with 20 or more predicates. We strongly believe that the proposed implementation is ready for production-grade database management systems.

1 INTRODUCTION

Query optimizers need precise cardinality estimates to generate query execution plans of high quality. Basic approaches to estimate result cardinalities rely on the assumptions that values are uniformly distributed and the selectivities of predicates are independent. Increasingly sophisticated techniques were proposed to address the uniform distribution assumption and also correlation between predicates, see [1, 6] for comprehensive surveys. However, the space consumption and maintenance effort for all combinations of multi-column histograms [12], samples [3], or statistics on views [7] exponentially grows with the number of columns considered. For this reason, these statistics are generated only for a few out of all possible column subsets. We address the challenge how to integrate estimates produced from these sources of statistics consistently. Note that in general sampling alone is not sufficient because it can result in highly imprecise estimates, and thus other synopsis have to be used [11, 15].

Markl et al. [9] observed that the query optimizer makes suboptimal plan choices despite the rich statistics at hand to find the optimal plan because *fleeing to ignorance* seems to be the most reasonable choice. They suggested the maximum entropy method to exploit all available knowledge and to handle inconsistent and missing information in a consistent way. Consider for example the following scenario. Assume we have three predicates p_0, p_1, p_2 whose selectivities are estimated to be $s_0 = 0.5$, $s_1 = 0.5$, and $s_2 = 0.5$. Further assume that the combined selectivity for $p_0 \wedge p_1$ is $s_{01} = 0.4$ and for $p_1 \wedge p_2$ is $s_{12} = 0.1$. These selectivities could be estimates produced from single column histograms, 2-dimensional histograms, and/or sampling. The question is what is the selectivity of the whole conjunct $p_0 \wedge p_1 \wedge p_2$? The answer given by entropy maximization (as proposed by Markl et al. [9]) is 0.08, which clearly deviates from the estimate $0.5 * 0.5 * 0.5 = 0.125$ produced under the independence assumption. Clearly, the estimate produced under the independence assumption is inconsistent since it is larger than the selectivity of $p_1 \wedge p_2$. Indeed, it is widely known that the independence assumption (1) does not hold in general and (2) leads to bad cardinality estimates and, consequently, (3) leads to suboptimal query execution plans [8].

In order to derive the missing selectivity values, Markl et al. propose to find the unique vector $x = (x_0, x_1, \dots x_{2^z-1})$ (for z predicates) that maximizes the entropy

$$H(s) = \sum_i -x_i \log x_i$$

subject to the constraints given by the known selectivities. A formal definition requires some preliminaries and will be given in Sec. 2. Maximizing entropy can be seen as a generalization of the independence assumption limited to the case of unknown selectivities. Since the known selectivities are possibly derived from several synopsis, the problem may become inconsistent. In this case, a *corrector step* is necessary. Since different correctors have been proposed in the literature (e.g., [9, 10]), we assume in this paper that the problem on hand is consistent.

To solve the entropy maximization problem, Markl et al. use *iterative scaling*. However, this algorithm is known to have very slow convergence [2, p82] and, additionally, has a relatively high asymptotic complexity of $O(m^2 * n)$ in each iteration, where m is the number of known selectivities, z the number of predicates and $n = 2^z$. For example, for eight predicates, iterative scaling needs on average 260 iterations and 115 ms whereas a Newton-based algorithm needs 10 iterations and 0.14 ms on a system with an Intel i7-4790 CPU. For scenarios with even more predicates iterative scaling quickly becomes too slow to be practical while our new Newton-based algorithm on the CPU and even more so on the GPU are able to calculate a solution in a few milliseconds. Hence, with our method we can avoid strategies like partitioning the set of predicates to reduce the problem size that can be found in real-world scenarios [9].

Notation	Description
$p_0, \ldots, p_{z-1}$	z predicates
$N = \{0, \ldots, z-1\}$	set of all predicate indices
$n = 2^z$	abbreviation
$T \subseteq 2^N$	set of indices of known selectivities
$m = \|T\|$	number of known selectivities
β_T	vector of known selectivities
C	complete design matrix
D	(partial) design matrix
$s(p)$	selectivity of predicate p
Bit-wise operations	**Description**
$\|$	bit-wise or
&	bit-wise and
˜	bit-wise complement
$i \subseteq j$	boolean function returning $j = (i\|j)$

Table 1: Notation

In this paper, we propose to use a Newton-based algorithm to solve the entropy maximization problem. We formalize the problem as a series of vector- and matrix operations. However, a naive implementation of these operations fails to achieve the performance requirements for this method. Hence, we discuss in depth how to efficiently implement this algorithm and show that it is vastly superior to both iterative scaling and the naive implementation of the Newton method. We elaborate on the efficient implementation for both the CPU and the GPU.

The rest of the paper is organized as follows. Section 2 formally introduces the problem as a series of vector- and matrix operations. Section 3 describes and evaluates the (almost) straightforward implementation and the optimized implementation of Newton's algorithm for the CPU. Section 4 describes and evaluates our GPU implementation of the algorithm. Section 5 reviews how entropy maximization can be integrated into query optimizers and concludes the paper.

2 PROBLEM FORMALIZATION

In this section, we present an elegant way to formalize the maximum entropy method for selectivity estimation. This is a necessity since standard entropy maximization algorithms require a matrix-based representation of the problem, which is not yet readily available. Only this matrix-based representation of the problem will allow us to derive an efficient algorithms.

2.1 Design Matrix

Since we need a matrix representation of the problem, we need to heavily deviate from the notation of Markl et al. [9]. However, in our opinion, the resulting representation is much more elegant. From the notation of Markl et al. [9], we only keep the letter T to denote the indices of the known selectivities. For convenience, the most important parts of the notation are summarized in Table 1. The lower part contains the notation for bit-wise operations, which will be required for our efficient implementations.

2.1.1 Conjunctions of (Simple) Predicates (β). Consider a conjunctive query

$$p_0 \wedge \ldots \wedge p_{z-1}$$

of z predicates. These may be selection predicates or join predicates [9].

Let $N = \{0, \ldots, z-1\}$ be the set of numbers from 0 to $z-1$. Then, all subsets $X \subseteq N$ can be represented as a bit-vector of length z denoted by bv(X) where the set bits indicate the indexes of those elements of N which are also included in the subset X. Further, this bit-vector can be interpreted as a binary number. We make no distinction between the bit-vector and the integer it represents and use whatever is more convenient. For example, we use the notation $i \subseteq j$ to denote the fact that i has a '1' only in those positions where j has a '1', i.e., $j = i|j$ holds.

For any $X \subseteq N$ define the formula

$$\beta(X) := \wedge_{i \in X}\, p_i$$

i.e., $\beta(X)$ is the conjunction of all predicates p_i whose index i is contained in X. The following table gives a complete overview for $z = 3$, where we order bits from least significant to most significant:

bv(X)	$\beta(X)$
1=100	p_0
2=010	p_1
3=110	$p_0 \wedge p_1$
4=001	p_2
5=101	$p_0 \wedge p_2$
6=011	$p_1 \wedge p_2$
7=111	$p_0 \wedge p_1 \wedge p_2$

where the first column gives the integer value and its bit-vector representation of the index set X and the second column the corresponding conjunction of predicates contained in X. We use $\beta(i)$ instead of $\beta(X)$ if i is the bit-vector/integer representation of some X.

The selectivity of $\beta(X)$, i.e., the probability of $\beta(X)$ being true is denoted by $\beta(X)$. A special case occurs for the empty set. The empty conjunct is always true. Thus $\beta(\emptyset) = \beta(0) = 1$.

2.1.2 Complete Conjuncts (γ). A conjunction of literals containing all predicates either positively or negatively is called *complete conjunct* (*atom* by Markl et al., also *minterm*). For $n = 3$, the following table contains a list of all complete conjuncts:

i	$\gamma(i)$				
0=000	$\neg p_0$	$\wedge$	$\neg p_1$	$\wedge$	$\neg p_2$
1=100	p_0	$\wedge$	$\neg p_1$	$\wedge$	$\neg p_2$
2=010	$\neg p_0$	$\wedge$	p_1	$\wedge$	$\neg p_2$
3=110	p_0	$\wedge$	p_1	$\wedge$	$\neg p_2$
4=001	$\neg p_0$	$\wedge$	$\neg p_1$	$\wedge$	p_2
5=101	p_0	$\wedge$	$\neg p_1$	$\wedge$	p_2
6=011	$\neg p_0$	$\wedge$	p_1	$\wedge$	p_2
7=111	p_0	$\wedge$	p_1	$\wedge$	p_2

Note that two different complete conjuncts can never be true simultaneously. The complete conjuncts have been indexed by their bit-vector representation, where a positive atom corresponds to '1' and a negative atom corresponds to '0'. For a given $X \subseteq N$, denote by $\gamma(X)$ the complete conjunct X:

$$\gamma(X) := \bigwedge_{i \in X} p_i \;\wedge\; \bigwedge_{i \notin X} \neg p_i$$

The probability of a complete conjunct $\gamma(X)$ for some X being true is denoted by $\gamma(X)$.

2.1.3 Correspondence between β and γ. For a given $X \subseteq N$, the bit-vectors y of the complete conjuncts $\gamma(Z)$ contributing to $\beta(X)$ can be expressed as all the bit-vectors y which contain a '1' at least at those positions where the bit-vector representation bv(X) of X contains a '1'. That is

$$\{y | y \supseteq \text{bv}(X)\}.$$

Consider $X = \{0\}$ ($\hat{=}100$). Then

$$\begin{array}{rcllllll} \beta(X) & = & s(& p_0 & \wedge & \neg p_1 & \wedge & \neg p_2 &)+ \\ & & s(& p_0 & \wedge & p_1 & \wedge & \neg p_2 &)+ \\ & & s(& p_0 & \wedge & \neg p_1 & \wedge & p_2 &)+ \\ & & s(& p_0 & \wedge & p_1 & \wedge & p_2 &), \end{array}$$

where $s(p)$ denotes the selectivity of the complete conjunct p. For $X = \{0, 1\}$ ($\hat{=}110$):

$$\beta(X) = s(p_0 \wedge p_1 \wedge \neg p_2) + s(p_0 \wedge p_1 \wedge p_2).$$

As a special case, we get for $X = \emptyset$ ($\hat{=}000$) that all complete conjuncts contribute to $\beta(\emptyset)$. Further, the sum of them must be one. Consequently, we always assume that the empty set is contained in the set of known selectivities T, i.e., $\emptyset \in T$.

2.1.4 Complete Design Matrix C. In case $T = 2^N$, all selectivities are known. Define $n = 2^z$. Then, we define the *complete design matrix* $A \in \mathbb{R}^{n,n}$ as

$$C = (c_{i,j}) = \begin{cases} 1 & \text{if } i \subseteq j \\ 0 & \text{else} \end{cases}$$

where we use indices in $[0, 2^z - 1]$. Note that C is unit upper triangular, nonsingular, positive definite, and persymmetric.

For $z = 3$, we have

$$C = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

This design matrix helps us to go from probabilities for complete conjuncts to selectivities for positive conjuncts. Let $b = (\beta(0), ..., \beta(n-1))^t$ the column vector containing all the selectivities $\beta(X)$ for all $X \in 2^N$ and $x = (\gamma(0), ..., \gamma(n-1))^t$ the column vector containing all the selectivities for all complete conjuncts. Then,

$$Cx = b$$

holds.

2.2 The (Partial) Design Matrix D

We first establish some notation to eliminate rows and columns in some matrix A. Let $A \in \mathbb{R}^{n,n}$ be some matrix. Let $T \subseteq \{0, \ldots, n-1\}$, $m := |T|$, be a set of column indices. Then, we denote by $A|_{c(T)}$ the matrix where only the columns in T are retained. Likewise, we denote by $A|_{r(T)}$ the matrix derived by retaining only the rows in T. These operations can be expressed via matrix multiplication. For an index set T with $m = |T|$, we define the matrix $E^{m,n,T} \in \mathbb{R}^{m,n}$ as

$$E_{m,n,T}(i, j) = \begin{cases} 1 & \text{if } j = T[i] \\ 0 & \text{else} \end{cases}$$

where $T[i]$ denotes the i-th element of the sorted index set T. For example, for $m = 4$, $n = 8$, $T = \{1, 3, 5, 7\}$, we get

$$E_{4,8,T} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

```
NewtonA($b(= \beta_T), T, \epsilon$)
1  $w = 0$
2  $x = \exp(-1)$
3  while ($\delta > \epsilon$)
4      $A = D\text{diag}(x)D^t$
5      solve $Ay = b - Dx$
6      $w = w + y$
7      $x = \exp(D^t w - 1)$
8      $\delta = ||b - Dx||$
9  return ($x$, $Cx$)
```

Figure 1: Newton Variant A [2, p73]

Then, for $A \in R^{n,n}$

$$\begin{array}{rcl} A|_{r(T)} & = & E_{m,n,T}A \\ A|_{c(T)} & = & A(E_{m,n,T})^t \end{array}$$

holds. For a given subset $T \subseteq \{0, \ldots, n-1\}$ (of known selectivities), we retain only those rows from the complete design matrix C for which there is an entry in T. We define the problem specific *(partial) design matrix* D for T as

$$D := C|_{r(T)} = E_{m,n,T}C \in \mathbb{R}^{m,n} \tag{1}$$

where $m := |T|$. Clearly, the rank of D is m.

2.3 Problem Definition

For z predicates, a given vector β_T of known selectivities and indices T thereof, the idea of Markl et al. is to find the solution to $Dx = \beta_T$ that maximizes the entropy of the solution vector x [9]. That is, the problem to solve can be specified as

$$\underset{x}{\text{argmax}} \sum_{i=0}^{n-1} -x_i \log x_i \text{ subject to } Dx = \beta_T \text{ and } x \geq 0 \tag{2}$$

where $n = 2^z$. Note that, we must have that $\sum_{i=1}^{n-1} x_i = 1$, but this is implied since we assume that $\emptyset \in T$ always holds.

3 EFFICIENT CPU IMPLEMENTATION

In this section, we first discuss an implementation of Newton's algorithm to solve the entropy maximization problem that is directly derived from [2]. Due to the matrix-based formalization of our problem, the algorithm is readily applicable and we call this Variant A, and it represents the state-of-the-art implementation of Newton's algorithm. This algorithm it's rather inefficient since its steps require multiplications of large vectors and matrices. We improve this by devising a method for how these matrix and vector operations can be computed very efficiently. This leads us to Variant B of Newton's algorithm. Finally, we evaluate the runtime of both variants on an Intel CPU and compare it with the iterative scaling which was used by Markl et al.

3.1 Newton Variant A

Markl et al. propose to use iterative scaling to solve the optimization problem in Eqn. 2 [9]. However, it is well-known that iterative scaling converges very slowly [2, p82]. In contrast, a Newton-based approach exhibits local quadratic convergence [2, p73]. We thus selected a Newton-based algorithm applied to the dual problem of Eqn. 2:

$$\underset{w}{\text{argmin}} \exp(D^t w - 1)^t \vec{1} - \beta_T^t w \tag{3}$$

as the basis of our work, where we suppose that the set $\{x \in \mathbb{R}^n : Dx = \beta_T,\ x \geq 0\}$ has a nonempty interior (see also [2, p55]).

Fig. 1 shows the code of a Newton-based algorithm to solve the maximum entropy problem defined in Sec. 2.3. As input, it receives the vectors b and T of known selectivities and their indices, and some $\epsilon > 0$ used in the stop criterion. It returns the solution x maximizing the entropy and the vector Cx containing the β-selectivities for all possible conjuncts. Although T does not occur in the body of Fig. 1, it is used in the definition of the design matrix D (see Eqn. 1).

The steps in the algorithm differ vastly in complexity. The initializations of w and x have complexity $O(n)$ and $O(m)$, respectively, and are thus rather uncritical. The calculation of $w = w + z$ in Line 6 has complexity $O(m)$ and is thus rather uncritical, too.

The calculation of $A = D\text{diag}(x)D^t$ in Line 4 can be very expensive if implemented literally. Note that diag(x) is a diagonal $(n \times n)$-matrix with x on its diagonal. Using standard matrix multiplication, the complexity of this step is $O(m{*}n^2{+}m^2{*}n)$. However diag(x) contains only zero's besides the diagonal and thus a more efficient procedure which does not rely on materializing diag(x) can be devised:

```
GET_DDIAGXDT($D, x$)
1  for ($0 \leq i < m, 0 \leq j < m$)
2      $s = 0$
3      for ($0 \leq k < n$)
4          $s$ += $D[i,k] * x[k] * D[j,k]$
5      $A(i,j) = s$
6  return $A$
```

This procedure has complexity $O(m^2 * n)$ and is thus far better than the naive approach using matrix multiplication.

In step (5), we need to solve $Ay = b - Dx$ for y. Calculating Dx has complexity $O(m * n)$. To solve the equation, note that the (m, m) matrix $A = D\text{diag}(x)D'$ calculated in step (2) is symmetric, non-singular, and positive definite. Thus, the efficient Cholesky decomposition [5, p237] can be applied to derive a lower triangular matrix L with $A = LL^t$. Then, we derive the solution y using back substitution [4, p89]. The complexity of this procedure is $O(m^3)$.

In step (7), we need to calculate $D^t w$, which has complexity $O(m * n)$. Step (8) with complexity $O(m)$ is uncritical again, as Dx has been calculated in step (5) already.

In step (9), we need to calculate the product of the complete design matrix C with the primal solution vector x. Using standard matrix multiplication this step has complexity $O(m * n)$.

The complexities of the steps become visible when profiling Newton Variant A for $z = 8 \ldots 10$: roughly 80% of the runtime is spent in procedure GET_DDIAXDT.

3.2 Newton Variant B

As we will see in below, a careful analysis of the structure of the complete design matrix C allows us to derive a reduction-based algorithm that avoids redundant computations resulting in an algorithm for Newton's method with lower computational complexity than the state-of-the-art algorithm from Sec 3.1.

3.2.1 Recursive Characterization of C. The complete design matrix C can also be defined recursively. Denote by $C_z \in \mathbb{R}^{n \times n}$ with $n = 2^z$ the complete design matrix for z predicates. Then

$$C_0 = (1)$$

and

$$C_{z+1} = \begin{bmatrix} C_z & C_z \\ 0 & C_z \end{bmatrix}$$

characterize the complete design matrix C. Another possibility to define C is to use the Kronecker product $\otimes$ [5, p337]. With

$$C_1 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

we have

$$C_{z+1} = C_1 \otimes C_z$$

3.2.2 Efficient Calculation of Cx and $C^t x$. Let us turn to calculating Cx for some vector $x \in \mathbb{R}^n$, which we need to do efficiently for our Newton-based algorithm. If we cut $x \in \mathbb{R}^n$ into two halves $x_1, x_2 \in \mathbb{R}^{n/2}$, we observe that

$$C_z x = \begin{pmatrix} C_{z-1} & C_{z-1} \\ 0 & C_{z-1} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} C_{z-1}x_1 + C_{z-1}x_2 \\ C_{z-1}x_2 \end{pmatrix} \qquad (4)$$

The term $C_{z-1}x_2$ occurs twice but has to be calculated only once. Based on this observation, it is easy to implement a recursive procedure calculating $C_z x$ in $O(z2^z)$, i.e. $O(nlogn)$ substituting $n = 2^z$. As a major contribution of this paper, we are now able to reduce the algorithmic complexity of the newton method from $O(n^2)$ down to $O(nlogn)$.

In order to avoid the overhead of recursion, we provide an efficient iterative algorithm. We assume that the in/out argument `Cx` has been initialized with x. Further, `vp_add` is an AVX2-based implementation to add two vectors of length h.

```
void get_Cx(double* Cx, uint z)
1  w = h = s = t = 0;
2  n = 1 << z;
3  for (w = 2; w <= n; w <<= 1) // width
4      for (s = 0; s < n; s+ = w) // start of first half
5          h = (w >> 1); // half of width
6          t = s + h; // start of second half
7          vp_add(Cx + s, Cx + t, h);
```

A procedure to efficiently calculate $C^t y$ can be devised similarly by replacing `Cx` by `Ctx` and `vp_add(Cx + s, Cx + t, h)` by `vp_add(Ctx + t, Ctx + s, h)`. We call this algorithm `get_Ctx` to w'.

3.2.3 Efficient Calculation of Dx and $D^t x$. First remember that for $n = 2^z$, z being the number of predicates, (1) the complete design matrix C is of dimension (n, n) and (2) the design matrix D is of dimension (m, n). where in typical applications m will be much smaller than $n = 2^z$.

As we have seen in Sec. 3.2.2, calculating Cx in Line 9 can be implemented very efficiently. By exploiting the definition of D in Eqn. 1, we can evaluate $Dx = E_{m,n,T}Cx$ efficiently by first calculating Cx and then picking the components contained in T. This has to be done only once to calculate the expressions Dx in Lines 5 and 8, and Cx in Line 9. Further, $C^t x$ can be calculated efficiently using algorithm `get_Ctx`. Thus, calculating $D^t w$ in step (7) can be implemented efficiently by exploiting the fact that $D^t = C^t E^T_{m,n,T}$. We can embed w into a vector w' in $\mathbb{R}^n$ via

$$w'[j] = \begin{cases} w[i] & \text{if } j = T[i] \text{ for some } i \\ 0 & \text{else} \end{cases}$$

$(0 \leq i < m, 0 \leq j < n)$ and apply algorithm `get_Ctx`.

3.2.4 Efficient calculation of $D\mathrm{diag}(x)D^t$. Next, we discuss an efficient implementation of step (4). As we have already calculated Cx, we now show that it is possible to calculate $(D\mathrm{diag}(x)D^t)$ from Cx. We start with an efficient algorithm to calculate

$$(C\mathrm{diag}(v)C^t).$$

Observe that $(\mathrm{diag}(v)C^t) = (C\mathrm{diag}(v))^t$. Further,

$$(C\mathrm{diag}(x))[j,k] = \sum_{l=0}^{n-1} c_{j,l}\mathrm{diag}(x)[l,k] = c_{j,k}x_k$$

Thus, using

$$\begin{aligned}(C\mathrm{diag}(x)C^t)[i,j] &= \sum_{k=0}^{n-1} c_{i,k}(C\mathrm{diag}(x))^t[k,j] \\ &= \sum_{k=0}^{n-1} c_{i,k}(C\mathrm{diag}(x))[j,k] \\ &= \sum_{k=0}^{n-1} c_{i,k}c_{j,k}x_k \\ &= \sum_{(i|j)\subseteq k} x_k \\ &= (Cx)[i|j]\end{aligned}$$

we can calculate $(C\mathrm{diag}(x)C^t)$ from Cx. Since

$$\begin{aligned}D\mathrm{diag}(x)D^t &= (E_{m,n,T}C)\mathrm{diag}(x)(E_{m,n,T}C)^t \\ &= E_{m,n,T}(C\mathrm{diag}(x)C^t)E^t_{m,n,T}\end{aligned}$$

we can use Cx to fill $(D\mathrm{diag}(x)D^t) \in \mathbb{R}^{m,m}$ via

$$(D\mathrm{diag}(x)D^t)[i,j] = (Cx)[T[i] \mid T[j]] \tag{5}$$

for $0 \le i, j < m$.

3.3 Evaluation

In order to evaluate the implementations of the two variants of Newton's algorithm and the iterative scaling used by Markl et al., we need to generate entropy maximization problems. Since generation of β selectivities easily leads to inconsistencies, we generate a random vector x of size n containing positive integers interpreted as cardinalities for all complete conjuncts γ. Dividing each x_i by $\sum_i x_i$ results in γ-selectivities. Calculating $b = Cx$ results in a complete set of consistent β-selectivities. From these, we select the subset T of known selectivities by extracting selectivities for single predicates and conjunctions of two or three predicates. In practice, not all pairs or triples will be available. Thus, the runtimes reported in the experiments below can be seen as loose upper bounds on the runtime in practice.

We use the stopping criterion $||b/Dx||_q \le \epsilon$ where b/Dx denotes component-wise division,

$$||y||_q := \max_i(\max(y_i, 1/y_i)),$$

and $\epsilon = 1 + 10^{-8}$.

We implemented iterative scaling and the two variants of our Newton-based algorithm in C++ and compiled them with g++ version 7.2.1 with option -O3. The experiments where run on a system with an Intel i7-4790 CPU. Note that this CPU with Haswell architecture had a better single-thread performance than a newer server CPU with Skylake architecture. We report the average execution time of 777 generated problems for each number z of predicates. Our implementation runs in single-threaded mode.

Figures 2 and 3 show the average runtime of our CPU implementation versus the runtime of iterative scaling (as proposed by Markl et al. [9]) if the set of known selectivities T contains all unary and additionally all binary or ternary conjuncts. Column z contains the number of predicates considered and column m contains the number of known selectivities. Besides the average runtime in milliseconds, we include the average number of iterations.

As one can see, our Newton-based implementation is much more efficient than the originally proposed iterative scaling algorithm. For ten conjuncts the runtime of iterative scaling already exceeds one second. Further, as expected, Newton Variant B is much more efficient than Newton Variant A.

		Newton Var. A	Newton Var. B		Iterative Scaling	
z	m	runtime [ms]		#itr	runtime [ms]	#itr
3	7	0.009	0.004	7.3	0.14	190
4	11	0.017	0.008	7.8	0.47	190
5	16	0.061	0.027	8.1	2.1	200
6	22	0.23	0.048	9	9.4	210
7	29	0.84	0.075	9.1	34	240
8	37	2.9	0.14	10	120	260
9	46	10	0.25	11	370	280
10	56	29	0.41	11	1100	310
11	67	98	0.73	12	—	—
12	79	310	1.4	13	—	—
13	92	1000	2.7	13	—	—
14	106	3300	5.3	14	—	—
15	121	11000	11	15	—	—
16	137	—	23	15	—	—
17	154	—	48	16	—	—
18	172	—	100	17	—	—
19	191	—	200	17	—	—
20	211	—	480	18	—	—

(Intel i7-4790, single-threaded, $T = \{t|\mathrm{popcnt}(t) \le 2\}$)

Figure 2: Newton vs. Iterative Scaling

		Newton Var. A	Newton Var. B		Iterative Scaling	
z	m	runtime [ms]		#itr	runtime [ms]	#itr
4	15	0.04	0.02	8.7	3	890
5	26	0.15	0.05	9	16	910
6	42	0.79	0.13	9.3	79	1000
7	64	3.9	0.33	10	360	1200
8	93	16	0.76	10	1600	1400
9	130	65	1.8	11	6700	1580
10	176	230	4.1	11	26000	1800
11	232	890	9.1	12	—	—
12	299	3400	20	13	—	—
13	378	11000	40	13	—	—
14	470	38000	80	14	—	—
15	576	120000	150	15	—	—
16	697	—	270	15	—	—
17	834	—	480	16	—	—
18	988	—	880	17	—	—
19	1160	—	1400	17	—	—
20	1351	—	2600	18	—	—

(Intel i7-4790, single-threaded, $T = \{t|\mathrm{popcnt}(t) \le 3\}$)

Figure 3: Newton vs. Iterative Scaling

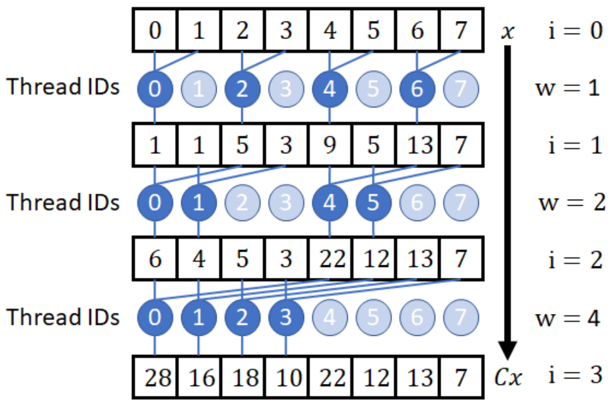


Figure 4: Scheme of the efficient GPU implementation of Cx **for an initial** $x = 0, 1, \ldots, 7$

Figure 2 shows that for up to ten predicates, the runtime to calculate all selectivities needed by the query optimizer is below 0.5 milliseconds if Newton Variant B is used, i.e. three orders of magnitude faster than iterative scaling. However, somewhere between 11 and 20 predicates, depending on the context (e.g., ad hoc queries vs. repeated execution), even the runtimes of our optimized Newton Variant B implementation becomes too high. In particular, Figure 2 indicates that the Newton Variant B with 13 predicates exceeds one second of runtime, while our new method B finishes even 20 conjuncts in less than a second.

In general the runtimes are higher when the problems contain all unary, binary or ternary conjuncts. As can be seen in Figure 3, iterative scaling needs almost 26 seconds for 10 predicates while our new Newton method with Variant B calculates these problems in about 4 milliseconds, i.e. more than 4 orders of magnitude faster. The naive implementation of Newton's method in Variant A needs more then ten seconds runtime for 15 predicates while Variant B is nearly 4 orders of magnitude faster for the same problems.

The increasing runtimes as we consider more and more complex predicates motivated us to pursue a GPU implementation.

4 EFFICIENT GPU IMPLEMENTATION

In this section we describe how the Newton algorithm can be implemented efficiently on a modern GPU. We first explain the multi-threaded GPU implementation of Variant B presented in Sec. 3.2. After that we present experimental results of our implementation using CUDA 10.0 on an NVIDIA Tesla V100 GPU.

4.1 Newton Variant B on the GPU

We discuss how Variant B of Newton's method can be implemented on an NVIDIA GPU. We focus our presentation on the implementation of Cx because, as we have seen in Sec. 3.2, this operation is at the heart of the implementation of steps (4), (7), (8) and (9) of the Newton algorithm presented in Figure 1. We also point out how the remaining step (5), the Cholesky decomposition, is implemented efficiently on the GPU. Finally we outline how we organize our code in kernels of the end-to-end implementation.

4.1.1 GPU Implementation of Cx and $C^t x$. As the NVIDIA V100 GPU used in our experiments offers an abundance of 5120 CUDA cores, we need to extend the implementation of `get_Cx` presented in Sec. 3.2.2 to support massive multi-threading. Figure 4 illustrates the parallelization scheme we use in our implementation. Here, the required operations for calculating Cx are shown for $x = \{0, 1, 2, 3, 4, 5, 6, 7\}$ and $z = 3$. Boxes represent the contents of x after each iteration i, and dark (light) blue circles represent active (inactive) CUDA threads. In each iteration, every active thread performs one addition and stores the result. The connecting blue lines indicate the flow of data. In every iteration half of the GPU threads are active while the other half is idle. While this may seem wasteful, it allows us to use a simple mapping from thread-id to accessed memory addresses. A more effective use of the GPU threads would require a more complex mapping. In fact, we did not find an efficient way to map thread-ids to memory addresses while keeping all threads active all the time. As the maximum number of threads per thread block for the Tesla V100 is 1024, the first ten iterations of our scheme can be performed without requiring communication between different thread blocks. During these ten iterations we make use of the GPUs shared memory, and access to global memory is only required once when loading x into shared memory and once when writing Cx back to global memory. This is beneficial because compared to global memory, shared memory on the NVIDIA V100 GPU offers lower latency and significantly higher bandwidth. Hence, for $z \leq 10$ we use the kernel using shared memory shown in Listing 1. In every iteration of the outer loop we advance with processing vector x by the number of available threads. Note, that for z predicates we have $n = 2^z$ elements to process, i.e. for $z = 15$ we have $2^{15} = 32768$ elements to process. The inner loop in Listing 1 adds the elements as illustrated in Figure 4.

For $z > 10$, no efficient shared memory implementation is possible as threads of one thread block would need to access shared memory allocated in another thread block. This is not possible, and as a consequence all memory accesses have to go to global memory. This requires global synchronization through individual kernel launches. We call this global kernel to compute Cx once for every $z > 10$. It is shown in Listing 2. In our implementation we use templates to generate these calls at compile time. The parameter `direction` allows us to not only calculate Cx but also to calculate $C^t x$. When `direction` is set to 1, the algorithm proceeds backwards, giving us $C^t x$. This is needed in step (7) of the Newton algorithm where we use the product $D^t w$.

Recall that steps (4), (7), (8) and (9) in the Newton algorithm shown in Figure 1 build upon or use the calculation of Cx. This is why we do not describe the implementation of these steps in detail here. The basic ideas are similar to the ones presented for the computation of Cx.

4.1.2 Cholesky Solver. As for the CPU implementation presented in Sec 3.1, solving $Ay = b - Dx$ for y in step (5) of the Newton algorithm shown in Figure 1 can be done using Cholesky decomposition [5, p237]. Fortunately, we can use the cuSolver library from the CUDA toolkit [16] for large problems, i.e. for $m \geq 40$. First, we rely on `cusolverDnDpotrf` to factorize A in a kernel call. Then, we call the kernel `cusolverDnDpotrs` where we pass $b - Dx$ as argument and get y as result of step (5).

As multiple kernel calls are involved in these steps, and each kernel call implies a call overhead of approximately $5 - 10\mu s$, we also implement a variant of the Cholesky decomposition using only a single kernel call. We use this kernel as a solver for small problems, i.e. $m < 40$. The implementation is based on [13] and calculates the solution of the system of equations via

Listing 1: Kernel to compute Cx in shared memory

```
template <int BLOCK_SIZE_X>
__global__ void getCxShared(double* __restrict__ const x, const unsigned int z,
                            const bool direction=0) {
    unsigned int stride = blockDim.x * gridDim.x;
    __shared__ double xShared[BLOCK_SIZE_X];
    unsigned int end = (1<<z);
    for(int globalIdx = threadIdx.x+blockDim.x*blockIdx.x; globalIdx < end; globalIdx+=stride) {
        xShared[threadIdx.x] = x[globalIdx];
        __syncthreads();
        for(int w = 1; w < BLOCK_SIZE_X; w<<=1) {
            if((threadIdx.x/w)%2 == direction) {
                xShared[threadIdx.x]+=xShared[threadIdx.x+w-direction*2*w];
            }
            __syncthreads();
        }
        x[globalIdx] = xShared[threadIdx.x];
    }
}
```

Listing 2: Kernel to compute Cx in global memory

```
template <unsigned int iteration>
__global__ void getCxGlobal(double* __restrict__ const x, const bool direction=0) {
    static constexpr auto offset = 1U << iteration;
    const int myGlobalIdx = threadIdx.x+blockDim.x*blockIdx.x;
    const int blockOffset = (blockIdx.x*1024/offset)*2*offset;
    const int myElementIdx = offset*direction + blockOffset + myGlobalIdx%offset;
    x[myElementIdx]+=x[myElementIdx+offset -2*direction*offset];
}
```

Gaussian elimination without pivoting. It is implemented to run in a single thread block using shared memory. In our experiments, this reduced the end-to-end runtimes of the Newton algorithm by $0.2 - 0.4ms$. However, as the CPU implementation is still faster than the GPU for such small problems this alternative is not really needed.

4.1.3 End-To-End GPU Implementation. We now describe how the various kernels are combined to implement Newton's algorithm on the GPU. In Figure 5, we can only present pseudo code as all the GPU code taken together is several hundred lines long. The initialization in steps (1) - (3) and the main loop are realized in function `NewtonB_GPU`.

While the logic of the loop is the same as in Figure 1 for the CPU code we organize the code to minimize the number of kernel calls. For example, in step (5) we compute both $D\text{diag}(x)D^t$ and also $b - Dx$ in a single kernel call to `buildMatrixA`. In this kernel we first compute Cx calling `getCxShared` and then, if $z > 10$, we call `getCxGlobal` in a loop for every $10 < w \leq z$. In the second step of kernel `buildMatrixA`, we gather from Cx the elements for Dx and $A = D\text{diag}(x)D^t$ as explained in Sec 3.2.3 and Eqn 5 in Sec 3.2.4. In Sec 4.1.2 we explain how we implement step (6) of the loop in function `NewtonB_GPU`, i.e. using the cuSolver library of CUDA for larger problems. Step (7) computes $w = w - y$ using thrust::transform from Thrust, the CUDA C++ template library [16]. Then, step (8) fuses steps (7) and the computation of $b - Dx$ in step (8) of the CPU-based code from Figure 1 into a single kernel `productOfDtw`. This kernel first distributes vector w into x, and then `productOfDtw` uses the logic of `get_Cx_GPU` to compute $D^t w$ using direction = 1 as parameter to handle the transposed matrix; see Sec 3.2.3. As part of this computation we can also calculate the vectors u_{old}, u_{new} and x in the same kernel. Notice, that after the call to `productOfDtw` the vector u_{old} contains the element-wise delta of the last loop iteration. We use this vector in step (9) to determine δ to check for convergence of the algorithm. In our GPU implementation we use the L_∞ norm and $\epsilon = 10^{-8}$. Because of the local quadratic convergence of the Newton algorithm we found that the norm used to check for convergence had virtually no impact on the convergence of the algorithm. If convergence is reached, we return the solution of the Newton algorithm in step (11) by doing one final call to get_Cx_GPU(x,0).

4.2 Evaluation

To evaluate the performance of the GPU-based implementation of the Newton algorithm presented in Sec 4.1, we generated the same entropy maximization problems as in Sec 3.3. We compiled the Newton algorithm using gcc 7.3.1 for the host code and CUDA 10.0 for the kernels on the GPU and compiled them with g++ -O3

The experiments where run on a system with an Intel Xeon E7-8890v3, i.e., using a CPU from the same hardware generation as we used for the evaluation of the CPU-based implementation in Sec 3.3. The system was equipped with a PCI-attached NVIDIA Tesla V100 GPU with 16GB of HBM2 memory. We report the average execution time of the generated problems for different numbers of predicates, z. During the experiments, the host code on the CPU was running in a single thread; virtually all computation was done on the GPU. We remark that the runtimes for the CPU implementations reported in Sec 3.3 used a single thread on the host. The GPU implementation we used here performs busy waiting on the host. With CUDA 10.1 the graph feature

```
GET_CX_GPU($x$, $direction$)
1  $y$ = getCxShared(x,direction)
2  for $w = 1$ to $z - 10$
3      $Cx$ = getCxGlobal<10 + w>($y$,direction)
4  return $Cx$

BUILDMATRIXA($b, x$)
1  $Cx$ = get_Cx_GPU(x,0)
2  (A, $Dx$) = distribute $Cx$ to $A$ and $Dx$ using Sec 3.2.3 and Sec 3.2.4
3  return (A, $Dx$)

PRODUCTOFDTW($w$)
1  $D^t w = 0$
2  distribute $w$ into $x$
3  $D^t w$ = get_Cx_GPU(x,1)
4  together with get_Cx_GPU(x,1), in the same kernel also compute
5      $x = \exp(-D^t w)$
6      $u_{new} = x/\exp(1)$
7      $u_{old} = u_{old} - u_{new}$
8  return ($D^t w, u_{old}, u_{new}, x$)

NEWTONB_GPU($b(= \beta_T), T, \epsilon$)
 1  $w = 0$
 2  $b = b * \exp(1)$
 3  $x = 1$
 4  while ($\delta > \epsilon$)
 5      (A, $Dx$) = buildMatrixA(b, x)
 6      solve $Ay = b - Dx$ for $y$ using cuSolver
 7      $w = w - y$
 8      ($D^t w, u_{old}, u_{new}, x$) = productOfDtw(w)
 9      $\delta = ||u_{old}||_\infty$
10      swap($u_{old}, u_{new}$)
11  return (get_Cx_GPU(x,0))
```

Figure 5: GPU version of Newton Variant B

became available which allows to model the graph of kernels and reduce the call overheads for the kernels. Furthermore, the graph recapture feature introduce with CUDA 10.2. supports passing parameters to these graphs further reducing the call overheads of the GPU. With this the GPU implementation may become faster for smaller problems, but an initial overhead to create and instantiate the graph of about 0.4ms would remain. For larger problems these overheads become insignificant.

In Figure 6 we present the runtime for configurations with different complexity. As in Sec 3.3 column z contains the number of predicates considered and column m contains the number of known selectivities.

The runtimes in the third column in Figure 6 are reported for problems where the set of known selectivities T contains all unary and additionally all binary conjuncts. In this setup, the GPU is faster than the fastest CPU implementation for 13 or more predicates. For 20 predicates the runtime of the fastest CPU implementation was 480 ms (see Figure 2) while the GPU implementation only needs 18 ms, i.e. speed-up of 27x. Furthermore, the NVIDIA V100 GPU is able to compute problems with 25 predicates in only 632 ms. In comparison, the state-of-the art method based on iterative scaling presented by Markl et al. [9] already needs more than one second to compute the result for only 10 predicates (see Figure 2).

The runtimes in the fifth column in Figure 6 refer to problems where the set of known selectivities T contains all unary, binary

	Newton GPU			
z	m	runtime [ms]	m	runtime [ms]
3	7	0.9	8	1.0
4	11	0.7	15	0.9
5	16	0.7	26	0.9
6	22	0.7	42	1.2
7	29	0.8	64	1.4
8	37	1.0	93	1.8
9	46	1.3	130	2.9
10	56	1.5	176	3.5
11	67	1.8	232	4.9
12	79	2.2	299	6.5
13	92	2.5	378	8.8
14	106	3.1	470	11
15	121	3.7	576	16
16	137	4.7	697	20
17	154	6.2	834	28
18	172	7.7	988	33
19	191	11	1160	46
20	211	18	1351	63
21	232	35	1562	90
22	254	63	1794	130
23	277	140	2048	220
24	301	310	2325	420
25	326	630	2626	760

(NVIDIA Tesla V100)

Figure 6: GPU implementation of Variant B of Newton's algorithm

and ternary conjuncts. Here, the GPU is faster than our fastest CPU-based implementation for 10 or more predicates. For 20 predicates the GPU-based implementation is more than 43 times faster than our fastest CPU-based implementation. Such a complex problem could not be solved in a reasonable time by the state-of-the art method based on iterative scaling [9]. According to Figure 3, that implementation processed problems with 10 predicates in almost 26 seconds while our GPU-based implementation finishes this task in only 3.5 ms, i.e. almost five orders of magnitude faster.

5 DISCUSSION AND CONCLUSION

Query optimizers rely on several sources to estimate the selectivity of complex conjunctive predicates. Many database systems use elaborate methods to serve selectivity estimation, e.g., multi-column histograms [12], samples [3], statistics on views [7] or even query feedback [14].

Entropy maximization as proposed by Markl et al. [9] considers all available information to derive a consistent estimate for all partial conjuncts of a predicate. However, as the runtimes for iterative scaling are prohibitively high already for 8 predicates, Markl et al. suggests to partition the problem into smaller conjuncts assuming independence between the selectivities of the predicates of the partitions. This risks loosing valuable information from the set of known selectivities.

With the formalization of the entropy maximization problem as a series of vector- and matrix operations we are able to derive efficient implementations for this problem using the Newton algorithm. As our CPU based algorithm is more than 4 orders of magnitude faster than the iterative scaling for the most complex problem it could handle, entropy maximization becomes a feasible option even for complex predicates without sacrificing the quality of the cardinality estimates. Even more, the new implementations can be applied to conjuncts with 18 predicates for the CPU or even 25 predicates for the GPU with runtimes of less than a second making partitioning the input problem irrelevant for virtually all scenarios. While Markl et al. already explained in detail how to integrate the maximum entropy method into query optimizers, we conclude that using the implementation techniques presented in this paper, entropy maximization is ready to be included into production-grade database management systems.

REFERENCES

[1] G. Cormode, M. Garofalakis, P. Haas, and C. Jermaine. 2012. *Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches.* NOW Press.

[2] S.-C. Fang, J. R. Rajasekera, and H. S. J. Tsao. 1997. *Entropy Maximization and Mathematical Programming.* Kluwer.

[3] Rainer Gemulla, Wolfgang Lehner, and Peter J. Haas. 2007. Maintaining bernoulli samples over evolving multisets. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS).* 93–102.

[4] G. Golub and C. van Loan. 1996. *Matrix Computations.* The John Hopkins University Press. Third Edition.

[5] D. Harville. 2008. *Matrix Algebra from a Statistician's Perspective.* Springer.

[6] Y. Ioannidis. 2003. The History of Histograms (abridged). In *Proc. Int. Conf. on Very Large Data Bases (VLDB).* 19–30.

[7] P.-Å. Larson, W. Lehner, J. Zhou, and P. Zabback. 2007. Cardinality estimation using sample views with quality assurance. In *Proc. of the ACM SIGMOD Conf. on Management of Data.* 175–186.

[8] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

[9] V. Markl, P.J. Haas, M. Kutsch, N. Megiddo, U. Srivastava, and T.M. Tran. 2007. Consistent Selectivity Estimation Via Maximum Entropy. *VLDB Journal* 16, 1 (January 2007), 55–76.

[10] G. Moerkotte, M. Montag, A. Repetti, and G. Steidl. 2015. Proximal operator of quotient functions with application to a feasibility problem in query optimization. *J. Computational Applied Mathematics* 285 (2015), 243–255.

[11] Magnus Müller, Guido Moerkotte, and Oliver Kolb. 2018. Improved Selectivity Estimation by Combining Knowledge from Sampling and Synopses. *PVLDB* 11, 9 (May 2018), 1016–1028. https://doi.org/10.14778/3213880.3213882

[12] V. Poosala and Y. Ioannidis. 1997. Selectivity Estimation Without the Attribute Value Independence Assumption. In *Proc. Int. Conf. on Very Large Data Bases (VLDB).* 486–495.

[13] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. 1988. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, New York, NY, USA.

[14] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 19–28.

[15] Xiaohui Yu, Nick Koudas, and Calisto Zuzarte. 2006. HASE: a hybrid approach to selectivity estimation for conjunctive predicates. In *International Conference on Extending Database Technology.* Springer, 460–477.

[16] CUDA Developer Zone. 2019. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/.

Weaving Enterprise Knowledge Graphs: The Case of Company Ownership Graphs[*]

Paolo Atzeni
Università Roma Tre

Luigi Bellomarini
Banca d'Italia

Michela Iezzi
Banca d'Italia

Emanuel Sallinger
TU Wien and
University of Oxford

Adriano Vlad
Università Roma Tre and
University of Oxford

ABSTRACT

Motivated by our experience in building the Enterprise Knowledge Graph of Italian companies for the Central Bank of Italy, in this paper we present an in-depth case analysis of company ownership graphs, graphs having company ownership as a central concept. In particular, we study and introduce three industrially relevant problems related to such graphs: company control, asset eligibility and detection of personal links. We formally characterize the problems and present Vada-Link, a framework based on state-of-the-art approaches for knowledge representation and reasoning. With our methodology and system, we solve the problems at hand in a scalable, model-independent and generalizable way. We illustrate the favourable architectural properties of Vada-Link and give experimental evaluation of the approach.

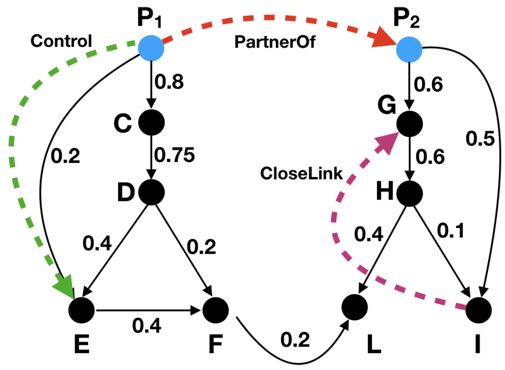


Figure 1: Sample excerpt of a company ownership KG.

1 INTRODUCTION

This paper is motivated by our experience in building the *Enterprise Knowledge Graph* of Italian companies for Banca d'Italia, the central bank of Italy.

Company ownership graphs are central objects in corporate economics [9, 19, 23, 36] and are of high importance for central banks, financial authorities and national statistical offices, to solve relevant problems in different areas: *banking supervision*, *credit-worthiness evaluation*, *anti-money laundering*, *insurance fraud detection*, *economic and statistical research* and many more. As shown in Figure 1, in such graphs, *ownership* is the core concept: nodes are companies and persons (black resp. blue nodes), and ownership edges (black solid links) are labelled with the fraction of shares that a company or person x owns of a company y. Company graphs are helpful in many situations.

One first important problem that can be solved with such graphs is *company control* (also effectively formalized in the context of logic programming [18]), which amounts to deciding whether a company x controls a company y, that is, x can push decisions through in y having the vote majority. Consider the graph in Figure 1: P_1 controls C, D (via C), E (since it controls D, which owns 40% of E and P_1 directly owns 20% of it), and F (via E and D). Similarly, P_2 controls all its descendants except for L. Apparently, P_1 exerts no control on L either.

A second particularly representative application of company graphs is in the context of *collateral eligibility* (also known as *asset eligibility* or *close link*) problem, which consists of estimating the risk to grant a specific loan to a company x that is backed by collateral issued by another company y. According to European Central Bank regulations [1], company y cannot act as a guarantor for x if it is too "close" to it in terms of ownership; the regulation gives a detailed definition for this concept of *closely-linked entity*, which includes: "the two companies must not be owned by a common third party entity, which owns more than 20% of both". With respect to Figure 1, we see that, for example, G and I are closely linked since P_2 owns more than 20% of both.[1]

Besides financial relationships, *personal or family connections* enable much broader use of such company graphs: detecting *family businesses* or studying the real dispersion of control [21] are just two such applications. In our example in Figure 1, knowing that P_1 and P_2 have personal connections –e.g., are married– allows to deduce that, in fact, P_1 and P_2 together control L. Likely, they act as a single center of interest: L is in fact a family business, with control in the hands of a single family, with P_1 and P_2 together controlling 60% of it. Similarly, although D and G do not strictly fulfil the definition of close link, as P_1 and P_2 have a personal connection, there is very low risk differentiation between them and so it is reasonable to prevent G from acting as a guarantor for D or vice versa.

In all the above settings, and indeed in many more, it is our experience that the links representing relevant relationships in the financial realm are not immediately available in data stores. For instance, there are no enterprise graphs readily providing company control relationships, close links or family connections. Reasons fall mostly into four categories: (i) such links represent non-trivial relationships, whose calculation is complex and is not

[*]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. This work is supported by the EPSRC grant EP/M025268/1, the Vienna Science and Technology Fund (WWTF) grant VRG18-013, and the EC grant 809965.

[1]Actually, here we are forcing the concept of close links to include individuals.

typically done while building the enterprise data stores, for example in the ETL (Extract, Transform, Load) jobs; (ii) the enterprise data stores are mainly relational and neglect specific connections because they are not easily navigable without a graph-based technology; (iii) edges between entities in different data stores are ignored because enterprise databases follow a siloed approach; (iv) edges are considered less trustworthy (lower data quality) than the entities they connect, and so are skipped.

Contribution. In this paper, we focus on the industrial context of the ownership graph of Banca d'Italia and provide the following contributions:

- We provide a compact but formal characterization of the problems of company control, asset eligibility and detection of personal connections over ownership graphs. We argue that the three problems belong to a much broader class, which we name *knowledge graph augmentation*, (KG augmentation). By KG augmentation, we wish to characterize a special case of *link prediction* [29], the key problem of predicting hidden links in network structures, where the emphasis is on the need for a careful combination of the extensional data (existing nodes and edges) and the available domain knowledge.
- We present Vada-Link, a framework for the solution of KG augmentation problems, leveraging state-of-the-art methodologies for *logic-based reasoning* [14, 24], which provide a very good balance between computational complexity of the reasoning task, being PTIME in data complexity, and expressive power, thanks to logic-based *Knowledge Representation and Reasoning* (KRR) languages such as Vadalog [14]. In fact, Vadalog captures full *Datalog* [18, 27] and is able to express SPARQL queries under set semantics and the entailment regime for OWL 2 QL [24].
- From the structured enterprise data stores, we build a *Property Graph* [3] (PG) –a graph with labelled nodes and edges– and model KG augmentation problems as *reasoning tasks* on a *Knowledge Graphs* (KG), based on such PG. In particular, here a KG is a data model combining an *extensional component*, that is, the data from the enterprise data stores, with a formal representation of the domain of interest, the *intensional component*, expressed in some KRR language. KGs typically represent domains that are well suited for being modeled as *property graphs* complemented with rules that express domain knowledge.
- We argue that our approach is general and can be used to augment KGs with new links that can be deterministically or heuristically deduced from a combination of extensional data and domain knowledge. In particular, we motivate that our approach is *schema independent*, as it is not coupled to a specific graph structure (i.e., specific types for nodes, edges and their properties), but relies on meta-level concepts describing graph constructs. At the same time, our solution is *problem aware*, as specific reasoning steps adopt *polymorphic behaviour*, depending on the specific problem. Moreover, our approach is *data model independent*, in the sense that the extensional component can be based on diverse data sources.
- With our KG-based approach, we build a *company knowledge graph*, where the PG of the ownership graph represents the extensional component of the KG and the hidden links, object of the above problems, are obtained by combining a logic-based intensional definition of their specification, i.e., the intensional component of the KG, with the extensional component. Our solution provides an approach to KG augmentation that shows at the same time high accuracy and efficiency, achieved by *multi-level clustering* of the search space, with techniques inspired by record linkage experience [20]. In particular, in the KG intensional component, we combine highly selective feature-based clustering with *neighborhood-preserving embeddings*: the former technique avoids quadratic blow-up of the search space, while the latter enhances accuracy by recognizing graph-based similarities of the entities.
- We exploit the framework to build our company knowledge graph. We motivate that our solution has very favourable properties: it is scalable, since it uses a tractable logic fragment, Vadalog [14] and reduces the search space via multi-level clustering; it is understandable, IT-independent and modifiable thanks to the adoption of a fully declarative approach; Vada-Link decisions are explainable and unambiguous, as the semantics of Vadalog is based on that of Datalog, well known in the database community [2].
- We provide extensive experimental evaluation of Vada-Link.

KG augmentation. In Vada-Link, we structure our link prediction tasks into two of subgoals. First, a *clustering task*, where we aim at grouping the nodes into candidate clusters in order to limit the search space of connected nodes. This allows to overcome the need to compare a quadratic number of nodes, which would make the approach infeasible. Clustering is performed with a combination deterministic rule-based choices and embedding techniques, e.g., *node2vec* [26], which allows to exploit not only the textual or numeric features of edges, but also their structural properties, such as: specific shared neighbours (along with their properties, etc.), topological role nodes and so on.

Clearly, clustering requires a preliminary *feature engineering phase*, needed to individuate the most informative features, which is however out of the scope of this paper. The second subgoal is a *multi-class classification problem*, where we aim at assigning each pair of nodes a link class, if any.

Related work. Different research areas and data management problems can be seen as related to Vada-Link. Link prediction is the problem of discovering links between nodes in network structures: it is relevant in social network analysis [29] and a variety of approaches have been recently proposed, e.g. [39]. In this work, we refer to a different setting, which we called Knowledge Graph augmentation: it is the problem of inferring connections that are necessarily present on the basis of a combination of extensional knowledge and domain experience. In this sense, we borrow techniques that are common in link prediction, e.g. node2vec [26] for clustering purposes, but we do not use them to make linking decisions, which are taken only via logic-based reasoning. In the context of KGs, there have been some proposals for link prediction approaches [28, 32]. However, unlike our approach, the emphasis is still on guessing new connections that cannot be deduced from existing data.

In this paper, for our goals, we borrow from the vast experience of the database community in *record linkage* [20], which is a different yet closely related problem. It consists in deciding whether two records without a common database identifier actually refer to the same real-world entity. Here, the problem is different as we are deciding whether two different real-world entities have some relationship and, if so, its type. Nevertheless, we inherit ideas for search space reduction, namely *blocking* and feature-based *probabilistic record linkage*, adapting them in the context of knowledge graphs. In doing so, we also rely in our experience in formulating and solving data science and data integration problems in a declarative context and in Vadalog in particular [11]. Finally, the general theoretical setting, including the definition of basic data models (e.g., property graphs), is also shared with the *graph querying* community [15].

Overview. The remainder of the paper is organized as follows. In Section 2 we describe our industrial setting of the company ownership graph at Banca d'Italia. In Section 3, we introduce the background of Vadalog-based KGs, graph embeddings as well as the foundations of our approach. In Section 4 we present the details of Vada-Link. In Section 5 we illustrate the architecture of the system, while Section 6 shows experimental evaluation. We draw our conclusions in Section 7.

2 INDUSTRIAL SETTING: AUGMENTING OWNERSHIP GRAPHS

A central bank needs data about companies to pursue a number of core institutional goals. In particular, the Bank of Italy owns the database of Italian companies, provided by the Italian Chambers of Commerce. Although the database is extremely rich and comprehensive, we shall see that many of the problems of interest that cannot easily be solved using traditional query languages can be concisely formulated as reasoning tasks on knowledge graphs. We show how to build such a company knowledge graph, in particular to identify hidden links between the involved entities in various scenarios. Such problems can be easily represented as a KG augmentation problem: given an input graph, we seek new edges that improve the connectivity of the graph to gain the ability to find new and undiscovered patterns. Let us start with a high-level description of the database.

The Italian Company Database. The database provided by the Italian Chambers of Commerce contains several features for each company such as legal name, address, incorporation date, legal form, shareholders and so on. A shareholder can be either a person or company. For persons, we find the associated personal data, such as first name, surname, date and place of birth, sex, home address, and so on. Detailed shareholding structure is also available: for each shareholder the database contains the actual share as well as the type of legal right associated to each share (ownership, bare ownership and so on).

The database at our disposal contains data from 2005 to 2018. If we see the database as a graph, where companies and persons are nodes and shareholding is represented by edges, on average, for each year the graph has $4.059M$ nodes and $3.960M$ edges. There are $4.058M$ Strongly Connected Components (SCC), composed on average of one node, and more than $600K$ Weakly Connected Components (WCC), composed on average of 6 nodes, resulting in an high level of fragmentation. Interestingly, the largest SCC has only 15 nodes, while the largest WCC has more than one million nodes. The *average in-* and *out-degree* of each node is ≈ 1 and that the *average clustering coefficient* is ≈ 0.0084, very low when compared to the number of nodes and edges. Furthermore, it is interesting to observe that the *maximum in-degree* of a node is more than $5K$ and the *maximum out-degree* is more than $28K$ nodes. We also observe a high number of self-loops, almost $3K$, i.e., companies that own shares of themselves, which could be referred to the buy-back phenomenon [4]. The resulting graph shows a scale-free network structure, as most real-world networks [7, 22, 34]: the degree distribution follows a power-law and there are several nodes in the network that act as hubs.

Use Cases. Let us introduce three relevant problems in the context of this database considering Figure 2.

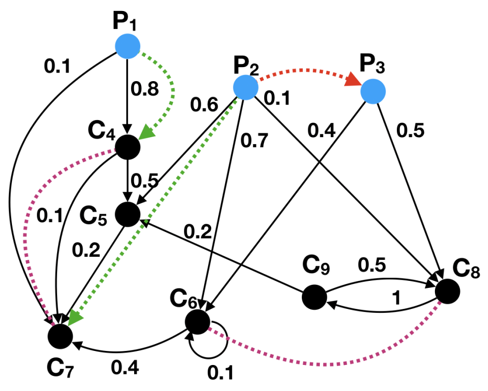


Figure 2: Italian Company Graph. Red-dashed edges represent personal connections, green-dashed edges represent control links, magenta-dashed edges are close-links.

(1) Does P_2 control C_7? More generally, we want to understand if a company or a person exerts control, through the majority of voting rights, on another company, i.e., the *Company Control* problem.
(2) Are companies C_6 and C_8 closely related? More generally, we want to understand whether there exists a link between two companies based on high overlap of shares, i.e., the *close link* problem.
(3) Are there personal/family links between the persons in the company graph? More generally, we wish to *detect personal connections* and label them, on the basis of the kind of relationship.

Solving the third problem allows us to see the first two under a new light: are there a groups of people (e.g., of the same family) in control of a certain company? Are two companies closely related because of the personal ties between their shareholders?

Towards a formalization of the above problems, we propose a graph-based representation of the company database, for which we need some preliminary notions.

Definition 2.1. A (regular) *Property Graph* (PG) is a tuple of the form $G = (N, E, \rho, \lambda, \sigma)$, where:

- N is a finite set of nodes;
- E (disjoint from N) is a finite set of edges;
- *incidence function* $\rho : E \longrightarrow N^n$ is a total function that associates each edge in E with an n-tuple of nodes from N –we will consider $n = 2$ from hereinafter;
- the *labelling function* $\lambda : (N \cup E) \longrightarrow \mathbf{L}$ is a partial function that associates nodes/edges with a label from a set $\mathbf{L}$;
- $\sigma : (N \cup E) \times \mathbf{P} \longrightarrow \mathbf{V}$ is a partial function that associates nodes/edges with properties from $\mathbf{P}$ to a value from a set $\mathbf{V}$ for each property.

The provided company database can be represented in terms of a property graph, as follows.

Definition 2.2. A *Company Graph* is a property graph $G = (N, E, \rho, \lambda, \sigma)$, such that:

- N contains companies and persons;
- E are shareholding edges, from companies to companies or persons to companies;
- the labeling set $\mathbf{L}$ is defined as $\{C, P, S\}$, where C stands for a Company node and P stands for a Person node; S represents a Shareholding edge;

- each node n has an identifier $x \in \mathbf{P}$, with value $\sigma(n, x)$, and a set of properties (or features) $f_1, \ldots, f_m$, with values $\sigma(n, f_1), \ldots, \sigma(n, f_m)$;
- each edge e has share amount w with value $\sigma(e, w) \in (0, 1] \subseteq \mathbb{R}$.

Figure 2 shows an example of the Company Graph. Let us introduce the definitions for the industrial use cases of our interest.

The Company Control Problem. Let us start with that of company control, with an effective formulation presented in the context of logic programming [18].

Definition 2.3. A company (or a person) x *controls* a company y, if: (i) x directly owns more than 50% of y; or, (ii) x controls a set of companies that jointly (i.e., summing the share amounts), and possibly together with x, own more than 50% of y.

Example 2.4. Referring to Figure 2: P_1 controls C_4 by means of a direct 80% edge; P_2 controls C_7, via C_5 and C_6. Green-dashed edges represent the resulting control links.

The Close Link Problem. For the second use case, we need to define the notion of *accumulated ownership* of a company or person x over a company y.

Definition 2.5. Let G be a Company Graph and let ϕ_{xy} be finite the set of all the s simple paths from x to y. Let ϕ_{ij}^m, with $1 \leq m \leq s$, be each of such paths.
The *accumulated ownership* $\Phi_G(x, y)$ of x over y is:

$$\Phi_G(x, y) = \sum_{\phi_{xy}^m \in \phi_{xy}} W(\phi_{xy}^m) \tag{1}$$

where:

$$W(\phi_{xy}^m) = \prod_{e \in \phi_{xy}^m} \sigma(e, w). \tag{2}$$

Starting from Definition 2.5, we can now define *Close Links*.

Definition 2.6. Given a Company Graph G, there is a *Close Link* relationship between a pair of companies x and y for a threshold $t \in (0, 1] \subseteq \mathbb{R}$ if: (i) $\Phi_G(x, y) \geq t$; or, (ii) $\Phi_G(y, x) \geq t$; or, (iii) there exists a person or company z s.t. $\Phi_G(z, x) \geq t$, $\Phi_G(z, y) \geq t$.

Example 2.7. Let us refer to Figure 2 and consider $t = 0.2$. We have that P_3 owns 40% of C_6 and 50% of C_8, therefore they are in close link relationship by Definition 2.6-(iii). Also, since $\Phi_G(C_4, C_7) = 0.2$, it follows that C_4 and C_7 are in close link relationships by Definition 2.6-(i).

Detecting Personal Connections. As we have seen, detecting personal connections enables insightful analyses, including the possibility to achieve more comprehensive view of company control and close links. In particular, we are interested in family relationships of various degrees, i.e. "PartnerOf", "SiblingOf", and so on. Many different models to predict family links exist. We adopt a simple one and, as common in these cases, we define the presence of a family link between x and y with a multi-feature Bayesian classifier. For a given feature f_i, we compute the conditional probability $p_i = P(L_{xy} \mid d(f_i^x, f_i^y) < T_f)$ of having a link L_{xy}, given that some distance between the feature values is under a given threshold for that feature (e.g., Levenshtein distance between two strings "name" of person). The probability p_i and threshold T_f can be estimated by observing $P(d(f_i^x, f_i^y) < T_f \mid L_{xy})$ from training data and given the a priori likelihood of $P(d(f_i^x, f_i^y) < T_f)$ and $P(L)$. Then, we combine all the conditional probabilities p_i into p by applying *Graham Combination* [25]:

$$p = \frac{\prod_{i=1}^n p_i}{\prod_{i=1}^n p_i + \prod_{i=1}^n (1 - p_i)} \tag{3}$$

Finally, a family link exists whenever $p > T$, where T is an established confidence threshold.

Knowing family connections, we can extend Definitions 2.3 and 2.6 to a more general setting. Let us define a family F as a set of persons directly or indirectly connected by personal links.

Definition 2.8. A family F *controls* a company y if: (i) a person $x \in F$ controls y according to Definition 2.3; or, (ii) F controls a (possibly empty) set of companies that jointly, and possibly together with direct ownerships of $\{x_1, \ldots, x_n\} \subseteq F$ (i.e., summing the share amounts), directly own more than 50% of y.

Definition 2.9. Given a Company Graph G, there is a *Close Link* relation between a pair of companies x and y for a threshold $t \in (0, 1] \subseteq \mathbb{R}$ if: (i) Definition 2.6 holds; or, (ii) there exist two persons z_1 and z_2 ($z_1 \neq z_2$) s.t. $z_1, z_2 \in F$, where F is a family, and $\Phi_G(z_1, x) \geq t$ and $\Phi_G(z_2, y) \geq t$.

Example 2.10. Referring to Figure 2, the red-dashed edge between P_2 and P_3 represent "PartnerOf" relationship. For example, P_2 and P_3 are in the same family. Neither P_2 nor P_3 control C_8 singularly. Yet, they have a jointly ownership of 60%; so, their family controls C_8. Also, C_5 and C_8 are closely linked, via P_2 that owns the 60% of C_5 and P_3 that owns 50% of C_8.

The major limitation to answer the illustrated business cases is the translation of all the above definitions into computationally infeasible algorithms due to the dimension of the graph and the complexity of the problems. For example, the close link definition resorts to the ownership definition, and not only: with regard to the third condition of Definition 2.6, the so-called third-party requires to look for a comparison of all existing ownership of companies linked to the third-party entity and a check of the fixed threshold passing. This involves finding all simple paths between pairs of nodes, a paradigmatic computationally hard problem in complexity theory, technically in the #P class [38]. Finding family connections is also challenging, requiring exhaustive comparison of all pairs of persons. Therefore, the intensional part of the company knowledge graph is urgently needed.

3 RELATIONAL GRAPH REPRESENTATION

In order to present the full detail of our KG-based solutions to the settings illustrated in Section 2, let us introduce some basic notions and see how we use them to model and reason on company graphs.

Knowledge Graphs (KG). Along the lines given in [10], we define a KG as a semi-structured data model composed of three components: (i) a *ground extensional component* (or simply extensional component), that is, a set of relational constructs for schema and data, which can be effectively modeled as a *property graph*; (ii) an *intensional component*, that is, a set of *inference rules* over the constructs of the ground extensional component; (iii) a *derived extensional component* that can be produced as the result of the application of the inference rules over the ground extensional component (with the so-called "reasoning" process).

In order to fulfil (i), we adopt a mapping of PG constructs into relational constructs. Let us introduce the background for both and describe our mapping.

Relational Foundations. Let **C**, **N**, and **V** be disjoint countably infinite sets of *constants*, *(labeled) nulls* and (regular) *variables*, respectively. A *(relational) schema* **S** is a finite set of relation symbols (or predicates) with associated arity. A *term* is either a constant or variable. An *atom* over **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 **S** associates to each relation symbol in **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.

Relational Representation for PGs. We map constructs of PGs into relational terms as follows. L-labelled nodes $n \in N$ (with $L_n \in \mathbf{L}$) are represented by facts $L(\hat{c}_x, c_f^1, c_f^2, \ldots, c_f^n)$ of predicate L, where for each property (feature) $f_i \in \mathbf{P}$, we have a constant term c_f^i of L of value $\sigma(n, f_i)$. Note that here we assume a total ordering of property names, so we can map them into positional atom terms and no ambiguity arises (a non-positional perspective could be easily adopted as well). We also assume every node has an identifier x, whose value $\hat{c}_x = \sigma(n, x)$ identifies its facts.

We map each L_e-labelled edge $e \in E$, into facts of predicate L_e: $L_e(\hat{c}_x^1, \ldots, \hat{c}_x^k, f_1, \ldots, f_m)$, where for each argument i of the incidence function ρ, there is a constant term $\hat{c}_x^i$ of L_e with value $\sigma(n, x)$, where $n = \rho(e)[i]$ and x is n identifier, and for each feature $f_i \in \mathbf{P}$ of e there is a constant c_f^i of L_e with value $\sigma(e, f_i)$.

Observe that node and edge labels operate at schema level and map into predicate names. Properties and identifiers are at instance level and define term values for facts representing nodes and edges. With this premises given, in our setting the extensional component of a KG is therefore a database instance representing the PG by means of the mapping described above.

Example 3.1. The extensional component of a KG based on the PG in Figure 1 has the following relational representation.
Company(C), Company(D), Company(E), Company(F), Company(G), Company(H), Company(I), Company(L), Person(P_1), Person(P_2), Own($P_1, C, 0.8$), Own($C, D, 0.75$), Own($D, E, 0.4$), Own($D, F, 0.2$), Own($E, F, 0.4$), Own($P_1, E, 0.2$), Own($P_2, G, 0.6$), Own($G, H, 0.6$), Own($H, L, 0.4$), Own($H, I, 0.1$), Own($P_2, I, 0.5$), Own($F, L, 0.2$).

Inference Rules. We describe the intensional components of KGs with logical inference rules in the Vadalog language. At the core of Vadalog, there is Warded Datalog$^{\pm}$ [12, 14] a language part of the Datalog$^{\pm}$ family [17]. Datalog$^{\pm}$ languages consists of *existential rules*, which generalize Datalog rules with existential quantification in rule heads, so that 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 with constants and variables. For brevity, we write this existential rule as $\varphi(\bar{x}, \bar{y}) \rightarrow \exists\bar{z}\, \psi(\bar{x}, \bar{z})$ and replace $\wedge$ with comma to denote conjunction of atoms. The semantics of such rule is intuitively as follows: for each fact $\varphi(\bar{t}, \bar{t}')$ that occurs in an instance I, then there exists a tuple $\bar{t}''$ of constants and nulls such that the facts $\psi(\bar{t}, \bar{t}'')$ are also in I. More formally, the semantics of a set of existential rules Σ over a database D, denoted $\Sigma(D)$, is defined via the well-known *chase procedure* [2]: new facts are added to D by the chase (possibly involving null values to satisfy existentially quantified variables) until $\Sigma(D)$ satisfies all the existential rules of Σ.

Example 3.2. The following rules define intensional edges linking persons with companies they are influential on (e.g., in the sense of control on company decisions). By Rule (1) a person x affects a company c she owns; her spouse also affects the company by Rule (2). Rules (3) and (4) generate Spouse edges, having a validity interval from t_1 to t_2, from Married edges.

(1) Person(x), Own(x, c, v) $\rightarrow$ Influence(x, c).
(2) Own(x, c, v), Spouse(x, y, t_1, t_2) $\rightarrow$ Influence(y, c).
(3) Married(x, y) $\rightarrow \exists t_1, t_2$ Spouse(x, y, t_1, t_2).
(4) Spouse(x, y, t_1, t_2) $\rightarrow$ Spouse(y, x, t_1, t_2).

4 THE VADA-LINK FRAMEWORK

In this section we focus on Vada-Link and describe in detail our technique for KG augmentation.

Algorithm 1 Basic KG augmentation algorithm.
Input: $G = (N, E, \rho, \lambda, \sigma)$, C: link classes
Output: $U = (N, E', \rho', \lambda', \sigma)$

```
1: $U \leftarrow G$
2: changed ← true
3: **while** changed **do**
4:     changed ← false
5:     $\mathbf{K} \leftarrow$ *GraphEmbedClust*($U$)
6:     **for** $K \in \mathbf{K}$ **do**                                  ▷ First level
7:         $\mathbf{B} \leftarrow$ *GenerateBlocks*($K$)
8:         **for** $B(N_B, E_B, \rho_B, \lambda_B, \sigma_B) \in \mathbf{B}$ **do**       ▷ Second level
9:             **for** $p_1, p_2 \in N_B, c \in C$ **do**
10:                **if** *Candidate*($p_1, p_2, c$) *and* $e \notin E_B$ **then**
11:                    add $e$ to $E_B$
12:                    set $\rho_B(e) = (p_1, p_2)$
13:                    set $\lambda_B(e, \text{TYPE}) = C$.
14:                    changed ← true
15: **return** U
```

Algorithm 1 presents a high-level overview of the approach. We take as input a property graph G and return a property graph U, which is obtained by adding the predicted edges. In our industrial case introduced in Section 2, the property graph is a company graph and the predicted links can be a control relationship, a close link relationship, or a family link.

The overall approach consists in a double level of clustering (which we will also call *blocking*) of the graph: after the first grouping, possible links between nodes are searched only within a single cluster in order to limit the number of needed comparisons. We start from a clustering **K** (line 5) performed by the function *GraphEmbedClust*. We compute a node embedding of the whole graph U: nodes are mapped into multi-dimensional vectors, whose distance reflects the similarity of the nodes, evaluated on the basis of both their features and role in the graph topology. Then each cluster K is in turn partitioned into a more specific clustering **B** (line 7) by the function *GenerateBlocks*. The search for nodes to be linked is then performed within each cluster B and for each type $c \in C$ of links by *Candidate* (line 10). If an edge is to be created for the considered nodes, it is labeled with the proper type (lines 11-13). Once all clusters B in **B** have been used to enrich U with new edges, first-level clustering via graph embedding is recursively applied (line 5). The algorithm proceeds until no changes occur. Finally, U is returned.

The Knowledge Graph. We implemented the core KG augmentation logic of Vada-Link described in Algorithm 1 with a Vadalog KG, where G is the ground extensional component and the intensional component is defined by three set of rules, encoding the prediction logic for the edges to be added to G to obtain U. The first set of rules (Algorithm 2) is the *input mapping*: they take as input the relational representation of one specific PG, as defined in Section 3 (for example that of the company graph) and transform it into higher-level concepts: generic nodes, generic edges and types and properties for those nodes and edges. The second set of rules (Algorithm 3) is the actual *link prediction logic*: it contains the core reasoning process giving rise to new edges; it operates on generic nodes and edges. Finally, the third set of rules (Algorithm 4) is the output mapping: it transforms the high-level generic links that have been created by the link prediction logic into their relational representation in the PG.

Let us now analyse the details of the three sets of rules.

Algorithm 2 Input mapping for the company PG.

```
(1) $\text{Company}(\text{name}, \text{addr}, \text{inc. date}, \text{leg. form}, \ldots),$
    $z = \#\text{sk}_c(\text{name}) \rightarrow$
    $\text{Node}(z, \text{name}, \text{addr}, \text{inc. date}, \text{leg. form}), \text{NodeType}(z, \text{Comp}).$
(2) $\text{Person}(\text{name}, \text{birth}, \text{addr}, \ldots),$
    $z = \#\text{sk}_p(\text{name}) \rightarrow$
    $\text{Node}(z, \text{name}, \text{birth}, \text{addr}, \ldots), \text{NodeType}(z, \text{Person}).$
(3) $\text{Own}(x, y, \text{amount}, \text{right}, \ldots), \text{right} = \text{pers.share} \rightarrow$
    $\exists z\ \text{Link}(z, \#\text{sk}_p(x), \#\text{sk}_c(y), \text{amount}, \text{right}, \ldots),$
    $\text{EdgeType}(z, \text{Shareholding}).$
(4) $\text{Own}(x, y, \text{amount}, \text{right}, \ldots), \text{right} = \text{comp.share} \rightarrow$
    $\exists z\ \text{Link}(z, \#\text{sk}_c(x), \#\text{sk}_c(y), \text{amount}, \text{right}, \ldots),$
    $\text{EdgeType}(z, \text{Shareholding}).$
```

The ground extensional component is modeled by means of three atoms: `Company`, `Person` and `Own`. `Company` and `Person` hold the basic features for companies and persons, respectively –the feature names are self explanatory– and are identified by `name`; `Own` connects a person/company with name x to a company with name y, when x is a shareholder of y and contains the respective features, such as `amount` and type (`right`) of share. Clearly, here we are assuming that `name` is a valid unique identifier for the sake of simplicity, while in practice fiscal code or other more appropriate codes are used. Rule (1) upgrades companies into generic `Nodes` of `NodeType` "Company" (quotes are omitted in the rules for readability). Similarly, Rule (2) upgrades persons. For simplicity of presentation and without loss of generality, here we assume that atoms are variadic (denoting extra terms with "…") so as to support an arbitrary number of features; coherence is guaranteed by the adoption of a positional perspective in our relational representation of PGs and by nodes and edges being typed. Clearly, we could replace variadic nodes and edges with $\text{Feature}(x, f, v)$ atoms, explicitly representing a feature f of value v for node x.

Both Rules (1) and (2) use functions denoted by #sk, namely *Skolem functors*, to generate node identifiers. Skolem functors are often used in variants of Datalog with OID invention, where identifiers need to be generated [5, 16] with specific properties: (i) *determinism*: repeated applications of the same functor on the same argument yield the same OID –this will be useful to generate edges; (ii) *injectivity*: there are no distinct domain elements yielding the same OID –for instance, no different companies will be assigned the same OID; (iii) *disjoint range*: different Skolem functors cannot produce the same OID –in case a company and a person have the same name, $\#\text{sk}_c$ and $\#\text{sk}_p$ will produce different OIDs. Vadalog supports Skolem functors in this form.

Rule (3) and Rule (4) upgrade ownership into generic `Links` with an `EdgeType` depending on the type of ownership: `pers.share` for persons holding companies, `comp.share`, in the case of a company owning a company. We adopt existential quantification to generate OIDs for `Links` and Skolem functors to obtain the OIDs of the `Nodes` associated to names x and y.

Observe that the application order of the rules is irrelevant: thanks to determinism of Skolem functors, `Links` can be generated even before the respective `Nodes` for companies and persons.

Algorithm 3 Vadalog KG augmentation logic.

```
(1) $\text{Node}(x, f_1^x, \ldots, f_n^x), \text{Link}(e, v, w, f_1^e, \ldots, f_m^e),$
    $\text{NodeType}(x, t_n), \text{EdgeType}(e, t_e),$
    $b_1 = \#\text{GraphEmbedClust}(f_1^x, \ldots, f_n^x, f_1^e, \ldots, f_m^e, t_n, t_e, \langle e \rangle),$
    $b_2 = \#\text{GenerateBlocks}(f_1^x, \ldots, f_n^x, t_n) \rightarrow \text{Block}(b_1, b_2, x)$
(2) $\text{Node}(x, f_1^x, \ldots, f_n^x), \text{Node}(y, f_1^y, \ldots, f_n^y),$
    $\text{NodeType}(x, t_n), \text{NodeType}(y, t_n), x \neq y,$
    $\text{Block}(b_1, b_2, x), \text{Block}(b_1, b_2, y), \text{LinkClass}(t),$
    $\text{Candidate}(x, y, t) \rightarrow \exists z\ \text{Link}(z, x, y, \ldots), \text{EdgeType}(z, t).$
```

Algorithm 3 represents the core prediction logic of Vada-Link. For every node x, Rule (1) considers all the edges e of the graph and positions x into a two-level nested clustering structure represented by the atom `Block`, where b_1 and b_2 are the clustering levels. The first clustering is established by applying the function #GraphEmbedClust, which wraps a function call to a specific clustering algorithm based on node embedding (details in Section 4.1). It takes as input the features of x, all the edges e of the graph along with their features, the respective types t_n and t_e and returns the identifier b_1 of the first-level cluster. The second-level clustering is determined by applying the function #Generate-Blocks, whose resulting cluster identifier b_2 only depends on the node properties and type.

Some discussion of the use of functions in our solution is needed. #GenerateBlocks is a *fact-level function* (details in Section 4.2) and its semantics is quite straightforward: for a specific binding of the function arguments that is part of its domain, a value for b_1 is produced. The function is in some sense polymorphic: depending on the type t_n of the involved nodes, a specific semantics is applied to decide the target cluster on the basis of the node features. For example, if the node represents a person, a specific algorithm may rely on last names or addresses; in case of companies, the industrial sector may be relevant, and so on.

#GraphEmbedClust is a *monotonic aggregation function*. Aggregation functions are adopted in various settings making use of logical formalism and the need for a careful definition arises in all of them, especially in the presence of model based semantics [31]. Among the various types of aggregation that Vadalog features [13], in our solution we adopt the monotonic one [37], which respects monotonicity w.r.t. to set containment. Intuitively, aggregation is provided in the form stateful fact-level functions, which memorize the current aggregate value; subsequent invocations of a function then yield updated values for the aggregate so as that the "final value" is the actually desired aggregate value. Thanks to monotonicity, such final value can be easily identified as the minimum/maximum value. In order to decide the first-level clustering for node x, the function #GraphEmbedClust takes as input the node features, an edge e with its features, and the node and edge types t_n and t_e; for a given node x, whenever the function is activated for a new edge e (notation $\langle e \rangle$ denotes that e is such an aggregation contributor), a more accurate clustering is possible and a new, greater value for b_1 is returned. In fact, #GraphEmbedClust wraps the invocation of a node2vec primitive (some details in Section 4.1), whose precision depends on the portion of the graph that is available to it.

For every second-level cluster defined by a `Block` fact, Rule (2) exhaustively considers all the pairs of nodes x and y and for every possible `LinkClass` t (wrt our case many exist: Control, CloseLink, ParentOf, PartnerOf, etc.); the `Candidate` predicate (details in Section 4.3) is used decide whether a `Link` from x to y must be produced or not. If the case, a new t-typed edge is created. Observe that Rule (2) compares only the pairs of nodes in the same sub-cluster (identified by b_1 and b_2).

Algorithm 4 Output mapping for the company PG.

```
(1) $\text{Link}(z, x, y), \text{EdgeType}(z, \text{Control}) \rightarrow \text{Control}(x, y)$.
(2) $\text{Link}(z, x, y), \text{EdgeType}(z, \text{CloseLink}) \rightarrow \text{CloseLink}(x, y)$.
(3) $\text{Link}(z, x, y), \text{EdgeType}(z, \text{ParentOf}) \rightarrow \text{ParentOf}(x, y)$.
(4) $\text{Link}(z, x, y), \text{EdgeType}(z, \text{PartnerOf}) \rightarrow \text{PartnerOf}(x, y)$.
```

Algorithm 4 is the output mapping, actually transforming the predicted edges back into the PG language. With reference to our industrial case, Rules (1) and (2) generate `Control` and `CloseLink` links, respectively. Rules (3) and (4) exemplify possible family links, and many more exist.

4.1 Clustering with Node Embeddings

Embeddings are mappings of real-world objects into high dimensional real-valued vectors that guarantee specific, e.g. geometric, properties reflecting the semantic relationships between the objects. Typically, embeddings are based on some similarity or neighbourhood notion, like in word embeddings [6]. The function #GraphEmbedClust in Rule (1) of Algorithm 3 implements a graph embedding, specifically a *node embedding*, that is, mappings of graph nodes into vectors so that network node neighbourhood is preserved. Specific algorithms learn node embeddings with different random walk strategies, resulting in the optimization of different measures as a consequence. In the function we adopt *node2vec* [26], a particularly interesting embedding which optimizes both network vicinity and network role of a node. We map graph nodes into vectors with 128 dimensions, in such a way that their distance preserves feature-based node similarity as well as neighbourhood, e.g., the so-called "homophily", that is, nodes having the same friend nodes are considered similar. We also apply a preliminary dimensionality reduction step based on T-SNE (*t-distributed stochastic neighbor embedding spectral clustering*) [35]. The #GraphEmbedClust is *polymorphic* in the sense that depending on the types t_n and t_e of involved nodes and edges, it adopts different embedding strategies, which highlight the topological peculiarity of the problem. For instance, it is indeed common that persons having largely overlapping groups of family members are in turn connected by a family relationship; conversely, companies in hold of overlapping sets of shares of other companies, tend to be part of the same group.

Clustering based on graph embeddings shows to be particularly useful in the industrial cases at hand, since it includes in the same clusters candidates that would be far in terms of their descriptive features. This goes beyond the textual or numeric features of companies and persons. Moreover, the interplay between different types of links is also interesting: for example, it is our experience that people in hold of overlapping sets of shares tend to be family members; vice versa, companies owned by overlapping sets of people tend to be part of the same group. The combination of quantitative features and structural graph property is also noteworthy. People owning certain patterns of shares of the same company tend to have family connections.

4.2 Generating Blocks

#GenerateBlocks in Rule (1) of Algorithm 3 reduces the search space for link candidates by sub-clustering. In particular, the function takes as input a vector of features of a node x (a person or a company in our case), belonging to first-level cluster b_1 and returns the identifier b_2 of a second-level cluster. Note that a careful choice the features, *feature engineering phase*, is fundamental: in VADA-LINK, we modularize out this highly domain dependent aspect into the specific implementations of #GenerateBlocks, so that the overall solution is general and ad-hoc tuning is possible whenever new business domains arise.

VADA-LINK provides different pluggable implementations for various domains. In most of the cases sub-clustering can be determined on the basis of well-known hashing or partitioning techniques. We support intuitive declarative specifications of auxiliary functions in VADALOG as follows:

$$\text{Hash}(h, f_1, \ldots, f_n), \text{Features}(f_1, \ldots, f_n) \rightarrow \text{Ans}(h).$$

Here, the `Feature` atom binds to the vector of features taken as input by #GenerateBlocks; with a join with the `Hash` atom, the functionally dependent hash value h is returned. The above rule is clearly complemented by the set of facts defining the underlying `hash` relation. Conventionally the `Ans` atom denotes the function return value. Alternative implementations could be based on Skolem functors as follows:

$$h = \#\text{Sk}_h(f_1, \ldots, f_n), \text{Features}(f_1, \ldots, f_n) \rightarrow \text{Ans}(h).$$

4.3 Generating Matching Candidates

The possible candidates to be linked are matched by the polymorphic `Candidate` predicate, which has different implementations, depending on type of link to be predicted. Let us show how this is applied in our cases.

Algorithm 5 `Candidate` predicate for company control.

```
(1) $\text{Node}(x, f_1, \ldots, f_n), \text{NodeType}(x, \text{Company}) \rightarrow$
        $\text{Candidate}(x, x, \text{Control})$.
(2) $\text{Candidate}(x, z, \text{Control}), \text{Link}(u, z, y, w)$
    $\text{EdgeType}(\text{u}, \text{Shareholding}), msum(\text{w}, \langle \text{z} \rangle) > 0.5$
        $\rightarrow \text{Candidate}(x, y, \text{Control})$.
```

The VADALOG rules in Algorithm 5 define candidate companies to be linked by a control relationship, according to Definition 2.3. Such relationship holds for a company on itself (Rule (1)); then, whenever a company x controls a set of companies z that jointly own more than 50% of a company y, then x controls y.

Algorithm 6 defines companies that are in close link relationships (with threshold T) according to Definition 2.6. Rule (1) and (2) calculate accumulated ownership for companies x and y, according to Definition 2.5. Rule (3) gives the base case for close links: accumulated ownership greater than or equal to 20% is a close link. Close links are symmetric, by Rule (4). Finally, if a company z owns a significant share of both x and y, they are a close link by Rule (5).

Algorithm 6 `Candidate` predicate for close links.

```
(1) $Link(z, x, y, w), EdgeType(z, Shareholding) \rightarrow AccOwn(x, y, w)$.
(2) $Link(u, x, z, w_1), EdgeType(u, Shareholding), AccOwn(z, y, w_2), v = msum(w_1 \cdot w_2, \langle z \rangle) \rightarrow AccOwn(x, y, v)$.
(3) $AccOwn(x, y, w), w \geq T \rightarrow Candidate(x, y, CloseLink)$.
(4) $Candidate(y, x, CloseLink) \rightarrow Candidate(x, y, CloseLink)$.
(5) $AccOwn(z, x, w_1), AccOwn(z, y, w_2), w_1 \geq T, w_1 \geq T, \rightarrow Candidate(x, y, CloseLink)$.
```

Algorithm 7 `Candidate` predicate for PartnerOf.

```
$Node(x, f_1^x \dots f_n^x), Node(y, f_1^y \dots f_n^y), NodeType(x, Person), NodeType(y, Person), \#LinkProbability(f_1^x \dots f_n^x, f_1^y \dots f_n^y) > T \rightarrow Candidate(x, y, PartnerOf)$.
```

Algorithm 7 defines a pair of candidate persons that have a family connection. In the example, the function #LinkProbability implements Equation 3 of Section 2. Here we consider the "PartnerOf" relationship, but similar algorithms are valid for all personal connection types.

We are now able to extend company control and close link detection to support the presence of families. For the sake of space, we omit here VADALOG rules generating "Family" nodes F and "Family" links connecting persons to their family.

Algorithm 8 `Candidate` predicate for family control.

```
(1) $Node(F, f_1^F, \dots, f_n^F), NodeType(F, Family), Link(z, x, F), EdgeType(z, Family), Link(v, x, y), EdgeType(v, Control) \rightarrow Candidate(F, y, Control)$.
(2) $Candidate(F, x, Control), Link(z, x, y, w), EdgeType(z, Shareholding), msum(w, \langle x \rangle) > 0.5 \rightarrow Candidate(F, y, Control)$.
(3) $Link(u, i, F), EdgeType(u, Family), Link(z, i, y, w), EdgeType(z, Shareholding), msum(w, \langle i \rangle) > 0.5 \rightarrow Candidate(F, y, Control)$.
```

Rules (1) Algorithm 8 implement condition (i) of Definition 2.8, while Rules (2) and (3) implement condition (ii) by accounting for the contribution on y of both the companies x controlled by F and the direct ownership of members of F. Technically, the two monotonic summations of Rules (2) and (3) contribute to the same total, one for each $\langle F, y \rangle$ pair.

Algorithm 9 `Candidate` predicate for family close link.

```
(1) $Link(z, i, F), EdgeType(z, Family), Link(k, j, F), EdgeType(k, Family), i \neq j, AccOwn(i, x, v), v \geq 0.2, AccOwn(j, y, w), w \geq 0.2, \rightarrow Candidate(x, y, CloseLink)$.
```

Finally, Algorithm 9 extends Algorithm 6 and implements part (ii) of Definition 2.9.

4.4 Discussion

We conclude the section with some informal arguments about termination, correctness and properties of our approach. I.e., Algorithms 2, 3 and 4 terminate and, in particular, Algorithm 3 correctly adds the required edges as defined in Algorithm 1.

Termination. As we have touched on in Section 3, the semantics of a set of existential rules Σ is given by the chase procedure, where new facts are added to database D (the extensional component), until $\Sigma(D)$ satisfies all the rules. We argue that in our case $\Sigma(D)$ is always finite.

- Algorithms 2 and 4 are non-recursive, therefore each rule adds a finite number of facts to $\Sigma(D)$, because of the finiteness of the extensional component.
- Algorithm 3 is recursive, as `Links` generated by Rule (2) appear in the body of Rule (1) –edges recursively improve the embeddings. The number of `Links` that can be generated by Rule (2) is finite and, in the worst case, it amounts to $|N|^2 \times C$, where N are the PG nodes and C is the number of possible link types. Therefore, Rule (2) produces a finite number of facts in $\Sigma(D)$, up to renaming of link identifiers z. Technically, the VADALOG chase procedure applies isomorphism check to prevent the generation of redundant facts. Therefore, Rule (1) produces a finite number of clusterings $\langle b_1, b_2 \rangle$, since it can fire in the worst case for every single edge in E plus all the ones introduced by Rule (2). Therefore Algorithm 3 always terminates.
- Special care must be paid in the specific polymorphic implementations of the `Candidate` predicate. Observe that in our settings, Algorithms presented in Section 4.3, in the worst case enumerate all the graph paths, and so always terminate.

Correctness. Let us show that each element (e.g., a company or a person) is correctly assigned to a single cluster and pairwise comparison is correctly performed inside each of them and for all possible link classes, as defined in Algorithm 1.

- The nested clustering is produced by the joint use of functions #GraphEmbedClust and #GenerateBlocks within Rule (1) of Algorithm 3. They are both applied to each node and the generated identifiers, b_1 and b_2, appear as terms the head of Rule (1) as well as the node x. As a consequence, because of set semantics, every node x is assigned to a unique pair $\langle b_1, b_2 \rangle$.
- In the body of Rule (2) of Algorithm 3, the two `Node` atoms operate on all pairs of nodes $\langle x, y \rangle$ such that x and y share the clustering configuration $\langle b_1, b_2 \rangle$. Therefore, no other elements are involved in the comparison performed by the `Candidate` predicate. Moreover, Rule (2) fires for each possible class t of `LinkClasses` and so, eventually, all possible triples $\langle x, y, t \rangle$, with $\langle x, t \rangle$ in the same cluster are evaluated.

A broader consideration of the overall correctness and completeness of the approach applied to the specific problems over company ownership graphs is necessary. All the implementations of the `Candidate` predicate we presented in Section 4.3 are deterministic, in the sense that they apply a priori conditions that encode the domain knowledge and produce a linking decision. Clearly, `Candidate` may be implemented as a statistical model, but still, this decision remains deterministic. The specific configuration of the clustering mechanism, i.e., the specific implementations of #GraphEmbedClust and #GenerateBlocks, determines which pairs of nodes are considered by `Candidate` and, as we will see in Section 6, this is the main key to achieving scalability. Hence, if two nodes x and y are supposed to be connected by a t-typed edge e and the recursive clustering mechanism always assigns them to different blocks, the final result will not be complete and miss e. It is the responsibility of the data engineer to strike an acceptable balance between completeness and granularity of clustering from case to case.

Finally, observe that the correctness of the links predicted by Vada-Link depends on the correctness/statistical robustness of the single implementations of `Candidate`, which is problem dependent and out of the scope of this paper.

Complexity. In the average case, clustering allows for linear behaviour, as we experimentally show in Section 6. In the worst case, i.e., all nodes are assigned to the same cluster, the approach performs $|N|^2 \times C$ comparisons, where N are the nodes in the extensional components and C the number of possible link types. However, with highly dense graphs, complexity is dominated by that of #GraphEmbedClust since node2vec has quadratic complexity in the graph branching factor [33]. Also, the complexity of the specific implementations of `Candidate` must be taken into consideration from case to case, as it could be dominating, for example with problems that require path enumeration. Full complexity characterization of company graph problems is beyond the scope of this paper. However, we point out that the complexity of `Candidate` can be controlled by a careful choice of the sub-fragment of Vadalog language. In fact, if the task is described in *Warded Datalog*$^\pm$, the fragment at the core of the Vadalog language, there is the formal guarantee of polynomial complexity [12].

Properties of the approach. Our approach is *schema and model independent*. It is schema independent in the sense that Vada-Link is able to perform KG augmentation regardless of the specific input PGs. This independence is achieved by means of a preliminary "promotion" of the relational representation of the extensional component into a generic graph model (Algorithm 2), with abstract constructs (nodes, edges, features, types). The link prediction logic is then applied within the generic graph, with specific polymorphic implementations of `Candidate` for each problem, encoded in Vadalog in terms of these high-level constructs. In this sense, the approach is also *problem aware*. Finally, the generated links are mapped again into the specific graph schema with Algorithm 4.

The approach is model independent in the sense that the extensional component can originate from heterogeneous data sources, even based on different data models (relational, object oriented, XML, NoSQL models), under the condition it can be imported into the relational representation of PGs described in Section 3.

Finally, it is worth remarking that our prediction technique is based on a kind of *reinforcement principle* because the positively predicted edges in turn help new predictions. In fact, the first-level clustering in Algorithm 3 is gradually improved with new edges being considered from both the ground ones and those generated by Rule (2).

5 THE ARCHITECTURE OF VADA-LINK

The development of enterprise applications using KGs is still a largely unexplored area and design methodologies as well as architectural patterns are gradually emerging. In [10], we proposed a set of architectural principles for the design of KG-based applications; in the following, we briefly motivate how those principles are satisfied by Vada-Link architecture.

- *Assign the immediately known, original information to the ground extensional component of the KG.* In fact, Algorithm 2 embodies the construction of a property graph exactly holding the information available in the company database. Besides the simple mapping, we clearly perform data cleaning and quality enhancement steps, whose details are omitted in this work.
- *For the schema design of the extensional component, adopt well-known conceptual modeling techniques.* In Vada-Link, we provide a relational representation of the extensional data (Section 3) that respects consolidated design and data normalization practices: relevant entities, persons, ownerships and companies in the case at hand, reflect into standalone relations, incorporating identifiers as well as the specific features as attributes.
- *Use extensional rules to model sophisticated and reasoning intensive business rules.* We represent the logic needed to generate the links in the form of declarative specifications. More concretely, with respect to the company KG, we consider multiple kinds of links (close links, company control, family relationships) and for each of them propose a Vadalog program representing it.
- *Keep business logic in the applications (do not let it drift into the KG intensional component).* We carefully separate the business logic of the client applications, that is, the software components using our KG for internal purposes (e.g., economic research, anti-money laundering, etc.) from any logic needed to generate the links. The Vadalog rules represent only the latter, whereas the application business logic resides within the application components. We choose not to implement each polymorphic variant of generation of clusters and evaluation of matching candidates (Sections 4.2 and 4.3) in a dedicated software module. First, such module would be highly coupled to data and hardly explainable or modifiable; more in general, the advantages of declarative approaches in the KG realm are largely acknowledged [30]. In our case, the adoption of logic-based rules appears particularly effective under three perspectives. *Understandability*: it is our experience that business users approach and appreciate human-readable rules instead of pure code; *modifiability*: given by the combination of high abstraction level and compactness of code (20-30 lines of Vadalog rules against 1K+ lines of Python code for the three cases at hand); *IT independence*: avoiding the strong coupling to a specific programming language.

Figure 3 shows the full functional architecture of Vada-Link. Its goal is building a KG from an existing source database. To this aim, data fetched from the RDBMS are enriched with features and extensions from external sources, with common ETL jobs.

The enriched dataset is then used as input to build the extensional component of the KG, that is, the property graph. Our *graph-building pipeline* takes as input an arbitrary database schema and maps it into the relational representation for PGs described in Section 3. The property graph is stored into a *Neo4J* server. The set of Vadalog rules in Algorithms 2, 3 and 4 are stored into a dedicated repository and executed by Vadalog. Enterprise applications interact with the KG via a *reasoning API*. Full details about Vadalog architecture are in [14].

6 EXPERIMENTS

In this section, we provide an experimental evaluation our approach to KG augmentation. The goal of the section is to highlight the scalability and accuracy of the overall approach in Vada-Link and not focussing on the specific Vadalog implementations of the three problems. In fact, out of the three KG augmentation problems we have presented, for the goal of this section we concentrate on the detection of family connections, which is at the same time straightforward but helpful to stress the system. Specific performance tests on complex reasoning tasks, including company control, can be found in [12, 14].

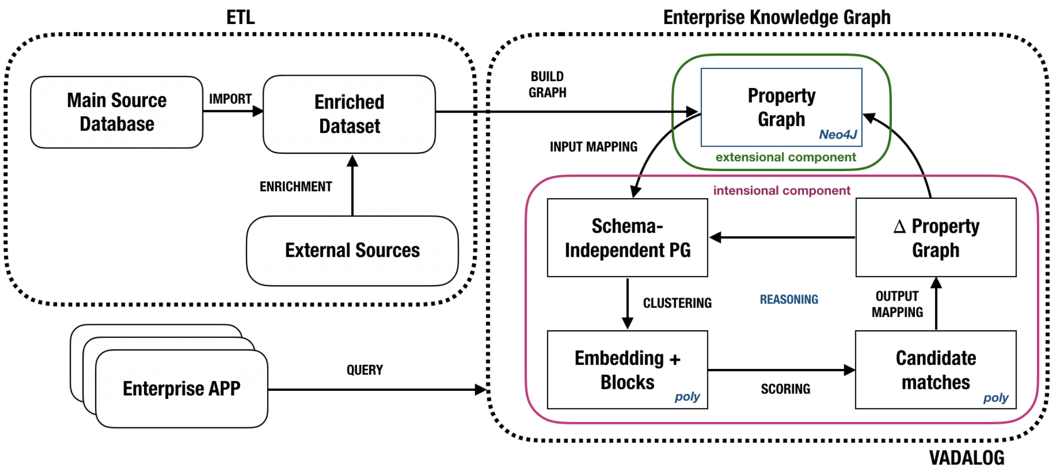


Figure 3: The functional architecture of Vada-Link**.**

We validate Vada-Link on both real-world and artificial data, showing that it exhibits good scalability and accuracy.

Datasets. The *real-world data* we use in the following experiments is the database of Italian companies of Banca d'Italia, as described in Section 2. For the *synthetic data*, we developed a graph generator. In particular, since company networks tend to be *scale-free networks* (see Section 2), we built different artificial graphs by adopting Barábasi algorithm [8] for the generation of scale-free networks, varying the number of nodes and the graph density. For each node, we randomly generated 6 features, out of distributions respecting their statistical properties.

Software and hardware configuration. We ran the experiments on a MacBook with 1.8 GHz Intel Core i5 and 4 GB 1600 MHz DDR3 memory. Vada-Link has been compiled with JDK 1.8.0_4 and clustering functions executed with Python 3.2.3.

6.1 Evaluating Scalability

We tested the scalability of Vada-Link in a number of settings, varying both the graph topology (e.g., number of nodes, density) and the data distribution, inducing different clustering structures.

Varying number of nodes. We investigate the impact of the number of nodes on the performance of Vada-Link. For the *real-world case*, we built 20 scenarios with subsets from the Italian company graph presented in Section 2, with ≈ 1-100k nodes representing persons. In order to stress the system even more with *synthetic data*, we built 6 artificial graphs with ≈ 1-10k nodes, having the same scale-free topology as the real-world graphs, but much higher density. We performed each experiment 10 times and averaged the elapsed times.

Results. The results for real-world data are reported in Figure 4(a). Vada-Link shows good scalability: execution time (blue line) grows slightly more than linearly with the number of nodes and remains under 20 seconds for 10k nodes. The trend is significantly far from quadratic growth, which we would expect with the naive approach consisting of exhaustive all-pairs comparison (red line). Figure 4(b) shows the results for synthetic data. Although the elapsed times are higher by one order of magnitude, which we explain with the highly dense topology of the generated artificial data, the trend is still linear and the approach effective.

Varying the number of clusters. We seek to observe the impact of the number of clusters on the overall execution time. To this end, we crafted a *real-world-like* experiment adopting the company graph presented in Section 2 and artificially tweaking the value of some features. As we have illustrated in Sections 4.1 and 4.2, our clustering technique is based on a recursive combination of node2vec (first-level clustering) and feature based blocking (second-level clustering); in particular, in Section 4.2 we have shown that the assignment to a second-level cluster is decided on the basis of a deterministic mapping –via hashing or Skolem functors– of a feature vector $f_1, \ldots, f_n$ into a cluster identifier. In this experiment, we alter the value of k of such n features in order to hijack the mapping into an increasing number of clusters of decreasing size and observe how this affects elapsed time. We extract values for the vector $f_1, \ldots, f_k$ from a discrete multivariate uniform distribution over the sample space $S_1, \ldots, S_k$. To induce more (fewer) and smaller (bigger) clusters, we restrict (expand) the cardinality of the domain of $S_1 \times \ldots \times S_k$. Specifically, we mapped $f_1, \ldots, f_n$ into ≈ 1-500 clusters.

Results. Figure 4(c) confirms that, as usual in clustering approaches, effective application of Vada-Link calls for a careful feature engineering phase: the selectivity of the features, i.e., the number of distinct values in the domain, and the cardinality of their domain affect the number and size of the clusters. For example, searching for the "siblingOf" relationship among people of the same last name and age range, would lead to clusters including thousands of persons, since certain last names are notably more common than others. Resorting to specific features, for example address vicinity or geographic area, could highly reduce the search space. Also, while the above considerations are somehow intrinsic and common to any clustering approach, in Vada-Link, the adoption of a hybrid node2vec/feature-based clustering combined within a recursive self-improving approach make it easier to strike a good balance between number of clusters (and hence elapsed time) and recall, as we shall see in Section 6.2.

Varying the density. In this experiment we investigate the impact of the density of the graph on the performance of Vada-Link. To this end, we built 4 artificial scenarios, *superdense*, *dense*, *normal*, *sparse*, corresponding to graphs of increasing density, and measured execution time for subset of these graphs of 1-1k nodes.

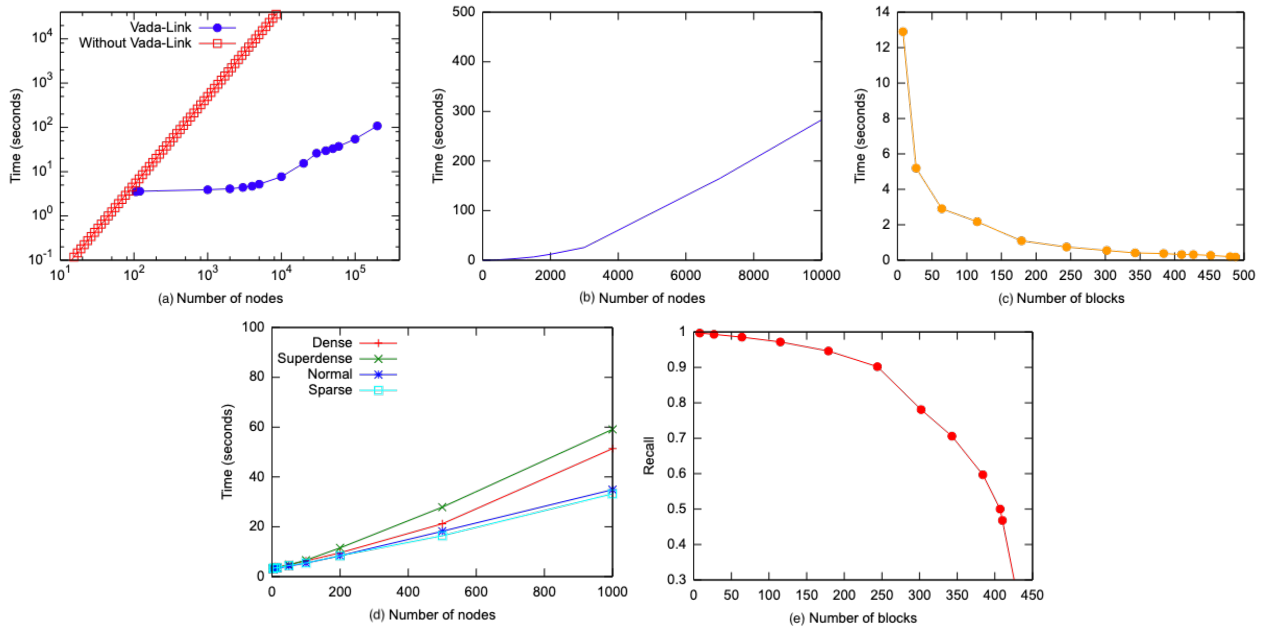


Figure 4: Analysis of Vada-Link execution time, depending on: (a) number of nodes in the graph (real-world data from the graph of Italian companies); (b) number of nodes in the graph (synthetic data); (c) number and size of the clusters (real-world data); (d) graph density (synthetic data). Analysis of Vada-Link recall, depending on: (e) number and size of the clusters (synthetic data).

Results. The results, shown in Figure 4(d), highlight that the performance of Vada-Link is influenced by the density of the considered graph. In particular, we notice an increase of the elapsed times, especially significant for more than 500 nodes. For lower values, sparse, normal and dense show similar trends, whilst superdense is slower, with ≈30 seconds for 500 nodes. The trend is amplified for greater values, with superlinear growth for dense and superdense. While for second-level clustering, #GenerateBlocks in Algorithm 3 is not affected by node density by construction, since it only considers node features, both first-level clustering (#GraphEmbedClust) and the implementations of `Candidate` predicate are. Node2vec needs to process a number of random walks that grows with the density; nevertheless, once a density is fixed, it scales almost linearly with the number of nodes, as also experimentally shown in [26]. Also the behaviour of `Candidate` is highly dependent on the considered KG augmentation problem. For example, detection of family connections have good scalability w.r.t. density as evident in Figure 4(d). Company control and close link detections are more challenging (specific experiments in [12, 14]). Nevertheless, thanks to clustering, Vada-Link can achieve good behaviour also in this case, clearly at the cost of loss in accuracy.

6.2 Evaluating Accuracy

As typically done in the validation of link prediction settings [26], we consider a graph with some edges arbitrarily removed; then, it is our goal to predict those missing edges and in so-doing we are interested in recall, i.e., how many of those removed edges we have recovered. We wish to infer the connections that need to be present and whose definitions are expressed by the polymorphic but deterministic implementations of the `Candidate` predicate, for example, Algorithm 7.

In some settings, `Candidate` implements certain models, like in the case of company control and asset eligibility, for which we would just need to discuss correctness from case to case; in others, `Candidate` encodes a classification model, for which the usual validation methodologies (confusion matrix, accuracy, precision, recall, ROC, AUC, etc.) would be valuable. This is indeed the case of the detection of family connections, our third setting. However, our goal here is not to concentrate on the statistical properties of specific link prediction models: in fact, besides our simple but effective Bayesian approach, more sophisticated models can be plugged in into Vada-Link, each with specific fine tuning. Instead, here our interest is in evaluating the overall tradeoff between scalability and recall.

Varying the number of clusters. We seek to observe the impact of the number of clusters on the recall of Vada-Link. We artificially built 10 random graphs S_i (with $1 \leq i \leq 10$) having the same number of nodes, topology and features as the real-world graph presented in Section 2. For each of them, we ran Vada-Link in “no cluster mode”, that is, we forced the system to concentrate all nodes inside one single cluster in order to produce all the theoretically possible links by means of the naive exhaustive comparison. Therefore, for each subgraph S_i we produced an augmented one $\hat{S}_i$. Then, from each $\hat{S}_i$, we randomly selected 10 edge sets Θ_{ij} (with $1 \leq j \leq 10$), each containing 20% of the predicted links and generated new subgraphs $S^{\Theta_{ij}}$ without those edges. For each $S^{\Theta_{ij}}$ we ran Vada-Link by varying the number of clusters with 20 configurations from 1 to 500, with the technique described in Section 6.1. For each case and cluster, we obtained an augmented graph $\hat{S}^{\Theta_{ij}}$ and computed the recall R_{ijc} as the fraction of edges cardinality $|E(\hat{S}^{\Theta_{ij}})|/|E(\hat{S}_i)|$, that is, the percentage of removed edges that have been recovered. For each of the 20 clusters, we averaged the 100 computed recall R_{ij}.

Results. Figure 4(e) shows how recall decreases with the number of clusters: It is clearly maximum for the single cluster case; then, the recall obtained for 20 clusters, 99.4%, is certainly acceptable for our use cases and, in general reasonable for industrial settings in ownership graphs. We then observe 98.6% for 50 clusters and the approach becomes ineffective for more than 400 clusters, with a recall steadily under 50%. A comparison with Figure 4(c) is interesting and confirms that with more than 10 clusters, processing time is under 10 seconds. This implies that for our case, an effective balance between efficiency and recall is between 10 and 20 clusters. More generally, Figure 4(e) shows a slow decrease of recall, which proves the robustness of Vada-Link. We explain that with the recursive interaction between first- and second-level clustering we have in Algorithm 3. Whenever in a second-level cluster new links are predicted, they are used by the aggregate function #GraphEmbedClust to improve the embedding and provide better first-level grouping and, as a consequence, increase the likelihood that in the second-level clustering, #GenerateBlocks considers new candidates. In other words, the recursive interplay between the two clustering functions compensate for increases in the number of clusters and contributes to a favourable balance between scalability and recall.

7 CONCLUSION

Company ownership graphs confront us with several problems, relevant in the financial realm. Discovering company control, reasoning on asset eligibility or finding out family connections are three interesting examples we focus on in this paper, motivated by the construction of the Enterprise Knowledge Graph of Banca d'Italia. Many more such settings exist, all with the common goal of enriching the company graph with new links. The need for actionable solutions is felt by the the national central banks of the European system, the national statistical offices and many more financial authorities

In this paper, we proposed Vada-Link, a new approach for the creation of valuable links in company graphs that leverages the vast amount of domain knowledge typically present in financial realm. On the basis of recent developments in the KG research area, we model the input graph as the extensional component of a logic-defined KG, where domain knowledge is encoded in Vadalog and link prediction is operated as a reasoning task within the KG. We discussed several favourable properties of our approach, which is general and applies to different data models and schemas. We also discussed the architecture of Vada-Link and provided experimental evaluation, highlighting good performance. We believe that with Vada-Link we are shedding light on problems of the financial realm that have specific scientific relevance per se and a graph- and logic-based view on them can certainly contribute to their full characterization.

REFERENCES

[1] 2014. GUIDELINE (EU) 2011/14 OF THE ECB. https://www.ecb.europa.eu/ecb/legal/pdf/l_33120111214en000100951.pdf. [Online; accessed 15-Feb-2019].

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

[3] Renzo Angles. 2018. The Property Graph Database Model. In *AMW.*

[4] Paul Asquith and David W. Mullins. 1986. Signalling with Dividends, Stock Repurchases, and Equity Issues. In *Financial Management.* 27–44.

[5] Paolo Atzeni, Luigi Bellomarini, Francesca Bugiotti, and Giorgio Gianforme. 2009. A runtime approach to model-independent schema and data translation. In *EDBT (ACM International Conference Proceeding Series)*, Vol. 360. ACM, 275–286.

[6] Dzmitry Bahdanau, Tom Bosc, Stanislaw Jastrzkebski, Edward Grefenstette, Pascal Vincent, and Yoshua Bengio. 2017. Learning to Compute Word Embeddings On the Fly. *CoRR* abs/1706.00286 (2017).

[7] Albert-László Barabási. 2009. Scale-Free Networks: A Decade and Beyond. *Science* 325, 5939 (2009), 412–413.

[8] Albert-Laszlo Barabasi and Reka Albert. 1999. Emergence of Scaling in Random Networks. *Science (New York, N.Y.)* 286 (11 1999), 509–12.

[9] Fabrizio Barca and Marco Becht. 2001. The Control of Corporate Europe. Oxford University Press, European Corporate Governance Network. (2001).

[10] Luigi Bellomarini, Daniele Fakhoury, Georg Gottlob, and Emanuel Sallinger. 2019. Knowledge Graphs and Enterprise AI: The Promise of an Enabling Technology. In *ICDE.* IEEE, 26–37.

[11] Luigi Bellomarini, Ruslan R. Fayzrakhmanov, Georg Gottlob, Andrey Kravchenko, Eleonora Laurenza, Yavor Nenov, Stéphane Reissfelder, Emanuel Sallinger, Evgeny Sherkhonov, and Lianlong Wu. 2018. Data Science with Vadalog: Bridging Machine Learning and Reasoning. In *MEDI (Lecture Notes in Computer Science)*, Vol. 11163. Springer, 3–21.

[12] Luigi Bellomarini, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. 2017. Swift Logic for Big Data and Knowledge Graphs. In *IJCAI.*

[13] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. [n.d.]. The Vadalog System (technical report). https://drive.google.com/drive/u/0/folders/0B2xKYNJdpJzQaEh5VzlhTURHU3M.

[14] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *PVLDB* 11, 9 (2018), 975–987.

[15] Angela Bonifati, George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs.* Morgan & Claypool Publishers.

[16] Luca Cabibbo. 1998. The Expressive Power of Stratified Logic Programs with Value Invention. *Inf. Comput.* 147, 1 (1998), 22–56.

[17] Andrea Calì, Georg Gottlob, Thomas Lukasiewicz, Bruno Marnette, and Andreas Pieris. 2010. Datalog+/-: A Family of Logical Knowledge Representation and Query Languages for New Applications. In *LICS.*

[18] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 2012. *Logic programming and databases.* Springer.

[19] Ariane Chapelle and Ariane Szafarz. 2005. Controlling firms through the majority voting rule. *Physica A: Statistical Mechanics and its Applications* 355, 2-4 (2005), 509–529.

[20] Peter Christen. 2012. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer. I–XIX, 1–270 pages.

[21] Caroline Fohlin. 1998. Trends in Business Organization: Do Participation and Cooperation Increase Competitiveness? Edited by Horst Siebert. Tubingen: J. C. B. Mohr, 1995. Pp. viii, 292. DM 108. *The Journal of Economic History* 58, 1 (1998), 292–293.

[22] Diego Garlaschelli, Stefano Battiston, Maurizio Castri, Vito Servedio, and Guido Caldarelli. 2004. The scale-free topology of market investments. http://goo.gl/h7tVVK. http://goo.gl/h7tVVK

[23] James B Glattfelder. 2010. *Ownership networks and corporate control: mapping economic power in a globalized world.* Ph.D. Dissertation. ETH Zurich.

[24] Georg Gottlob and Andreas Pieris. 2015. Beyond SPARQL under OWL 2 QL Entailment Regime: Rules to the Rescue. In *IJCAI.* 2999–3007.

[25] Paul Graham. 2003. Better bayesian filtering. In *Proceedings of Spam Conference*, Vol. 10 (8). 707–710.

[26] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *KDD.* ACM, 855–864.

[27] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and emerging applications: an interactive tutorial. In *SIGMOD.*

[28] Seyed Mehran Kazemi and David Poole. 2018. SimplE Embedding for Link Prediction in Knowledge Graphs. In *NeurIPS.* 4289–4300.

[29] David Liben-Nowell and Jon M. Kleinberg. 2003. The link prediction problem for social networks. In *CIKM.* ACM, 556–559.

[30] Maximilian Marx, Markus Krötzsch, and Veronika Thost. 2017. Logic on MARS: Ontologies for Generalised Property Graphs. In *IJCAI 2017.* 1188–1194.

[31] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. 1990. The magic of duplicates and aggregates. In *VLDB.* 264–277.

[32] Enrico Palumbo, Giuseppe Rizzo, Raphaël Troncy, Elena Baralis, Michele Osella, and Enrico Ferro. 2018. Knowledge Graph Embeddings with node2vec for Item Recommendation. In *ESWC (Satellite Events) (Lecture Notes in Computer Science)*, Vol. 11155. Springer, 117–120.

[33] Tiago Pimentel, Adriano Veloso, and Nivio Ziviani. 2018. Fast Node Embeddings: Learning Ego-Centric Representations. In *ICLR (Workshop).* OpenReview.net.

[34] Cesar A. Hidalgo R. and Albert-László Barabási. 2008. Scale-free networks. *Scholarpedia* 3, 1 (2008), 1716.

[35] Nicoleta Rogovschi, Jun Kitazono, Nistor Grozavu, Toshiaki Omori, and Seiichi Ozawa. 2017. t-Distributed stochastic neighbor embedding spectral clustering. In *IJCNN.* IEEE, 1628–1632.

[36] Andrea Romei, Salvatore Ruggieri, and Franco Turini. 2015. The layered structure of company share networks. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA).* IEEE, 1–10.

[37] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. 2015. Optimizing recursive queries with monotonic aggregates in DeALS. In *ICDE.* 867–878.

[38] Leslie G. Valiant. 1979. The complexity of enumeration and reliability problems. In *Journal on Computing*, Vol. 8. SIAM, 410–421.

[39] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *NeurIPS.* 5171–5181.

Expanding Query Answers on Medical Knowledge Bases

Chuan Lei
IBM Research - Almaden
chuan.lei@ibm.com

Vasilis Efthymiou
IBM Research - Almaden
vasilis.efthymiou@ibm.com

Rebecca Geis*
IBM Germany
rebecca.geis@de.ibm.com

Fatma Özcan
IBM Research - Almaden
fozcan@us.ibm.com

ABSTRACT

Medical knowledge bases (KBs) are known to be vital for tasks like clinical decision support and medical question answering, since they provide well-structured relational information between entities. One of the main challenges for querying a medical KB is the mismatch between the terms in the KB and the colloquial and imprecise terminology used in user queries. To address this challenge, we propose a domain-specific query relaxation approach that leverages rich medical domain vocabularies and their semantic relationships from external knowledge sources, such as taxonomies, ontologies, and semantic networks, to expand the vocabulary of KBs. Our main goal is to expand both the set of queries that we can answer, as well as the set of answers to the queries, over the medical KB. We introduce a lightweight adaptation method to customize and incorporate external knowledge sources to work with the existing KB, and propose a novel similarity metric to leverage the information content in the KB, the structural information in the external knowledge source, and the contextual information from user queries. We implement our proposed techniques for a medical KB, and use SNOMED CT as the external knowledge source. We experimentally demonstrate the effectiveness of our proposed method and the improved quality of query results in terms of both precision and recall, compared to state-of-the-art approaches. Finally, we conduct user studies to evaluate how much a conversational interface can benefit from our proposed method in terms of its query capability on the medical KB.

1 INTRODUCTION

Medical knowledge bases (KBs) provide structured information about medical entities (such as drugs and diseases) and their relationships, which are invaluable in medical applications. Such KBs are often created from medical information sources, including medical literature, patient data, claims data, etc, and offer deep domain specialization with rich and detailed information, which is known to be vital for domain-specific tasks like clinical decision support and medical question answering. The medical KBs are different from cross-domain large-scale KBs such as DBpedia [5] and Freebase [9] which provide well-structured, encyclopedic knowledge but with less detail and precision.

When querying medical KBs, the users do not always formulate their queries precisely to match the terms in the KB, especially when they use natural language. For example, users are likely to use informal words, phrases, or abbreviations of certain terms in their natural language queries, which makes matching the mentioned entities to the medical KB a non-trivial task. Query relaxation [18] is one of the most prominent techniques used for query answering, allowing more domain-specific terms in user queries. Instead of returning no or incomplete answers, query relaxation can transform the query in a way that the user's intent is better represented, greatly improving the flexibility and usability of a medical KB.

The problem of query relaxation has been extensively studied in information retrieval and database systems with the goal of returning information beyond what is specified by a standard query [17, 26]. However, the techniques, traditionally designed for formally defined query languages such as SQL, cannot handle the complexity from natural language queries that involve complex semantic constraints and logic [37, 43]. Hence, they often fail to ensure query answering with high precision and recall. Recent work [3, 8, 14] demonstrated that deep learning models built at word or sentence level can be used for semantic similarity estimation. However, these methods demand high-quality training data, which is critical and expensive in reality.

In this paper, we focus on a medical KB (*MED*) which contains medication, disease and toxicology information to support informed diagnosis and treatment decisions for evidence-based clinical decisions and patient education. We observe that this and similar medical KBs can be further enriched by external knowledge sources, such as medical ontologies, taxonomies, and semantic networks (e.g., Unified Medical Language System [40], SNOMED Clinical Terms [38], and Gene Ontology [12]). These knowledge sources can be exploited by query relaxation to expand query answers.

We introduce a novel query relaxation method that leverages rich domain vocabularies and their semantic relationships from external medical knowledge sources, which largely consist of subsumption relationships (e.g., $A \sqsubseteq B$, where A and B are concepts in the external knowledge source). We first find the concept corresponding to a given query term in the external knowledge source, and then relax the term by exploring the concept's neighborhood to identify semantically related concepts.

The rich domain vocabulary and structural information of external knowledge sources empower query relaxation to generalize or specialize query terms beyond syntactic matching. However, external knowledge sources such as SNOMED CT are often not customized to the application's requirements. Using external knowledge sources without proper adaptation may introduce semantically unrelated information into the results, leading to low precision and recall. To provide high-quality results, a query relaxation method has to address the following challenges.

External knowledge source ingestion. External knowledge sources are often comprehensive, consisting of an excessive amount of information describing a domain. The given KB is often substantially smaller than the external knowledge sources.

*Rebecca Geis is currently a graduate student at Technische Universität Darmstadt.

This makes it challenging to identify semantically related results from the external knowledge source. For example, given a query *"what drugs treat pertussis"*, there might be no drug directly associated with *"pertussis"* in the given KB. Instead, a generalized clinical finding, *"bronchitis"*, in the KB has corresponding drug information. However, the distance (i.e., the number of hops) between *"pertussis"* and *"bronchitis"* in SNOMED CT is large, making it difficult for query relaxation to identify the semantic similarity between the two terms. Worse yet, many findings closer in distance but not semantically related might be returned as well, causing low-precision results.

Exploiting the query context. Contextual information has significant impact on the semantic correctness of the relaxed results. For example, a user may ask *"what drugs treat psychogenic fever"*, in which case the context is *"treatment"* and *"psychogenic fever"* is a query term in the given medical KB. This term appears in SNOMED CT as the name of a clinical finding, with both *"hyperpyrexia"* and *"hypothermia"* being similar findings. However, in the context of *"treatment"*, drugs for *"hypothermia"* should not be returned, as *"hypothermia"* is the opposite of *"hyperpyrexia"* and *"psychogenic fever"*.

To address these challenges, we propose a novel two-phase query relaxation method, consisting of external knowledge source ingestion phase and the online query relaxation phase. We implemented our techniques for the medical KB (*MED*), and used SNOMED CT as the external knowledge source. The main contributions of this paper are as follows.

• We present a lightweight, yet effective offline ingestion process that customizes the external knowledge source to the given KB.

• We propose a novel similarity metric to identify semantically related concepts, leveraging (*i*) the information content in the KB, (*ii*) the structural information in the external knowledge source, and (*iii*) the contextual information from the user query.

• We introduce a programmatic way to incorporate our method into two state-of-the-art systems, a conversational system [21] and a natural language query (NLQ) system [23, 35] for the medical KB, using SNOMED CT as the external knowledge source.

• Our experiments show that our query relaxation method for the medical KB outperforms state-of-the-art methods, including deep learning-based ones, in precision and recall. We also conduct a user study demonstrating how our query relaxation method improves the response quality of a conversational system.

Outline. The rest of the paper is organized as follows. In Section 2, we introduce the basic concepts used in this paper, and in Section 3, we provide an overview of our query relaxation approach. Section 4 describes context generation, extraction, and management. Section 5 introduces our query relaxation method in detail. We explain how to integrate our query relaxation technique into two natural language interface systems in Section 6, and provide experimental results in Section 7. We review related work in Section 8, and conclude in Section 9.

2 BACKGROUND

2.1 Knowledge Base

Following the standard notation of description logic [6], we assume that a KB is given in the form of TBox and ABox. In this paper, TBox is referred to as *domain ontology* and ABox is referred to as *instances* or data.

The domain ontology describes the concepts relevant to the domain, and the relationships (roles) among different concepts. The concepts associated with a relationship are provided by the domain (i.e., source) and range (i.e., destination) constraints of this relationship. The *context* of a query term used in a query can be represented by a relationship and its associated concepts from the domain ontology.

Figure 1 shows a fragment of a sample medical domain ontology. The concept *"Finding"* connects to both concepts *"Indication"* and *"Risk"* through the relationship *"hasFinding"*. This shows that *"Finding"* can be potentially used in two different contexts (i.e., *Risk-hasFinding-Finding* and *Indication-hasFinding-Finding*). Two example queries could be *"which drugs have the risk of **causing** diabetes"* and *"which drugs **treat** diabetes"*, where the query term is *"diabetes"*.

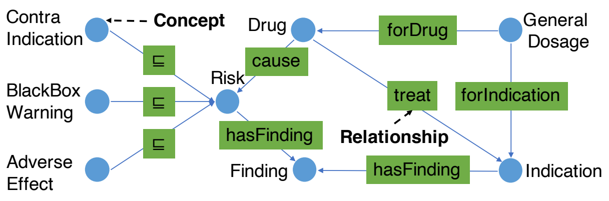


Figure 1: Snippet of a medical ontology.

The instances (data) of the given KB are stored separately for query answering as shown in Figure 3. For example, *"fever"* and *"renal impairment"* are two instances of *"Finding"*. We assume that our input is in the form of a query term and its associated context. Following the previous example, the input to our query relaxation method would be [diabetes, *Risk-hasFinding-Finding*] or [diabetes, *Indication-hasFinding-Finding*]. Recent technologies [1, 21, 28] have been designed to extract the contextual information from natural language questions. In this paper, our method is integrated with Watson Assistant [21] to receive the contextual information, and finds semantically related instances for a given query term with high precision and recall. We provide more details on how we bootstrap the conversation space in Section 4.

2.2 External Knowledge Source

In this work, we utilize the rich medical domain vocabularies and their semantic relationships from external knowledge sources such as ontologies, semantic networks, and knowledge graphs. In particular, we are interested in the subsumption relationships in the form of $A \sqsubseteq B$, where A and B are concepts in the external knowledge source. In this case, we say that A *specializes* B, and that B *generalizes* A. We refer to the direct and implied (by transitivity) specializations of a concept A, excluding A, as the *descendants* of A. We assume that the external knowledge source is a directed acyclic graph (DAG), in which a top concept (owl:Thing in OWL) is the root and every concept is a descendant of the root. To avoid confusion with the concepts of the domain ontology, we refer to the concepts in the external knowledge source as *external concepts*.

2.3 Semantic Similarity Measures

Semantic similarity measures estimate the similarity between concepts, and are commonly used in various processing tasks (e.g., entity resolution [10, 15], link prediction [27], change detection [22]). The knowledge-based approach to semantic similarity exploits taxonomies like WordNet. Typically, path finding measures and information content (IC) measures are two common categories in knowledge-based approaches [19]. In addition to a knowledge source, the IC approach can leverage a frequency value $freq(A)$, accounting for the number of times a concept A is mentioned in a document corpus, to compute the similarity between concepts. Specifically, the IC of a concept A is defined as the inverse of the log of the concept's frequency [25, 34]:

$$IC(A) = -log(freq(A)), \tag{1}$$

where $freq(A)$ is recursively defined as:

$$freq(A) = |A| + \sum_{A_i \sqsubseteq A} freq(A_i), \tag{2}$$

with $|A|$ being the number of times concept A is directly mentioned in the document corpus, and A_i being the direct descendants of A in the taxonomy. The intuition is that the more general a concept is, the more likely it is that the concept or its descendants appear in the corpus. We describe how we compute this equation in the next section.

The IC-based similarity measure compares the IC of a pair of concepts to the IC of their Least Common Subsumer (LCS)[1]. The greater the IC of the LCS (i.e., the more specific the LCS), the more similar is the pair of concepts:

$$sim_{IC}(A, B) = \frac{2 \times IC(lcs(A, B))}{IC(A) + IC(B)}. \tag{3}$$

In general, the IC similarity measure is shown to outperform other approaches on various semantic similarity benchmarks [2, 19, 29]. Hence we adopt the above IC similarity measure and further integrate it with the structural information in the external knowledge source, as well as the contextual information from the natural language query.

3 APPROACH OVERVIEW

In this section, we provide an overview of our query relaxation method, as shown in Figure 2. We propose a two-phase approach: an offline phase, in which we construct context specifications, as well as incorporate an external knowledge source into the given KB, and an online phase, in which we take a query term associated with a context, and return the semantically related results as answers for the query.

Offline phase. In the offline phase, also called external knowledge source ingestion, we perform the following tasks: (i) we initialize a set of possible contexts based on the domain ontology, and optionally generate training examples for context classification (if needed by natural language interface (NLI) system), (ii) we compute the frequency of each external concept in the external knowledge source with respect to the associated contexts, and (iii) we generate mappings between instance data in the knowledge base and external concepts in the external knowledge source.

To initialize the set of possible contexts, we traverse the domain ontology and return all the relationships, along with their source and destination concepts. Those relationships constitute the set of possible contexts, which we provide to the NLI system. We can also provide labeled data for training a context classifier in the NLI system if required (described in Section 4).

To compute the frequency of each external concept, we leverage the document corpus from which the given knowledge base

[1]A LCS of two concepts always exists in the external knowledge source. When multiple LCSs exist, we choose the one with the shortest path to the pair of concepts. If multiple LCSs have equal distance to the pair of concepts, we use the average IC of these LCSs for the similarity measure.

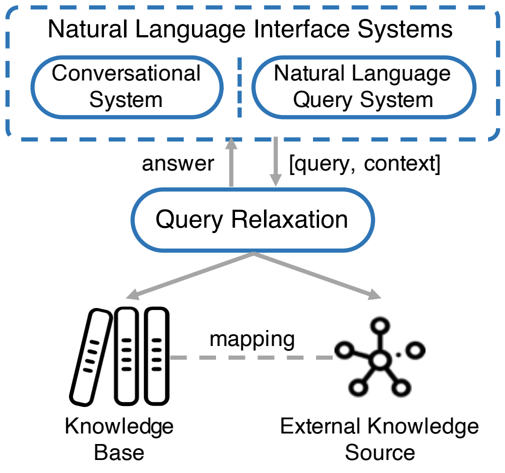

Natural Language Interface Systems
Conversational System
Natural Language Query System
answer
[query, context]
Query Relaxation
mapping
Knowledge Base
External Knowledge Source

Figure 2: Approach overview.

is curated. Additionally, we differentiate the frequency of the external concepts with respect to different contexts, as described in Section 5.1.

To map the instance data from the given KB to the external concepts in the external knowledge source, we provide three alternative methods, depending on the accuracy requirement from the application. Specifically, these methods include matching the instance data and external concepts with exactly the same names (exact match), very similar names in terms of edit distance, or similar names in terms of word embeddings, as described in Section 7.2.

Online phase. In the online phase, also called online query relaxation, we receives as input a [query term, context] pair and perform the following tasks: (i) we search for an external concept Q corresponding to the query term in the external knowledge source, and (ii) we retrieve the top-k similar external concepts having corresponding matching instances in the KB to Q as the answers.

To identify an external concept Q that corresponds to the input query term, we follow a similar process used in the offline external knowledge source ingestion: we identify Q as the external concept whose name either matches with the exact query term, or is very similar in terms of either edit distance or word embeddings. Additionally, several knowledge sources (e.g., SNOMED CT[2], DrugBank[3], DBpedia[4]) may offer a more sophisticated lookup service, which we can also utilize to find such mappings.

Finally, having identified the external concept Q that corresponds to the input query term, we retrieve its top-k similar external concepts that have corresponding instances in the given KB. For the similarity computation, we leverage the information content from the KB, the structural information in the external knowledge source, as well as the contextual information from the query, as described in Section 5.2.

[2]https://browser.ihtsdotools.org/
[3]https://www.drugbank.ca
[4]https://github.com/dbpedia/lookup

4 CONTEXT SPECIFICATION

As explained in Section 2.1, the context can be represented by a relationship and its associated concepts from the domain ontology. In this section, we provide a brief overview of: (*i*) how our method provides the necessary information that an NLI system requires for context recognition, and (*ii*) how the conversational context differs from simple question answering. We note that the process of context recognition is orthogonal to our method, and we refer the reader to [33] for more details.

Context generation and extraction. Context reflects the intent or goal expressed in the user query/input[5]. NLI systems typically use a learning-based model to identify the intent for a given user query within the current conversation. As a consequence, most of these systems require as input the specification of all possible contexts expected in a given workload with labeled query examples for training the intent classifier. These contexts are usually based on (*i*) the purpose of the application and the scope of questions that it intends to handle, (*ii*) the anticipated set of questions that the users might ask within the scope of the application.

To feed such NLI system with training data, we need to follow a two-step process. The first step is to generate all possible contexts, based on the domain ontology. For this step, we traverse the domain ontology and extract all relationships, along with their associated concepts, i.e., their source and destination concepts. Since a context can be represented by a relationship, we define the set of possible contexts (i.e., possible labels for training data) as the set of relationships.

The second step is to associate a query workload to the generated contexts. There are different options for this step, which go beyond the scope of this work. One simple approach is to retrieve an existing query workload, and ask domain experts to label each query in the workload with the most relevant context. Once we have such an annotated query workload, we can either stop the process here, or we can further enrich the query workload. For enriching the query workload, we can replace identified instances with other instances of the same concept. For example, we can generate more queries in our workload from a given query *"what drugs treat fever"*, labeled with the context *Indication-hasFinding-Finding*), by replacing *"fever"* with other instances of *"Finding"*, such as *"headache"*, *"sore throat"*, and *"pain in throat"*.

The result of this two-step process is a set of queries, each labeled with a context, which we can provide to a NLI system as training data for context recognition.

Context management. In Figure 2, we show two alternative NLI systems that can benefit from our query relaxation method. The main difference between a conversational system and a natural language query system is that the latter can be stateless. Namely, a conversational system needs to keep track of the conversational flow, i.e., the state of the dialogue and its history. This way, the current context can be inferred from the previous state, even if not explicitly mentioned in the current query. For example, if the current query is *"what about fever?"*, there is no clear context, if this query is processed individually. However, if it is processed as part of a conversation, in which the previous query was *"which drugs treat diabetes"*, then a conversational system can infer that the previous context *Indication-hasFinding-Finding*) remains unchanged. More details on that subject can be found in [23]. On the other hand, a natural language query system typically handles contexts in one-shot queries without considering previously asked queries.

[5]We use the terms context and intent interchangeably in this section.

5 QUERY RELAXATION METHOD

Our query relaxation method has two phases, the offline external knowledge source ingestion and the online query relaxation. In this section, we first focus on describing how to customize and incorporate an external knowledge source into the given KB, and then we show how to use the adapted knowledge sources in online query relaxation.

5.1 External Knowledge Source Ingestion

The external knowledge source ingestion addresses the following two issues. First, we need to count the frequency $freq(A)$ of an external concept A (Equation 2) based on the corpus from where the given medical KB is curated. Its frequency should reflect the context in which the concept is used. Second, the external knowledge source typically contains an excessive amount of information in terms of domain vocabulary and relationships. It is imperative to customize and incorporate the external knowledge source in accordance with the given KB for query relaxation.

Concept frequency. First, we need to map the instances from the given KB to their corresponding external concepts in the external knowledge source, as illustrated in Figure 3. A variety of techniques can be leveraged to produce such mappings, ranging from exact string matching, approximate string matching using edit distance, to word embeddings. In this paper, we use these techniques in a pluggable fashion, and we compare the effectiveness of these algorithms in the experimental evaluation. If an instance is mapped to an external concept, then this concept is marked with a flag (the concepts in yellow in Figure 3). The online query relaxation only returns flagged concepts as semantically related results to a given query term, since the given KB only contains information about those concepts.

Next, as mentioned earlier, a concept could be used in different contexts depending on the natural language query, and the semantic meaning can be completely different in different contexts. For example, a condition treated by a drug is different from an adverse effect (i.e., condition) caused by the same drug. In this case, having a single frequency associated with a concept (Equation 2) would not be sufficient to capture the semantic differences over all possible contexts.

To resolve this issue, we identify all the contexts where an external concept A can be used. Specifically, we use the relationships associated to a concept in the domain ontology as the contexts of A, if an instance of this concept is mapped to A. Then, we compute the concept frequency with respect to each context. The online query relaxation phase chooses the appropriate concept frequency according to the query context as described later in this section.

To compute the frequency of external concepts, we assume that the KB is curated based on a document corpus, and we count the number of times that each external concept name is mentioned within this corpus. To account for the sparsity of certain concept names in the corpus, the concept frequency is further adjusted based on the number of documents in which the concept name appears. For example, *"asthma"* is mentioned in 54 drug descriptions in DrugBank [16], whereas *"lung cancer"* has only a handful of associated specialty drugs. Hence, we utilize the commonly used tf-idf weighting to alleviate this bias.

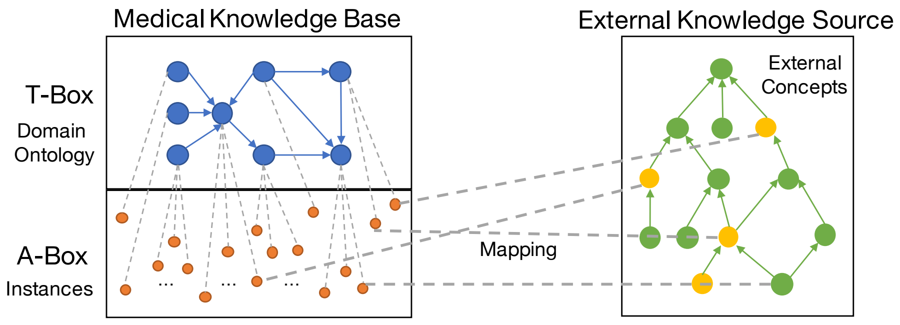


Figure 3: External knowledge source ingestion.

Example 1. In the medical ontology depicted in Figure 1, the concept *"Finding"* is connected to both *"Indication"* and *"Risk"*. In this case, the external concepts associated with instances of *"Finding"*, have two concept frequencies corresponding to the contexts *"Indication-hasFinding-Finding"* and *"Risk-hasFinding-Finding"*. Depending on the query context, one of the concept frequencies is used in query relaxation. In Figure 4, we show a snippet of SNOMED CT with the external concept frequency populated. The external concepts in brackets are all mapped from different instances of *"Finding"* in the domain KB, so they can be used in two different contexts (*"Indication-hasFinding-Finding"* and *"Risk-hasFinding-Finding"*). Hence, they are associated with two concept frequencies. For example, *"headache"* is the only direct descendant of *"craniofacial pain"*, and the frequency of *"craniofacial pain"* is the frequency of itself, together with the one of *"headache"*. Accordingly, the frequency of *"pain of head and neck region"* is a summation of the frequencies of *"craniofacial pain"*, *"pain in throat"*, and itself, which is 19164 (i.e., 18878 + 283 + 3) in the context of *"Indication-hasFinding-Finding"* and 1656 in the context of *"Risk-hasFinding-Finding"*.

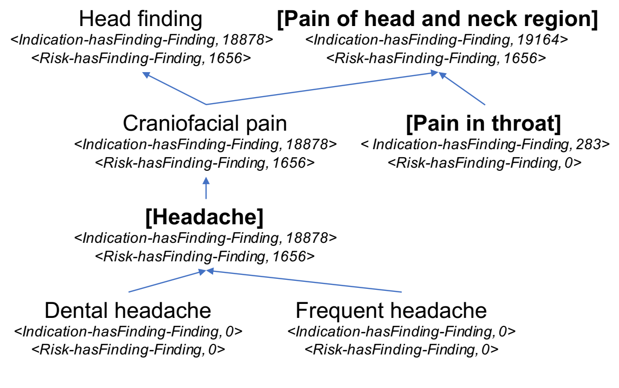

Head finding
<Indication-hasFinding-Finding, 18878>
<Risk-hasFinding-Finding, 1656>
[Pain of head and neck region]
<Indication-hasFinding-Finding, 19164>
<Risk-hasFinding-Finding, 1656>
Craniofacial pain
<Indication-hasFinding-Finding, 18878>
<Risk-hasFinding-Finding, 1656>
[Pain in throat]
< Indication-hasFinding-Finding, 283>
<Risk-hasFinding-Finding, 0>
[Headache]
<Indication-hasFinding-Finding, 18878>
<Risk-hasFinding-Finding, 1656>
Dental headache
<Indication-hasFinding-Finding, 0>
<Risk-hasFinding-Finding, 0>
Frequent headache
<Indication-hasFinding-Finding, 0>
<Risk-hasFinding-Finding, 0>

Figure 4: Snippet of SNOMED CT with frequencies.

Finally, all of these frequencies are normalized between [0, 1], which corresponds to the probability of a concept appearing in the corpus. The root concept has the highest normalized frequency of 1, because all concepts in the external knowledge source are its descendants.

Sparsity of external knowledge source. The commonly used external knowledge sources, such as SNOMED CT, are comprehensive, consisting of an excessive amount of information describing the domain. They often consist of rich domain vocabularies associated with deep hierarchies. On the contrary, the given KB is often relatively smaller compared to the external knowledge source. Therefore, only a small subset of the external concepts may have corresponding instances in the KB. Any pair of concepts could be connected through multiple intermediate ones, which makes finding semantically related concepts prohibitively time-consuming for an online system. One straightforward approach would be computing the pairwise similarity between all concepts offline. However, this leads to unnecessary computations and space consumption, since most of these precomputed similarities may not even be used during the online query relaxation.

In the offline ingestion phase, we alleviate the above issue by introducing additional application-specific edges to the external knowledge source. Specifically, an additional directed edge is introduced from an external concept A to an external concept B, if all the following conditions are satisfied: (1) A and B are not directly connected (i.e., one-hop neighbors), (2) A is a descendant of B, and (3) at least one of the two concepts has a corresponding instance in the given KB. Consequently, they become one-hop neighbors with respect to the application. The distance between two external concepts is attached to the new edge so that the original semantic information between two concepts is preserved. This way, external concepts come closer, avoiding unnecessary delays in the online query relaxation phase.

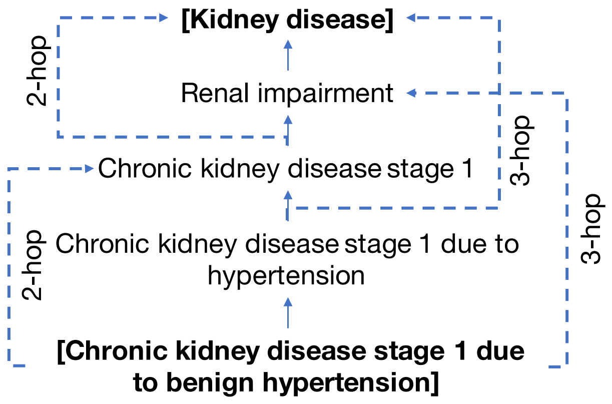

[Kidney disease]
Renal impairment
Chronic kidney disease stage 1
Chronic kidney disease stage 1 due to hypertension
[Chronic kidney disease stage 1 due to benign hypertension]
2-hop
2-hop
3-hop
3-hop

Figure 5: External knowledge source customization.

Example 2. In Figure 5, *"chronic kidney disease stage 1 due to hypertension"* is 3 hops away from *"kidney disease"*, which has a corresponding instance in the KB. By introducing an additional edge (the dashed line) between these two external concepts, they are only 1 hop away. Therefore, more semantically related concepts are within a close distance, and the semantic similarity between two external concepts remains unchanged since the original path information (3-hop) between them is attached to the new edge.

Overall, the offline external knowledge source ingestion process is summarized in Algorithm 1. The algorithm receives as input the medical KB (as domain ontology O and instances I) and the external knowledge source EKS, and it returns the set of possible contexts C, the frequencies F of the external concepts for each context, the mappings M from instances to external concepts, and the set of external concepts that are marked with a flag FEC. The algorithm consists of 3 almost independent procedures: context generation, mappings, and concept frequency. Additionally, for efficiency, sparsity of external knowledge source is also handled in the same loop as concept frequency, even though one does not depend on the other.

In Lines 1-4, we create the set of possible contexts, based on the domain ontology's relationship, along with their domains (source concepts) and ranges (destination concepts). In Lines 5-11, we find an external concept A as a mapping for every instance $i \in I$, if such exists, based on a chosen mapping function, and return the set of mappings M. At the end of this loop, the set FEC contains all the external concepts that have been marked with a flag, i.e., all the external concepts that have a corresponding instance in I. In Line 12, we sort the external concepts in topological order, such that the descendants precede their ancestors (note that EKS is a DAG). This way, we can easily compute the frequency of each external concept for a given context (Lines 14-18), using the recursive function of Equation 2. Since the external concepts are already topologically sorted, we add application-specific edges to alleviate the sparsity of EKS in the same loop (Lines 19-23). Specifically, in Line 21, we add an edge from external concept A to external concept B, when all three conditions (previously described) are satisfied, while attaching their original distance (as $|shortestPath(A, B)|$) to the new edge.

Time complexity analysis. The time complexity of external knowledge source ingestion can be broken down into the following parts. First, creating all possible contexts requires iterating through all relationships ($|R|$) in the domain ontology. Hence the time complexity is $\Theta(|R|)$. Second, the time complexity of finding an external concept A for every instance depends on the chosen method. For example, using word embeddings to find mappings requires $\Theta(|I| \cdot Cost(lookup))$, where $|I|$ denotes the total number of instances and $Cost(lookup)$ denotes the constant cost of embedding lookup for each instance. If an approximate string matching algorithm is used, then the time complexity becomes $O(|I| \cdot mn)$, where $O(mn)$ denotes the time complexity of typical approximate string matching algorithm (m and n are the lengths of two strings). Third, topological sorting of all external concepts requires $O(|V| + |E|)$ time complexity, assuming that V is the set of concepts and E is the set of relationships in the external knowledge source. Fourth, the time complexity of computing frequency of each external concept for all possible context is $O(|V| \cdot AVG(contexts))$, where $AVG(contexts)$ denotes the average number of contexts per concept in the domain ontology. Regarding adding application-specific edges to EKS, we ignore its time complexity since the number of concepts satisfying is much less than $|V|$. In summary, the total time complexity is $\Theta(|R|) + \Theta(|I| \cdot Cost(lookup)) + O(|V| + |E|) + O(|V| \cdot AVG(contexts))$. Note that the external knowledge source ingestion is an offline process that is executed only once.

Algorithm 1 Knowledge Source Ingestion Algorithm.

```
Input: Domain ontology $O$, Instances $I$, External Knowledge
       Source $EKS$
Output: (Set of contexts $C$), External concept frequencies $F$,
       Mappings $M$, Flagged external concepts $FEC$
    ▷ Context generation
 1: $C \leftarrow \emptyset$
 2: for each $r \in Relationships(O)$ do
 3:     $C \leftarrow C \cup \{(domain(r), r, range(r))\}$
 4: end for
    ▷ Mappings
 5: $M \leftarrow \emptyset$
 6: $FEC \leftarrow \emptyset$                          // Flagged external concepts
 7: for each $i \in I$ do
 8:     $A \leftarrow mapping(i, EKS)$  // map $i$ to an external concept $A$
 9:     $M \leftarrow M \cup \{(i, A)\}$
10:     $FEC \leftarrow FEC \cup \{A\}$
11: end for
    ▷ Concept frequency
12: $Q \leftarrow topol.Sort(Concepts(EKS))$  // children before parents
13: $F \leftarrow \emptyset$                            // Frequencies wrt. context
14: while $Q$ is not empty do
15:     $A \leftarrow Q.next()$
16:     for each $c \in C$ do
17:         $F \leftarrow F \cup \{(A, c, freq(A))\}$          // see Equation 2
18:     end for
        // External knowledge source customization
19:     for each $B \in ancestors(A, EKS) \setminus parents(A, EKS)$ do
20:         if $A \in FEC$ or $B \in FEC$ then
21:             $EKS.addEdge(A, |shortestPath(A, B)|, B)$
22:         end if
23:     end for
24:     $Q.remove(A)$
25: end while
26: return $C, F, M, FEC$
```

5.2 Online Query Relaxation

Given a query term, the goal of online query relaxation is to identify the semantically related instances contained in the given KB by leveraging the external knowledge source. In this subsection, we present a novel similarity measure that leverages the information content from the medical KB, the structural information in the external knowledge source, as well as the contextual information from the query.

Contextual information. As described earlier, the possible contexts of a query term mapped to a concept in the domain ontology, are the relationships of the concept to its adjacent concepts. With the contextual information, the online query relaxation phase can choose the appropriate concept frequency to use in Equation 2.

Example 3. For the query *"what are the risks of using aspirin"*, the context is *"Drug-cause-Risk"*. As shown in Figure 1, *"Risk"* has three descendants *"Black Box Warning"*, *"Adverse Effect"*, and *"Contra Indication"*. Assuming that the query term *"aspirin"* cannot be found in the given KB, then the online query relaxation would consider related conditions in the context of *"Drug-cause-Risk"*. Hence, the concept frequency used for the similarity measure should be the total frequency of all three descendants of *"Risk"*.

In case the contextual information is not available for online query relaxation, our method can aggregate the frequencies (i.e., all possible contexts) associated with an external concept. As verified in the experimental evaluation section, the contextual information greatly improves the quality of the results.

Structural (Path) information. As described above, external knowledge sources contain generalization and specialization relationships between concepts. Generalizing the query term in a user query may cause information loss [24]. For example, as shown in Figure 4, *"headache"* can be generalized to *"craniofacial pain"*, which can in turn be generalized as *"pain of head and neck region"* including *"pain in throat"*. Apparently, *"pain in throat"* no longer describes pain in head, even if it is as close to *"craniofacial pain"* as *"frequent headache"*.

In this case, solely relying on the IC similarity measure (Equation 3) with contextual information can be insufficient as it cannot differentiate the semantic difference between specialization (i.e., the opposite direction of subsumption edges in the external knowledge source) and generalization (i.e., following the direction of subsumption edges). We tackle this challenge by assigning different weights to generalization and specialization in the external knowledge source. The weight of a path connecting two external concepts A and B is thus computed as:

$$p_{A,B} = \prod_{i}^{|D|} w_i^{D-i}, \tag{4}$$

where the distance between external concepts A and B is $|D|$, and w_i indicates the weight of the i-th edge from A to B. The intuition is that we penalize a generalization and the penalty is more if it appears early on in a path from A to B. In fact, such distinction helps us to better capture the semantic similarity between a pair of concepts based on their relative locations in the external knowledge source.

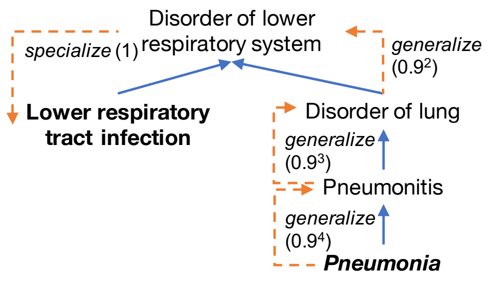
To learn the weights of both generalization and specialization, simple statistical regression analysis [7] such as logistic regression can be used. In our empirical study, the weights of generalization and specialization are set to 0.9 and 1, respectively.

Example 4. In Figure 6, the penalty associated with the path (dashed orange lines) connecting two external concepts can be different, depending on which concept corresponds to the query term. In this example, there are 4 hops between *"pneumonia"* and *"lower respiratory tract infection"*. If the query term is *"pneumonia"*, it would be penalized more as the first 3 hops in the path starting from *"pneumonia"* to *"lower respiratory tract infection"* are all generalizations (Figure 6(a)). On the other hand, if the query term is *"lower respiratory tract infection"*, it only suffers from one generalization at the beginning (Figure 6(b)).

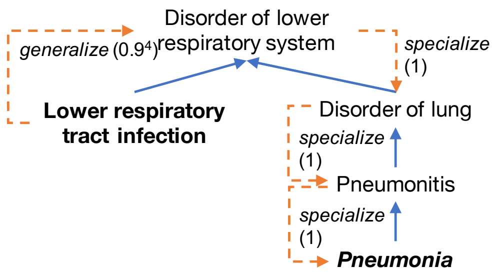
Putting it all together. Overall, the online query relaxation uses a novel similarity measure, which takes as input the information content, the contextual information, and the structural information to find semantically related concepts.

$$sim(A, B) = p_{A,B} \times sim_{IC}(A, B). \tag{5}$$

For a given query term, the query relaxation method first finds the corresponding external concept A in the external knowledge source. Then, it searches for the concepts within a distance r from A. Last, our method retrieves the pre-computed similarity between A and each external concept in its neighborhood. Top-k relaxed results are returned based on their similarity scores, where k is application-specific and defined by users. The radius r can be set in different ways. Namely, it can be set as a fixed value by empirical studies, or dynamically decided if a fixed r cannot provide k results.



(a) Path 1: From Pneumonia (query term) to Lower respiratory tract infection.



(b) Path 2: From Lower respiratory tract infection (query term) to Pneumonia.

Figure 6: Example of paths between two external concepts.

Overall, the online query relaxation process is summarized in Algorithm 2. The algorithm receives as input a query term q, along with its associated context c, the instances I from the given KB, the external knowledge source EKS, the set of external concepts that are marked with a flag FEC, the mappings M from I to EKS concepts, the radius r and an integer k, and it returns the top-k results Res from I.

Algorithm 2 Online Query Relaxation Algorithm.

```
Input: Query term $q$, Context $c$, Instances $I$, External Knowledge
    Source $EKS$, Flagged external concepts $FEC$, Mappings $M$,
    radius $r$, integer $k$
Output: Top-$k$ results $Res \subseteq I$
1: $A \leftarrow mapping(q, EKS)$ // concept $A$ corresponds to $q$ in $EKS$
   ▷ candidates are flagged concepts within radius $r$ from $A$
2: $Candidates \leftarrow neighbors(A, EKS, r) \cap FEC$
3: $sort(Candidates, sim(A, B))$          // Equation 5 for context $c$
4: $Res \leftarrow \emptyset$
5: while $|Res| \leq k$ and $|Candidates| > 0$ do
6:     $B \leftarrow Candidates.pop()$ // get next element and remove it
7:     $Res \leftarrow Res \cup \{i|(i, B) \in M\}$
8: end while
9: return $Res$
```

In Line 1, we retrieve the external concept A that corresponds to the query term q, using the same mapping function as in Algorithm 1. Then, we get the set of candidate external concepts within radius r from A, which are marked with a flag, i.e., members of FEC (Line 2), and we sort them in descending order of similarity to A (Line 3). Finally, we iterate through the sorted candidates and add to the results Res the instances i that map to those candidates, until k results have been retrieved, or there are no more candidates (Lines 5-8).

Time complexity analysis. For online query relaxation, we again assume that V is the set of concepts in the external knowledge source and the total number of flagged external concepts FEC is N. Then finding a corresponding external concept A corresponding to the query term q requires $O(|V|)$ time in the worst case. Returning all flagged external concepts within radius r from A requires $O(N)$ time in the worst case (i.e., r is large enough to include all flagged external concepts). Sorting these candidates and returning the top-k results take $\Theta(NlogN)$ and $\Theta(k)$ time, respectively. Hence, the total time complexity of online query relaxation is $\Theta(NlogN)$.

6 APPLICATIONS OF QUERY RELAXATION

Our proposed method is applicable and beneficial to various natural language interface systems, including conversational systems, question and answer systems, as well as natural language query systems to KBs. In this section, we describe how to integrate the query relaxation method with two state-of-the-art systems for the medical data set *MED*, that is used to support evidence-based clinical decisions and patient education, and SNOMED CT as the external knowledge source, since it is one of the most comprehensive and widely used medical knowledge sources.

6.1 Integration with a Conversational System

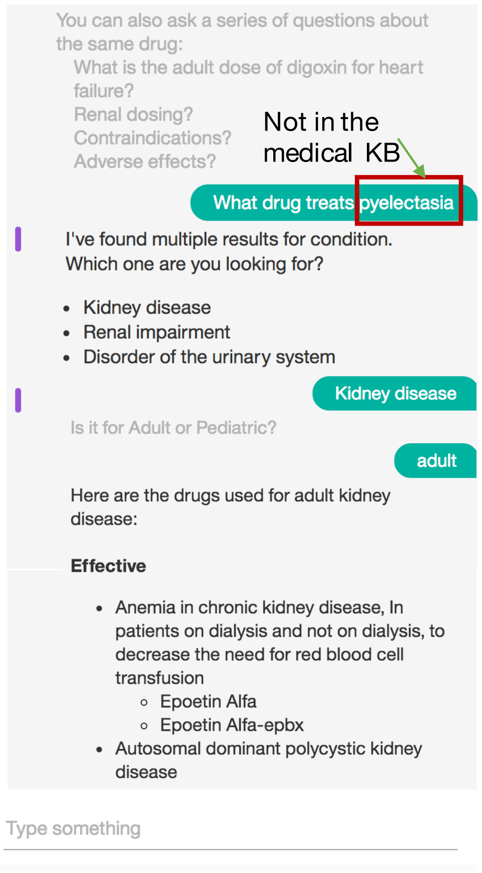
In the following, we describe how we extended a conversational system [33] that is built on top of IBM Watson Assistant [21], to include query relaxation. The query relaxation method is implemented in Java and is deployed on IBM Cloud™ to interact with Watson Assistant. The medical data set is stored in IBM Db2® Database and the external knowledge source (SNOMED CT) is stored in a graph database (i.e., JanusGraph[6]).

As described in Section 4, the possible intents (i.e., contexts) are bootstrapped based on the domain ontology, and our query relaxation method provides training examples to Watson Assistant for intent classification. At query time, Watson Assistant provides the input to our query relaxation method in the form of a [query term, context] pair.

In this case, the context comes from the intent classifier of the conversational system. Regarding the query term, Watson Assistant extracts entity mentions from an input natural language query[7] and passes the unknown entity mentions as query terms to our relaxation method. Next, we showcase two scenarios in which our query relaxation are used (Figures 7 and 8).

The first scenario is to expand the set of queries by using query relaxation when there is no answer in the KB for the user query. For example, when the query term (*"pyelectasia"*) is unknown (i.e., no matching concept in the given KB), Watson Assistant triggers the query relaxation method to find a list of semantically related concepts that are contained in the given KB by utilizing the external knowledge source (i.e., SNOMED CT). As shown in Figure 7, these additional concepts are then used as a means to "repair" the conversation and smoothly redirect the user to the information contained in the KB. Consequently, the conversation can continue with follow-up questions around the expanded result, *"kidney disease"*. Without our query relaxation, Watson

[6]https://janusgraph.org/
[7]For a demo, refer to https://natural-language-understanding-demo.ng.bluemix.net.


You can also ask a series of questions about the same drug:
What is the adult dose of digoxin for heart failure?
Renal dosing?
Contraindications?
Adverse effects?

What drug treats pyelectasia
I've found multiple results for condition. Which one are you looking for?
- Kidney disease
- Renal impairment
- Disorder of the urinary system

Kidney disease
Is it for Adult or Pediatric?
adult
Here are the drugs used for adult kidney disease:
Effective
- Anemia in chronic kidney disease, In patients on dialysis and not on dialysis, to decrease the need for red blood cell transfusion
  - Epoetin Alfa
  - Epoetin Alfa-epbx
- Autosomal dominant polycystic kidney disease

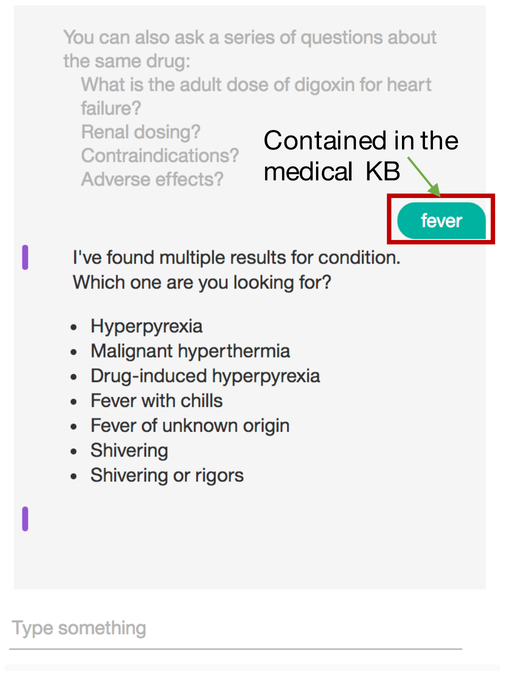
Figure 7: Integration with Watson Assistant (Scenario 1).

Assistant would not be able to return any useful information except replying messages such as *"I don't understand"*. Worse yet, it may return irrelevant and incorrect information to the user, as illustrated in Section 7.2.

In the second scenario, we use query relaxation to expand query answers beyond what matches to the query term in the KB directly. As seen in Figure 8, the query term (*"fever"*) is identified by Watson Assistant as an instance of the concept (*Finding*) in the medical KB. Without query relaxation, identifying *"fever"* as *Finding* triggers one predefined intent *"Indication-hasFinding-Finding"* in Watson Assistant. Hence, the information such as syndromes and treatments for *"fever"* would be returned to the user. With our query relaxation method for concept expansion, 7 additional concepts related to *"fever"* are returned before providing any information specific to *"fever"*. Hence, it offers more opportunities for the user to explore the information contained in the given knowledge base.

6.2 Integration with a Natural Language Query System

Currently, we are also working on incorporating our method with a natural language query (NLQ) system [23, 35]. The NLQ system is different from the previously described conversational system as it targets one-shot complex queries. In this case, the NLQ system receives a natural language query as input and interprets it over the domain ontology to produce a structured query such as SQL. The proposed query relaxation method is utilized to increase the flexibility and robustness of query interpretation.


You can also ask a series of questions about the same drug:

What is the adult dose of digoxin for heart failure?

Renal dosing?

Contraindications?

Adverse effects?

fever

I've found multiple results for condition. Which one are you looking for?

- Hyperpyrexia
- Malignant hyperthermia
- Drug-induced hyperpyrexia
- Fever with chills
- Fever of unknown origin
- Shivering
- Shivering or rigors

Figure 8: Integration with Watson Assistant (Scenario 2).

We outline the solution with a running example from the medical domain as captured in Figure 1 for the query: *"What are the risks caused by using Aspirin with pyelectasia"*.

Evidence generation. In the very first step, the NLQ system tries to identify all the different mentions of ontology elements in the input natural language query [35]. The NLQ system iterates through all the word tokens in the query and collects evidences of one or more elements which have been referenced in the input query. These elements can be concepts or relationships in the domain ontology, as well as the instances of those concepts in the knowledge base.

In general, a token can match multiple elements in the ontology. For example, the token *"risks"* is mapped to a concept *"Risk"*, and the phrase *"caused by"* is mapped to *"cause"* relationship in the ontology. *"Aspirin"* is mapped to the concept of *"Drug"*.

There are two types of evidence: (*i*) a *metadata* evidence is generated by matching the token to the ontology elements (e.g., *"Risk"*), and (*ii*) a *data-value* evidence is generated by looking up a token in the knowledge base (e.g., *"Aspirin"*). The evidence for a token can either be *metadata* or *data-value*, but not both [35].

Due to the colloquial and imprecise terminology used in natural language queries, our query relaxation method is particularly useful to increase the capability of query understanding. The NLQ system relies on the semantically related results from our query relaxation method to match a token to either a metadata or a data-value evidence. The NLQ system associates these semantically relevant results with ontology elements on the fly, as shown in Figure 9. Again, if *"pyelectasia"* is not contained in either domain ontology or knowledge base, our query relaxation returns semantically relevant results (e.g., *"kidney disease"*, *"nephropathy"*, etc.), which can be then mapped to the concept of *"Finding"* in the domain ontology.

Interpretation generation. Note that only one element from the evidence set of each token corresponds to the correct query. In this phase, the NLQ system tries all such combination of elements

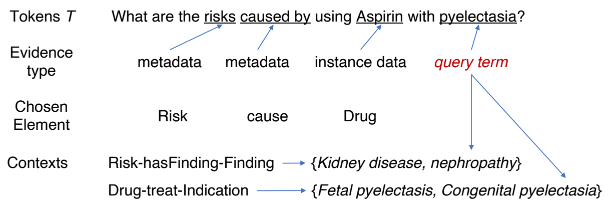


Figure 9: Evidence set.

from each evidence set. Each such combination, called *selection set*, is used to generate an interpretation, which is represented as a sub-graph in the semantic graph connecting one evidence for each token for each ontology element [35]. This semantically grounds the words in the input query to specific contexts by referring to elements in the semantic graph. Connecting these referred elements produces a unique interpretation for the given natural language query based on the ontology semantics.

For each selection set, a sub-tree, called *Interpretation Tree*, is computed, which uniquely identifies the relationship paths among the evidences in the selected set [35]. It is computed by connecting all the elements in the selected set in the semantic graph and satisfying the following constraint. The NLQ system uses a Steiner-tree-based algorithm [35] and ranks the interpretations according to their compactness to generate an interpretation of minimal size for a selected set. Note that query relaxation returns a similarity score associated with each result value. We are currently extending the ranking algorithm to take this ranking score into account, in addition to compactness.

For example, the top ranked interpretation as found from selected set is *ITree* = { (*Drug* ⟶ *cause* ⟶ *Risk* ⟶ *hasFinding* ⟶ *Finding*), (*Drug* ⟶ *treat* ⟶ *Indication* ⟶ *hasFinding* ⟶ *Finding*) }. Two interpretations have the same compactness. In this case, if we take into consideration the similarity scores associated with the relaxed results, the former interpretation would be more preferable since *"Kidney disease"* is the most semantically similar concept to the search term *"pyelectasia"*.

7 EXPERIMENTAL EVALUATION

In this section, we describe experiments using a medical data set (*MED*) to show the efficacy of our proposed query relaxation method in terms of precision, recall, and F1-score. We also conduct user studies, in which we use Watson Assistant as the conversational interface.

7.1 Experimental Setup

Data set. We use a medical data set (*MED*) that is used to support evidence-based clinical decisions and patient education. The total size of this data set is around 1.2 GB. The ontology corresponding to *MED* consists of 43 concepts and 58 relationships. We chose SNOMED CT as the external knowledge source since it consists of comprehensive information regarding terms, synonyms, and definitions used in clinical documentation and reporting.

Users. 20 Subject Matter Experts (SMEs) participated in our experiments. They all have deep knowledge and understanding of the medical domain, and are able to distinguish between a correct and a wrong answer.

Methodologies. We conducted two sets of experiments to evaluate the efficacy of our proposed query relaxation method with respect to precision and recall. First, we show the superiority of our method compared to alternative methods. Second, a user study demonstrates the benefit of applying our query relaxation method in a conversational system.

7.2 Evaluation Results

Precision and recall. We chose 100 commonly used concepts of medical conditions, and used the methods described below to identify the semantically related concepts. The participants were asked to evaluate whether these relaxed concepts are indeed related to the given ones.

We first study the effectiveness of the methods used for mapping instances from the given KB to the external knowledge source, including exact string matching (*EXACT*), approximate string matching with an edit-distance threshold $\tau = 2$ (*EDIT*), and a variant of word embedding to support longer pieces of text [3] (*EMBEDDING*). Table 1 reports *Precision*, *Recall* as well as *F1-score* of these three mapping methods.

We observe that word embedding achieves the best overall result quality. This provides our query relaxation method a solid foundation to identify semantically related concepts in the neighborhood of these concepts. On the contrary, exact and approximate string matching suffer from lower recall compared to the word embedding method. Hence we used word embeddings in the rest of the experiments as the matching method.

Table 1: Accuracy of mapping methods.

Methods	Precision	Recall	F1
EXACT	**100**	83.33	90.01
EDIT	96.36	88.33	92.17
EMBEDDING	96.49	**91.67**	**94.02**

Next, we compare our proposed query relaxation method (*QR*) against its variants as well as alternative approaches, including our proposed method without the frequency information from the corpus (*QR-no-corpus*), our proposed method without the contextual information (*QR-no-context*), a baseline IC-based semantic measure (*IC*) [2], a baseline method [3, 8] using both pre-trained embeddings [32] (*Embedding-pre-trained*) and embeddings we trained on a given medical document corpus (*Embedding-trained*).

Table 2: Overall effectiveness.

Methods	P@10	R@10	F1
QR	**90.51**	**82.64**	**86.40**
QR-no-context	85.45	77.27	81.15
QR-no-corpus	78.23	70.91	74.39
IC	75.55	68.18	71.68
Embedding-pre-trained	66.14	60.13	62.99
Embedding-trained	79.37	71.81	75.40

Table 2 reports *Precision@10*, *Recall@10* as well as *F1-score* against the *MED* data set. *Precision@10* corresponds to the number of relevant results among the top 10 returned concepts, *Recall@10* is the proportion of relevant results found in the top 10 returned concepts to the total number of relevant results, and *F1-score* is the harmonic mean of *Precision@10* and *Recall@10*. Our proposed methods including (*QR-no-corpus* and *QR-no-context*) are more accurate than the baseline *IC*. Specifically, we observe that *QR-no-context* still returns higher quality results than the baseline *IC*. This shows that differentiating specialization and generalization relationships helps capturing the semantic similarity between a pair of concepts. It is not surprising that *QR-no-context* further improves the result quality when the frequency information is available from the corpus. Regarding *Embedding-pre-trained*, we observe that it achieves the lowest precision and recall among all methods. This is expected as the model was trained on a different medical corpus and many of the words contained in SNOMED CT are out of its vocabulary. For the embedding of multi-word query terms, we used the average its words' embeddings. The result quality of *Embedding-trained* is much improved as we trained the embedding model on our medical document corpus, and the embedding of a (multi-word) query term is further computed based on [3]. However, without the contextual information from the query, many concepts in the given KB are cluttered with the query term in the low-dimensional vector space. Hence the quality of *Embedding-trained* is still not as good as *QR*. In summary, our method *QR*, which incorporates both the frequency information from the corpus, as well as the context from the query, achieves the highest precision and recall.

User study. In this user study, participants were asked to complete two tasks to evaluate the query understanding capability of a conversational system with and without our query relaxation method. The participants were allowed to get familiar with the conversational system over the given KB. In task 1 (*T1*), we asked participants to come up with 20 questions around 20 given concepts (i.e., condition names). For example, the participant may ask "what drugs are used to treat [*condition*]" or "what drugs cause [*condition*]". In task 2 (*T2*), the participants were allowed to come up with 10 questions of their own choice about anything in *MED*.

The participants were then asked to grade the quality of the relaxed results in a scale of 1-5. If the system returns a correct response in the first attempt, it receives 5 points. If the system fails to return a correct response, the participants can rephrase their questions for at most 4 more times. Each time a wrong result is returned, the participant subtracts a point. For example, if the correct answer is returned after 3 attempts (i.e., 2 failed attempts) the participant gives in total 5-2 = 3 points. In addition to the score, we also asked the participants to provide detailed feedback.

Table 3: Watson Assistant with and without QR.

	QR		***no QR***	
Score	***T1***	***T2***	***T1***	***T2***
1 (Very dissatisfied)	2.1%	10.55%	13.06%	11.11%
2 (Dissatisfied)	10.35%	11.07%	16.87%	38.26%
3 (Okay)	25.59%	29.33%	36.29%	30.85%
4 (Satisfied)	35.21%	33.37%	18.25%	12.47%
5 (Very satisfied)	26.85%	15.68%	15.53%	7.31%
AVG	3.73	3.31	3.06	2.67

Table 3 shows the aggregated grades received by our query relaxation method for the two tasks described above. The numbers in the table show the percentage of each particular grade. Clearly, the conversation system with query relaxation achieves a substantially higher score than the one without query relaxation in both tasks. The average grades of the system with query relaxation in both tasks are 20% higher than the ones without relaxation. Specifically, our query relaxation method performed slightly better in *T1* than *T2*. The reasons are that the questions used in *T2* were completely from the participants and a non-trivial number of questions (9 out of 200) do not have an answer in the given KB, as opposed to *T1*, where 20 concepts were provided to the users. Moreover, the feedback from the participants indicates that the lower grades in Table 3 are due to other reasons orthogonal to the quality of our query relaxation method.

Specifically, there are 7 incidences in which the expected answers are not contained in the given KB (*MED*). There are 11 incidences where the users complained about the conversational flow irrespective of the query relaxation results. For example, some users prefer smaller number of interactions with the conversational system in order to complete their tasks. Some users failed to follow the instructions, and hence ran into unexpected follow-up questions. Moreover, there are 10 incidences that the users did not provide any negative feedback but gave a low grade such as 1 or 3. Last but not the least, the SMEs reported 6 instances that the amount of information returned is overwhelming even though the relaxed results are semantically correct. All these cases resulted in low grades, including 1, 2, and 3. Note that all the above cases are counted in Table 3. One straightforward solution to address these issues would be to incorporate the user's relevance feedback [39] in the query relaxation method, and to progressively improve the relaxed results.

8 RELATED WORK

Natural language querying over KBs. Several approaches have been recently investigated to build natural language interfaces to KBs. In [13], query templates are learned from KBs and question answering corpora. Wang et al. [41] leverage terms and their relationships from a web corpus and map them to related concepts using a KB. Then, a random walk-based algorithm is proposed for understanding the terms in a given query. The semantic parsing methods proposed in [4] use a domain-independent representation derived from combinatory categorical grammar parsers. Most recently, a supervised learning framework [20] is introduced to exploit sentence embedding for the medical question answering task. However, they usually require a large labeled corpus or pairs of questions and answers. Our approach is complementary to these methods since they only focus on answering queries with 'strict' execution, which often results in no answers. Our query relaxation expands the domain vocabulary used in queries and provides more semantically related results.

Query relaxation for databases. The database community has developed query relaxation methods that return information beyond a standard query. Query relaxation expands the query selection criteria to include additional relevant information, often by consulting a semantic model of the data domain. Gaasterland [18] introduces query relaxation techniques in deductive databases, using logic rules to specify legal relaxation constraints. Query relaxation [11] is introduced to relational databases using type-abstraction hierarchies to find semantically similar query results. A taxonomy-based relational algebra is proposed to extend the capability of selection and join by relating values occurring in the tuples with values in the query using the taxonomy [26]. Poulovassilis et al. [31] applied query approximation and query relaxation techniques based on RDFS inference rules to the evaluation of conjunctive regular path queries (CRPQs) over graph data. Our approach is different from the above work as we leverage external knowledge sources to expand the domain vocabulary. Further, our approach uses the domain ontology and context information to differentiate the semantic subtleties among instances.

Semantic similarity measures. Among various semantic measures, path finding measure [42] is based on the shortest path separating concepts, which traverses the LCS of two concepts. IC-based measures can be estimated solely from the structure of a taxonomy [36], or from the distribution of concepts in a text corpus and a taxonomy [34]. Our semantic similarity measure is designed upon these measures and overcomes their limitations by utilizing the domain ontology to differentiate the semantic subtleties. Hence our method can achieve significant gain of recall without sacrificing the precision.

Recent works demonstrated that deep learning models built at word [8, 30] or sentence [3, 14] level can be used for semantic similarity estimation. However, these methods demand high quality training data sets, which is critical and expensive in reality. Moreover, directly applying word or sentence embeddings to our problem is not sufficient since the structural and contextual information are not considered when the model is trained. Hence, we use word and sentence embeddings for linking the given KB to the external knowledge source and build our similarity measure on top of it.

9 CONCLUSION

In this paper, we present a novel two-phase query relaxation method that leverages external knowledge sources to expand answers for querying medical KBs. We introduce a novel similarity metric to empower our query relaxation method to identify semantically related concepts. Our method is successfully integrated with two exemplary systems, a conversational system and a natural language query system, respectively. Our experiments show that our query relaxation method for the medical KB outperforms state-of-the-art methods, including deep learning-based ones, in precision and recall. We also conduct a user study demonstrating how our query relaxation method expands the query results and improves their quality for medical KBs.

REFERENCES

[1] Amazon. 2019. Amazon Comprehend Medical. https://aws.amazon.com/comprehend/medical/.

[2] Mohamed Ben Aouicha and Mohamed Ali Hadj Taieb. 2016. Computing semantic similarity between biomedical concepts using new information content approach. *Journal of Biomedical Informatics* 59 (2016), 258–275.

[3] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. 2017. A Simple but Tough-to-Beat Baseline for Sentence Embeddings. In *ICLR*.

[4] Yoav Artzi, Nicholas FitzGerald, and Luke Zettlemoyer. 2013. Semantic Parsing with Combinatory Categorial Grammars. In *ACL (Tutorials)*.

[5] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. 2007. DBpedia: A Nucleus for a Web of Open Data. In *ISWC*. 722–735.

[6] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider (Eds.). 2003. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press.

[7] Christopher M. Bishop. 2007. *Pattern recognition and machine learning, 5th Edition.* Springer.

[8] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Enriching Word Vectors with Subword Information. *arXiv preprint arXiv:1607.04606* (2016).

[9] Kurt D. Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*. 1247–1250.

[10] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. 2015. *Entity Resolution in the Web of Data.* Morgan & Claypool Publishers.

[11] Wesley W. Chu, Hua Yang, Kuorong Chiang, Michael Minock, Gladys Chow, and Chris Larson. 1996. CoBase: A Scalable and Extensible Cooperative Information System. *J. Intell. Inf. Syst.* 6, 2-3 (1996), 223–259.

[12] The Gene Ontology Consortium. 2019. The Gene Ontology knowledgebase. http://geneontology.org/.

[13] Wanyun Cui, Yanghua Xiao, and Wei Wang. 2016. KBQA: An Online Template Based Question Answering System over Freebase. In *IJCAI*. 4240–4241.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. 4171–4186.

[15] Xin Luna Dong and Divesh Srivastava. 2015. *Big Data Integration.* Morgan & Claypool Publishers.

[16] DrugBank. 2019. DrugBank. https://www.drugbank.ca/.

[17] Marcus Fontoura, Vanja Josifovski, Ravi Kumar, Christopher Olston, Andrew Tomkins, and Sergei Vassilvitskii. 2008. Relaxation in text search using taxonomies. *PVLDB* 1, 1 (2008), 672–683.

[18] Terry Gaasterland. 1997. Cooperative Answering through Controlled Query Relaxation. *IEEE Expert* 12, 5 (1997), 48–59.

[19] Vijay Garla and Cynthia Brandt. 2012. Semantic similarity in the biomedical domain: An evaluation across knowledge sources. *BMC bioinformatics* 13 (2012), 261.

[20] Yu Hao, Xien Liu, Ji Wu, and Ping Lv. 2019. Exploiting Sentence Embedding for Medical Question Answering. In *AAAI*. 938–945.

[21] IBM. 2019. Watson Assistant. https://www.ibm.com/cloud/watson-assistant/.

[22] Danai Koutra, Neil Shah, Joshua T. Vogelstein, Brian Gallagher, and Christos Faloutsos. 2016. DeltaCon: Principled Massive-Graph Similarity Function with Attribution. *TKDD* 10, 3 (2016), 28:1–28:43.

[23] Chuan Lei, Fatma Özcan, Abdul Quamar, Ashish R. Mittal, Jaydeep Sen, Diptikalyan Saha, and Karthik Sankaranarayanan. 2018. Ontology-Based Natural Language Query Interfaces for Data Exploration. *IEEE Data Eng. Bull.* 41, 3 (2018), 52–63.

[24] Jiaqing Liang, Yi Zhang, Yanghua Xiao, Haixun Wang, Wei Wang, and Pinpin Zhu. 2017. On the Transitivity of Hypernym-Hyponym Relations in Data-Driven Lexical Taxonomies. In *AAAI*. 1185–1191.

[25] Dekang Lin. 1998. An Information-Theoretic Definition of Similarity. In *ICML*. 296–304.

[26] Davide Martinenghi and Riccardo Torlone. 2014. Taxonomy-based relaxation of query answering in relational databases. *VLDB J.* 23, 5 (2014), 747–769.

[27] Víctor Martínez, Fernando Berzal, and Juan Carlos Cubero Talavera. 2017. A Survey of Link Prediction in Complex Networks. *ACM Comput. Surv.* 49, 4 (2017), 69:1–69:33.

[28] Microsoft. 2019. Language Understanding (LUIS). https://www.luis.ai/home.

[29] Ted Pedersen. 2010. Information Content Measures of Semantic Similarity Perform Better Without Sense-tagged Text. In *NAACL*. 329–332.

[30] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global Vectors for Word Representation. In *EMNLP*. 1532–1543.

[31] Alexandra Poulovassilis, Petra Selmer, and Peter T. Wood. 2016. Approximation and relaxation of semantic web path queries. *Journal of Web Semantics* 40 (2016), 1 – 21.

[32] Sampo Pyysalo, Filip Ginter, Hans Moen, et al. 2013. Distributional semantics resources for biomedical text processing. In *Proceedings of Languages in Biology and Medicine.*

[33] Abdul Quamar, Chuan Lei, Dorian Miller, Fatma Özcan, Jeffrey Kreulen, Robert J Moore, and Vasilis Efthymiou. 2020. An Ontology-Based Conversation System for Knowledge Bases. *Under review* (2020).

[34] Philip Resnik. 1995. Using Information Content to Evaluate Semantic Similarity in a Taxonomy. In *IJCAI*. 448–453.

[35] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R. Mittal, and Fatma Özcan. 2016. ATHENA: An Ontology-Driven System for Natural Language Querying over Relational Data Stores. *PVLDB* 9, 12 (2016), 1209–1220.

[36] Nuno Seco, Tony Veale, and Jer Hayes. 2004. An Intrinsic Information Content Metric for Semantic Similarity in WordNet. In *ECAI*. 1089–1090.

[37] Jaydeep Sen, Fatma Özcan, Abdul Quamar, Greg Stager, Ashish R. Mittal, Manasa Jammi, Chuan Lei, Diptikalyan Saha, and Karthik Sankaranarayanan. 2019. Natural Language Querying of Complex Business Intelligence Queries. In *SIGMOD*. 1997–2000.

[38] SNOMED. 2019. SNOMED Clinical Terms. https://www.snomed.org/snomed-ct/what-is-snomed-ct.

[39] Yu Su, Shengqi Yang, Huan Sun, Mudhakar Srivatsa, Sue Kase, Michelle Vanni, and Xifeng Yan. 2015. Exploiting Relevance Feedback in Knowledge Graph Search. In *KDD*. 1135–1144.

[40] UMLS. 2019. Unified Medical Language System. https://www.nlm.nih.gov/research/umls/.

[41] Zhongyuan Wang, Kejun Zhao, Haixun Wang, Xiaofeng Meng, and Ji-Rong Wen. 2015. Query Understanding Through Knowledge-based Conceptualization. In *IJCAI*. 3264–3270.

[42] Zhibiao Wu and Martha Palmer. 1994. Verbs Semantics and Lexical Selection. In *ACL*. 133–138.

[43] Pengcheng Yin, Zhengdong Lu, Hang Li, and Ben Kao. 2016. Neural Enquirer: Learning to Query Tables in Natural Language. In *IJCAI*. 2308–2314.

VAP: A Visual Analysis Tool for Energy Consumption Spatio-temporal Pattern Discovery

Xiufeng Liu
Technical University of Denmark
Lyngby, Denmark
xiuli@dtu.dk

Zhibin Niu
Technical University of Denmark
Lyngby, Denmark
zhini@dtu.dk

Yanyan Yang
School of Computing, University of Portsmouth
Portsmouth PO1 3HE, UK
linda.Yang@port.ac.uk

Junqi Wu
Tianjin University
Tianjin, China
wujunqi@tju.edu.cn

Dawei Cheng
Shanghai Jiaotong University
Shanghai, China
dawei.cheng@sjtu.edu.cn

Xin Wang
Southwest Jiaotong University
Chengdu, China
xinwang@swjtu.cn

ABSTRACT

In the context of urbanization and the rapid growth of energy demand, understanding the spatial and temporal dynamics of urban energy use is crucial for identifying energy-saving potentials. In this demo, we present a visual analysis tool, VAP, that allows users to explore the dynamics of urban energy use at different spatial and temporal scales. In contrast to traditional statistical and machine learning methods, the visual analysis based tool focuses on analytical thinking, user interactions and answering business questions by examining different visual analysis views. In the demonstration, conference attendees will interact with VAP and learn its capabilities in discovering typical consumption patterns and spatio-temporal shift patterns from a real-world case study of electricity.

1 INTRODUCTION

In recent years, the availability of high-resolution energy consumption data has exploded at an unprecedented rate, along with the diffusion of smart metering technology. The energy sector is increasingly in need of advanced tools and methods to gain insights from big smart meter data sets for decision-making purposes [1]. However, traditional statistical and machine learning methods fail or are too complex to answer some central business questions such as *"What is the consumption trend or pattern over time?"* and *"Does mass mobility affect energy demand?"*. On the other hand, these questions can be much easier to answer through visual analysis supported by human cognitive capabilities. The visual analysis focuses on analytical reasoning, facilitated by interactive user interfaces, and exploring the views that most effectively answer these questions [2]. The visual analysis encompasses several disciplines, including geographic information systems, information visualization, and data computing. Visual analysis has been employed in bioinformatics, physics, astronomy, and climate, but until now rarely in energy.

The most relevant work we have found is the tool implemented in [3] to study energy consumption in the Chicago area in connection with census data. This tool supports the disaggregation analysis on several spatial levels, but without further functionalities, e.g. for the analysis of spatio-temporal patterns and demand shifts caused by mass behaviors. However, much research has been attempted on spatio-temporal data analysis, which involves the identification of object spatial positions at specific moments, e.g., [2], and the detection of anomalies such as traffic congestion [4], cyber attacks [5], medical diagnoses [6] and more. We believe that there is great potential for visual analysis in the applications of energy management systems, because especially many areas in this sector can achieve better results efficiently and effectively with the help of visual analysis. Among others, these include the study of consumer behaviors and living habits, the planning of energy supply, the development of energy strategies and the design of personalized services.

In this demo, we introduce a visual analysis tool, *VAP*, to support the study of urban energy consumption patterns and dynamics on different spatial and temporal scales. This tool is unique in at least two ways:

- Unlike traditional segmentation analysis, which uses clustering algorithms to find typical patterns, VAP supports typical pattern recognition through visual mining. High-dimensional time series are first reduced to low-dimensional data points, and closely placed together on a view according to their similarity, then users can identify patterns by interactively selecting the points on the view. Therefore, pattern recognition is an interactive process embedded in human cognitive recognition. The identified patterns represent customers with similar consumption behaviors or habits, which can be used to develop targeting demand-response programs, forecast energy consumption, and provide personalized services.
- VAP supports the analysis of energy shift patterns in different spatial and temporal scales. The variation in energy demand over time has been studied intensively, while the variation in demand across different spatial spaces has rarely been studied. The demand shift patterns can be identified and visualized with VAP, and the shifts in high energy demand over time can help utilities plan energy distribution and improve energy flexibility.

In Section 2, we will describe the visual analysis framework, the tool, and the pattern discovery methods. In Section 3, we will outline the demonstration scenarios that illustrate how VAP can be used to discover typical patterns and shift patterns of energy demand through visual analysis in a real-world case study. The conference attendees can interact with VAP by first asking business questions, then probing the answers through interactions with the tool, and finally gaining knowledge from the visualized outcomes. This demo presents the elements that drive research for visual analysis in the energy sector, and provides an outlook

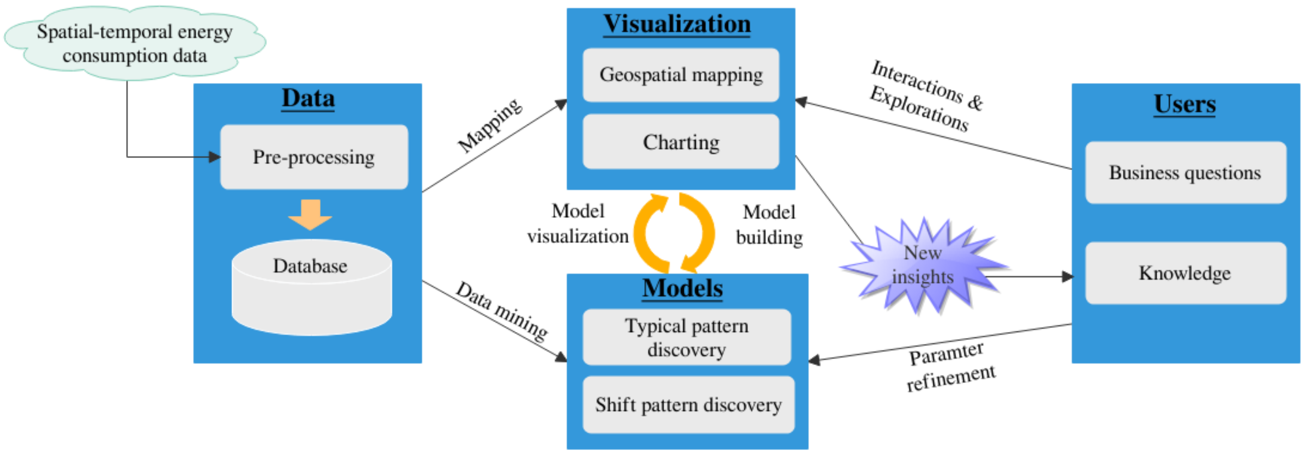


Figure 1: Overview of the visual analysis framework for spatio-temporal pattern discovery

on the use potentials on a higher spatial scale as well as on other urban energy uses.

Through the demonstration of VAP, we hope to increase the awareness of the emerging technologies for the domain-specific applications within the data management community. We also hope that this demonstration will stimulate the research of applying new database technologies in the applications for smart energy systems, including data acquisition, processing, storage, analysis and visualization. We plan to make the code open-source and create an online demonstration version for others to use, experience and extend.

2 APPROACH

Visual analysis aims to help users gain insights into the data by interacting with the visual diagrams that interpret the data. The overview of the visual analysis framework is illustrated in Figure 1, which integrates the components including Data, Models, Visualization and Users. The data for this demo is spatio-temporal energy consumption data from smart meters. The data were preprocessed, including removal of anomalies and correction of missing values. To derive knowledge from the data, the models including typical pattern recognition and spatio-temporal shift pattern recognition were implemented. The visualization aids in the presentation of the analysis results, knowledge generation and communication with users. Users typically gain knowledge by first asking business questions and then exploring analytical views to answer their questions. This is an iterative process of discovering knowledge from the data and refining parameters of the models.

2.1 Pattern recognition models

VAP supports typical consumption pattern discovery and spatial-temporal shift pattern discovery of energy demand using the visual analysis method. The two pattern recognition models are described as follows.

Typical pattern discovery model. Typical consumption patterns are often used to segment customer groups for offering personalized services in the energy sector. In order to visually analyze typical consumption patterns, high-dimensional time series must first be reduced to a lower dimension so that it can be displayed in a low-dimensional space. The proposed model supports the t-distributed Stochastic Neighbor Embedding (t-SNE) [7] and the Multi-Dimensional Scaling (MDS) [8] for high-dimensional time series data reduction. The positions of the resulting low-dimensional data points on the view are determined by minimizing the Kullback-Leibler divergence defined as follows.

$$KL(P\|P') = \sum_{i\neq j} P_{ij} \log \frac{P_{ij}}{P'_{ij}} \tag{1}$$

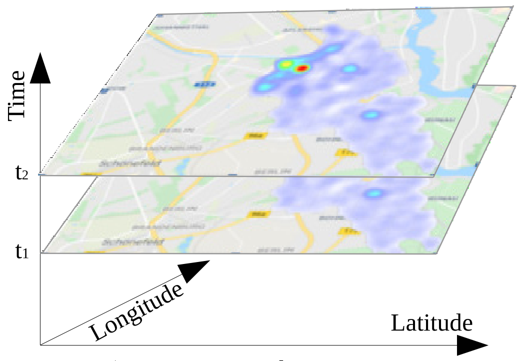
where P_{ij} is the similarity probability distribution between the high-dimensional objects, o_i and o_j, while P'_{ij} is the similarity probability distribution between the reduced low-dimensional objects, o'_i and o'_j. Here, the Pearson correlation coefficient is used as the distance metric for calculating the similarity as it can be better to reflect the correlation of the trend between two time series [9]. The similarity probability distribution of the low-dimensional objects can be obtained by:

$$P'_{ij} = \frac{\left(1 + \left\|o'_i - o'_j\right\|^2\right)^{-1}}{\sum_{k\neq l}\left(1 + \left\|o'_k - o'_l\right\|^2\right)^{-1}} \tag{2}$$

where k and l are the indices of the objects; and $\|\cdot\|$ represents the distance. The similarity probability determines whether the two objects should be placed closely or far away on the view, i.e. the more similar, the closer.

With the reduced data points, typical patterns of energy consumption can be interactively identified by selecting the closely placed points on the view in a low-dimensional space.

Shift pattern discovery model. This model is used to capture the changes of high energy demand locations over time. The shift pattern of energy demand is essential for planning the energy supply between different geographical areas. For example, the high-demand area may shift from commercial to residential when people go home after work. This can be happening within the time interval of 1 - 2 hours, but the detection of demand shifts is helpful for energy supply planning. Here, we use the flow map method [10] to visualize the spatial migration of high-energy demand flow, which implies the flow between spatially different areas. Figure 2 illustrates the flow map method with a schematic diagram. First, the spatio-temporal distributions of the discrete energy demand are expressed as two different density-strength maps over the time from t_1 to t_2 (see Figure 2a). The density strength map can be obtained using a kernel density estimation method (KDE). Then, the flows are obtained by the difference between the strength maps in two time steps. The resulting flow



a) Density strength map over time

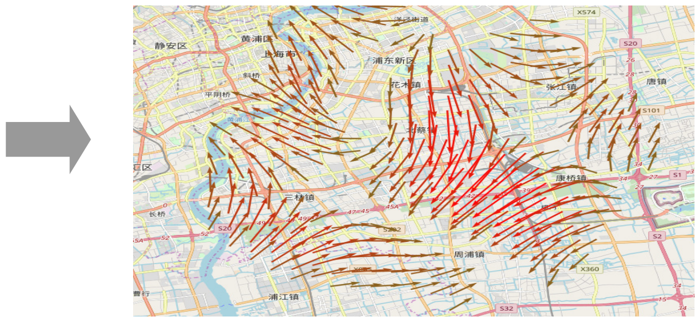
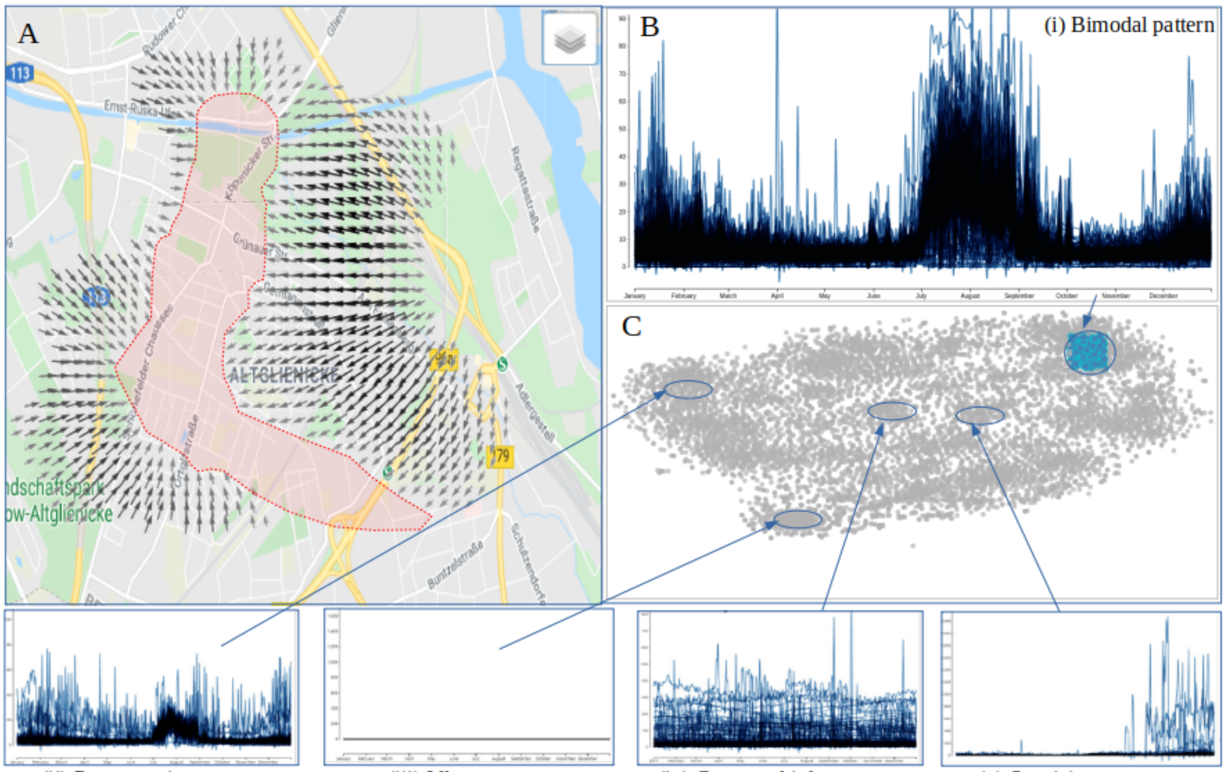

b) Spatio-temporal shift patterns

Figure 2: Flow map method for shift pattern discovery



(ii) Power-saving pattern (iii) Idle pattern (iv) Constant high pattern (v) Suspicious pattern

Figure 3: Spatio-temporal pattern discovery using VAP

map represents the spatio-temporal shifts of high energy demand (see Figure 2b).

The method is formalized in the following. Let $x_1, x_2, ..., x_n$ be the discrete geographical locations of the customers, x_i denoted as a vector $(lon_i, lat_i)^T$ of longitude and latitude. The density of a position of x in a 2D space is defined as follows:

$$\widehat{f}_{2D}(x) = \frac{1}{n}\sum_{i=1}^{n} c_i K_h(x - x_i) = \frac{1}{nh}\sum_{i=1}^{n} c_i K\left(\frac{x - x_i}{h}\right) \quad (3)$$

where n is the sample size, c_i is a normalized value of the average energy consumption used to re-weight the demand strength with respect to the geographical distribution, and h is the bandwidth of the kernel K_h. Gaussian kernel is used in the implementation, but note that other kernels can also be used. Since the Gaussian kernel can cover a larger spatial area for the changes, and it has a lower computation complexity compared with other kernels with exponential functions. The flow patterns representing the energy demand shift between t_1 and t_2 can be expressed as the density difference as follows:

$$Shift(x)|_{(t_1,t_2)} = \widehat{f}_{2D}(x)|_{t_2} - \widehat{f}_{2D}(x)|_{t_1} \quad (4)$$

2.2 Visual analysis tool

The visual analysis tool VAP is implemented as a web application with an architecture of three layers *data layer, logic layer* and *presentation layer*. In the data layer, PostgreSQL is used as the database management system, with PostGIS added to support spatial data processing. In the logic layer, all algorithms are implemented in Python, including the typical pattern discovery and spatio-temporal RESTful APIs are implemented to exchange JSON-formatted data between client and server. In the presentation layer, HTML5, CSS and JavaScript are used to implement the user interface (see Figure 3). Especially the JavaScript library, Leaflet.js [11], is used for the visualization of Scalable Vector Graphics (SVG) and the mapping. The flow patterns are displayed as colored arrows on the map, and the color depth represents the rate of change of the flow patterns; the darker the color, the higher the rate. The JavaScript library d3.js [12] is used for time series visualization.

Figure 3 shows the main user interface of VAP, which consists of the three views: A, B, and C. View A displays the spatial information of the clustered customers. This view supports users in selecting different map types, displaying the geographical positions of customers with markers, and visualizing the spatial distribution density with a heat map and spatio-temporal shift patterns with flow vectors. View B shows the time series for the customers selected in view C. This view visualizes the typical consumption pattern for all selected customers. View C is an interactive navigator that allows users to explore different energy consumption patterns by selecting the points by clicking and dragging. The closer the points are to each other, the more similar the patterns will be.

VAP supports visual analysis for any type of energy consumption with spatial information. The design of the database model can be interpolated with specific energy data sets in real use cases. Figure 3 illustrates an example for the visual analysis of an electricity consumption data set (note: the coordinates are offset for anonymization), where five typical patterns were discovered, including (i) bimodal pattern, (ii) energy-saving pattern, (iii) idle pattern, (iv) constant high pattern and (v) suspicious pattern. View A and B shows the flow map and the aggregated consumption pattern for the customers selected in the reduced 2D space in view C, respectively. The pattern is a bimodal pattern with a peak in winter and summer, respectively, which may be caused by the use of electrical heating and cooling appliances. The other four typical patterns can be interpreted in a similar way by examining the consumer behaviors or habits of customers, which will be explained in the demonstration. In view A, the area covered by the arrows is a commercial area, while the area pointed to by the arrows is a residential area (i.e. the light red area). This flow map indicates that the area with high energy demand is shifted to the residential area when people go home after work.

3 DEMONSTRATION

We will use a real-world electricity consumption data set for the demonstration. We will introduce the system architecture, the visual analysis process and how to use the tool to solve practical problems in energy planning. Conference attendees will interact with the tool to perform visual analysis on the data, i.e., ask business questions, answer questions through the exploration of the views and acquire knowledge. In particular, the participants will interactively discover typical and shift patterns of electricity consumption with VAP. We aim to help participants learn more about visual analysis for energy decision makings, explore the approaches for energy demand-side management, and raise their awareness of energy savings. Two demo scenarios will be presented during the conference.

S1: Typical patterns discovery. In this demo scenario, conference attendees will interact with VAP by investigating typical energy consumption patterns and identifying the spatial distribution of customers in the study area. We will explain to the participants the meaning of each identified pattern and the reason behind it. First, an attendee can start by asking questions, e.g., *who are the early birds with a morning peak between 5:00–7:00?* They investigate the customers of interest by selecting the points at different places on the view. Second, the attendees will study the transition of consumption patterns based on point similarity (or point spacing) in 2D space. They select the closely placed points continuously, and observe the pattern transition over the spatial space. Third, they can select the scatter plots generated by different dimensional reduction methods, including t-SNE and MDS, observe difference and compare capabilities in typical pattern discovery. Fourth, we will run the k-mean algorithm on the sampled data to discover typical patterns, compare the results, and explain the advantages of using the visual analysis method.

S2: Spatio-temporal shift pattern discovery. In this demo scenario, conference attendees will examine the patterns of the energy demand interactively, and we will explain the discovered shift patterns accordingly. First, an attendee examines the shift patterns by varying the temporal granular intervals, including hourly, every four hours, daily, weekly, monthly, quarterly, and yearly. They can then learn the sensitivity of the shift pattern changes against different time granularity. Second, they select different customer groups according to the consumption intensity in a quartile value ranging from 30% to 90%. They can then learn the sensitivity against different energy consumption intensities. Third, we can further demonstrate the shift pattern dynamics through a simulation. If, for example, the data are fed to the system in a short time interval, e.g., every 10 seconds, we can observe the changes of patterns in near real time.

4 CONCLUSION

We presented a visual analysis framework and a tool that supports both spatial and temporal pattern analysis for smart energy systems. We described the technique of dimension reduction and discussed how to reduce and visualize high dimensional data to a low dimensional space. We described and demonstrated the discovery of typical consumption patterns and spatial-temporal shift patterns of energy demand using a real case study. The demonstration validated the plausibility of the proposed visual analysis framework and the effectiveness of the tool in knowledge discovery through user interactions.

Acknowledgements. This research was supported by the EU H2020 Research and Innovation Program under the Marie Sklodowska-Curie Grant Agreement (754462), the Flexible Energy Denmark project funded by Innovationsfonden (8090-00069B) and the project FlexSUS: Flexibility for Smart Urban Energy Systems (91352) funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No 775970.

REFERENCES

[1] N. Iftikhar, X. Liu, F. E. Nordbjerg, and S. Danalachi. 2016. A Prediction-based Smart Meter Data Generator. In Proc. of NBIS, 2016, 173–180.
[2] N. Andrienko, and G. Andrienko. 2013. A visual analytics framework for spatio-temporal analysis and modelling. Data Mining and Knowledge Discovery 27, 1 (2013), 55–83.
[3] J. T. Trabucco, D. Lee, S. Derrible, and G. E. Marai. 2019. Visual Analysis of a Smart City's Energy Consumption. Multimodal Technologies and Interaction 3, 2 (2019), 30.
[4] F. Zhou, W. Huang, Y. Zhao, Y. Shi, X. Liang, and X. Fan. 2015. Entvis: a visual analytic tool for entropy-based network traffic detection. IEEE graphics and applications 35, 6 (2015), 42–50.
[5] L. Layman, N. Zazworka. 2014. InViz: instant visualization of security attacks. In HotSoS, (2014), 15.
[6] F. Ritter, T. Boskamp, A. Homeyer, H. Laue, M. Schwier, F. Link, and HO. Peitgen. 2011. Medical image analysis. IEEE pulse 2, 6 (2011), 60–70.
[7] LVD. Maaten, and G. Hinton. 2008. Visualizing data using t-SNE. Journal of machine learning research, 9 (2008), 2579–2605.
[8] J. B. Kruskal. 1964. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis Psychometrics, 29 (1964), 1–27.
[9] X. Liu, N. Iftikhar, H. Huo, R. Li, and P.S. Nielsen. 2019. Two Approaches for Synthesising Scalable Residential Energy Consumption Data. Future Computer Systems 95, (2019), 586–600.
[10] D. Guo, and X. Zhu. 2014. Origin-destination flow data smoothing and mapping. IEEE Transactions on Visualization and Computer Graphics 20, 12 (2014), 2043–2052.
[11] Crickard III P. 2014. Leaflet. js essentials. Packt Publishing Ltd, 2014.
[12] F. Bao, and J. Chen. 2014. Visual framework for big data in d3. js. In Proc. of IEEE Workshop on Electronics, Computer and Applications, 2014, 47–50.

Chronos: The Swiss Army Knife for Database Evaluations

Marco Vogt Alexander Stiemer Sein Coray Heiko Schuldt
{firstname.lastname}@unibas.ch
Databases and Information Systems Research Group
Department of Mathematics and Computer Science
University of Basel, Switzerland

ABSTRACT

Systems evaluations are an important part of empirical research in computer science. Such evaluations encompass the systematic assessment of the run-time characteristics of systems based on one or several parameters. Considering all possible parameter settings is often a very tedious and time-consuming task with many manual activities, or at least the manual creation of evaluation scripts. Ideally, the thorough evaluation of a complete evaluation space can be fully automated. This includes the set-up of the evaluation, its execution, and the subsequent analysis of the results. In this paper, we introduce Chronos, a system for the automation of the entire evaluation workflow. While Chronos has originally been built for database systems evaluations, its generic approach also allows its usage in other domains. We show how Chronos can be deployed for a concrete database evaluation, the comparative performance analysis of different storage engines in MongoDB.

1 INTRODUCTION

Scientific practice considers the development of novel theories and their empirical evaluation. In computer science, empirical evidence is to a large extent obtained by the systematic evaluation of systems. Essentially, such a systematic evaluation includes the thorough assessment of the quantitative behavior of a system based on one or several parameters. This means that systems need to be run over and over again with a modified set of parameters. This is a tedious and highly repetitive task, but essential for obtaining insights into the run-time characteristics of the system under evaluation (SuE).

In the database community, systems evaluations are mostly based on benchmarks that combine test data and certain access characteristics (queries). Systems evaluations need to be tailored to the characteristics of an SuE and to the parameters of the latter that determine their behavior. Nevertheless, the overall evaluation workflows, seen from a high-level perspective, show quite some commonalities even across systems. First, the SuE needs to be set up with the exact parameters of a particular evaluation run. This also contains the configuration of the SuE — in the database world, this includes the generation of benchmark data and their ingestion into the system. Second, the SuE needs to undergo a warm-up phase, for instance filling internal buffers, to make sure that the behavior of the SuE reflects a realistic use. Third, the actual evaluation is run. In the case of a database benchmark, this is the execution of the predefined queries in a given query mix. The evaluation finally generates data which at the end needs to be analyzed.

In most cases, these steps are implemented to a large extent by means of manual activities and are highly repetitive. Even in cases of (semi-)automated evaluations, they need to be re-started over and over again with varying parameters.

Ideally, a complete set of evaluation runs that systematically and thoroughly assess a given parameter space of an SuE can be fully automated. The only requirement on an SuE is that its evaluation workflow does not require any human interaction, allowing such SuEs to be evaluated using a generic evaluation toolkit. In order to provide systems *Evaluations-as-a-Service (EaaS)*, such a toolkit, once linked to the SuE, has to fulfill the following requirements: (i) It has to feature an easy to use UI for defining new experiments, for scheduling their execution, for monitoring their progress, and for analyzing their results. (ii) The toolkit has to support different SuEs at the same time and in particular parallel executions of benchmarks of these systems. (iii) To allow SuEs to be considered in long-running evaluations, the toolkit has to exhibit a high level of reliability. This includes an automated failure handling and the recovery of failed evaluation runs. (iv) The toolkit has to provide mechanisms for archiving the results of the evaluations as well as of all parameter settings which have led to these results. (v) Already existing evaluation clients should be easily integrable into the toolkit which also has to support developers in building new clients. (vi) The toolkit has to offer a large set of basic analysis functions (e.g., different types of diagrams), support the extension by custom ones, and provide standard metrics for measurements (e.g., execution time). Finally, (vii) it should be easy to apply the toolkit to new SUEs.

In this paper, we introduce Chronos, a generic evaluation toolkit for systematic systems evaluations. The contribution of this paper is twofold: First, we describe the functionality of Chronos which automates the entire evaluation workflow and assists users in the analysis and visualization of the evaluation results. Second, we demonstrate how Chronos can be configured for the evaluation of different SuEs and for the systematic analysis of a complete evaluation space by using two different storage engines in MongoDB[1] as running example.

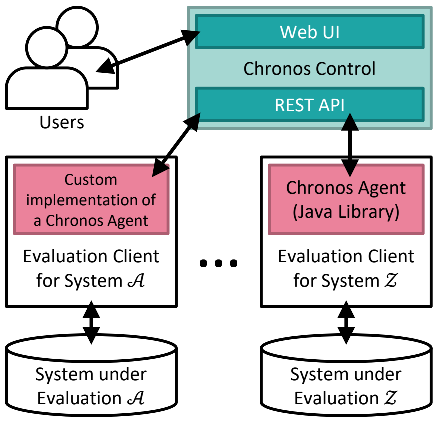
Since its first version, Chronos has been used for the evaluation of several research prototypes, including ADAM$_{pro}$ [6], Beowulf [9], Icarus [10], and Polypheny-DB [11] as well as in various student evaluation activities. It has been released on GitHub[2] under the MIT open source license.

The remainder of this paper is structured as follows: In Section 2 we introduce Chronos and Section 3 shows Chronos at work. Section 4 discusses related work and Section 5 concludes.

2 CHRONOS

Chronos is a toolkit for defining, monitoring, and analyzing the results of evaluations. It consists out of two main building blocks (see Figure 1): First, *Chronos Control* (in green), the heart of the evaluation toolkit that provides a web UI and a RESTful API for managing evaluations; second, the *Chronos Agents* (in red)

[1] https://www.mongodb.com
[2] https://github.com/Chronos-EaaS



Users
Web UI
Chronos Control
REST API
Custom implementation of a Chronos Agent
Evaluation Client for System $\mathcal{A}$
Chronos Agent (Java Library)
Evaluation Client for System $\mathcal{Z}$
System under Evaluation $\mathcal{A}$
System under Evaluation $\mathcal{Z}$

Figure 1: Chronos Toolkit Architecture Overview

which interact with the (existing) evaluation clients and which leverage the REST API to get the required information to perform the evaluations. While it is possible to write a custom Chronos Agent, there is also a generic Chronos Agent available for Java. This allows to easily integrate Chronos into existing projects. In this section, we present the implementation of Chronos and how to integrate it into existing projects.

2.1 Data Model

The data model of Chronos contains projects, experiments, evaluations, jobs, systems, and deployments.

Project. A project is an organizational unit which groups experiments and allows multiple users to collaborate on a specific evaluation. Access permissions are handled at the level of projects so that every member of a project has access to all experiments, evaluations, and their results. Users can archive entire projects, i.e., make their evaluation settings and the results persistent.

Experiment. An experiment is the definition of an evaluation with all its parameters; when executed, it results in the creation of an evaluation. Like projects, experiments can be archived.

Evaluation. An evaluation is the run of an experiment and consists of one or multiple jobs. If the objective of an evaluation is, for example, to compare the performance of two storage engines of a database system for different numbers of threads, every job would execute the benchmark for a specific number of threads for each engine. Depending on the evaluation, the execution of jobs can be parallelized if there are multiple identical deployments of the SuE.

Job. A job is a subset of an evaluation, e.g., the run of a benchmark for a specific set of parameters and a given DB storage engine. The result of every job is stored together with its log output. A job can be in one of the following states: *scheduled*, *running*, *finished*, *aborted*, or *failed*. Jobs which are in the status *scheduled* or *running* can be aborted and those which are *failed* can be re-scheduled.

Result. A result belongs to a job and consists of a JSON and a zip file. Every data item which is required for the analysis within Chronos Control is stored in the JSON file. Additional results can be stored in the zip file (e.g., for further analysis outside of Chronos).

System. A system is the internal representation of an SuE. For every SuE, it is defined which parameters the SuE expects, how the results are structured, and how they should be visualized.

Deployment. A deployment is an instance of an SuE in a specific environment. There can be multiple deployments of an SuE at the same time. Deployments serve two purposes: First, they allow to simultaneously execute evaluations in different (hardware) environments or different versions of the SuE; second, they allow to parallelize the evaluation in case of multiple identical deployments.

2.2 Implementation

In the following section, we give details on the implementation of Chronos' architecture as depicted in Figure 1.

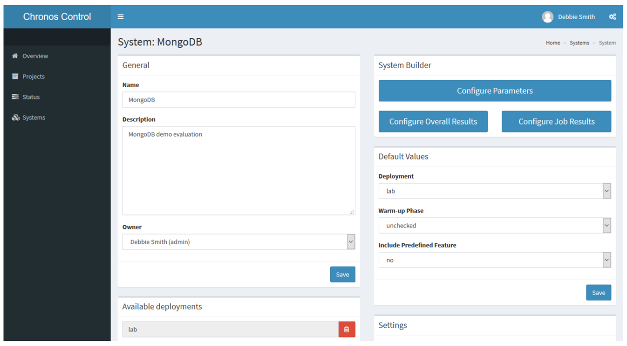
Chronos Control. It is designed as a web application allowing the management and analysis of evaluations using common web browsers. It offers a RESTful web service for clients benchmarking the SuEs. Chronos Control has only a few run-time requirements: Apache HTTP Server[3], PHP[4], MySQL[5] or MariaDB[6], and git[7]. For the SuE extensions, Mercurial[8] is also supported. The provided installation script handles the complete setup of Chronos including the creation of the necessary database schema and user account.

User Interface. Chronos' web UI is based on Bootstrap[9] and comes with an advanced session and role-based user management to support the deployment in a multi-user environment. A key feature of the Web UI is its modular architecture which enables the easy integration of different SuEs. For every SuE, only the available parameters for an experiment and information on how the results are to be visualized have to be specified. Both can be done completely UI-based. Chronos Control therefore provides several parameter and diagram types. Parameter types include Boolean, check box, and value types as well intervals and ratios. For the result visualization, Chronos provides bar, line, and pie diagrams. If more parameter types and diagrams are required, the built-in set of types can be extended by providing an external repository containing PHP scripts with additional implementations.

REST Interface. Chronos' REST API is used for both the clients requesting information to perform their benchmarks (e.g., requesting job descriptions or submitting results) and for the integration of the Chronos toolkit into existing evaluation workflows. For this, the API offers methods to, for example, schedule an evaluation which is caused by a successful build of the SuEs build bot. To support the smooth evolution of Chronos, the API is versioned. This allows new clients to simultaneously use the newly developed features while other clients still use older versions of the REST API.

Chronos Agent. Chronos Agents are clients or client libraries connecting to Chronos' REST API that perform or trigger the actual evaluation workload. Agents are essential to link existing or newly developed evaluation clients of the SuEs with Chronos. An agent can be implemented in any language supporting the

[3] https://httpd.apache.org/
[4] https://secure.php.net/
[5] https://mysql.com/
[6] https://mariadb.org/
[7] https://git-scm.com/
[8] https://mercurial-scm.org/
[9] https://getbootstrap.com/



System: MongoDB

General

Name: MongoDB

Description: MongoDB demo evaluation

Owner: Debbie Smith (admin)

Available deployments: lab

Default Values

Deployment: lab

Warm-up Phase: unchecked

Include Predefined Feature: no

Figure 2: Configuration of a System in Chronos' Web UI

access to the RESTful web service. Along with Chronos Control, we provide a reference implementation of a generic agent library written in Java (also available on GitHub[2]). This reference implementation handles all the communication with Chronos Control including the upload of the results via HTTP or FTP. The latter allows to use a different server or a NAS for storing the results which also reduces the load and storage requirements on the Chronos Control server. During its run, the agent periodically sends the output of the logger to Chronos Control and the agent library allows to easily update the progress of the evaluation. Furthermore, the agent library already measures basic metrics which are returned to Chronos Control along with the results.

Integrating the Chronos Agent library into an existing evaluation client is the only part which requires programming. All other steps can be done completely UI-based. However, the required amount of programming is rather negligible, since the agent library already provides an interface with all necessary methods to be implemented. Depending on the existing evaluation client, this usually narrows down to calling already existing methods of the evaluation client.

3 CHRONOS AT WORK

In this section, we present the two main workflows supported by Chronos: First, the registration of an SuE in the Chronos toolkit, and second, the necessary steps to perform an actual evaluation.

The first workflow demonstrates how easy it is to integrate the Chronos Agent into an existing evaluation client using the Java library. After building a Chronos-enabled evaluation client and setting-up an instance of the SuE, it needs to be registered in Chronos Control. This can be done by either specifying the parameters required for the evaluation client and the SuE in Chronos Control or by providing a path to a git or mercurial repository. Figure 2 shows the overview page of a system in the Chronos Web UI. Once the system has been created and its parameters have been configured, the first workflow is finished and Chronos is ready for executing evaluations against this SuE.

The second workflow starts with the creation of a project. Afterwards, one or multiple experiments are defined (Figure 3a). To schedule work for a Chronos Agent, an evaluation needs to be created which consists of one or multiple jobs as depicted in Figure 3b. Figure 3c shows the overview page of a job providing the current status of the job including its progress and the log output. Furthermore, it allows to abort a scheduled or running job or to reschedule a failed one. The timeline shows all events associated with this job. When the Chronos Agent has finished its work, the evaluation results are visualized (Figure 3d).

The separation of experiments and evaluations comes in handy if certain evaluations need to be repeated multiple times. This is the case during the development of an SuE, for example, in the bug-fixing phase, or for the quality assurance monitoring the performance of an SuE over subsequent change sets.

A demonstration that has been prepared to show Chronos' capabilities considers two workflows using the comparative evaluation of two storage engines of MongoDB (wiredTiger and mmapv1) as an example. This demonstration allows to create short running evaluations for the two MongoDB deployments and to directly analyze the results in the Chronos Web UI.

The MongoDB Chronos agent is available on GitHub[2] and the demo that will be presented at the conference is summarized in a short video[10] that shows how Chronos can be used for the comparative evaluation of the two storage engines of MongoDB.

4 RELATED WORK

Like the Chronos toolkit, the *PEEL framework* introduced in [1] automates the evaluation process and helps improving the reproducibility of evaluations. PEEL, however, is designed with a focus on Machine Learning applications running on frameworks like Hadoop MapReduce, Spark, Flink, and others. In PEEL, the SuE and the experiments are described using XML or Scala classes. These documents are compiled into a bundle containing the PEEL framework, the required configuration, and the evaluation data sets. The bundle is then deployed on the target machine running the SuE.

PROVA! [7] is a distributed workflow and system management tool for benchmarks on HPC systems. It allows to easily benchmark applications on different systems and architectures. Similar to Chronos it visualizes the results. While Chronos is a general-purpose tool for all kinds of evaluations and benchmarks with a focus on database evaluations, PROVA! is specifically tailored to the HPC domain.

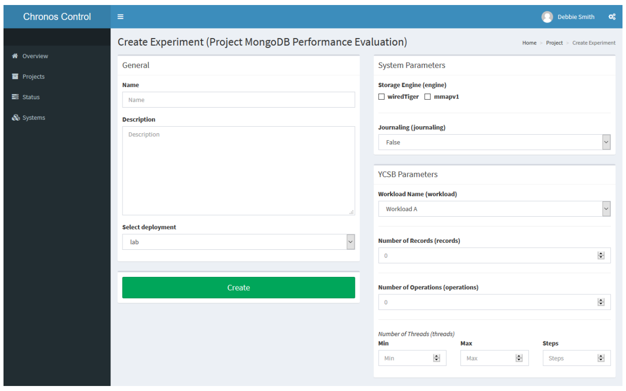
OLTP-Bench [5] is a benchmarking framework which provides implementations for in total 15 different transactional, web-oriented, and feature testing benchmarks including YCSB [4], TPC-C[11], and CH-benCHmark [3]. In contrast to OLTP-Bench, Chronos addresses the complete evaluation workflow and thus also includes the definition of the experiments and the analysis of the results. However, a combination of Chronos (automation of the evaluation workflow) and OLTP-Bench (definition of various benchmarks) would even further facilitate the definition, set-up, execution, and analysis of evaluations. In our future work, we thus plan to develop a Chronos Agent that wraps the OLTP-Bench so as to combine both systems.

The *TREET* testbed [8] allows developers of trust and reputation systems (e.g., online marketplaces) to evaluate their systems using standardized test cases. Further, TREET can be flexibly extended with custom agents and test cases allowing the testing of the developer's application. Like Chronos, but for a different domain, TREET supports the execution of experiments and the comparison between different trust and reputation systems.

5 CONCLUSION

In this paper, we have presented the evaluation toolkit Chronos, a first step towards the concept of Evaluation-as-a-Service. Chronos automates the entire evaluation workflow and assists users in the results analysis. The Chronos toolkit is available on

[10]https://youtu.be/fNmsZH4HOl0
[11]http://www.tpc.org/tpcc/



Create Experiment (Project MongoDB Performance Evaluation)

General

Name: Name

Description: Description

Select deployment: lab

System Parameters

Storage Engine (engine): ☐ wiredTiger ☐ mmapv1

Journaling (journaling): False

YCSB Parameters

Workload Name (workload): Workload A

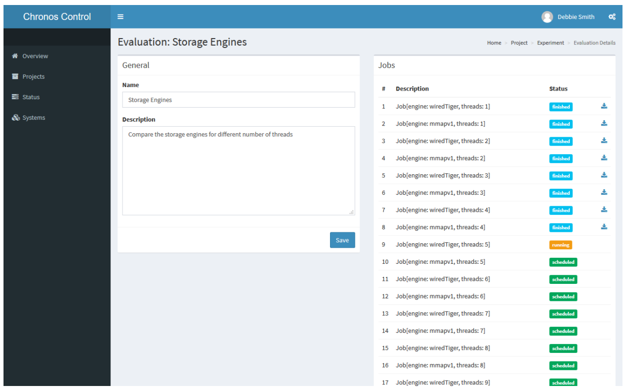
Number of Records (records): 0

Number of Operations (operations): 0

Number of Threads (threads)

Min: Min | Max: Max | Steps: Steps

(a) Creation of an Experiment



Evaluation: Storage Engines

General

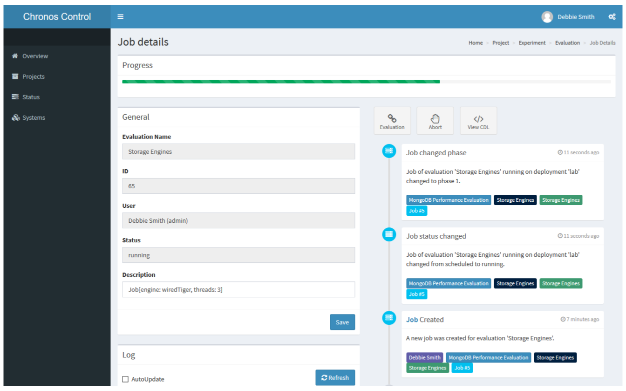
Name: Storage Engines

Description: Compare the storage engines for different number of threads

Jobs

#	Description	Status
1	Job[engine: wiredTiger, threads: 1]	finished
2	Job[engine: mmapv1, threads: 1]	finished
3	Job[engine: wiredTiger, threads: 2]	finished
4	Job[engine: mmapv1, threads: 2]	finished
5	Job[engine: wiredTiger, threads: 3]	finished
6	Job[engine: mmapv1, threads: 3]	finished
7	Job[engine: wiredTiger, threads: 4]	finished
8	Job[engine: mmapv1, threads: 4]	finished
9	Job[engine: wiredTiger, threads: 5]	running
10	Job[engine: mmapv1, threads: 5]	scheduled
11	Job[engine: wiredTiger, threads: 6]	scheduled
12	Job[engine: mmapv1, threads: 6]	scheduled
13	Job[engine: wiredTiger, threads: 7]	scheduled
14	Job[engine: mmapv1, threads: 7]	scheduled
15	Job[engine: wiredTiger, threads: 8]	scheduled
16	Job[engine: mmapv1, threads: 8]	scheduled
17	Job[engine: wiredTiger, threads: 9]	scheduled

(b) Details of a Running Evaluation



Job details

Progress

General

Evaluation Name: Storage Engines

ID: 65

User: Debbie Smith (admin)

Status: running

Description: Job[engine: wiredTiger, threads: 3]

Log

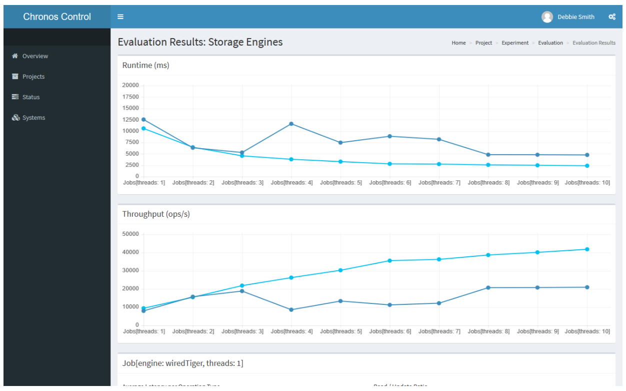
☐ AutoUpdate

Job changed phase — 11 seconds ago
Job of evaluation 'Storage Engines' running on deployment 'lab' changed to phase 1.
MongoDB Performance Evaluation | Storage Engines | Storage Engines | Job #5

Job status changed — 11 seconds ago
Job of evaluation 'Storage Engines' running on deployment 'lab' changed from scheduled to running.
MongoDB Performance Evaluation | Storage Engines | Storage Engines | Job #5

Job Created — 7 minutes ago
A new job was created for evaluation 'Storage Engines'.
Debbie Smith | MongoDB Performance Evaluation | Storage Engines | Storage Engines | Job #5

(c) Details of a Running Job



(d) Basic Result Analysis done by Chronos Control

Figure 3: Basic Evaluation Workflow

GitHub[2]. Future releases of Chronos will be extended with the functionality for setting up the infrastructure of an SuE automatically, for example, in an on-premise cluster or in the Cloud. Also, we plan to release additional reference implementations of agent libraries, for instance for Python.

ACKNOWLEDGMENTS

The work was partly funded by the Swiss National Science Foundation (SNSF) in the context of the project Polypheny-DB (contract no. 200021_172763). The authors would like to thank Filip-Martin Brinkmann and Ivan Giangreco. Their feedback on the evaluation of ADAM$_{pro}$ [6] and PolarDBMS [2] on the basis of Chronos has significantly improved the system. Moreover, the authors like to thank Dina Sayed for her helpful comments on an earlier version of this paper.

REFERENCES

[1] Christoph Boden, Alexander Alexandrov, Andreas Kunft, Tilmann Rabl, and Volker Markl. 2017. PEEL: A Framework for Benchmarking Distributed Systems and Algorithms. In *Performance Evaluation and Benchmarking for the Analytics Era (TPCTC)*. Springer, Munich, Germany, 9–24. DOI: http://dx.doi.org/10.1007/978-3-319-72401-0_2

[2] Filip-Martin Brinkmann and Heiko Schuldt. 2015. Towards Archiving-as-a-Service: A Distributed Index for the Cost-effective Access to Replicated Multi-Version Data. In *Proceedings of the 19th International Database Engineering & Applications Symposium (IDEAS'15)*. ACM, Yokohama, Japan, 81–89. DOI: http://dx.doi.org/10.1145/2790755.2790770

[3] Richard Cole, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, Florian Waas, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, and Thomas Neumann. 2011. The Mixed Workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*. ACM, Athens, Greece, 1–6. DOI: http://dx.doi.org/10.1145/1988842.1988850

[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of the 1st ACM Symposium on Cloud Computing*. ACM, Indianapolis, Indiana, USA, 143–154. DOI: http://dx.doi.org/10.1145/1807128.1807152

[5] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288. DOI: http://dx.doi.org/10.14778/2732240.2732246

[6] Ivan Giangreco and Heiko Schuldt. 2016. ADAM$_{pro}$: Database Support for Big Multimedia Retrieval. *Datenbank-Spektrum* 16, 1 (2016), 17–26. DOI: http://dx.doi.org/10.1007/s13222-015-0209-y

[7] Danilo Guerrera, Antonio Maffia, and Helmar Burkhart. 2019. Reproducible stencil compiler benchmarks using prova! *Future Generation Computer Systems* 92 (2019), 933–946. DOI: http://dx.doi.org/10.1016/j.future.2018.05.023

[8] Reid Kerr and Robin Cohen. 2010. TREET: the Trust and Reputation Experimentation and Evaluation Testbed. *Electronic Commerce Research* 10, 3 (2010), 271–290. DOI: http://dx.doi.org/10.1007/s10660-010-9056-y

[9] Alexander Stiemer, Ilir Fetai, and Heiko Schuldt. 2016. Analyzing the Performance of Data Replication and Data Partitioning in the Cloud: The BEOWULF Approach. In *IEEE International Conference on Big Data*. IEEE, Washington DC, USA, 2837–2846. DOI: http://dx.doi.org/10.1109/BigData.2016.7840932

[10] Marco Vogt, Alexander Stiemer, and Heiko Schuldt. 2017. Icarus: Towards a Multistore Database System. In *IEEE International Conference on Big Data*. IEEE, Boston, MA, USA, 2490–2499. DOI: http://dx.doi.org/10.1109/BigData.2017.8258207

[11] Marco Vogt, Alexander Stiemer, and Heiko Schuldt. 2018. Polypheny-DB: Towards a Distributed and Self-Adaptive Polystore. In *IEEE International Conference on Big Data*. IEEE, Seattle, WA, USA, 3364–3373. DOI: http://dx.doi.org/10.1109/BigData.2018.8622353

skyex: an R Package for Entity Linkage

Suela Isaj
Aalborg University
suela@cs.aau.dk

Torben Bach Pedersen
Aalborg University
tbp@cs.aau.dk

ABSTRACT

As the data is becoming bigger, more heterogeneous, and originating from different sources, the availability of the same information in different forms leads to various entity linkage problems. We demonstrate our skyex package, an R package that supports all three steps of entity linkage: blocking, pairwise comparison, and labeling. Thus, the user can solve the whole process using skyex, but not necessarily; the skyex modules are independent, meaning that the user can easily integrate them with other packages or even other environments. Additionally, we are the first to provide the implementation of two skyline-based algorithms (*SkyEx-F* and *SkyEx-D*) that can label the compared pairs without the need for weights, scoring functions, etc. skyex supports the typical workflow of entity linkage, using minimalist, user-friendly function calls.

1 INTRODUCTION

The *entity linkage* problem, sometimes called *data matching*, *entity resolution*, *duplicate detection*, *reconciliation*, etc., detects different records that belong to the same entity. Even though the process varies in different domains, the main steps are the same: blocking, pairwise comparison, and labeling the pairs (Fig.1). The entity linkage process starts with a set of entities that might contain duplicates. First, a blocking method is used to group entities that show a certain level of similarity and are of interest to compare further. Then, the pairwise comparison step compares the entities in the same blocks, e.g., using similarity metrics of the attributes of the entities or comparing the structure of their connections. Finally, the labeling step decides whether a pair of candidates belongs to the same entity or not. The entity linkage process results in a set of labeled pairs.

We present an R package, skyex, that supports all three steps of the entity linkage problem. In the labeling step, we provide the novel *SkyEx-F* and *SkyEx-D* algorithms in [6, 8]. The R language is in the top five languages of data science, and even more importantly, R is the second most used software in data science scientific papers, corresponding to 50,000 articles [1]. Moreover, R is used by different industries besides academia, such as healthcare, government, insurance, etc., where entity resolution is a common obstacle [2]. The current entity linkage tools [1–4, 9–11] offer rule-based solutions with blocking and comparison functions [3, 10], crowdsourcing solutions [4, 9], or machine learning solutions [1, 2, 11]. In contrast to all the current tools, we contribute with a Labeling module that implements two novel algorithms (*SkyEx-F* and *SkyEx-D*) [6, 8], which can label the pairs without the need of weights, scoring functions, or exhaustive experiments. In order to support the full entity linkage workflow, we provide functions to perform blocking based on text and spatial attributes, and we offer a module for textual, spatial, semantic pairwise comparison. Similarly to [2], we support analysis and visualization functions that assist in the interpretation of the results and assessing the quality of the labeling. Analogously to [11] that uses Python, skyex uses the R ecosystem and can be easily integrated with other packages, in contrast to the current standalone tools. Finally, we demonstrate the different scenarios that can be supported by our tool using three real-world datasets. Overall, skyex solves the entity linkage problem with minimal effort and background knowledge.

The remainder of the paper continues with the functionalities covered by our skyex package in Section 2, a description of our on-site demonstration in Section 3, and finally, concluding in Section 4.

[1]http://r4stats.com/articles/popularity/
[2]https://stackoverflow.blog/2017/10/10/impressive-growth-r/

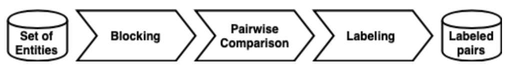
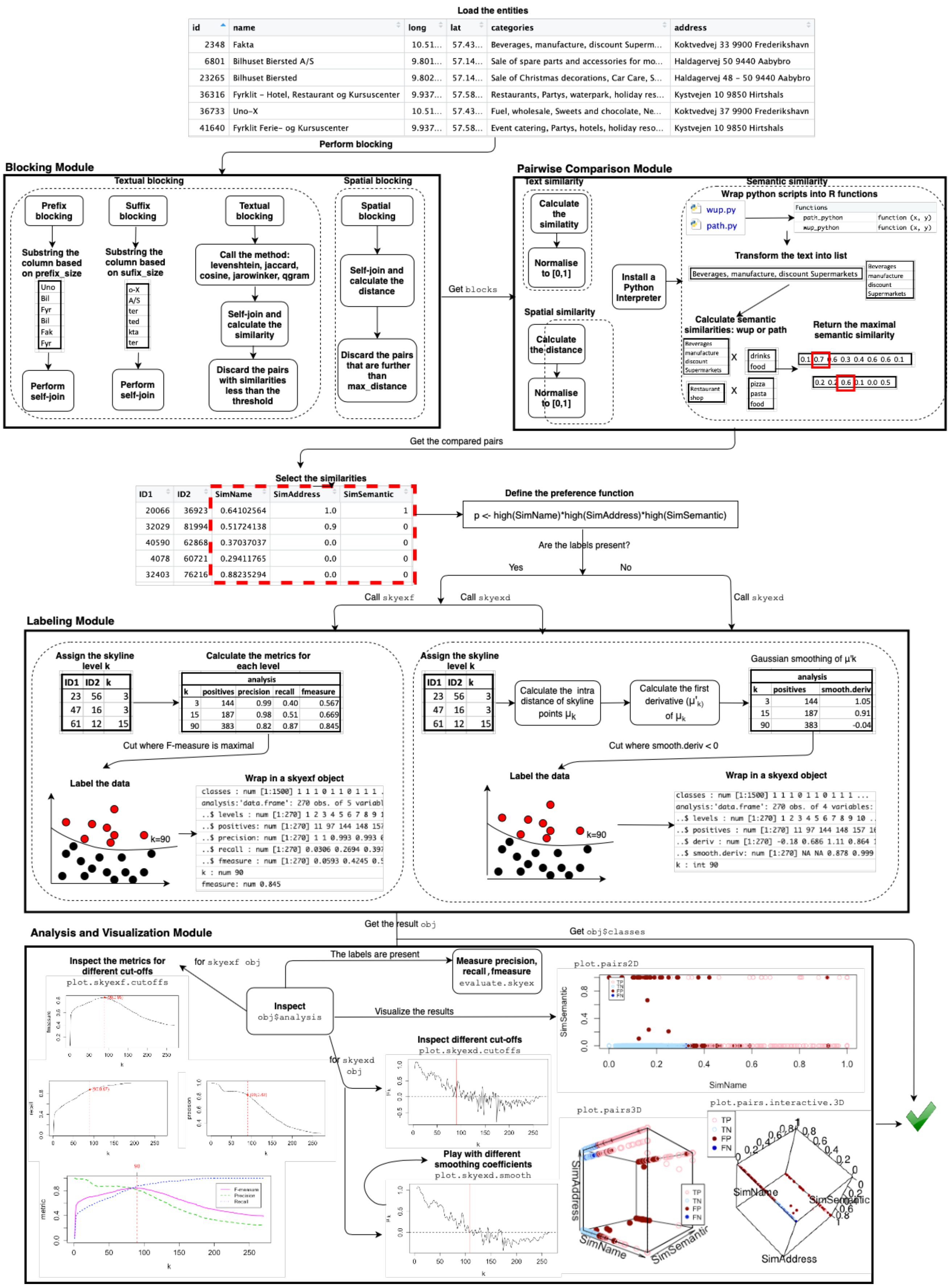


Figure 1: The entity linkage process

2 SKYEX PACKAGE FUNCTIONALITIES

The skyex package is composed of 17 functions corresponding to four main modules: Blocking, Pairwise Comparison, Labeling, and Analysis and Visualization. The workflow of using skyex is illustrated in Fig. 2. The user starts with a dataframe (a common data type for storing tables in R) of entities. In order to illustrate the workflow and our functions, we will use a real-world dataset of spatial entities extracted as in [5] and used in the experiments of [8]. The dataset contains spatial entities in the North Denmark region, originating from four sources, Google Places, Yelp, Krak (online yellow pages in Denmark, www.krak.dk), and Foursquare. We also introduce the running example of six records of entities (entities) from this dataset in Fig. 2, which are identified by an ID, by geographic coordinates latitude and longitude, categories that explain the type of spatial entity, and the address.

Blocking module. After loading the data, we can use a blocking technique (textual or spatial) from the Blocking module. The textual blocking is executed by the textual.blocking function, choosing a similarity metric among *levenshtein*, *cosine*, *jaccard*, *jaro-winker*, and *qgram*, and setting a maximal distance allowed. For example, textual.blocking on the attribute "name" with *levenshtein* and maximal distance 4 will group the entities with names "Bilhuset Biersted A/S" and "Bilhuset Biersted" (from entities in Fig. 2). Note that textual.blocking is accurate but time-consuming. Alternatively, prefix.blocking and suffix.blocking produce faster results. Besides, in some domains, e.g., for species names, these methods can be more relevant than textual blocking. For spatial entities, being spatially close is often a better indicator of block quality than the name. For example, two records with the same name, e.g., Fakta supermarkets in different cities, are two different entities. spatial.blocking creates blocks of entities that are at most *max_distance* meters apart. The code snippets for these blocking methods are as follows:


Load the entities
Perform blocking
Blocking Module
Textual blocking
Spatial blocking
Prefix blocking
Substring the column based on prefix_size
Uno
Bil
Fyr
Bil
Fak
Fyr
Perform self-join
Suffix blocking
Substring the column based on sufix_size
o-X
A/S
ter
ted
kta
ter
Perform self-join
Textual blocking
Call the method: levenshtein, jaccard, cosine, jarowinker, qgram
Self-join and calculate the similarity
Discard the pairs with similarities less than the threshold
Spatial blocking
Self-join and calculate the distance
Discard the pairs that are further than max_distance
Get blocks
Pairwise Comparison Module
Text similarity
Calculate the similatity
Normalise to [0,1]
Spatial similarity
Calculate the distance
Normalise to [0,1]
Install a Python Interpreter
Semantic similarity
Wrap python scripts into R functions
wup.py
path.py
Functions
path_python function (x, y)
wup_python function (x, y)
Transform the text into list
Beverages, manufacture, discount Supermarkets
Beverages
manufacture
discount
Supermarkets
Calculate semantic similarities: wup or path
Return the maximal semantic similarity
X
drinks
food
0.1 0.7 0.6 0.3 0.4 0.6 0.6 0.1
Restaurant
shop
X
pizza
pasta
food
0.2 0.2 0.6 0.1 0.0 0.5
Get the compared pairs
Select the similarities
Define the preference function
p <- high(SimName)*high(SimAddress)*high(SimSemantic)
Are the labels present?
Yes
No
Call skyexf
Call skyexd
Call skyexd
Labeling Module
Assign the skyline level k
Calculate the metrics for each level
Cut where F-measure is maximal
Label the data
k=90
Wrap in a skyexf object
Assign the skyline level k
Calculate the intra distance of skyline points μk
Calculate the first derivative (μ'k) of μk
Gaussian smoothing of μ'k
Cut where smooth.deriv < 0
Label the data
k=90
Wrap in a skyexd object
Get the result obj
Get obj$classes
Analysis and Visualization Module
Inspect the metrics for different cut-offs
plot.skyexf.cutoffs
for skyexf obj
The labels are present
Inspect obj$analysis
Measure precision, recall, fmeasure
evaluate.skyex
Visualize the results
for skyexd obj
Inspect different cut-offs
plot.skyexd.cutoffs
Play with different smoothing coefficients
plot.skyexd.smooth
plot.pairs2D
SimSemantic
SimName
TP
TN
FP
FN
plot.pairs3D
SimAddress
SimName
SimSemantic
plot.pairs.interactive.3D
SimName
SimSemantic
SimAddress
fmeasure
recall
precision
metric
F-measure
Precision
Recall
k

id	name	long	lat	categories	address
2348	Fakta	10.51…	57.43…	Beverages, manufacture, discount Superm…	Koktvedvej 33 9900 Frederikshavn
6801	Bilhuset Biersted A/S	9.801…	57.14…	Sale of spare parts and accessories for mo…	Haldagervej 50 9440 Aabybro
23265	Bilhuset Biersted	9.802…	57.14…	Sale of Christmas decorations, Car Care, S…	Haldagervej 48 – 50 9440 Aabybro
36316	Fyrklit – Hotel, Restaurant og Kursuscenter	9.937…	57.58…	Restaurants, Partys, waterpark, holiday res…	Kystvejen 10 9850 Hirtshals
36733	Uno-X	10.51…	57.43…	Fuel, wholesale, Sweets and chocolate, Ne…	Koktvedvej 37 9900 Frederikshavn
41640	Fyrklit Ferie- og Kursuscenter	9.937…	57.58…	Event catering, Partys, hotels, holiday reso…	Kystvejen 10 9850 Hirtshals

ID1	ID2	SimName	SimAddress	SimSemantic
20066	36923	0.64102564	1.0	1
32029	81994	0.51724138	0.9	0
40590	62868	0.37037037	0.0	0
4078	60721	0.29411765	0.0	0
32403	76216	0.88235294	0.0	0

ID1	ID2	k
23	56	3
47	16	3
61	12	15

k	positives	precision	recall	fmeasure
3	144	0.99	0.40	0.567
15	187	0.98	0.51	0.669
90	383	0.82	0.87	0.845

```
classes : num [1:1500] 1 1 1 0 1 1 0 1 1 1 .
analysis:'data.frame': 270 obs. of 5 variabl
..$ levels : num [1:270] 1 2 3 4 5 6 7 8 9 1
..$ positives: num [1:270] 11 97 144 148 157
..$ precision: num [1:270] 1 1 0.993 0.993 0
..$ recall : num [1:270] 0.0306 0.2694 0.397
..$ fmeasure : num [1:270] 0.0593 0.4245 0.5
k : num 90
fmeasure: num 0.845
```

ID1	ID2	k
23	56	3
47	16	3
61	12	15

k	positives	smooth.deriv
3	144	1.05
15	187	0.91
90	383	-0.04

```
classes : num [1:1500] 1 1 1 0 1 1 0 1 1 1 ...
analysis:'data.frame': 270 obs. of 4 variables:
..$ levels : num [1:270] 1 2 3 4 5 6 7 8 9 10 ..
..$ positives : num [1:270] 11 97 144 148 157 16
..$ deriv : num [1:270] -0.18 0.686 1.11 0.864 1
..$ smooth.deriv: num [1:270] NA NA 0.878 0.999
k : int 90
```

Figure 2: skyex workflow

```
#Textual blocking using levenshtein distance and max_distance=4
blocks<-textual.blocking(data=entities, column = "name",
         method = "levenshtein", max_distance = 4)
#Prefix blocking for the first 4 characters
blocks<-prefix.blocking(data=entities, column = "name", prefix_size = 4)
#Spatial blocking for entities at most 50 m apart
blocks<-spatial.blocking(data=example, longitude = "long",
         latitude = "lat", max_distance = 50)
```

Pairwise Comparison module. The Blocking module outputs a dataframe of pairs, which saves the user from the task of having to create the pairs from each block. The Pairwise Comparison module offers three functions that compare text syntactically and semantically, as well as spatial attributes. Moreover, all three functions output normalized values, which can be directly used in the Labeling module. `text.similarity` calculates the similarity of the pairs based on a text attribute using similarity metrics such as *levenshtein*, *cosine*, *jaccard*, *jaro-winker*. *levenshtein* similarity is calculated using the formula in [7, 8] in order to return a normalized value. `spatial.similarity` also requires a maximal distance for the normalization. For example, for a $max\_distance = 70$, "Uno-X" and "Fakta" will have a similarity of 0.0, because their distance of 83 meters is beyond the threshold. In the case of Bilhuset Biersted A/S and Bilhuset Biersted, this distance is 63 meters, which translates to a similarity of 0.09.

Regarding the semantic similarity, our work in [8] uses the Wu&Palmer metric from Wordnet. There exists a `wordnet` library in R, but it does not provide the metrics. Moreover, Wu&Palmer needs the whole path of both words that need to be compared, which in R, it could be resolved only through recursive calls. Through experimentation, this implementation turned out to be non-efficient. Thus, we include two Python scripts in the `skyex` package for two different metrics in Wordnet. These scripts are wrapped in R functions; thus, the user only needs to have a Python interpreter installed and give its path to the R function. The code for the pairwise comparisons is as follows:

```
#Text similarity using cosine
blocks$SimName<-text.similarity(data=blocks, method = "cosine",
                 column1 = "name.x", column2 = "name.y")
#Spatial similarity with max_distance=70
blocks$SimSpatial<-spatial.similarity(data=blocks, lat1 = "lat.x",
                    long1 = "long.x", lat2 = "lat.y", long2 = "long.y",
                    max_distance = 70)
#Semantic similarity with Wu&Palmer
blocks$SimSemantic<-semantic.similarity(data=blocks,
                     column1 = "categories.x", column2 = "categories.y",
                     pythonpath = "/Users/..", method = "wup" )
```

Labeling module. After the pairs are compared, the user can decide which similarities should go into the labeling process. Usually, he would select those similarities that are likely to indicate a match, e.g., the similarity of the names of the entities. We will consider the similarity of the name "SimName", the similarity of the address "SimAddress", and the semantic similarity of the categories "SimSemantic" as in [8]. Then, the user decides on the preference function for the Pareto Optimality calculations. In our case, we prefer a high value for each similarity. Depending on the availability of the labels, the user can choose between running `skyexf` or `skyexd`, corresponding to the threshold-based *SkyEx-F*, or to the fully unsupervised *SkyEx-D*, respectively [6]. *SkyEx-F* finds that skyline level k that separates best the classes and maximizes the F-measure. It starts with assigning the skyline to all the points and then checking different cut-offs while measuring precision, recall, and f-measure. Finally, it labels the data, and the `skyexf obj` is returned, containing the classes, an analysis data frame, the proposed cut-off k, and the corresponding f-measure.

For unlabeled data, *SkyEx-D* finds the skyline level k where the mean distance of the points in the positive class starts to drop, meaning that we are entering the denser area of the negative class. It starts by assigning the corresponding skyline to each point; then, calculating the cumulative mean distance in the positive class and its first derivative; later, finding where the first derivative becomes negative for the first time. Finally, *SkyEx-D* labels the data and wraps the classes, the analysis data frame, and the proposed cut-off k in a `skyexd obj`. Detailed explanations about both algorithms can be found in [6]. Our `skyex` package hides all the details above from the user, meaning that the processes inside the dotted line boxes (Fig. 2) are performed simply by the `skyexf` and `skyexd` function calls. The script for running both algorithms, storing the results of each labeling algorithm in separate objects, and attaching the predicted classes to the dataset is as follows:

```
#Define the preference
p<-high(SimName)*high(SimSemantic)*high(SimAddress)
#Call SkyEx-F algorithm and store the result in f.obj
f.obj<-skyexf(data=blocks, p=p, label="Class",posclass=1, negclass=0)
#Call SkyEx-D algorithm and store the result in d.obj
d.obj<-skyexd(data=blocks, p=p, simlist=c("SimName", "SimSemantic",
        "SimAddress"), posclass=1, negclass=0, smooth.coefficient=5)
blocks$fpred<-f.obj$classes
blocks$dpred<-d.obj$classes
```

We thus provide a labeling procedure that can be used with only two lines of code: defining the preference and calling the labeling function. However, for a more knowledgeable user, we offer the possibility to do analysis and visualize the results through the Analysis and Visualization module.

Analysis and Visualization module. To illustrate the analysis of the labeling, we will use the 1500 manually-labeled pairs in [8]. Additionally, this dataset is also available in our package under the name `pairsManual` and can be loaded simply by `data(pairsManual)`. The Analysis and Visualization module needs the output of the Labeling module as input, which is a `skyexd` or `skyexf` object. The raw analysis can be accessed simply by calling the dataframe `analysis` from *obj* (inspect `obj$analysis` in Fig. 2). In the case of a `skyexf` object, `analysis` contains all the cut-offs, the size of the positive class, precision, recall, and f-measure. In order to facilitate the exploration of `analysis`, the user can call `plot.skyexf.cutoffs`, which produces graphs that monitor the evolution of the metrics when passing to the deeper skylines (see Fig. 2). `plot.skyexf.cutoffs` by default plots the f-measure. However, it is possible to plot the precision and the recall separately, and also all metrics together. The code snippets for plotting the f-measure (first two), the precision, the recall, and all the metrics are as follows:

```
plot.skyexf.cutoffs(f.obj)
plot.skyexf.cutoffs(f.obj, "fmeasure")
plot.skyexf.cutoffs(f.obj, "precision")
plot.skyexf.cutoffs(f.obj, "recall")
plot.skyexf.cutoffs(f.obj, "all")
```

The resulting plots from the above script on `pairsManual` are shown in Fig. 2 in the Analysis and Visualization module. Understandably, precision is high in the first skylines because it is very likely that the pairs in the first skylines that we label as positives are actual positives, but it degrades while moving in deeper cut-offs. On the contrary, recall is always increasing, the more we label as positive, the more likely it is to find an actual positive. The F-measure gives the trade-off between both metrics. All graphs show the suggested cut-off by `f.obj` in the red dotted line. However, the user can explore different trade-offs for his problem. In that case, plotting all metrics in a graph (the last script) gives a better overview.

In the case of a `skyexd` object, `analysis` keeps the cut-offs, the size of the positive class, the first derivative, and the smoothed values. Similarly, `plot.skyexd.cutoffs` aids exploring the raw `analysis` by plotting the smoothed first derivative function for each cut-off. If the plot looks too "smoothed" or too "raw", it is possible to play with different smoothing coefficients without having to re-run `skyexd` again by calling `plot.skyexd.smooth`. (see the code below). Fig. 2 shows the analysis of `skyexd`, which was run with `smooth.coefficient=5`, and also the results of `plot.skyexd.smooth(d.obj, 10)`. The higher the smoothing coefficient, the higher the cut-off k, since smoother values push the cut-off towards deeper skylines.

```
#Plot the first derivative and the current cut-off k
plot.skyexd.cutoffs(d.obj)
#Smooth the first derivative with 10
plot.skyexd.smooth(d.obj, 10)
```

`evaluate.skyex` can also be called as in the code below, to measure precision, recall, and f-measure when the labels are available. The values of these metrics will be printed in the console.

```
evaluate.skyex(prediction=d.obj$classes, labels=data$Class, posclass = 1)
```

Additionally, we offer user-friendly functions to plot the data and the *obj* results. We offer 2D plots, 3D plots, and interactive 3D plots, where the user can play and move the dimensions while looking at the data. The color of the points reflects if the pair is a true positive TP (an actual positive labeled as positive), a true negative TN (an actual negative labeled as negative), a false positive FP (an actual negative labeled as positive), and a false negative FN (an actual positive labeled as negative). The user can decide to change the colors of the points based on his preference. The code for these plots is as follows:

```
#Plot 2D using SimName and SimSemantic
plot.pairs2D(data=data, sim1="SimName", sim2="SimSemantic",
          prediction=f.obj$classes, labels=data$Class, posclass = 1)
#Plot 3D using SimName, SimSemantic, and SimAddress
plot.pairs3D(data=data, sim1="SimName", sim2="SimSemantic",sim3="SimAddress",
          prediction=f.obj$classes, labels=data$Class, posclass = 1)
#Plot 3D interactive plot using SimName, SimSemantic, and SimAddress
plot.pairs.interactive.3D(data=data, sim1="SimName", sim2="SimSemantic",
          sim3="SimAddress", prediction=f.obj$classes,
          labels=data$Class, posclass = 1)
```

Fig. 2 shows the results of `pairsManual` with the three types of plots. These graphs can also be considered as an analysis since they show the problems with labeling and where to locate them. For example, it is noticeable that we have a bigger problem with the false positives then with the false negatives, thus if precision is fundamental to the domain, we could go back to the analysis and evaluation module and consider a smaller k for the cut-off. The interactive 3D plot offers a better view of the data points since it is possible to move and rotate the graph.

Summary. The workflow of `skyex` supports typical entity linkage tasks, from blocking to evaluating the quality of the labels. The Blocking, Pairwise Comparison, and Labeling modules are completely independent, which means that the user can decide to perform his own methods and still be able to connect to the workflow of `skyex`. The labeling task can be as simple as just calling two lines of code to get the classes and as detailed as performing analysis, playing with the parameters, visualizing the labels, and highlighting the errors, etc. Moreover, the user can always go back, choosing new similarities and new preferences until the results are satisfactory. The `skyex` package is dependent on `rPref`, `dplyr`, `fields`, `rgl`, `plot3D`, `smoother`, `fuzzyjoin`, `stringr`, `stringdist`, `geosphere`, `reticulate`, and `pracma` which support some basic functionalities in our functions. `skyexf` and `skyexd`) scale relatively well for an R environment; e.g. they run in less than a minute for 50,000 pairs, less than 15 minutes for 150,000 pairs, and around 1 hour for 300,000 pairs.

3 DEMONSTRATION OVERVIEW

In the on-site demonstration, the user can download `skyex`[3], which is publicly available in GitHub, by following the README instructions, or use our pre-installed R environment. We will provide three datasets: `entities` (2814 spatial entities in the North Denmark region with an ID, name, categories, and address) [8], `restaurants`[4] (a collection of 864 restaurant records with name, address, city, and type), and `pairsManual` (1500 labeled pairs with pre-compared similarities of the name, address, and categories) [8]. Additionally, we have published a full video[5] demonstrating our functionalities for all three datasets, and a short video[6] for the `restaurants` dataset. We will provide example scripts, which the user can adapt based on his preference. The user will start with different blocking techniques on `entities` and `restaurants`, discussing with us what would be a good blocking technique for this dataset. Afterwards, he can play with different similarity metrics and different thresholds for the pairwise comparison. Later, the user can decide either to continue with the dataset of pairs he created so far from `entities` and `restaurants`, or move to the pre-compared `pairsManual` and play with the labeling parameters. The user can try both algorithms and will be guided by us through the Analysis and Visualization module. He can try the visualizations (including the interactive plotting) in order to detect problems with the labeling. Finally, he can discuss with us the applicability of the method across domains and possibilities for improvement.

4 CONCLUSIONS AND FUTURE WORK

We introduced the `skyex` package, a user-friendly R package that supports all three steps of the entity linkage process. We demonstrated the functions of `skyex` with scripts and sample data, and supported the full workflow of the user. We showed that our Labeling module could solve the labeling problem with only two lines of code, but at the same time, offer the possibility for deeper analysis for the knowledgeable user. As future work, we intend to work on the scalability of our tool for big data, as well as on a similar package in Python.

REFERENCES

[1] E. C. Dragut et al. 2016. ORLF: A flexible framework for online record linkage and fusion. In *ICDE*. 1378–1381.
[2] A. Ebaid et al. 2019. EXPLAINER: Entity Resolution Explanations. In *ICDE*. 2000–2003.
[3] A. Elmagarmid et al. 2014. NADEEF/ER: Generic and interactive entity resolution. In *SIGMOD*. 1071–1074.
[4] Y. Govind et al. 2018. Cloudmatcher: a hands-off cloud/crowd service for entity matching. *PVLDB* 11, 12 (2018), 2042–2045.
[5] S. Isaj and T. B. Pedersen. 2019. Seed-Driven Geo-Social Data Extraction. In *SSTD*. 11–20.
[6] S. Isaj, T. B. Pedersen, and E. Zimányi. 2019. Multi-Source Spatial Entity Linkage. arXiv:1911.09016
[7] S. Isaj, N. B. Seghouani, and G. Quercini. 2019. Profile Reconciliation Through Dynamic Activities Across Social Networks. In *CAiSE*. 126–141.
[8] S. Isaj, E. Zimányi, and T. B Pedersen. 2019. Multi-Source Spatial Entity Linkage. In *SSTD*. 1–10.
[9] X. Ke et al. 2018. A demonstration of PERC: probabilistic entity resolution with crowd errors. *PVLDB* 11, 12 (2018), 1922–1925.
[10] L. Kolb, A. Thor, and E. Rahm. 2012. Dedoop: Efficient deduplication with hadoop. *PVLDB* 5, 12 (2012), 1878–1881.
[11] P. Konda et al. 2016. Magellan: toward building entity matching management systems over data science stacks. *PVLDB* 9, 13 (2016), 1581–1584.

[3]https://github.com/suelai/skyex
[4]source: https://www.cs.utexas.edu/users/ml/riddle/data.html
[5]https://youtu.be/TdxVsUtKRjw
[6]https://youtu.be/Zn8FOOh_xwA

Data Quality Checking for Machine Learning with MeSQuaL

[Demonstration paper]

Ugo Comignani
Aix Marseille Univ, Université de Toulon, CNRS, LIS, DIAMS
Marseille, France
ugo.comignani@lis-lab.fr

Noël Novelli
Aix Marseille Univ, Université de Toulon, CNRS, LIS, DIAMS
Marseille, France
noel.novelli@lis-lab.fr

Laure Berti-Équille
IRD, UMR ESPACE DEV, Montpellier
Univ de Toulon, AMU, CNRS, LIS, DIAMS, Marseille, France
laure.berti@ird.fr

ABSTRACT

This demo proposes MeSQuaL, a system for profiling and checking data quality before further tasks, such as data analytics and machine learning. MeSQuaL extends SQL for querying relational data with constraints on data quality and facilitates the verification of statistical tests. The system includes: (1) a query interpreter for SQuaL, the SQL-extended language we propose for declaring and querying data with data quality checks and statistical tests; (2) an extensible library of user-defined functions for profiling the data and computing various data quality indicators; and (3) a user interface for declaring data quality constraints, profiling data, monitoring data quality with SQuaL queries, and visualizing the results via data quality dashboards. We showcase our system in action with various scenarios on real-world data sets and show its usability for monitoring data quality over time and checking the quality of data on-demand.

1 INTRODUCTION

Assessing data quality is challenging and requires the detection and elimination of a variety of data quality problems, such as errors, duplicate, inconsistent, obsolete, and incomplete information [3, 10]. A wide range of methods for statistical analysis, constraint mining, consistency checking, and duplicate elimination has to be used [6] and their specifications can be complex for various reasons:

• *Data quality checking is a highly domain- and task-specific problem.* Data quality is multidimensional. A plethora of measurable dimensions can be used to characterize the quality of data with various indicators (e.g., value accuracy, consistency, completeness, freshness, or absence of duplicate records). Multiple techniques can be implemented to evaluate each dimension whose specification ultimately depends on the requirements of the user, the task at hand, and the application domain. Moreover, depending on the machine learning (ML) task, statistical assumptions must be verified before applying a given ML model, and the test results may ultimately influence the data preparation with a selection of specific data transformations accordingly.

• *Data quality checking is inherently a human-in-the-loop (HIL) process.* The user needs a tool offering a flexible and declarative way to define, evaluate, and check various data quality indicators and query the data with some data quality requirements in mind that can be made explicit.

• *Data quality checking is a continuous process.* Data quality may vary over time due to the temporal and dynamic nature of the data and the evolving real world, but also as a consequence of various data cleaning and repairing actions. This bears the need

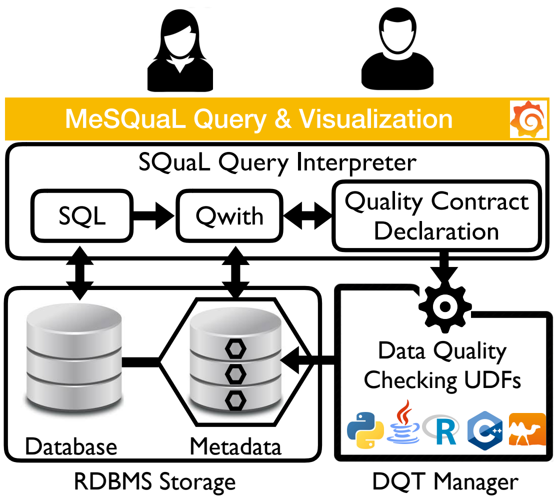

MeSQuaL Query & Visualization
SQuaL Query Interpreter
SQL
Qwith
Quality Contract Declaration
Data Quality Checking UDFs
Database
Metadata
RDBMS Storage
DQT Manager

Figure 1: MeSQuaL Architecture.

for monitoring the quality of different versions of the database and the quality of query results.

Our Approach. To deal with these challenges, we believe that it is essential to build tools that enable the data scientists to specify and verify data quality requirements in a declarative way. In particular, tools for analyzing data quality, testing statistical assumptions, and monitoring data quality continuously to provide insights about the data glitches and help in selecting appropriate data preparation and cleaning strategies. In this demo, we present MeSQuaL that we built for this purpose.

2 MESQUAL OVERVIEW

Architecture. As in Fig. 1, MeSQuaL consists of two main components: (1) the SQuaL query interpreter enabling the declaration of contracts for data quality checking and SQL queries extended with QWITH statement; (2) the Data Quality and Tests (DQT) `Manager` that operates over three types of RDBMS (Oracle, MySQL, and PostgreSQL) storing the data and related metadata. Our framework provides: (1) several built-in functions for data quality checking and statistical tests; (2) the possibility to call functions in `Python`, `Java`, `C++`, `R`, and `OCaml`; or (3) the possibility to define custom command-line calls to a UDFs (User-Defined Functions). Once a quality contract is declared in SQuaL with a list of dimensions, associated UDFs, and constraints, the `DQT Manager` computes the corresponding data quality indicators and stores them as metadata. SQuaL query result and visualization are displayed via a Grafana graphical interface[1].

[1] https://grafana.com/

```
<squal-script> ::= (<squal-query> | <contract-type> | <contract>)*
<squal-query> ::= '{' <sql-query> '}'  QWITH <qwith-formula> ';'
<sql-query> ::= <select-clause><from-clause>[<where-clause>] [<group-by-clause>] [<having-clause>] [<order-by>] [<sql-sets-operator><sql-query>]* ';'
<select-clause> ::= <attribute-name> [ = <sql-subquery>]* [, <attribute-name> [ = <sql-subquery>]* ]*
<from-clause> ::= <from-element> [, <from-element> | <join-operator> <from-element>  ON <join-equality>]*
<from-element> ::= (<sql-subquery> AS <name>|<relation-name> [AS <name>])
<where-clause> ::= WHERE [NOT] EXISTS <sql-subquery> | WHERE <expression> [NOT] IN <sql-subquery>
                   | WHERE <expression> <comparison-operator> [ANY|ALL] <sql-subquery>
<having-clause> ::= <having-element> [AND <having-element>]*
<having-element> ::=  <having-expression> <comparison-operator> (<compared-value>|<sql-subquery>)
<sql-subquery> ::= (<sql-query>|<squal-query>)
<qwith-formula> ::= <qwith-element> [ AND <qwith-element>]*
<qwith-element> ::= (<contract-name> | <constraint>)
<contracttype> ::= (CREATE | REPLACE) CONTRACTTYPE <contracttype-name> <dimension> [',' <dimension>]* ';'
                    | DELETE CONTRACTTYPE <contracttype-name>  ';'
<dimension> ::= <dimension-name> BY FUNCTION <binary-path> LANGUAGE <language>
<contract> ::= (CREATE | REPLACE) CONTRACT <contract-name> '('<constraint> ('AND' <constraint>)* ');'
                | DELETE CONTRACT <contract-name> ';'
<constraint> ::=  <contract-name> | [<contract-name> '.'] <dimension-name> <comparison-operator> <reference-value>
<comparison-operator> ::=  '<>' | '=' | '!=' | '>' | '<' | '<=' | '>='
```

Figure 2: SQuaL Syntax in EBNF

```
CREATE CONTRACTTYPE StatTests (
   autocorrelation BY FUNCTION 'durbinWatsonTest.py' LANGUAGE PYTHON,
   multicollinearity BY FUNCTION 'varInflationFactor.py' LANGUAGE PYTHON,
   heteroscedasticity BY FUNCTION 'BreuschPaganTest.py' LANGUAGE PYTHON,
   KMerrorNormality BY FUNCTION 'KolmogorovSmirnov.py' LANGUAGE PYTHON,
   SWerrorNormality BY FUNCTION 'ShapiroWilkTest.py' LANGUAGE PYTHON);
```

```
CREATE CONTRACTTYPE CheckQDB (
   completeness BY FUNCTION 'completeness.py' LANGUAGE PYTHON,
   uniqueness BY FUNCTION 'uniqueness.py' LANGUAGE PYTHON,
   consistency BY FUNCTION 'consistency.py' LANGUAGE PYTHON,
   outlyingness BY FUNCTION 'outlyingness.py' LANGUAGE PYTHON);
```

Figure 3: CONTRACTTYPE Examples

```
CREATE CONTRACT RegressionAssumptions (
   StatTests.autocorrelation > 0
   AND StatTests.autocorrelation < 4
   AND StatTests.multicollinearity <= 4
   AND StatTests.heteroscedasticity < 0.05
   AND StatTests.SWerrorNormality < 0.05);
```

```
CREATE CONTRACT CheckBeforeAnalysis (
   RegressionAssumptions
   AND CheckQDB.consistency > 0.9
   AND CheckQDB.outlyingness < 0.2);
```

Figure 4: CONTRACT Examples

```
{  SELECT timestamp, node_id,value_raw,valuehrf
   FROM ChicagoDataset
   WHERE ChicagoDataset.sensor = 'o3'
}
QWITH CheckBeforeAnalysis
   AND CheckQDB.completeness> 0.95;
```

Figure 5: SQuaL query example

SQuaL Syntax. Each step of a data quality checking scenario can be expressed in SQuaL, the SQL-extended language we implemented on top of each RDBMS. The grammar of SQuaL is provided in Fig. 2. A user can easily express data quality concerns and requirements either re-using the library of UDFs we provide with MeSQuaL or, depending on his/her programming skills, adding new functions and codes in `Python`, `Java`, `C++`, `R`, or `OCaml` to check the quality of data and test other hypotheses.

• **Contract type.** Data quality measures and indicators can be expressed via the declaration of *quality contract types*. In the ⟨`contracttype`⟩ statement of the grammar in Fig. 2, a contract type statement creates (or replaces) a contract type as a list of quality dimensions of interest. Each dimension indicator is computed by a UDF. Once the *contract type* is created and UDFs are loaded, it can be instantiated as a contract with constraints on the pre-declared dimensions and later on, invoked in SQuaL queries. A dimension is defined by a ⟨*dimension-name*⟩, the path of its UDF, and the language in which the UDF is implemented. Fig. 3 presents several examples of *contract type* declarations.

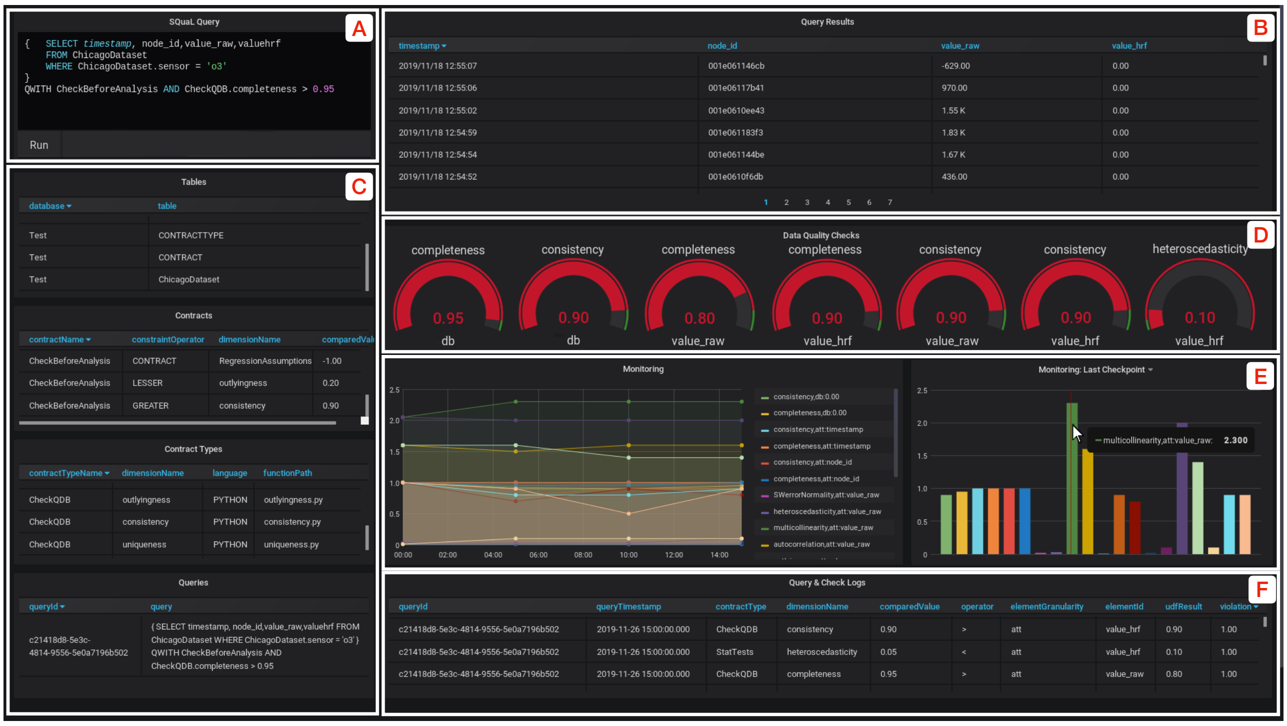
• **Contract.** A quality contract, described by the ⟨*contract*⟩ statement in Fig. 2, derives from one (or more) pre-existing contract type(s) and is a set of one-sided range constraints on the dimensions declared in the contract type(s). Constraints are simple comparison expressions involving the pre-declared dimension, and a reference value. Fig. 4 presents several examples of contract declaration based on the contract types declared in the examples of Fig. 3. The `DQT Manager` executes the contracts on-demand when they are invoked in the QWITH statement of a SQuaL query.

• **Qwith Query.** In a SQuaL query, described in the grammar in Fig. 2 by the ⟨*squal-query*⟩ statement, the QWITH operator can be used to extend and constraint a regular SQL query result as illustrated in Fig. 5. It can be applied to the whole database or to a query as it adds constraints to the semantics of the SQL query and returns the query result that satisfies the quality requirements defined in the contract types and contract instances with the UDFs executed by the `DQT Manager`. Additionally, QWITH can be used inside nested SQL queries.

Note that before declaring and creating data quality contract types and contract instances using SQuaL, an important and challenging task in dealing with real-world (possibly dirty) data is data exploration to better understand the reasons for poor data quality and how it can affect the processes that consume the data. Since data exploration is out of the scope of this demo and not directly enabled by our system, we assume that data exploration should be achieved before and externally for the adequate specifications of the contract types, instances, and constraints.

Goals. In this demo, we showcase the principal features of MeSQuaL:

• *Seamless integration of user-defined functions and constraints in the query language* to compute and check various dimensions of data quality. A common way to use our system is to declare the data quality dimensions of interest, bind and invoke the functions that compute relevant quality indicators (as shown in Fig. 3 with `CheckQDB` contract type for example), and query the data with constraints on these indicators using QWITH statement of our query language (as illustrated in Fig. 5);


A
B
C
D
E
F
Run
Data Quality Checks
completeness 0.95 db
consistency 0.90 db
completeness 0.80 value_raw
completeness 0.90 value_hrf
consistency 0.90 value_raw
consistency 0.90 value_hrf
heteroscedasticity 0.10 value_hrf
Monitoring
consistency,db:0.00
completeness,db:0.00
consistency,att:timestamp
completeness,att:timestamp
consistency,att:node_id
completeness,att:node_id
SWerrorNormality,att:value_raw
heteroscedasticity,att:value_raw
multicollinearity,att:value_raw
autocorrelation,att:value_raw
Monitoring: Last Checkpoint
multicollinearity,att:value_raw: 2.300

SQuaL Query

```
{   SELECT timestamp, node_id,value_raw,valuehrf
    FROM ChicagoDataset
    WHERE ChicagoDataset.sensor = 'o3'
}
QWITH CheckBeforeAnalysis AND CheckQDB.completeness > 0.95
```

Query Results

timestamp	node_id	value_raw	value_hrf
2019/11/18 12:55:07	001e061146cb	-629.00	0.00
2019/11/18 12:55:06	001e06117b41	970.00	0.00
2019/11/18 12:55:02	001e0610ee43	1.55 K	0.00
2019/11/18 12:54:59	001e061183f3	1.83 K	0.00
2019/11/18 12:54:54	001e061144be	1.67 K	0.00
2019/11/18 12:54:52	001e0610f6db	436.00	0.00

1 2 3 4 5 6 7

Tables

database	table
Test	CONTRACTTYPE
Test	CONTRACT
Test	ChicagoDataset

Contracts

contractName	constraintOperator	dimensionName	comparedVal
CheckBeforeAnalysis	CONTRACT	RegressionAssumptions	-1.00
CheckBeforeAnalysis	LESSER	outlyingness	0.20
CheckBeforeAnalysis	GREATER	consistency	0.90

Contract Types

contractTypeName	dimensionName	language	functionPath
CheckQDB	outlyingness	PYTHON	outlyingness.py
CheckQDB	consistency	PYTHON	consistency.py
CheckQDB	uniqueness	PYTHON	uniqueness.py

Queries

queryId	query
c21418d8-5e3c-4814-9556-5e0a7196b502	{ SELECT timestamp, node_id,value_raw,valuehrf FROM ChicagoDataset WHERE ChicagoDataset.sensor = 'o3' } QWITH CheckBeforeAnalysis AND CheckQDB.completeness > 0.95

Query & Check Logs

queryId	queryTimestamp	contractType	dimensionName	comparedValue	operator	elementGranularity	elementId	udfResult	violation
c21418d8-5e3c-4814-9556-5e0a7196b502	2019-11-26 15:00:00.000	CheckQDB	consistency	0.90	>	att	value_hrf	0.90	1.00
c21418d8-5e3c-4814-9556-5e0a7196b502	2019-11-26 15:00:00.000	StatTests	heteroscedasticity	0.05	<	att	value_hrf	0.10	1.00
c21418d8-5e3c-4814-9556-5e0a7196b502	2019-11-26 15:00:00.000	CheckQDB	completeness	0.95	>	att	value_raw	0.80	1.00

Figure 6: Screenshot of MeSQuaL's User Interface with Dynamic Dashboards

• *Continuous data-quality checking and monitoring.* MeSQuaL results can help the user in defining or comparing various methods for checking and profiling the quality of data over time, selecting the most appropriate ones, or refining the data quality checking process, and select the most appropriate contracts to monitor continuously;

• *Statistical testing.* Interactive hypothesis testing allows the user to check various statistical assumptions on the data distributions and make sure that the data conform to the requirements of a given ML model (as in Fig. 3 with StatTests contract type);

• *Efficient profiling of data quality indicators and visualization of static and dynamic profiles* showing the evolution of data quality over time with the Monitoring Panel $\boxed{\text{E}}$ in Fig. 6.

To increase the automation of data quality checking, we claim that there is a need for augmenting database management systems with a flexible and declarative way to declare, check, and monitor the quality of data, independently from the data model, format, or application, and this actually motivated the design of MeSQuaL.

3 DEMONSTRATION SCENARIOS

We will demonstrate MeSQuaL using two domain-specific data sets: the clinical database MIMIC-III[2] [7] and the ChicagoDataset from the Array of Things[3] real-time urban data (AoT) [5]. The users can examine and query the data sets and explore the data quality checking functions available in MeSQuaL to gain a sense of its usability. We will guide the users through the following scenarios.

[2]https://physionet.org/content/mimiciii/1.4/
[3]https://aot-file-browser.plenar.io/data-sets

1) **Declaring data quality indicators and constraints.** This scenario is dedicated to showcase the use of the SQuaL language for creating and using relevant contract types and contract instances to specify data quality checks and submit SQL queries extended with QWITH statement. In Panel $\boxed{\text{C}}$ of Fig. 6, the users can explore available contract types and contract instances predefined for the data set, like CheckQDB contract type and CheckBeforeAnalysis contract proposed in Fig. 3 and Fig. 4 that will check the completeness, uniqueness, consistency, or outlyingness of the data. The users will be able to declare new data quality checks and rules, and query the data sets with SQuaL in Panel $\boxed{\text{A}}$. For the query of Fig. 5, Panel $\boxed{\text{B}}$ presents the results on ChicagoDataset and red gauges of Panel $\boxed{\text{D}}$ show that neither the constraints on data consistency and completeness are satisfied by the queried data of ozone sensors ('o3'), nor the constraints defined in RegressionAssumptions contract. As presented in Table 1, other SQuaL queries including nested SQuaL queries (e.g., Q_8) will be tested to show the usability of our system.

2) **Evaluating the applicability of learning models with declarative statistical hypothesis testing.** In this scenario, the user will see in more detail how MeSQuaL can be used to declare various statistical tests, and visualize which data lead to violations of some statistical assumptions or other requirements of various ML models. For example, before the application of a linear regression over a data set, at least four critical assumptions need to be verified: normality, linearity, homoscedasticity, and absence of multicollinearity. Using StatTests contract type instantiated by RegressionAssumptions contract (defined in Fig. 3 and Fig. 4) available in the contract explorer (Panel $\boxed{\text{C}}$), the user will easily check if the statistical properties are met by the query results. The attendees will notice that new contracts can be added in a flexible and modular way, and they will explore our library of tests, inspect the logs of previous queries and

Table 1: SQuaL Query examples of the demo

	Query#	SQuaL Query	SQuaL Query time (ms)	SQL query time (ms)	UDF time (ms)	Total time (ms)
	Q_1	CREATE CONTRACTTYPE CheckQDB2 (completeness2 FLOAT ON DATABASE BY FUNCTION ’completeness.py’ LANGUAGE PYTHON, uniqueness2 FLOAT ON DATABASE BY FUNCTION ’uniqueness.py’ LANGUAGE PYTHON, consistency2 FLOAT ON DATABASE BY FUNCTION ’consistency.py’ LANGUAGE PYTHON, outlyingness2 FLOAT ON DATABASE BY FUNCTION ’outlyingness.py’ LANGUAGE PYTHON);	24.6	-	2482.4	2507
	Q_2	CREATE CONTRACT RegressionAssumptions (StatTests.autocorrelation > 0 AND StatTests.autocorrelation < 4 AND StatTests.multicollinearity <= 4 AND StatTests.heteroscedasticity < 0.05 AND StatTests.SWerrorNormality < 0.05);	28.3	-	2550.8	2579.1
AoT	Q_3	{ SELECT * FROM ChicagoDataset } QWITH CheckQDB.completeness> 0.95;	32.9	587.2	566.2	1186.3
	Q_4	{ SELECT * FROM ChicagoDataset } QWITH CheckBeforeAnalysis AND RegressionAssumptions;	274.6	499.6	500.4	1274.6
	Q_5	{ SELECT timestamp, node_id,value_raw,valuehrf FROM ChicagoDataset WHERE ChicagoDataset.sensor = ’o3’ } QWITH CheckBeforeAnalysis AND CheckQDB.completeness> 0.95;	214.2	21.5	784.7	1020.4
MIMIC-III	Q_6	{ SELECT * FROM Admissions } QWITH CheckQDB.completeness> 0.95;	42.1	346.8	472.3	861.2
	Q_7	{ SELECT * FROM Admissions WHERE Admissions.insurance = ’Private’ } QWITH CheckBeforeAnalysis AND CheckQDB.completeness> 0.95;	237.0	154.1	493.8	884.9
	Q_8	{ SELECT gender, dob, admittime FROM Admissions INNER JOIN (SELECT * FROM Patients WHERE dob < ’2090-12-12 00:00:00’ QWITH CheckQDB.completeness> 0.95) as Pat ON Admissions.subject_id=Pat.subject_id; } QWITH CheckQDB.completeness> 0.95;	318.9	16.6	1610.1	1945.6

checks (Panel $\boxed{\text{F}}$), and reload the visualization of some previous SQuaL queries (from Panel $\boxed{\text{C}}$). MeSQuaL combines the declarative and scripting approaches for querying the data and checking various assumptions simultaneously. It facilitates notably the data preparation choices to meet the requirements of some ML algorithms. Since the declared contracts are independent of the data sets, they can be reused whenever the assumptions and data quality checks need to be checked.

3) **Monitoring the evolution of data quality indicators.** In this last scenario, the attendees will see the possibilities offered by MeSQuaL for monitoring the evolution of data quality indicators in Panel $\boxed{\text{E}}$. Once declared, contract types and contract instances are stored as metadata and executed regularly by MeSQuaL’s `DQT Manager` via its configuration to schedule recurring SQuaL queries. The user can act as a DBA and define various thresholds for the declared data quality indicators to alert when some results are suspicious. Using the `AoT ChicagoDataset`, the attendees will see how MeSQuaL facilitates the continuous monitoring of data quality indicators to detect, for instance, inconsistent data from neighboring air pollution sensors or intermittent sensor failures with the use of the library of UDFs provided by MeSQuaL. Using MeSQuaL’s user interface shown in Fig. 6, the user can visualize the declared data quality indicators and spot the periodic or punctual errors in the data over time in Panel $\boxed{\text{E}}$.

4 RELATED WORK

Data quality has been extensively studied by the database community in the last decades [3, 10] with a line of data cleansing commodity systems and tools that can detect anomalies and reduce the burden on data scientists for data repairing and data preparation in the context of ML pipelines [1, 2, 8, 9, 13]. MeSQuaL is similar to some extent to two main operational data validation systems: (1) Google TensorFlow Data Validation (TFDV) system [4], used in production, is a library for exploring and validating ML data, including schema inspection and anomaly detection, such as missing features, out-of-range values, or wrong feature types, and (2) the Amazon system implementing unit-tests for data verification has been proposed in [11, 12]; it offers a declarative API that allows users to define checks on their data by composing a variety of available constraints. However, the main drawback of these systems is that they do not provide the user with (1) the possibility to interact with the data quality checking process, (2) the flexibility to declare new data quality metrics with user-defined functions, or constraints for data quality checks, and (3) the extension of the query language to check the results with respect to data quality requirements and constraints.

The key novelty of MeSQuaL is to provide the user with a framework for checking the quality of their relational data by declaring UDFs and constraints using SQuaL, an SQL-like query language extension for querying data and checking on-demand the quality of the results.

5 ACKNOWLEDGEMENTS

This work is funded by the French National Agency ANR Quali-Health 18-CE23-0002.

REFERENCES

[1] L. Berti-Équille. Learn2Clean: Optimizing the Sequence of Tasks for Web Data Preparation. In *Proc. of the The Web Conf 2019*, 2019.

[2] L. Berti-Équille. Reinforcement learning for data preparation with active reward learning. In *Internet Science - 6th International Conference, INSCI 2019, Perpignan, France, December 2-5, 2019, Proceedings*, pages 121–132, 2019.

[3] L. Berti-Équille. Quality Awareness for Data Management and Mining. Habilitation à Diriger des Recherches, Univ. Rennes 1, France, June 2007, http://pageperso.lis-lab.fr/~laure.berti/pub/Habilitation-Laure-Berti-Equille.pdf.

[4] E. Breck, M. Zinkevich, N. Polyzotis, S. Whang, and S. Roy. Data validation for machine learning. In *Proc. of SysML*, 2019.

[5] C. E. Catlett, P. H. Beckman, R. Sankaran, and K. K. Galvin. Array of things: A scientific research instrument in the public way: Platform design and early lessons learned. In *Proc. of the 2nd International Workshop on Science of Smart City Operations and Platforms Engineering*, SCOPE ’17, pages 26–33, New York, NY, USA, 2017. ACM.

[6] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, 2015.

[7] A. E. Johnson, T. J. Pollard, L. Shen, H. L. Li-wei, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L. A. Celi, and R. G. Mark. MIMIC-III, a freely accessible critical care database. *Scientific data*, 3:160035, 2016.

[8] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg. Activeclean: Interactive data cleaning for statistical modeling. *Proc. VLDB Endow.*, 9(12):948–959, Aug. 2016.

[9] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

[10] S. W. Sadiq, T. Dasu, X. L. Dong, J. Freire, I. F. Ilyas, S. Link, M. J. Miller, F. Naumann, X. Zhou, and D. Srivastava. Data quality: The role of empiricism. *SIGMOD Record*, 46(4):35–43, 2017.

[11] S. Schelter, F. Biessmann, D. Lange, T. Rukat, P. Schmidt, S. Seufert, P. Brunelle, and A. Taptunov. Unit testing data with deequ. In *Proc. of the 2019 International Conference on Management of Data*, SIGMOD ’19, pages 1993–1996, New York, NY, USA, 2019. ACM.

[12] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Bießmann, and A. Grafberger. Automating large-scale data quality verification. *PVLDB*, 11(12):1781–1794, 2018.

[13] M. Yakout, L. Berti-Équille, and A. K. Elmagarmid. Don’t be scared: use scalable automatic repairing with maximal likelihood and bounded changes. In *Proc. of the ACM SIGMOD*, pages 553–564, 2013.

MALOS: A Movement-Aware Location Selection System

Di Yang, Hui Li
yanw.d@foxmail.com,hli@xidian.edu.cn
School of Cyber Engineering, State Key Laboratory of Integrated Services Networks
Xidian University
Xi'an, China

Meng Wang
School of Computer Science
Xi'an Polytechnic University
Xi'an, China
wamengit@sina.com

Dan Li, Jiangtao Cui
i.danli@outlook.com,cuijt@xidian.edu.cn
School of Computer Science and Technology
Xidian University
Xi'an, China

ABSTRACT

Facility placement has been receiving considerable research attention due to the proliferation of GPS equipped mobile devices. Establishing new facilities has a wide spectrum of applications including billboard placement, wildlife monitoring, supermarket/restaurant placement, etc. In all these applications, finding the best position to place a facility can lead to the optimal benefit given plenty of target users, which are associated with many historical position logs, e.g., user check-in data, wildlife monitor logs. In this work, we demonstrate a general system, namely MALOS, for answering the majority of facility placement problems given movement logs of a series of target users. Specifically, MALOS leverages on three academic works in the following scenarios. Find the best one/k position/s to place one/k new facilities in order that the overall benefit can be maximized; find one position to place an extra facility given that there are k existing facilities in different places, such that the overall benefit can be maximized. Notably, in MALOS we consider only the geographical issues that are common in all facility placement applications and choose not to take into account the specialized factors that vary across applications, such as billboard size, restaurant type, etc. We shall provide an opportunity for demonstration users to experience all kinds of these facility placement queries with a real-world historical movement logs and corresponding maps interface.

1 INTRODUCTION

Facility placement has been receiving considerable research attention due to the proliferation of GPS equipped mobile devices. Establishing new facilities has a wide spectrum of applications, for example, setting up billboards, monitoring wildlife, running restaurants, building charge stations and urban planning. Majorities of existing research work focus on solving a specific problem, such as setting up billboards or running restaurants, etc. If we change the application, these researches cannot be applied directly to the new one. Therefore, we are motivated to design a general system to address facility placement problem, namely Movement-Aware LOcation Selection system (MALOS).

Compared with the maximum coverage problem of existing work [7], the advantages of MALOS are as follows. Firstly, as a ready-to-use integrated system, instead of loading offline dataset as input, MALOS incorporates a data acquisition module that collects historical check-in data from the Internet and is updated in real-time; secondly, [7] can only be applied in outdoor advertising applications but fails to fit in other applications. In comparison, we take into account the common factor of different facility placement applications, i.e., distance, and allows user-configuration for other factors specialized for particular scenarios. Thirdly, MALOS is a general system that supports up to three different use cases, while [7] considers only one of them.

MALOS is a *general* movement-aware facility placement system, where *general* can be interpreted as follows: 1) We generalize the common spatial-correlated issues within facility placement applications, and avoid falling into application-dependent factors, *e.g.,* size and price of billboard, categories of restaurant and etc., which can be extremely different across applications. 2) We consider the following representative scenarios in facility placement, given the historical movement logs for a large scale of users.

- *Place-One*: query the optimal location which can influence the maximum moving objects [6].
- *Place-k*: return k locations which can influence the largest number of moving objects [4].
- *Incremental-One*: add a location into the existing locations set which could provide the best marginal [1].

Our key contributions in this work are summarized as follows.

- We develop a general system based on user movement data, namely MALOS, answering general facility placement queries with user-friendly browser-based interface.
- The system provides three different queries to meet the major requirements for facility placement applications.
- We provide a module to further incorporate check-ins data for each moving object in real-time.

The rest of the demonstration proposal is organized as follows. Section 2 formally defines the specific queries supported by MALOS. Then, we introduce MALOS system in Section 3. Section 4 offers the demonstration details. Finally, Section 5 concludes the demonstration.

2 DEFINITION OF QUERIES

In this section, we will introduce the definitions of three queries mentioned before.

Definition 2.1 (Place-One). Given a set of candidate locations C, a set of moving objects Ω, a certain distance-based probability function PF and a user-specified influence threshold τ, the *Place-One* query aims to find the optimal candidate $c \in C$ such that $\forall c' \in C - \{c\}, inf(c) \geq inf(c')$, where $inf(c)$ is the number of moving objects in Ω that are influenced by c [6].

Definition 2.2 (Place-k). Given a set of candidate locations $C = \{c_1, c_2, ..., c_n\}$, a set of moving objects Ω and the budget number of new facilities $k(k \leq n)$. *Place-k* aims to find $\exists S \subseteq C$ to maximize $\sigma(S)$. $\sigma(S)$ denotes the total number of moving objects that are influenced by candidate set S [4].

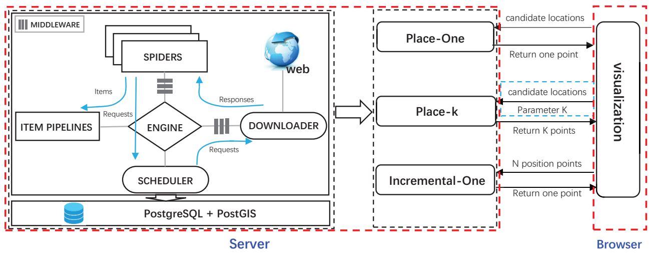


Figure 1: Architecture of MALOS.

Definition 2.3 (Incremental-One). Given a set of candidate locations C, a spatial network graph $G(V, E)$, a set of existing facilities F and a set of moving objects Ω, each of whose movement can be modeled as a set of reference locations, *Incremental-One* aims to find the optimal location c among a set C of query candidates such that $\forall c' \in C, \Delta(c) \geq \Delta(c')$, where $\Delta(c)$ is expected reduction of total distance for c [1].

In particular, in our default settings, we follow the same suggestions from original corresponding academic works for different queries. For instance, in *Place-One*, the probability of a user checking-in at a point-of-interest decays as the power-law of the distance between them. We set the distance-based probability function PF as $\rho(d_0 + dist(c, p))^{-\lambda}$ by default as suggested in [6], where ρ is a factor to describe behavior pattern, d_0 is a distance factor, $dist(c, p)$ is the distance between candidate c and position of moving object p, λ is a exponential factor to configure the mapping from distance to influence probability. In MALOS system, the default values of the number of candidates, probability threshold τ, behavior factor ρ, and exponential factor λ are 600, 0.7, 0.9, and 1.0, respectively. In fact, it can also be manually adapted to other functions by MALOS users.

3 MALOS PROTYPE

In this section, we cover the framework of MALOS, the acquisition of check-in data, modeling of check-in data in database and interaction with data.

3.1 The MALOS architecture

MALOS adopts the browser-server model and is implemented with JavaScript and Java, built on top of PostgreSQL. MALOS implements three algorithms: PINOCCHIO algorithm [6], GreedyPS algorithm [4] and Local Network Based (LNB) algorithm [1]. However, MALOS is not a simple aggregation of these three algorithms, which consider only how to address particular use cases given the data sources. Instead, as a real-time and ready-to-use system, MALOS has to additionally address two other grand challenges, 1) where and how to get the data these algorithms rely on; 2) how to store the data such that these algorithms can be efficiently carried out. Our solutions towards both challenges are discussed in detail within Section 3.2 and 3.3, respectively.

The architecture of MALOS is shown in Figure 1, which includes data acquisition, query processing and visualization modules. The data acquisition module obtains the historical position logs from third-party resources (e.g., Sina Weibo, Twitter). In order to efficiently crawl data from the web, we use the Scrapy architecture, which can efficiently obtain historical location check-in data to ensure real-time performance [5]. Users submit query requests through the browser, which executes a particular facility placement algorithm (*Place-One*, *Place-k* or *Incremental-One*) based on the request. Finally, the results are returned to the users, and are displayed over a Map interface (*e.g.,* Google Maps, Baidu Map) in the browser.

3.2 Data acquisition

How to obtain moving users' historical position data is a first challenge for MALOS. At present, there are two direct ways to address it, one is to be authorized to obtain real-time data directly from giant LBS (location-based service) enterprises (e.g., Uber, Google), the other is to cooperate with government departments. However, neither of them is easy to be carried out for ordinary companies. In this end, MALOS system aims to provide a friendly usage for general public users, who are unable to get the data through the ways discussed above. Therefore, we developed a general web crawler to obtain the historical locations of people through third-party services.

In MALOS system, we use Scrapy framework [5] to collect data. Scrapy is an excellent open source framework for quick crawling of web sites and extracting structured data. There are mainly seven components in Scrapy. Scrapy *Engine* is responsible for controlling the data flow between all components of the system. *Scheduler* receives requests from the engine and enqueues them for further usage by the engine. *Downloader* is responsible for fetching web pages and feeding them to the engine which, in turn, feeds them to the spiders. *Spiders* are custom classes written by us to parse responses and extract items from them or additional URLs to follow. *Item Pipeline* is responsible for processing items once they have been extracted by the spiders. *Downloader Middlewares* are specific hooks that sit between the Engine and the Downloader; and process requests (resp., responses) when they pass from the Engine to the Downloader (resp., Downloader to the Engine). *Spider Middlewares* are specific hooks that sit between the Engine and the Spiders; they are able to process spider input (responses) and output (items and requests). The framework also provides a convenient mechanism for extending Scrapy functionality by plugging custom code.

We present a feasible crawling strategy for MALOS (By default, the crawling site of the MALOS system is Weibo). First, we select one (or more) seed users in a region as the initial target, which is determined by the geographic location label in the personal information list. For each crawled object, we grab personal information in turn, locations of historical check-ins in all Weibo pages, as well as fans list and follower list. Then, add the two lists in the upper level watchlist to the crawl object list. Following this iteration, the reptile radiation can be formed, which takes the seed user as the core and diffuses outward layer by layer. In fact, there are a lot of records with noise information, such as some vulgar marketing numbers. Therefore, we need to set a threshold value for the captured object to determine whether the user is a target user. In MALOS, we parameterize a target moving user as follows: the number of message < 5000, fans < 5000, follower < 5000.

Given the above strategy, Scrapy in MALOS works as follows (illustrated in the left part of Figure 1). When the web *Spider* starts, it will extract each URL from the start URL of a seed user and encapsulate it into a request, which will be sent to the *Engine*. Then the requests are passed over to the *Scheduler* and form a queue. The first item in the queue will be sent to *Downloader* for retrieving the corresponding content. The response file returned from the *Downloader* will be handed over to a parse method. Finally, the parse method calls the a predefined XPath to extract the check-in data from the crawled page content.

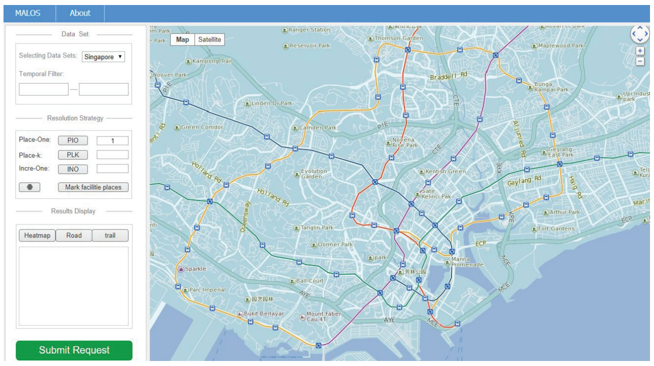


Figure 2: The browser interface of MALOS.

In order to meet our requirements, we customize these components by defining some parameters as followings.

ROBOTSTXT_OBEY = False
DOWNLOAD_DELAY = 0.1
CONCURRENT_REQUESTS = 16

3.3 Storage of check-in data

The check-in data reflects the spatio-temporal behavior of the moving object. It generally includes user ID, check-in location, timestamp, check-in area, etc. The check-in location sequence with time mark constitutes the user's historical positions logs. Based on the check-in records, we are able to construct historical positions logs for each individual Weibo user we have crawled. In particular, in PostgreSQL we present the following schema for user historical positions logs.

CI_PAIR (USER_ID varchar(40), LAT real, LNG real, CI_Time timestamp, CITY varchar(256));

Each CI_PAIR object stores the position of a moving user at a particular timestamp. Multiple records of a user are combined to form a spatial-temporal movement history.

We store the dataset in PostgreSQL (version 11.6) extended by PostGIS[1]. In addition to using the embedded distance calculation function ST_Distance ($geometry\ g1, geometry\ g2$), MALOS system extends the integration of PLACEONE($dataset$), PLACEK($dataset, parameter\_k$) and INCREONE ($dataset$, $parameter\_k$) functions into the PostgreSQL database. The extensions are basically customized functions that define specific query processing algorithms. As a mild coupling external module, the extensions can be easily distributed and maintained.

3.4 Usage of MALOS in facility placement

In MALOS system, data operations are initiated by users from browser side. The browser side offers two panels, the function panel and the map panel [2]. The function panel provides the interfaces to users for obtaining check-in data and generating queries. The map panel shows a map that illustrates the returned objects given the required query input of system users.

Users interact with the system through the function panel. The function panel provides data acquisition interaction function and location query interaction function. Note that in case that API authentication settings for real-time crawling mode may require troublesome modifications, in the demonstration we also provide an alternative mode to allow the system to load offline track data.

The MALOS system mainly provides users with three functions: *Place-One*, *Place-k* and *Incremental-One*. *Place-One* and *Place-k* functions are mainly for users who have not yet deployed any facility and want to select locations to deploy new ones in some candidate areas, which can also be manually set by the system users. *Place-One*, which aims to find the optimal location from a set of candidates to place a new facility such that a score (*i.e.,* benefit or influence on some given objects) can be maximized. Further, a user can specify a constraint k to perform *Place-k*, which indicates the number of candidate locations the user plan to choose. Considering that there are existing facilities, and users may intend to expand the scale of the facility network by opening a new one, the system provides *Incremental-One* function, which allows users to enter a series of existing facilities locations as constraints for location selection. Different algorithms with respect to each of these use cases have been implemented in MALOS system.

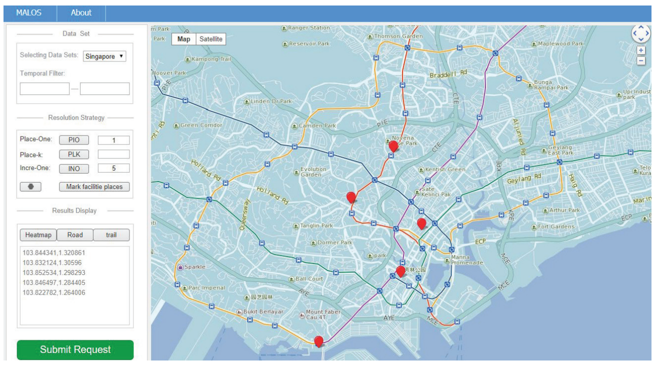
The *Place-One* function is addressed using PINOCCHIO algorithm. PINOCCHIO algorithm employs the R-tree structure to build geographic data index, then leverages two pruning rules based on a novel distance measure. With the help of influence arcs (IA) rule, it identify the candidates that influence the object. For the remnant candidates, non-influence boundary (NIB) rule is used to exclude those cannot influence the object. Lastly, the remnant candidates are verified.

The *Place-k* function is solved by GreedyPS algorithm. GreedyPS algorithm first calculates the set of objects affected by all candidates. Then it maps these sets to w bitmaps. Finally, it selects the candidates with the largest number of '1' in bitmaps until K is in each iteration process.

The *Incremental-One* function is addressed via LNB algorithm. It first constructs an index structure, Local Network Table (LNT), based on network locality. With the help of LNT, we initialize a Max Heap LNH ordered by $\Delta^+(v_i v_j)$, which is the upper bound of expected reduction of total distance (ERD) if a facility is built on edge $v_i v_j$. Then it iteratively checks the top candidate c in LNH. If $\Delta(ol)$ of current optimal candidate ol is greater than ERD upper bound for the top edge in LNH, the validation is finished. Otherwise, for every reference location whose local network covers $v_i v_j$, it needs to calculate $\Delta(c)$ through relevant rules. If a better candidate is found, it updates ol by c and eventually the optimal answer can be obtained, where $\Delta(c)$ is ERD for c.

After the request is processed by the server, the objects retrieved are returned. In order to enable users to observe the data intuitively, we need to visually present the spatial and temporal distribution of trajectory data. Visual comparison is one of the most fundamental and common visualization tasks [3]. Thus, we add a thermal layer, where we provide users with two types of density maps, namely, heatmap and roadmap. The thermal layer demonstrates the distribution density of the target trajectory in the geographic location by overlaying different color blocks on the map. In case that the volume of the check-in location data heat layer does not clearly reflect the trajectories of the crowd, we extensively add a trajectory layer to the system. The trajectory layer distinguishes the trajectories of different people by different colors, and reflects the overall trend of the trajectories by sampling method, which enhances the interaction experience for users. The marker layer provides a solution for visually presenting the results returned by the server and marking the input location points by users. For the *Incremental-One* function, a user does not need to enter the specific latitude and longitude coordinates of the existing facility in the system function panel. It is only necessary to find the location of the facility in the map panel and perform correlated marking on the marker layer.

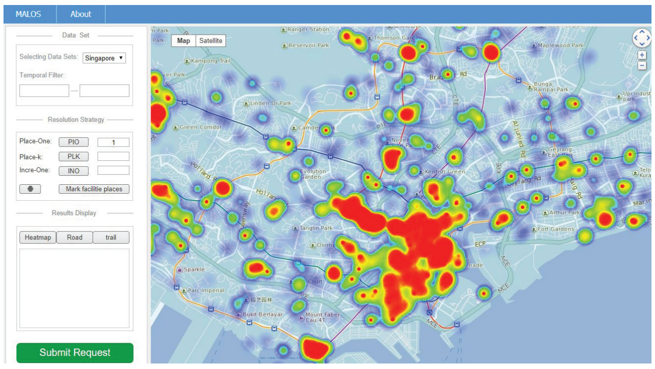
[1]www.postgis.net/



Selecting Data Sets: Singapore
Temporal Filter:
Place-One: PIO 1
Place-k: PLK
Incre-One: INO 5

103.844341,1.320861
103.832124,1.30596
103.852534,1.298293
103.846497,1.284405
103.822782,1.264006

Figure 3: Illustration of *Incremental-One* function.



Selecting Data Sets: Singapore
Temporal Filter:
Place-One: PIO 1
Place-k: PLK
Incre-One: INO

Figure 4: A visualization method of MALOS system(heatmap).

4 DEMONSTRATION

In our demonstration, as discussed before, we additionally implement an offline mode within MALOS system, in case there is troublesome modification for crawler API settings that is brought by the network connection issues. In offline mode, we adopt real-world check-in data in Singapore[2] to give users the opportunity to interact with MALOS and experience how the system can be used to select the location of facilities. The browser interfaces of MALOS has been shown in Figure 2. Users can use the system to find the best location of facilities in various scenarios, including *Place-One*, *Place-k* and *Incremental-One*.

4.1 Data loading and request submission

First, the user needs to select a target city to place the facility, and then select a function in the function panel to initiate the request. In MALOS system, we provide a list of cities for users to choose. For offline mode, the city can be only configured as Singapore. When the user selects a region where he wants to deploy the facilities, the map panel will automatically switches to focus on the corresponding area, and the server loads the historical check-in locations within the region. As shown in Figure 2, when the system loads the offline Singapore dataset, and the map panel also switches to Singapore. To specify the interested location, an audience can select a fine-grained location by drawing a rectangle on map panel (the latitude and longitude of the location are obtained using the Map API). For the *Incremental-One* function, the MALOS system allows users to enter some request constraints before running it. A user can enter a constant k to indicate the number of locations need to be picked. As illustrated in Figure 3, user enters a constant of 5 and marks five locations on the map to indicate the currently deployed facilities. The results display list also shows the longitude and latitude of these five facilities for user input verification.

4.2 Visualization and results display

Users can also explore the distribution of urban population information by visualization. There are three kinds of visualization methods, namely, heatmap, roadmap and trajectory-map for users to choose. As shown in Figure 4, heatmap is shown. In heatmap and roadmap, the darker the color, the greater the population density. The trajectory map shows the migration path of the crowd, and users can view the spatial and temporal distribution of urban population. As illustrated in Figure 3, when the query results are returned to the browser, the result objects are displayed in both the map panel and the result list. At the same time, users can further explore the surrounding environment of the recommended location by the system with the visualization.

[2]The default trajectory data source can be found in the github of PINOCCHIO listed in our MALOS project homepage: https://lihuixidian.github.io/malos/

5 CONCLUSION

In this demonstration, we present a general facility placement system, namely MALOS. MALOS solves a group of representative facility placement problems based on historical check-in location data that comes from the Internet and is updated in real time. MALOS adopts the browser-server model and provides an easy-to-use interface to answer *Place-One*, *Place-k* and *Incremental-One* queries. The queries cover the majority of facility placement problems given historical movement logs of massive users. Moreover, the system focuses on geographical issues that are general in all facility placement applications, and avoids taking into account application-dependent factors, such as the size of billboard, etc. In this way, the system can be used as a basic framework and easily adapted to different applications by taking into account extra specific application-aware factors.

ACKNOWLEDGMENTS

The work is supported by National Natural Science Foundation of China (No. 61672408, 61972309, 61976168), National Engineering Laboratory for Public Safety Risk Perception and Control by Big Data (PSRPC) and China 111 Project (No. B16037).

REFERENCES

[1] Jiangtao Cui, Meng Wang, Hui Li, and Yang Cai. 2018. Place Your Next Branch with MILE-RUN: Min-dist Location Selection over User Movement. *Inf. Sci.* 463-464 (2018), 1–20.
[2] Piotr Jankowski, Natalia V. Andrienko, and Gennady L. Andrienko. 2001. Map-centred exploratory approach to multiple criteria spatial decision making. *International Journal of Geographical Information Science* 15, 2 (2001), 101–127.
[3] Johannes Kehrer and Helwig Hauser. 2013. Visualization and Visual Analysis of Multifaceted Scientific Data: A Survey. *IEEE Trans. Vis. Comput. Graph.* 19, 3 (2013), 495–513.
[4] Dan Li, Hui Li, Meng Wang, and Jiangtao Cui. 2019. k-Collective Influential Facility Placement over Moving Object. *The 20th IEEE International Conference on Mobile Data Management* (2019).
[5] Jing Wang and Yuchun Guo. 2012. Scrapy-Based Crawling and User-Behavior Characteristics Analysis on Taobao. In *2012 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC 2012, Sanya, China, October 10-12, 2012.* 44–52.
[6] Meng Wang, Hui Li, Jiangtao Cui, Ke Deng, Sourav S. Bhowmick, and Zhenhua Dong. 2016. PINOCCHIO: Probabilistic Influence-Based Location Selection over Moving Objects. *IEEE Trans. Knowl. Data Eng.* 28, 11 (2016), 3068–3082.
[7] Yipeng Zhang, Zhifeng Bao, Songsong Mo, Yuchen Li, and Yanghao Zhou. 2019. ITAA: An Intelligent Trajectorydriven Outdoor Advertising Deployment Assistant. *Proc. VLDB Endow.* 12, 12 (2019), 1790–1793.

RRAMEN: An Interactive Tool for Evaluating Choices and Changes in Transportation Networks

Camila F. Costa
University of Alberta, Canada
camila.costa@ualberta.ca

Theodoros Chondrogiannis
University of Konstanz, Germany
theodoros.chondrogiannis@uni.kn

Mario A. Nascimento
University of Alberta, Canada
mario.nascimento@ualberta.ca

Panagiotis Bouros
Johannes Gutenberg University of Mainz, Germany
bouros@uni-mainz.de

ABSTRACT

This demonstration paper focuses on transportation-related queries within a city that go beyond simple routing and that are of interest to different types of users. For instance, individual users could be interested in which modes of transport are more effective to reach a set of alternative locations at a given time of the day, whereas urban planners could be interested in the effect that adding/removing a bus line would have in connecting regions of a city, e.g., a residential neighborhood and downtown. Given that context and using real data from the city of Berlin, we introduce RRAMEN, an interactive tool which is well equipped to support different city-scale mobility-related queries by different types of users.

1 MOTIVATION

Mobility within a city is both an important problem from an individual point of view as well as a higher level concern from a planning perspective. In sync with the current efforts towards mitigating climate change, we believe that there should be, whenever possible, a concerted effort to incentivize the use of public transportation systems. A few of the many worthy goals that can be accomplished with better public transportation systems are reducing traffic, therefore gas consumption, pollution and noise, reducing the need to dedicate large spaces for parking, thus creating more space for people, reducing costs associated with road maintenance, etc. Hence, there is a clear need for better (or complementary) tools that can support/promote a shift from using private vehicles on a regular basis towards public transit. This is the context that implicitly motivates our discussions and contributions in this paper. Also, in keeping with the above and for the purpose of this demonstration, but without loss of generality, we constrain ourselves to two modes of transport, public transit and private vehicles, and two types of users, individuals and urban planners.

Individual users are likely familiar with mobile routing solutions, e.g., Google Maps[1]. Those apps are typically designed for end-to-end routing (possibly setting some intermediate points, between origin and destination). Some of these apps also allow users to compare the efficiency of different modes of transport, e.g., public transit, private cars, bike and walking. However, as we will discuss shortly, comparing modes of transport is a use-case that is not easily contemplated by current apps but which is nonetheless of practical interest to individuals in the sense that it can, even if in a subtle manner, persuade them towards using public transportation. Additionally, we consider the role of urban planners. These individuals are interested in transportation from a collective rather than individual perspective and could make use of tools that help them in that respect.

Many existing systems, e.g., SANET [5], TransCad[2], and ISOGA [4], provide algorithmic solutions that enable the evaluation of public transportation systems through accessibility and/or reachability analysis [2, 8]. Another common approach employed by systems such as TRANSIMS [7], is the analysis of transportation systems through simulation. While the aforementioned systems can provide insight for the design and evaluation of transportation networks, they also come with the shortcoming of examining modes of transport in *isolation*. In many real-world scenarios though, users are more interested in the efficiency of a mode of transport in comparison to other available ones.

Towards the goal of investigating the compromises between different modes of transport from different perspectives, we recently proposed the notion of Relative Reachability [3]:

> Given a set of modes of transport and a source location, the *Relative Reachability* (RR) determines the mode of transport which minimizes arrival time at a given destination location. (As we shall discuss later, this concept can be extend to multiple destinations and/or regions.)

Given this context, our main goal in this demonstration paper is to present RRAMEN[3], a web-based tool which leverages on the RR concept in order to support different types of users with different needs. Towards this goal, in what follows, we discuss the data model underneath RRAMEN, along with its potential users and queries of interest. Next, we discuss in more detail a few queries reflecting the motivation above, while at the same time illustrating how RRAMEN can be used to answer these queries.

2 OVERVIEW OF RRAMEN

RRAMEN is a tool for Relative Reachability Analysis on Multimodal NEtworks that enables users to process a variety of RR queries and visualize the results. Our system is implemented in Python using the NetworkX[4] package for modelling the network, uses PostGIS[5] to store road and public transportation network data, and employs Mapbox[6] to display the map and visualize results. Figure 1 shows RRAMEN's interface. On the top left-hand side the user can define the source(s) and the destination(s), which

[1]https://www.google.com/maps

[2]https://www.caliper.com/tcovu.htm
[3]https://github.com/camilaferc/rramen
[4]https://networkx.github.io
[5]http://postgis.net
[6]https://www.mapbox.com

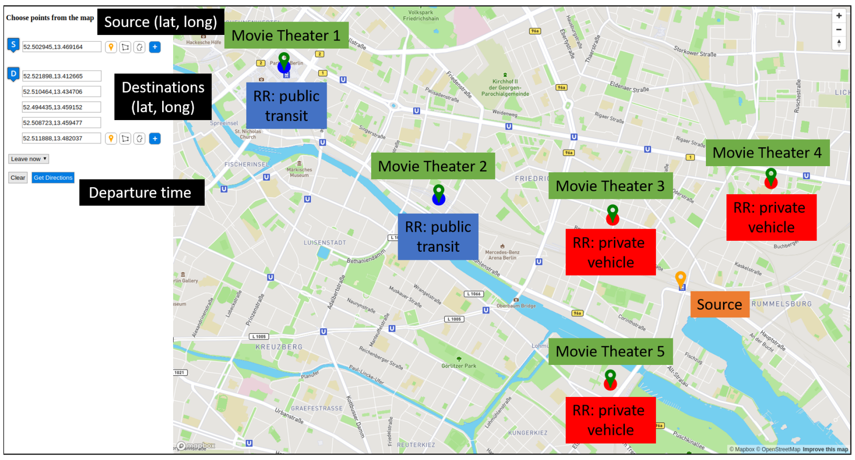


Figure 1: Illustration of RRAMEN's interface as well as the scenario discussed in Section 3.1.

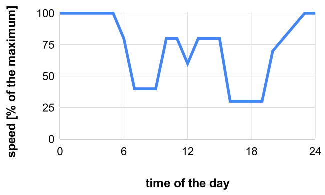


Figure 2: Speed over an edge during the day as %-age of maximum speed.

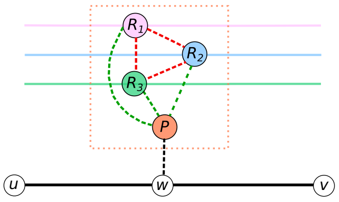


Figure 3: Stop with three routes, R_1, R_2 and R_3, and a parent node P connected to the road network.

can be either a single location, multiple locations or a region. The user can easily select these by clicking on the map. We note that while the processing of some queries is a computationally interesting topic, this paper focuses on discussing their applicability and on how RRAMEN can be used in practice.

In order to answer the queries supported by RRAMEN, we first obtain the road network (including footpaths) from Open Street Map (OSM)[7]. A PostGIS table is created to store the road networks' edges with their length, maximum speed, allowed transportation modes and geometry. To mimic how traffic fluctuates during a day, we assign to each edge of the network a speed distribution similar to the one illustrated in Figure 2. To compute such a distribution, we obtain the maximum allowed speed for each edge from OSM, split the day into a fixed number of time intervals (twenty four) and introduce a penalty that reduces the maximum speed during rush hour. Naturally, this assumption is orthogonal to RRAMEN's operation, i.e., should one have the actual speed distribution for each edge of the network, that could be easily integrated into RRAMEN.

Next, we obtain public transit data from GTFS[8] feeds, which include transit information such as stops, routes, trips and schedules. Such information is stored in a set of relational tables in PostGIS. For each route, we first extract the stop sequence covered by it. Then, for each existing route-stop pair, we create a node and add it to the public transit network. For instance, consider the stop shown in Figure 3 represented by a dotted box. There are three routes, R_1, R_2 and R_3, passing through the stop and, consequently, three nodes are created for that stop. Moreover, all nodes within a stop are directly connected to each other through an edge, the cost of which is given by the time to transfer from one route to another. Such a transfer time is extracted from the GTFS feed whenever available or set to 0 otherwise. Next, consecutive stops of a route are connected through an edge which is associated with a timetable containing the departure/arrival times for the corresponding route represented by that edge.

Finally, we build a multimodal network following the time-dependent model [6]. To connect the public transit network with the road network we create links from each stop to its closest road edge, as shown in Figure 3. More specifically, we first create a parent node P for each stop that acts as an entrance point, and we connect all route nodes within the stop to P with zero-cost edges. Then we look for the closest road edge (u, v) to P. If the closest point to P is u (or v), we add a link edge from P to u (or v). Otherwise, a new node w is created along with two new edges (u, w) and (w, v) and a link edge is added from P to w. The cost of the link edge is given by the travel time on foot.

To process RR queries, RRAMEN employs variants of Barrett's algorithm for the language constrained shortest path [1]. We note that RRAMEN can build routes that use a combination of public transit modes, i.e., bus, train or tram, and walking, as well as driving and walking. In the latter case, we assume that a user can walk to where his/her car is parked, drive and possibly walk again to the destination. Due to lack of more fine-grained data, we make the optimistic assumption that when using a private car one parks as close as possible to the source location and to the destination.

RRAMEN can support a variety of types of users and queries. Table 1 shows two types of users and queries that would be well supported by RRAMEN. Note that while many of the requirements of individual users can also be addressed using existing route planning systems by executing multiple (independent) queries, RRAMEN makes the exploration and the decision making process much easier. Furthermore, while it is true that some users will choose a transportation mode regardless of the RR of the destination, we believe that allowing one to make such choices quickly and easily is of practical value. In particular, it may incentivize one to make choices that favour the use of public transit.

3 DEMONSTRATION SCENARIOS

In this section, we use real data from the city of Berlin to demonstrate how RRAMEN can be used in practice by individual users as well as urban planners. Note that while in the following discussion each query is associated with a particular type of user, RRAMEN imposes no such binding by design.

3.1 Individual users

Single source-multiple destinations. Consider a user who is at home and wishes to watch a movie which is showing in a

[7]https://www.openstreetmap.org/
[8]https://developers.google.com/transit/gtfs/reference

Table 1: Sample users and queries supported by RRAMEN

Users		Queries (location-wise)	
Individual Users	**Urban Planners**	**One-to-many / Many-to-one**	**Many-to-many**
Can use RRAMEN to find easily reachable facilities within a city, or to make decisions related to their commute.	Can study transportation systems and the impact of changes on them.	Queries from a single source to multiple destinations or from multiple sources to a single destination.	Queries from multiple sources to multiple destinations (which can also model regions).

number of movie theatres. The choice of theatre could be based on how convenient it would be to reach that theatre w.r.t. the means of transportation considered. Figure 1 illustrates this type of one-to-many scenario, where the source is denoted by an orange marker and the destinations are denoted by green markers. The result of this query reveals that the two locations on the left half of the map (denoted by blue dots) have an arrival time earlier by public transportation than by car. Likewise, the other three possible destinations have an earlier arrival time by car.

In addition to determining the RRs for all destinations, the user may be interested in the actual arrival time at a given destination (using either means of transportation). For that he/she would simply click on the destination which would cause not only the arrival times to be displayed but also the actual suggested routes. This is illustrated in Figure 4 where we "zoom in" on the map in order to show only the relevant part of the interface[9].

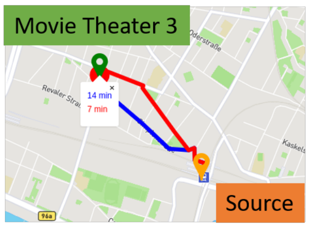


Figure 4: Arrival times for Movie Theatre 3, along with the routes by both public transit and private car.

Now, one may ask, what is the practical relevance of such a query? Naturally, there are trade-offs to be considered. For instance, if a user measures convenience by not having to look and (very likely) pay for parking and/or being able to have a drink or two after watching the movie, he/she would choose to go to some movie theatre using public transit regardless of its RR. Also, if a movie theatre that can be reached faster by public transit is far away, the user may choose a closer one to be reached by car, even with the associated overhead. Either way, RRAMEN empowers the user to consider such tradeoffs by him/herself.

Multiple sources-single destination. Let us now consider a scenario that is sort of the "reverse" of the one above. Consider a user that is moving to a city to work at a certain location and is looking for a place to rent. One criterion to choose where to live may be how convenient it would be for him/her to reach his/her workplace by public transit whenever his/her working shift starts. In this case, it would be useful to see the RRs of different rental units. An important difference of this scenario w.r.t. the previous one, is that in the previous scenario the RRs are computed w.r.t. the destinations (which theatre would be more convenient to reach from home), whereas now the RRs are computed w.r.t. the sources (from which potential apartment it would be more convenient to reach the workplace).

[9]Due to limited space, in what follows we show just the map part of the interface.

Figures 5 and 6 show the RRs of five apartments (orange markers) w.r.t. the working place of the user (green marker) at two different times, 7:30am and 5:30pm, respectively. By comparing the two figures we observe that the RR of all apartments but #4 remain the same. That is, if the user is interested in using public transportation, then apartments #3 and #5 are the best choices. We can imagine the case where apartment #4 may be a good choice if one is typically carpooling in the morning but riding a bus in the afternoon. The main point here is that, again, a single query returns the best available options to the user, leaving the final decision to him/herself.

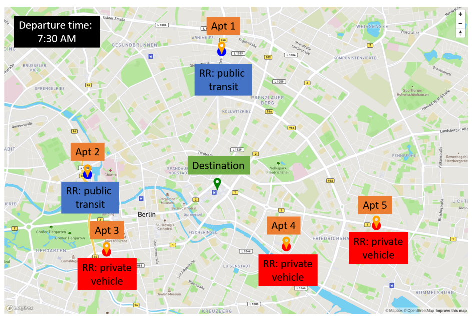


Figure 5: RR for apartments early in the morning.

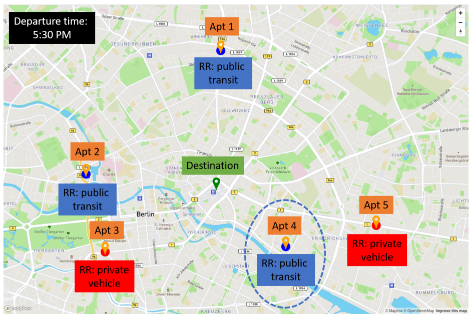


Figure 6: RR for apartments late in the afternoon. Note that Apartment #4's RR is different from earlier in the day.

Single source-region destination. Similar to the example above, one may want to consider living in a neighborhood that offers good transportation options towards a region on weekends. Figure 7 shows such a scenario at 8am, where the source is denoted by the orange marker and the destination region (say, the entertainment district) is determined by a polygon drawn by the user. We note that, besides selecting one or more destination points or sources as done in the scenarios above, RRAMEN allows the user to draw a region or select from a pre-determined set of regions, e.g., municipalities. As one can see in Figure 7, each point (node in the network) is either coloured blue or red (depending on whether the arrival time is earlier by public transit or

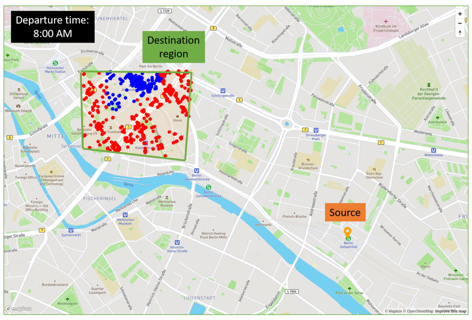


Figure 7: RR of points in a region w.r.t. a single source location at 8am.

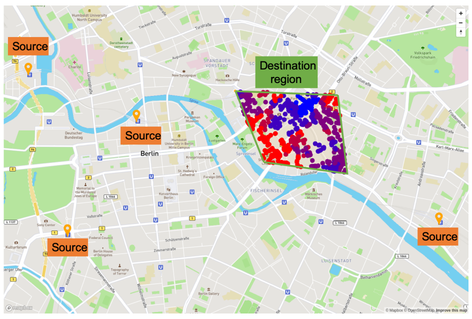


Figure 9: RR of all points within a region w.r.t. multiple source locations

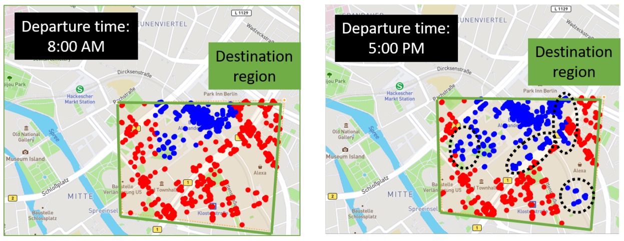
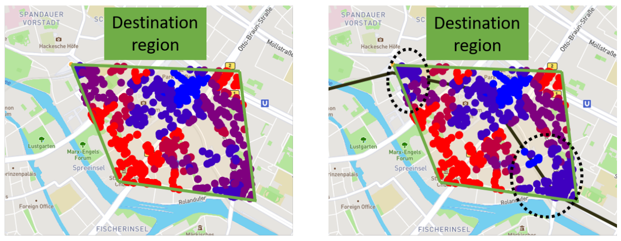


Figure 8: Same scenario of Figure 7 for 8am and 5pm. The dotted circles highlight the changed RRs.



Current RRs (i.e., before adding the new train line) Resulting RRs <u>after</u> adding the new train line

Figure 10: Detailed view before and after adding a new train line. The dotted circles highlight the changed RRs.

car, respectively). Interestingly, if the departure time were to be set to 5pm, the RRs within the destination region change quite a lot, as shown in Figure 8. Once again, by changing a single parameter in RRAMEN's interface the user can be better informed before making his/her decision.

3.2 Urban Planner

Multiple sources-region destination. Here we envision a scenario where a planner would select a number of representative points in the city, e.g., shopping malls, and identify their RR w.r.t. a region, e.g., downtown. While on the surface this may seem similar to the multiple sources-single destination query discussed earlier, there is a fundamental difference. For the earlier query each source location had its own RR, whereas for the current query, the RR for each one of the potential destination locations within the target region reflects an average over the RRs computed from all different source locations. Figure 9 illustrates this scenario, where the colors of each point reflect how much more likely it is for a transportation mode to be the most efficient w.r.t. the source locations. That is, the closer to red (or blue) the more likely it is, over the set of source locations, that private cars (or public transit) reach that point earlier.

Multiple sources-region destination before-after analysis. The following discussion is a natural follow-up from the previous one and depicts a scenario that may reflect best the usefulness of RRAMEN for an urban planner. What would happen if more resources were allocated to the public transportation system? For example, what would happen if one new train line was added? Figure 10 illustrates such a before-after comparison. One can clearly see that the northwest and southwest corners of the region in the figure on the right have more blue points than in the figure on the left. That is, by adding one additional train line, the RR of some locations would "flip" towards public transit, which we believe should be one of the goals of an urban planner. A similar analysis can be done in case one removes transit lines and/or removes (or adds) thoroughfares for private cars.

4 CONCLUSION

We presented RRAMEN, a web-based tool based on the concept of Relative Reachability, the main goal of which is to aid individual users and urban planners (among others) in making informed choices and evaluating changes w.r.t. a city's transportation network. There are a few interesting directions for further work, such as taking into account the demographics of a city when evaluating how changes impact a city's population at large. From a computational aspect, it would be also interesting to consider incremental and parallel/distributed computation models to scale up the overall efficiency of the the tool.

ACKNOWLEDGMENTS

This work is partially supported by CNPq Brazil, NSERC Canada Deutsche Forschungsgemeinschaft (DFG, Grant No. GR 4497/2).

REFERENCES

[1] Chris Barrett et al. 2000. Formal-Language-Constrained Path Problems. *SIAM J. Comput.* 30, 3 (2000), 809–837.
[2] Yung-Hsiang Cheng and Ssu-Yun Chen. 2015. Perceived accessibility, mobility, and connectivity of public transportation systems. *Transportation Research Part A* 77 (2015), 386–403.
[3] Theodoros Chondrogiannis et al. 2019. Relative Reachability Analysis as a Tool for Urban Mobility Planning: Position Paper. In *IWCTS @ ACMSIGSPATIAL*.
[4] Markus Innerebner et al. 2013. ISOGA: A System for Geographical Reachability Analysis. In *W2GIS*. 180–189.
[5] Atsu Okabe et al. 2015. SANET A Spatial Analysis along Networks (Ver.4.1). http://sanet.csis.u-tokyo.ac.jp
[6] Evangelia Pyrga et al. 2008. Efficient models for timetable information in public transportation systems. *JEA* 12 (2008), 2–4.
[7] L. Smith et al. 1995. *TRANSIMS: Transportation analysis and simulation system.* Technical Report LA-UR-95-1641. Los Alamos National Lab.
[8] Jeffrey Xu Yu and Jiefeng Cheng. 2010. Graph Reachability Queries: A Survey. In *Managing and Mining Graph Data*. Chapter 6, 181–215.

JedAI^3: beyond batch, blocking-based Entity Resolution

George Papadakis[1], Leonidas Tsekouras[2], Manos Thanos[3], Nikiforos Pittaras[2], Giovanni Simonini[5], Dimitrios Skoutas[6], Paul Isaris[7], George Giannakopoulos[2,7], Themis Palpanas[4], Manolis Koubarakis[1]
[1]National and Kapodistrian University of Athens, Greece {gpapadis,koubarak}@di.uoa.gr
[2]NCSR "Demokritos", Greece {ltsekouras, pittarasnikif, ggianna}@iit.demokritos.gr
[3]KU Leuven, Belgium emmanouil.thanos@kuleuven.be
[4]Paris Descartes University, France themis@mi.parisdescartes.fr
[5]University of Modena and Reggio Emilia, Italy simonini@unimore.it
[6]IMSI, Athena Research Center, Greece dskoutas@imis.athena-innovation.gr
[7]SciFY, Greece paul@scify.org

ABSTRACT

JedAI is an open-source toolkit that allows for building and benchmarking thousands of schema-agnostic Entity Resolution (ER) pipelines through a non-learning, blocking-based end-to-end workflow. In this paper, we present its latest release, JedAI^3, which conveys two new end-to-end workflows: one for budget-agnostic ER that is based on similarity joins, and one for budget-aware (i.e., progressive) ER. This version also adds support for pre-trained word or character embeddings and connects JedAI to the Python data analysis ecosystem. Overall, these enhancements provide JedAI with features offered by no other ER tool, especially in the schema- and domain-agnostic context.

1 INTRODUCTION

Entity Resolution (ER) aims to detect *duplicates*, i.e., different entity profiles that describe the same real-world objects. It is a core data integration task, with many applications that range from knowledge bases to question answering [8]. Yet, the available systems focus exclusively on *batch ER*, which is carried out in a budget-agnostic, offline way that imposes no strict constraints on temporal or computational resources. This means that they do not support *progressive ER*, which is carried out in a budget-aware manner that determines specific time frames or resources (e.g., by executing a fixed number of comparisons).

Even for batch ER, though, the existing tools have significant limitations: they cover the end-to-end pipeline partially, they constitute stand-alone systems with a limited variety of methods (usually the methods proposed by their creators), they apply only to a specific data type (i.e., structured or semi-structured data), or they require power users, providing insufficient guidelines on how to perform ER efficiently and effectively [8]. Another major disadvantage of existing tools is that they typically disregard the bulk of similarity join algorithms that allow for detecting pairs of duplicates in a rather efficient way.

Magellan resolves most of these issues, but is restricted to *Clean-Clean ER*, i.e., the task of detecting duplicate profiles across two overlapping, but individually duplicate-free datasets. This means that Magellan cannot tackle *Dirty ER*, i.e., the task of resolving the entities of a single dataset that contains duplicates in itself. Moreover, Magellan applies exclusively to relational data, it lacks a GUI, merely offering a command-line interface, and

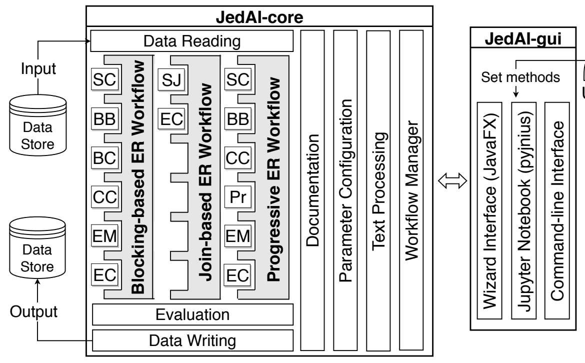


Figure 1: JedAI's architecture.

requires heavy user involvement. Its goal is actually to facilitate the development of tailor-made methods for the data at hand [8].

Java gEneric DAta Integration (JedAI) Toolkit [15] goes beyond existing ER tools by focusing on non-learning, schema- and structure-agnostic methods, which apply seamlessly to both structured and semi-structured data. At the same time, JedAI requires minimal human intervention, as neither domain knowledge nor training sets are needed. At its core lies a blocking-based end-to-end ER workflow, which is implemented by JedAI-core[1], conveying numerous state-of-the-art methods in each step. These methods can be mixed and matched to form thousands of ER pipelines that can be easily benchmarked through the wizard-like interface of its desktop application, JedAI-gui[2]. Thus, JedAI fulfills the main challenges arising in data integration [5]: the development of extensible, open-source[3] tools and the provision of solutions that apply to data of any structuredness - even unstructured (free-text) data. These new capabilities are exhibited through a live demonstration that involves user interaction.

In more detail, this demonstration presents the latest release of JedAI^3, which significantly enhances the core blocking-based workflow: it connects it with the Python ecosystem (see Section 5.1) and enriches it with pre-trained embeddings and with new techniques and capabilities per workflow step (see Section 2.1). Most importantly, JedAI now supports two new workflows: one based on similarity joins (Section 3) and one implementing budget-aware ER pipelines (Section 4). These enhancements equip JedAI with unique features that are offered by no other ER tool, especially in a schema- and domain-agnostic context.

[1]https://github.com/scify/JedAIToolkit
[2]https://github.com/scify/jedai-ui
[3]All code is available under Apache License V2.0, which supports both academic and industrial applications.

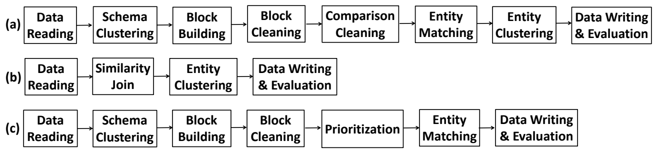


Figure 2: ER workflows in JedAI: (a) blocking-based; (b) join-based (*new*); (c) progressive (*new*).

Id	Rule	Dataset	Recall	Precision	F1
R1	0.59 < JaccardSim (title1, title2) & 0.26 < JaccardSim (authors1, authors2)	DBLP-ACM	0.926	0.930	0.928
R2	0.53 < JaccardSim (title1, title2)	Cora	0.855	0.749	0.799
R3	0.25 < JaccardSim (all_tokens1, all_tokens2)	10K census	0.969	0.995	0.982

Figure 3: Three matching rules along with their performance over established datasets.

2 BATCH, BLOCKING-BASED ER WORKFLOW

Figure 2(a) depicts the budget-agnostic, blocking-based end-to-end ER workflow of JedAI. It consists of the following steps.

1) *Data Reading* loads from disk the dataset(s) to be processed and the golden standard. The JedAI data model accommodates both structured (relational databases, CSV) and semi-structured (SPARQL endpoints, CSV, XML, OWL and RDF) data as well as any mixture of those.

2) *Schema Clustering* groups together attributes that share similar names and/or values, but are not necessarily semantically equivalent. It is an optional step that is suitable for highly heterogeneous datasets with hundreds of different attribute names. In these settings, it significantly improves the overall precision with no impact on recall [15, 18].

3) *Block Building* clusters similar entities into blocks so as to drastically reduce the candidate match space. It includes most of the state-of-the-art methods, using their schema-agnostic adaptation [13], which extracts multiple blocking keys from each entity and places it into multiple blocks. The resulting *overlapping* blocks contain high levels of redundancy, achieving high recall at the cost of low precision [13], due to two types of unnecessary comparisons [13]: (i) the *redundant* ones, which are repeated across different blocks, and (ii) the *superfluous* ones, which involve non-matching entities.

4) *Block Cleaning* is an optional step that cleans the original blocks from both types of unnecessary comparisons, improving their precision at a negligible cost in recall. All available methods are complementary and can be combined.

5) *Comparison Cleaning* is another optional step that targets both types of unnecessary comparisons. It operates at the level of individual comparisons, achieving higher accuracy at the cost of a higher time complexity. Several methods are included, primarily Meta-blocking techniques [14, 18].

6) *Entity Matching* implements schema-agnostic methods for assessing the value similarity of all entity pairs in the final set of blocks. These methods can be combined with various similarity measures and graph or bag representation models from the Text Processing component (see Figure 1). The end result is a *similarity graph*, where the nodes correspond to entities, and the weighted edges connect compared entities.

7) *Entity Clustering* includes the main methods that are typically used for partitioning the nodes of the similarity graph into equivalence clusters, such that every cluster corresponds to a distinct real-world object [6].

8) *Evaluation* uses the golden standard of the selected dataset in order to compute several measures for effectiveness and time efficiency. The user may store intermediate or end results through the *Data Writing* functionality.

2.1 New features

This workflow has been enriched with support for embeddings, which lie at the core of the latest ER works that are based on deep learning [4, 12]. JedAI supports any pre-trained character- (e.g., fastText [1]) and word-level embeddings (e.g., Glove [17], word2vec [11]). The user is only required to provide a path to an embeddings file in CSV form. These embeddings can be used in two steps: (i) in Block Building, combining them with LSH as in [4], and (ii) in Entity Matching, providing external, contextual information for the computation of value similarities, which is particularly useful for noisy entity profiles [12].

Another major improvement is the combination of multiple Entity Matching methods. It is now possible to select any (reasonable) number of algorithms, similarity measures and representation models for comparing the candidate matches that are contained in a set of blocks. For example, it is possible to combine the traditional bag-of-words model with the popular word2vec embeddings, each coupled with a different similarity measure. JedAI combines the similarity scores produced by the individual methods and normalizes them into the $[0, 1]$ interval.

Finally, several steps of this workflow have been enriched with new techniques. For example, Data Reading and Data Writing now support HDT and JSON files, while Entity Clustering conveys two new algorithms for Clean-Clean ER that are designed for solving the assignment problem: an efficient approximation called *Row-Column Proxy Clustering* [9], and a heuristic algorithm called *Assignment Problem Heuristic Clustering* [2].

3 JOIN-BASED ER WORKFLOW (NEW)

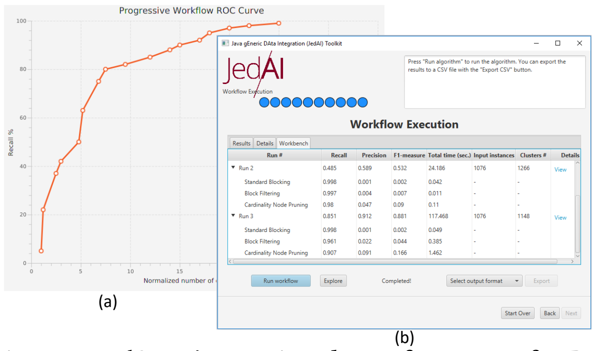
Similarity joins [7, 10] constitute a rather efficient alternative to blocking-based ER, especially for structured data that conforms to a schema of known quality. These joins accelerate the execution of matching rules and are combined with an Entity Clustering algorithm for high effectiveness [6], as shown in Figure 2(b). For example, consider the atomic (R1, R2) and composite (R3) matching rules in Figure 3, which exhibit very high effectiveness in combination with Connected Components clustering over established benchmark datasets [13].

JedAI-core now implements the workflow in Figure 2(b), conveying a library with the state-of-the-art string [7] and set [10] similarity join techniques. Most of them require that the user is familiar with the schema describing the entity profiles so as to select the most suitable attribute names. JedAI-gui facilitates this process through the *data exploration* feature, which allows for observing the schema and the values of entity profiles. Most importantly, JedAI goes beyond this schema-aware approach, applying similarity joins to semi-structured data through a schema-agnostic transformation that considers all tokens (or q-grams) in all attribute values (e.g., R2 in Figure 3). The only caveat comes from the resulting low similarity thresholds that render inapplicable character-based and token-based methods that are crafted for much larger thresholds [7, 10]. JedAI covers such cases by incorporating novel join techniques that inherently support low similarity thresholds, like SilkMoth [3] and Atlas [21].

The list of the character- and token-based similarity joins that are currently supported by JedAI appears in Figure 4. It entails

Similarity Join Methods		Prioritization Methods
Token-based	Character-based	1) Local Progressive Sorted Neighborhood
1) AllPairs	1) AllPairs	2) Global Progressive Sorted Neighborhood
2) PPJoin	2) FastSS	3) Progressive Block Scheduling
3) SilkMoth	3) PassJoin	4) Progressive Entity Scheduling
	4) EdJoin	5) Progressive Global Top Comparisons
		6) Progressive Local Top Comparisons

Figure 4: The methods available for the workflow steps Similarity Join (Section 3) and Prioritization (Section 4).



Press "Run algorithm" to run the algorithm. You can export the results to a CSV file with the "Export CSV" button.

Workflow Execution

Run #	Recall	Precision	F1-measure	Total time (sec.)	Input instances	Clusters #	Details
▼ Run 2	0.485	0.589	0.532	24.186	1076	1266	View
Standard Blocking	0.998	0.001	0.002	0.042	-	-	
Block Filtering	0.997	0.004	0.007	0.011	-	-	
Cardinality Node Pruning	0.98	0.047	0.09	0.11	-	-	
▼ Run 3	0.851	0.912	0.881	117.468	1076	1148	View
Standard Blocking	0.998	0.001	0.002	0.049	-	-	
Block Filtering	0.961	0.022	0.044	0.385	-	-	
Cardinality Node Pruning	0.907	0.091	0.166	1.462	-	-	

Figure 5: JedAI-gui reporting the performance of a Progressive ER workflow (a) and its benchmark screen (b).

most state-of-the-art approaches, as determined by recent experimental analyses [7, 10]. Any combination of matching rules and similarity join techniques is allowed to form atomic and composite schema-based or schema-agnostic pipelines. These can be readily juxtaposed to more complex blocking-based ones, providing a unique feature that is offered by no other relevant tool.

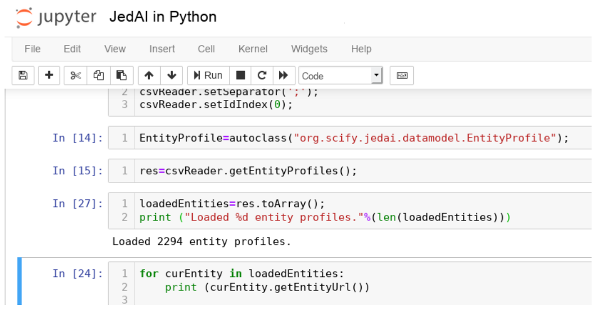
4 PROGRESSIVE ER WORKFLOW (NEW)

Progressive ER applies to *budget-aware applications*, which have limited computational or time resources. These limitations can only be addressed in a *pay-as-you-go* way that provides the best possible *partial solution* in the context of the available resources. For example, the Google dataset search system has indexed ~26 billion datasets [5], which can only be resolved progressively.

Schema-based progressive methods have been proposed [16, 20], but JedAI exclusively considers domain-agnostic ones [19], implementing the workflow in Figure 2(c). The first four steps are common with the blocking-based ER workflow, which is depicted in Figure 2(a). The only difference is that Data Reading also receives as input the user-specified *budget*, either in terms of the maximum running time or the maximum number of executed comparisons. Next, *Prioritization* is applied, assigning a weight to all entities, comparisons or blocks in order to schedule their processing in decreasing likelihood that they involve duplicates. Then, the top-weighted entity pairs are iteratively emitted, one at a time, in order to compare the corresponding entity descriptions (Entity Matching). Finally, the Evaluation estimates the rate of detected duplicates per comparison, i.e., the evolution of recall as more comparisons are executed. The resulting diagram is used for estimating the area under curve, which is analogous to performance - see Figure 5(a).

JedAI implements the Prioritization methods in Figure 4, which can be distinguished into two types:

i) Inspired by Sorted Neighborhood (SN), the *sort-based methods* rely on the similarity of blocking keys. They produce a list of entities by sorting all descriptions alphabetically, according to the corresponding blocking keys. In the schema-agnostic context, every token forms a blocking key and thus, every entity appears in the list as many times as the number of its distinct attribute value tokens. To avoid the large number of redundant



```
    csvReader.setSeparator(';');
    csvReader.setIdIndex(0);

In [14]:  EntityProfile=autoclass("org.scify.jedai.datamodel.EntityProfile");

In [15]:  res=csvReader.getEntityProfiles();

In [27]:  loadedEntities=res.toArray();
          print ("Loaded %d entity profiles."%(len(loadedEntities)))

          Loaded 2294 entity profiles.

In [24]:  for curEntity in loadedEntities:
              print (curEntity.getEntityUrl())
```

Figure 6: JedAI in Python.

comparisons and the arbitrary ordering of entities with identical keys, *Local Schema-agnostic Progressive SN* [19] weights all comparisons within the current window size via a schema-agnostic function that considers the frequency of appearance of every entity in the list along with the co-occurrence frequency of entity pairs for the current window size. The *Global Schema-agnostic Progressive SN* [19] does the same, but for a predetermined range of windows, eliminating their redundant comparisons.

ii) The *hash-based methods* are based on identical, schema-agnostic blocking keys and the resulting overlapping blocks. Similar to Meta-blocking, they assign a weight to every pair of candidate matches, assuming that their similarity is proportional to the number of blocks they share [14]. *Progressive Block Scheduling* [19] orders the blocks in ascending number of comparisons and then prioritizes all comparisons in the current block, by ordering them in decreasing weight. *Progressive Profile Scheduling* [19] orders entities in decreasing average weight of the corresponding candidate matches and then prioritizes all comparisons involving the current entity by ordering them in decreasing weight. *Progressive Global (Local) Top Comparisons* considers the top-K weights over the entire blocking graph (per entity), where K is derived from the given budget.

5 USER INTERFACE

Up to v2.1, JedAI offered two interfaces for user interaction [15]: (i) the desktop application, JedAI-gui, which offers a graphical interface, and (ii) the command-line interface implemented by JedAI-core. Both of them allow for constructing any combination of the available methods in the context of the blocking-based end-to-end ER workflow. Especially, JedAI-core conveys the Documentation module (see Figure 1), which facilitates the use and configuration of ER methods. It also enables the benchmarking of different workflows or configurations over a particular dataset through the workbench window, which summarizes the outcome of all runs and maintains details about the performance and the configuration of every step [15] (see Figure 5(b)).

5.1 New features

JedAI[3] extends both interfaces so that they cover all new features discussed above. The command-line interface has also been enriched with documentation support: at any step, the user is able to retrieve information about individual methods or specific parameters, thus facilitating their use. JedAI-core has also been augmented with a Python wrapper based on `pyjnius`, thus facilitating its adoption by the large user base of Python data analytics (see Figure 6). For example, the new wrapper allows for integrating JedAI with popular frameworks like `scikit`[4] for machine learning and `NLTK`[5] for natural language processing.

[4]https://scikit-learn.org
[5]https://www.nltk.org

6 DEMONSTRATION SCENARIOS

In this demo, we will present JedAI through a live interaction with users so as to highlight all new features discussed above. First, the user is asked to choose among the available interfaces: command-line, JedAI-gui or a Jupyter notebook. Then, she is asked to select one dataset among a set of carefully selected ones, which are easy to comprehend, yet interesting for an ER application, while involving both a structured and a semi-structured part (e.g., CSV-XML). These settings lay the ground for the following demo scenarios that showcase JedAI's functionalities.

1. Versatility. In this scenario, we compare the new ER workflows supported by JedAI with the blocking-based one that lies at the core of the previous version. Four different workflows are involved in this process:

(1) The user builds a traditional blocking-based workflow to be applied to the selected dataset.
(2) The same workflow is enhanced with word embeddings so as to examine the effect of deep learning techniques on improving accuracy.
(3) The user is asked to form a join-based end-to-end ER workflow with matching rules of arbitrary complexity; this might seem a complex procedure, but in reality, JedAI's data visualization feature simplifies it to a large extent.
(4) The user forms a Progressive ER workflow and applies it to the same dataset.

In all cases, the user can manually configure all methods, or use the recommended default parameter values. JedAI's workbench functionality allows to easily compare the performance of these fundamentally different workflows, assessing their pros and cons.

2. Automatic Configuration. In this scenario, we fine-tune the configuration parameters of the above four workflows in one of the available automatic ways, i.e., through random or grid search in a step-by-step or a holistic approach. Thus, the goal of this scenario is three-fold: (i) to test how well users can manually tune the parameters, (ii) to evaluate how close the default parameter values are to the "optimal" ones, and (iii) to compare the four workflows in terms of their best possible performance. For example, the performance of the fine-tuned progressive workflow demonstrates the minimum number of comparisons that are required for achieving sufficiently high recall. How close are the other three workflows to this ideal case? Again, JedAI's workbench functionality, the first for such a tool, renders these complex comparative process into a simple procedure.

3. Scalability. This last scenario focuses on the time efficiency of the selected workflows. The results of the previous scenarios advise us beforehand about the relative running time of the selected workflows, as JedAI's workbench feature reports both effectiveness and efficiency measures. Here, though, we examine how well each workflow scales to datasets of increasing size (10K, 50K, 100K, 500K, 1M and 2M entities), which have been derived from the selected dataset using artificial noise. This scenario demonstrates how workflows with similar running times over small datasets might end up differing by orders of magnitude. Most importantly, JedAI's workbench reports the running time per workflow step, thus facilitating the detection of bottlenecks.

Finally, it is worth stressing that during our demonstration, we will take special care to explain to users how to make the most of JedAI's functionalities, emphasizing the new features. Note also that most of the features listed above are demonstrated for the first time, and are not supported by any other ER system.

7 CONCLUSIONS

JedAI[3] is an open-source ER tool with 4 unique characteristics: (i) Based on blocking and similarity joins, it implements two different end-to-end workflows for batch ER that allow for composing millions of pipelines. (ii) It supports pre-trained embeddings of any kind. (iii) It supports budget-aware ER, enabling thousands of schema-agnostic progressive workflows. (iv) It can be integrated with Python's data analysis ecosystem. These capabilities will be exhibited through a live demonstration with user interaction.

Acknowledgements. This work was partially funded by the EU H2020 projects ExtremeEarth (825258) and SmartDataLake (825041).

REFERENCES

[1] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *TACL*, 5:135–146, 2017.
[2] Y. Chuang, H. Lee, and S. Tan. A robust heuristic method on the clustering-assignment problem model. *Computers & Industrial Engineering*, 98:63–67, 2016.
[3] D. Deng, A. Kim, S. Madden, and M. Stonebraker. Silkmoth: An efficient method for finding related sets with maximum matching constraints. *PVLDB*, 10(10):1082–1093, 2017.
[4] M. Ebraheem, S. Thirumuruganathan, S. R. Joty, M. Ouzzani, and N. Tang. Distributed representations of tuples for entity resolution. *PVLDB*, 11(11):1454–1467, 2018.
[5] B. Golshan, A. Y. Halevy, G. A. Mihaila, and W. Tan. Data integration: After the teenage years. In *PODS*, 2017.
[6] O. Hassanzadeh, F. Chiang, R. Miller, and H. Lee. Framework for evaluating clustering algorithms in duplicate detection. *PVLDB*, 2(1), 2009.
[7] Y. Jiang, G. Li, J. Feng, and W. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.
[8] P. Konda, S. Das, et al. Magellan: Toward building entity matching management systems. *PVLDB*, 9(12):1197–1208, 2016.
[9] J. M. Kurtzberg. On approximation methods for the assignment problem. *J. ACM*, 9(4):419–439, 1962.
[10] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *PVLDB*, 9(9):636–647, 2016.
[11] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.
[12] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep learning for entity matching: A design space exploration. In *SIGMOD*, pages 19–34, 2018.
[13] G. Papadakis, G. Alexiou, G. Papastefanatos, and G. Koutrika. Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data. *PVLDB*, 9(4):312–323, 2015.
[14] G. Papadakis, J. Svirsky, A. Gal, and T. Palpanas. Comparative analysis of approximate blocking techniques for entity resolution. *PVLDB*, 9(9):684–695, 2016.
[15] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, and M. Koubarakis. The return of jedai: End-to-end entity resolution for structured and semi-structured data. *PVLDB*, 11(12):1950–1953, 2018.
[16] T. Papenbrock, A. Heise, and F. Naumann. Progressive duplicate detection. *IEEE TKDE.*, 27(5):1316–1329, 2015.
[17] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *EMNLP*, pages 1532–1543, 2014.
[18] 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.
[19] G. Simonini, G. Papadakis, T. Palpanas, and S. Bergamaschi. Schema-agnostic progressive entity resolution. In *ICDE*, pages 53–64, 2018.
[20] S. E. Whang, D. Marmaros, and H. Garcia-Molina. Pay-as-you-go entity resolution. *TKDE*, 25(5):1111–1124, 2013.
[21] J. Zhai, Y. Lou, and J. Gehrke. ATLAS: a probabilistic algorithm for high dimensional similarity search. In *SIGMOD*, 2011.

RetroLive: Analysis of Relational Retrofitted Word Embeddings

Michael Günther
Database Systems Group, Technische Universität Dresden
Dresden, Germany
Michael.Guenther@tu-dresden.de

Maik Thiele
Database Systems Group, Technische Universität Dresden
Dresden, Germany
Maik.Thiele@tu-dresden.de

Erik Nikulski
Database Systems Group, Technische Universität Dresden
Dresden, Germany
Erik.Nikulski@tu-dresden.de

Wolfgang Lehner
Database Systems Group, Technische Universität Dresden
Dresden, Germany
Wolfgang.Lehner@tu-dresden.de

ABSTRACT

Text values are valuable information in relational database systems for analysis and machine learning (ML) tasks. Since ML techniques depend on numerical input representations, word embeddings are increasingly utilized to convert symbolic representations such as text into meaningful numbers. However, those models do not incorporate the context-specific semantics of text values in the database. To significantly improve the representation of text values occurring in DBMS, we propose a novel retrofitting approach called Retro which considers both, the semantics of the word embedding and the relational schema. Based on this, we developed RetroLive, an interactive system, that allows exploring how the retrofitted embeddings improve the performance for various ML and integration tasks. Moreover, the demo includes several interactive visualizations to explore the characteristics of the adapted vectors and their connection to the relational database.

1 INTRODUCTION

Due to their appealing properties, word embedding techniques such as word2vec [8], GloVe [9] or fastText [2] have become conventional wisdom in many research fields such as NLP or information retrieval to represent text values. These word vectors are trained by processing large text corpora, e.g. the GloVe embedding[1] was trained on a corpus with 840 billion tokens. These embeddings are typically used to convert text values in a meaningful numerical representations used as input for ML tasks. However, a naïve application of a word embedding model is not sufficient to represent the meaning of text values in a database which is often more specific than the meaning in the natural language. Thus, we aim for additionally incorporating information given by the disposition of the text values in the database schema into the embedding, e.g. which words appear in the same column or are related. Therefore, we developed a relational retrofitting approach called Retro [5] which is able to automatically derive high-quality numerical representations of textual data residing in databases without any manual effort.

[1]https://nlp.stanford.edu/projects/glove/

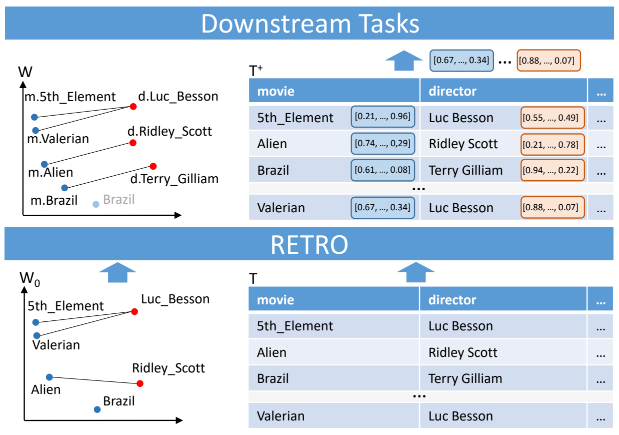


Figure 1: Relational Retrofitting: basis word embedding W^0 and relation T (left), retrofitted word embedding W and augmented relation T^+

Relational Retrofitting. Figure 1 provides a small example sketching the main principles of the relational retrofitting process. Retro expects a database and a word embedding as input, e.g. a movie table T that should be retrofitted into a pre-trained word embedding W_0. W_0 can be generated using well-known text embedding techniques. Those models are able to generate embeddings capturing syntactic and semantic relations between words, such as word analogies, gender-inflections, or geographical relationships. To provide vector representations for textual information in databases one could simply reuse the vectors of pre-trained embeddings, e.g. map each term from T to a term-vector pair in W_0.

However, there often will be a semantic mismatch between word embeddings and textual information in databases:

1) Given the movie table, it is known that all entities within the movie column must be movies, although some of the titles, such as "Brazil" or "Alien", may be interpreted differently by the word embedding model.

2) T provides a specific amount of relation types like the movie-director, whereas in the word embedding representation W_0 large amounts of implicit relations are modeled, e.g. if the director of a movie is also the producer or one of the actors this might be represented in the word embedding although not explicitly visible.

3) Terms in T which occurring infrequent in the general domain can not be modeled accurately by word embedding models.

For instance word2vec has a limited vocabulary according to a frequency threshold. Many terms appearing in a database will therefore have no counter-part within the embedding.
Therefore, we developed Retro [5] which incorporates the database semantics and augments all terms in database relation by a dense vector representation. In the context of our movie example, Retro would generate a new embedding for "Terry Gilliam" which should be close to other directors and their respective vectors. Furthermore, Retro would create vectors for all other textual values in the movie table that encode the semantics given by the pre-trained word embeddings and the database. On one hand, this ensures that vectors appearing in the same column, such as movies or directors, are close to each other. On the other hand, this ensures that movie-director pairs can be resolved. These vectors are ready-to-use for a wide range of ML, retrieval and data cleaning tasks such as classification, regression, null value imputation, entity resolution, and many more.
Demonstration. Our web-based demo called RetroLive showcases our novel retrofitting approach and guides the user through the whole retrofitting process. The demo provides different database schemas and target word embeddings the users can choose from and allows to explore their data statistics and characteristics. Additionally, RetroLive allows to set and fine-tune the various hyperparameters of the retrofitting learning problem. The impact of the hyperparameter values, the input data characteristic and target embeddings can be studied in detail using either 2-dimensional projections or by using word similarity and analogy benchmarks. To demonstrate the usefulness of the embedding generated by the relational retrofitting approach, RetroLive comes with various pre-defined extrinsic evaluation tasks for classification and regression. Furthermore, the users are able to define their own machine learning tasks. Finally, the demo includes several baseline approaches such as the node embedding technique DeepWalk [10] and the original retrofitting approach [4] from Faruqui et al.

2 RETRO – BACKGROUND

Retro aims at combining word embeddings with relational text data to generate good vector representations for text values residing within databases. Therefore, our relational retrofitting approach learns a matrix of vector representations $W = (\boldsymbol{v_1}, \dots \boldsymbol{v_n})$ with $\boldsymbol{v_i} \in \mathbb{R}^D$ for every unique text value $T = (\boldsymbol{t_1}, \dots \boldsymbol{t_n})$ in a database. To find initial word vectors for every text value, we tokenize the them based on the vocabulary of the basis word embedding model and build centroid vectors which is a convenient way to obtain a representation of text values consisting of multiple tokens [1, 11]. These vectors are stored in a matrix $W^0 = (\boldsymbol{v}'_1, \dots \boldsymbol{v}'_n)$ forming the basis for the retrofitting process. Besides, *columnar and relational connections* are extracted from the database (see Section 2.1). This encompasses semantic relations between text values, which are derived from the relational schema. Those connections are used to create a representation capturing the context of the text value in the database and thus help to preserve their semantics more accurately compared to a plain word embedding representation. In order to apply node embedding techniques like DeepWalk [10], all information is condensed into a common graph representation shortly presented in Section 2.2. The core procedure of the relational retrofitting is the adaption of the basis vectors W^0. This is performed by solving an optimization problem detailed further in Section 2.3.

2.1 Extracting Relational Information

One can derive different structural relations from the alignment of text values in the relational schema.
Columnar Connections: Text values with the same attribute, i.e. appearing in the same column, usually form hyponyms of a common hypernym (similar to subclass superclass relations). Thus, they share a lot of common properties which typically leads to similarity. We capture this information and assign each text value t_i to its column $C(i)$.
Relational Connections: Relations exhibit from the co-occurrence of text values in the same row as well as from foreign key relations. Those relations are important to characterize the semantics of text value in the database. We define a set of relation types R which is specific for a certain pair of text value columns. Those columns are related because they are either part of the same table or there exists a foreign key relationship between their tables. For every relation type $r \in R$ there is a set E_r containing the tuples of related text value ids. Relation types are directed. Accordingly, there is an inverted counterpart $\bar{r}$ for each relation r with $E_{\bar{r}} = \{(j,i)|(i,j) \in E_r\}$.

2.2 Graph Generation

A graph representation is a pre-requisite for training node embeddings such as DeepWalk [10] which serve as a strong baseline for Retro. To compile a graph $G = (V, E)$ the text values extracted by the tokenization process together with columnar and relational connections (Section 2.1) are combined. The node-set $V = V_C \cup V_T$ consists of text value nodes V_T for every distinct text value in a database column and blank nodes for every column V_C. The edge set $E = \bigcup_{r \in R} E_r \cup E_C$ consists of a set of edges E_r for every relational type and edges E_C connecting text values of one column to a common category node.

2.3 Optimization Problem

Retro considers relational and columnar connections (see Section 2.1) to retrofit an initial embedding. Accordingly, we define a loss function Ψ adapting embeddings to be similar to their basis word embedding representation W^0, the embeddings appearing in the same column, and related embeddings.

$$\Psi(W) = \sum_{i=1}^{n} \left[\alpha_i ||\boldsymbol{v_i} - \boldsymbol{v}'_i||^2 + \beta_i \Psi_C(\boldsymbol{v_i}, W) + \Psi_R(\boldsymbol{v_i}, W)\right] \tag{1}$$

The columnar loss is defined by Ψ_C and treats every embedding $\boldsymbol{v_i}$ to be similar to the constant centroid $\boldsymbol{c_i}$ of the basis embeddings of text values in the same column $C(i)$.

$$\Psi_C(\boldsymbol{v_i}, W) = ||\boldsymbol{v_i} - \boldsymbol{c_i}||^2 \quad \boldsymbol{c_i} = \frac{\sum_{j \in C(i)} \boldsymbol{v}'_j}{|C(i)|} \tag{2}$$

The relational loss Ψ_R treats embeddings v_i and v_j to be similar if there exists a relation between them and dissimilar otherwise. E_r is the set of tuples where a relation r exists. $\widetilde{E_r}$ is the set of all tuples $(i, j) \notin E_r$ where i and j are part of relation r. Thus, each of both indices has to occur at least in one tuple of E_r.

$$\Psi_R(\boldsymbol{v_i}, W) = \sum_{r \in R} \left[\sum_{\substack{j:(i,j)\\ \in E_r}} \gamma_i^r ||\boldsymbol{v_i} - \boldsymbol{v_j}||^2 - \sum_{\substack{k:(i,k)\\ \in \widetilde{E_r}}} \delta_i^r ||\boldsymbol{v_i} - \boldsymbol{v_k}||^2\right] \tag{3}$$

α, β, γ and δ are hyperparameters. Ψ has to be a convex function. In [5] we proved the convexity of Ψ for hyperparameter configurations fulfilling the following inequation:

$$\forall r \in R, i \in \{1, \ldots, n\} \quad (\alpha_i \geq 0, \quad \beta_i \geq 0, \quad \gamma_i^r \geq 0) \qquad (4)$$

$$\forall \boldsymbol{v_i} \in W \quad (4\alpha_i - \sum_{r \in R} \sum_{j:(i,j) \in \widetilde{E_r}} \delta_i^r \geq 0)$$

Given the property of convexity, an iterative algorithm can be used to minimize Ψ. Using sparse matrix calculations this can be done in parallel with linear time complexity according to the number of text values in W. Details are outlined in [5].

3 SYSTEM OVERVIEW

RetroLive is a fully functional system built on top of PostgreSQL. The front-end is a web application managing the provided input databases and embeddings (Section 3.1), controlling the relational retrofitting process (Section 2) and performs several ML tasks (Section 3.2). The system overview is depicted in Figure 2.
Based on an input *database schema* (①) and a target word embedding wodel (②) a so-called *basis word embedding* (③) is created which encodes each text value in the database as a dense vector. These vectors together with the extracted relational schema information (④) comprise the input for the relation retrofitting which returns a retrofitted embedding (⑤). In addition, a *graph representation* (⑦) of the text value relations (see Section 2.2) is generated and used to create node embeddings (⑧) which serve as a strong baseline for our approach. Moreover, the *graph representation* is used to apply the original retrofitting approach of Faruqui et al. (⑥). The basis word embeddings, node embeddings, simple and relational retrofitted embeddings are used to train and test ML models (⑨).

3.1 Datasets

The deployment used for the demonstration contains several preloaded database schemas and target word embeddings. Additionally, it provides several baseline embeddings.
Relational datasets: We prepared three datasets and imported them into PostgreSQL: The Movie Database (TMDB)[2], Google Play Store Apps[3] and Open Food Facts [4] which are all very popular datasets on Kaggle with a significant portion of textual data.
Target Word Embeddings: We used the popular 300 dimensional Google News embeddings[5] and an English fastText[6] model as target word embeddings for our retrofitting process.
Baseline Embeddings: We applied the original retrofitting approach [4] using the proposed standard parameter configuration of $\alpha_i = 1$ and the reciprocal of the outdegree of i for β_i within 20 iterations. DeepWalk (DW) is trained with its standard parameters and a 300 dimensions representation size.

3.2 ML Tasks

To demonstrate the usefulness of the relational retrofitting, the user can perform different *extrinsic* evaluation tasks, such as binary classification, missing value imputation and link prediction, on the embeddings generated with the relational retrofitting approach as well as the various baseline embeddings.
RetroLive comes with a set of pre-defined ML models:
Binary Classification: Text values can be assigned to two distinct classes by a classifier, e.g. we defined the task to distinguish

[2]https://www.kaggle.com/rounakbanik/the-movies-dataset
[3]https://www.kaggle.com/lava18/google-play-store-apps/
[4]https://www.kaggle.com/openfoodfacts/world-food-facts
[5]https://code.google.com/archive/p/word2vec/
[6]https://github.com/facebookresearch/fastText/blob/master/docs/pretrained-vectors.md

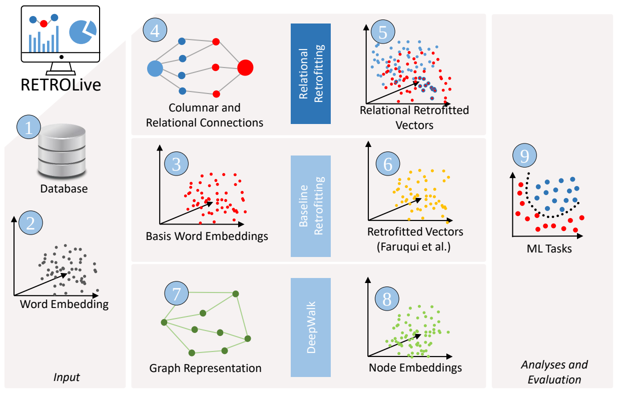


Figure 2: System Overview

US-American and non-US-American directors in TMDB.
Regression: Regression tasks assign input data points to a related numerical value, e.g. the budget or revenue in US dollar for a movie title.
Null Value Imputation: Text values are assigned to one category out of a finite set of category values, e.g. the "original language" of a movie or the app category for apps in the Google Play Store Apps dataset.
Link Prediction: The link prediction problem is typically defined on graphs where the goal is to predict links that are missing or likely to be created in the future. In our case, we consider the link prediction task for a specific relation to predict missing foreign key relations.
Our models consist of different multi-layer dense neural networks that can be applied on the generated node embeddings, pre-trained word embeddings, and our retrofitted embeddings. Details on the specific neural network architectures for all ML models are given in [5]. All experiments are repeated multiple times to obtain the distribution of the accuracy values.

4 DEMONSTRATOR WALKTHROUGH

Users access RetroLive through an interactive web interface shown in Figure 3 where they can configure and explore the whole relational retrofitting process. There are six main views:
Config and Retrofit: Those views allow to choose the database (three are pre-defined) and the target word embedding and configure the retrofitting process by setting the hyperparameters (Ⓐ). If the retrofitting is done on the same data used for the ML Task (e.g. to predict the genres of movies expressed in the database by foreign key relations), users can blacklist specific relations for the retrofitting in the config view. Further, the relational retrofitting process can be triggered and monitored (Ⓑ).
Results: In this view the user can inspect the extracted relational information (see Section 2.2) from the input schema in form of graph (Ⓒ). Additionally, the embedding statistics for each text column are presented (Ⓓ).
Analysis: In the analysis view, the users can inspect the characteristic of the retrofitted embeddings and compare them with the plain input word embedding. An interactive histogram (Ⓔ) shows the distribution of the cosine similarity between the plain word vectors and the retrofitted vectors, i.e. to which degree certain vectors have been changed during the retrofitting process. The user can click on the individual bins to see additional information, e.g. in how many relations certain terms have been

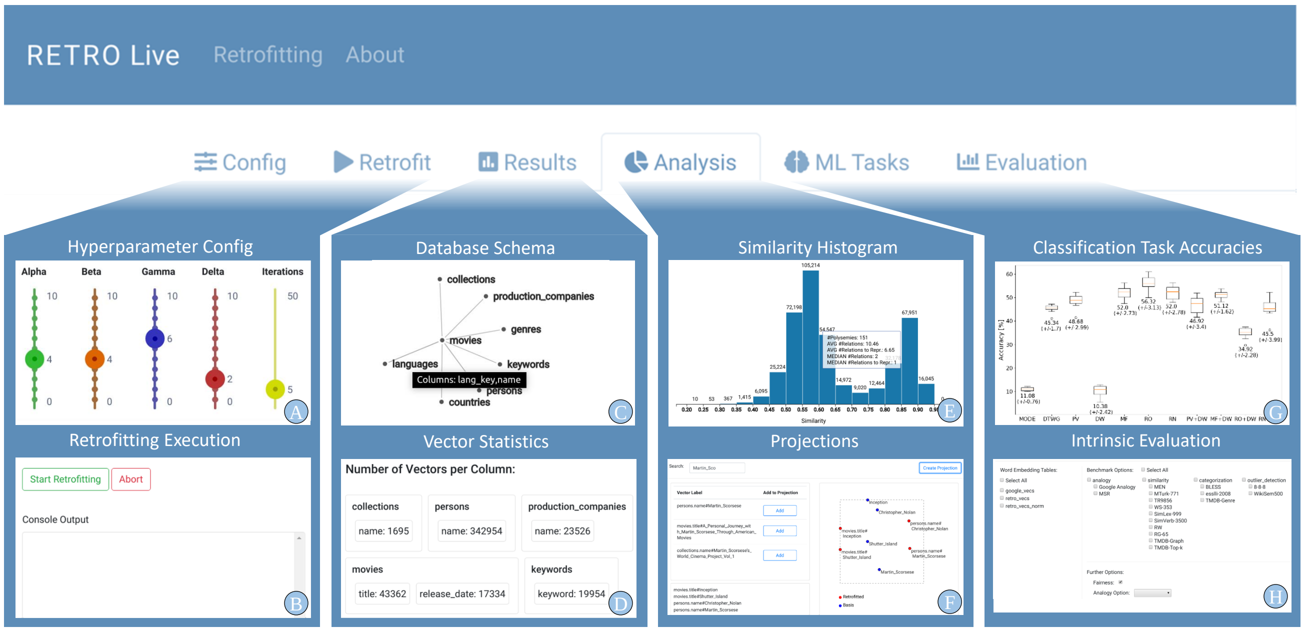


Figure 3: Demo User Interface

involved or in how many different columns a term appeared or to get a complete list of embeddings in the selected bin. Moreover, a 2-dimensional projection (PCA) shows the user-selected plain text and retrofitted vectors (F). In context of the TMDB database example, it can be seen that the vectors for movies and directors are arbitrarily distributed in the word embedding (red). However, after applying relational retrofitting the movie and director vectors (blue) are clustered and the difference vectors between movie and director pairs are of same length and orientation.
ML Tasks: To show the benefits of relation retrofitting the users can run different ML tasks (Section 3.2). The users select the embedding model (retrofitted, node, plain, etc.) they want to use for the given task. Training and testing data is retrieved from the database using pre-defined SQL queries which can be also modified by the user. Diagrams visualize the results (G).
Evaluation: Our demonstrator includes 14 intrinsic evaluation tasks to test word similarity, e.g. SimLex999 [6] MEN [3] or analogies, e.g. Google Analogy [7] (H). Here, the users can investigate whether original retrofitting and relational retrofitting affect the intrinsic task performance compared to plain word embeddings.

5 CONCLUSION

In this paper, we presented RetroLive, a novel system which allows generating retrofitted embeddings for an arbitrary database in a fully automated fashion. Participants can experience first-hand the power of relational retrofitting by tuning different hyperparameters, playing with different input datasets and investigating the effect of using a rich set of visualizations. Our demonstration highlights the potential of relational retrofitted vectors which achieve very good results for machine learning tasks like regression, binary, and multi-class classification. RetroLive also implements state-of-the-art baselines such as DeepWalk and the approach of Faruqui et al. which are both out-performed. Our system is publicly available under the permissive MIT license[7].

[7] https://github.com/guenthermi/postgres-retrofit

ACKNOWLEDGMENTS

This work is funded by the German Research Foundation (DFG) within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907) and by Intel®AI Research.

REFERENCES

[1] Abdulaziz Alghunaim, Mitra Mohtarami, Scott Cyphers, and Jim Glass. 2015. A Vector Space Approach for Aspect Based Sentiment Analysis. In *Proceedings of the 1st Workshop on Vector Space Modeling for Natural Language Processing*. 116–122.
[2] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.
[3] Elia Bruni, Nam Khanh Tran, and Marco Baroni. 2014. Multimodal Distributional Semantics. *J. Artif. Int. Res.* 49, 1 (Jan. 2014), 1–47.
[4] Manaal Faruqui, Jesse Dodge, Sujay Kumar Jauhar, Chris Dyer, Eduard Hovy, and Noah A Smith. 2015. Retrofitting Word Vectors to Semantic Lexicons. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 1606–1615.
[5] Michael Günther, Maik Thiele, and Wolfgang Lehner. 2019. RETRO: Relation Retrofitting For In-Database Machine Learning on Textual Data. arXiv:1911.12674
[6] Felix Hill, Kyunghyun Cho, Sébastien Jean, Coline Devin, and Yoshua Bengio. 2015. Embedding Word Similarity with Neural Machine Translation. In *ICLR 2015*.
[7] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781
[8] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3111–3119.
[9] 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.
[10] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 701–710.
[11] Xinjie Zhou, Xiaojun Wan, and Jianguo Xiao. 2015. Representation Learning for Aspect Category Detection in Online Reviews. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

RulER: Scaling Up Record-level Matching Rules

Luca Gagliardelli
Università degli Studi
di Modena e Reggio Emilia
Italy
luca.gagliardelli@unimore.it

Giovanni Simonini
Università degli Studi
di Modena e Reggio Emilia
Italy
simonini@unimore.it

Sonia Bergamaschi
Università degli Studi
di Modena e Reggio Emilia
Italy
sonia.bergamaschi@unimore.it

ABSTRACT

Record-level matching rules are chains of similarity join predicates on multiple attributes employed to join records that refer to the same real-world object when an explicit foreign key is not available on the data sets at hand. They are widely employed from data scientists and practitioners that work with data lakes, open data, and data in the wild.

We present RulER, the first tool that allows to efficiently execute record-level matching rules on parallel and distributed systems—we developed that on top of Apache Spark to leverage on its vast ecosystem of libraries and wide adoption. In this demo, we show how RulER can be easily employed for data preparation tasks (i.e., to join data sets to be consumed by data analytic tasks) and to support the user in debugging record-level matching rules. Finally, we demonstrate how our execution strategy of the record-level matching rules—introduced by RulER—is up to 3 times faster than the naïve concatenation of similarity join predicates.

1 INTRODUCTION

Combining data sets that bare information about the same real-world objects is an everyday task for practitioners that work with structured and semi-structured data. Frequently (e.g., when dealing with data lakes or when integrating open data with proprietary data) data sets do not have explicit keys that can be used for a traditional *equi-join* [4, 8, 9]. When that happens, a common solution is to perform a *similarity join* [6], i.e., to join records that have an attribute value similar above a certain threshold, according to a given similarity measure, as in the following example:

Example 1.1 (Similarity Join). *Given two product data sets, join all the record pairs with the Jaccard similarity of the product names above 0.8.*

A plethora of algorithms have been proposed in the last decades to efficiently execute the similarity join considering a single attribute, i.e., *attribute-level matching rules* (see [6] for a survey). At their core, all these algorithms try to prune the candidate pairs of records, on the basis of a single-attribute predicate—to alleviate the quadratic complexity of the problem.

Interestingly, only a few works had been focused on studying how to execute *record-level matching rules*, i.e., the combination of multiple similarity join predicates on multiple attributes (see section 2.1.1). Yet, this kind of rules permits to specify more flexible rules to match records, as in the following example:

Example 1.2 (Record-level matching rule). *Given two product data sets, join all the record pairs that have a Jaccard similarity of the product names above 0.8, or that have a Jaccard similarity of the description that is above 0.6 and the edit distance of the manufacturer lower than 3.*

Furthermore, record-level matching rules can be used to represent *decision trees* [1], hence learned with machine learning algorithms when training data is available. As a matter of fact, a decision tree for binary classification (i.e., classification of *matching/not-matching* records) can be naturally represented with DNF (disjunctive normal form) predicates—the same consideration can be done for a forest of trees.

To the best of our knowledge, no techniques have been proposed to leverage on distributed and parallel computing for scaling record-level matching rules. The benefit is twofold: *(i)* distributed computation allows to scale to large data sets that cannot be handled with a single machine; *(ii)* parallel execution reduces the execution times (3 time faster in our experiments). As a matter of fact, being able to efficiently execute similarity join is crucial when time is a critical component, e.g., when users are involved in the process. For instance, in exploratory search in a data lake [7], users typically look for related data sets and low latency in performing similarity join is required for enabling the user's interactive exploration. Also, when debugging record-level matching rules, users typically try different configurations of similarity metrics, thresholds, and attributes. Hence, enabling fast execution of such rules can significantly save user' time.

Contribution. We present RulER, a tool that enables users to efficiently execute record-level matching rules to join large data sets on distributed parallel systems. In particular, we implemented RulER on top of Apache Spark[1], to leverage on its vast ecosystem of libraries and tools for data preparation, and machine learning. In this demonstration, we will showcase RulER on several real-world data sets; attendees will write their own matching rules through Jupyter notebooks, and explore and analyze the results. We will demonstrate how to employ RulER both in a data preparation pipeline and to debug record-level matching rules. We implemented state-of-the-art similarity join algorithms for Spark that can be employed to build chains of similarity join predicates (i.e., to mimic record-level matching rules). Attendees will run such similarity join chains and verify that RulER is actually significantly faster (up to an order of magnitude) and more convenient to program such rules thanks to its APIs.

2 TOOL ARCHITECTURE

2.1 Preliminaries

2.1.1 Record-level matching rules. In RulER, matching rules are written in Disjunctive Normal Form (DNF), i.e., as a *disjunction* (logical *OR*) of *conjunctions* (logical *AND*) of similarity join predicates on multiple attribute (i.e., at the *record level*). This design choice is driven by the fact that DNF matching rules are easy to read and thus to debug, in practice. Moreover, DNFs can be employed to represent the trained model of a decision tree (or

[1]https://spark.apache.org/

of a random forest), hence suitable for exploiting labelled data. In this demonstration, we focus on how to scale DNF matching rules and we do not investigate how to generate *good* DNFs (i.e., decision trees/random forests) starting from training data.

To the best of our knowledge, the only related work tackling this problem is [1], which focuses on single-node execution, borrowing optimization techniques from the traditional relational database approaches. Similarly, [5] focuses on how to optimize multi-attribute similarity join, but only for conjunctions of predicates (i.e., not for DNF).

2.1.2 Similarity join with prefix index. The naïve solution for similarity join (i.e., each predicate of a DNF) is to enumerate and compare every pair of records, i.e., highly inefficient and not feasible on large data sets. To reduce the task complexity, different approaches were proposed in literature [2, 10–12]. All these approaches adopt a filter-verification pattern: *(i)* first an index is used to obtain a set of pre-candidates (e.g., prefix filter); *(ii)* the pre-candidates are filtered using a set of filters that are fast to apply; *(iii)* the resulting candidate pairs are probed with the similarity function to generate the final result.

The most efficient technique to obtain the pre-candidates efficiently is the *prefix filter* [2], which works as follows. Given a set of strings (the values of an attribute in a table, for instance), a pre-processing function is applied to each string to obtain a set of elements (e.g., tokens or n-grams, etc.). These elements are then sorted according to a global order, usually by their not decreasing document frequency of the elements (i.e., $1/\#(strings\ containing\ that\ element)$)—typically infrequent elements yield fewer candidate pairs [2]. Then, for each sorted set of elements only the first π are considered, i.e., the *prefixes*. A pair of element $\langle r_i, r_j \rangle$ can be safely pruned if their prefixes have no common elements. The prefix size depends on the adopted similarity function and threshold. For example, the prefix filter for the overlap similarity is defined as follows: given two sets, r_i and r_j, and an overlap threshold t; if $|r_i \cap r_j| \geq t$, then there is at least one common element within the π_{r_i}-prefix of r_i and the π_{r_j}-prefix of r_j, where $r = |r_j| - t + 1$ and $s = |r_j| - t + 1$.

An example of prefix filtering is reported in Figure 1. The prefixes, assuming an overlap threshold $t = 4$ are highlighted in grey. Since the two prefixes do not share any element, the pair $\langle r_i, r_j \rangle$ can be pruned. The intuition behind this is that the 3 remaining elements to check can provide at most a similarity of 3, that is not enough to reach the requested threshold t.

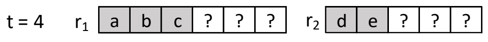


Figure 1: Prefix filtering example: The pair $\langle r_i, r_j \rangle$ can be pruned since the prefixes (in grey) have no common elements. The elements to check can provide at most an overlap similarity of 3 (or a Jaccard similarity of $3/8$).

The prefix filter can be adapted to work with many similarity measures like Jaccard, Dice, Cosine, Overlap [11] and Edit Distance [10], and it is employed by best performing similarity join algorithms [6].

The state-of-the-art distributed and parallel similarity join algorithms [3] partition the candidate pair of records according to the entry in the prefix index, i.e., for each element in the index, all the corresponding pairs of candidates are assigned to a computational node—more optimizations can be performed, but at their core, this is how parallelization is achieved by existing

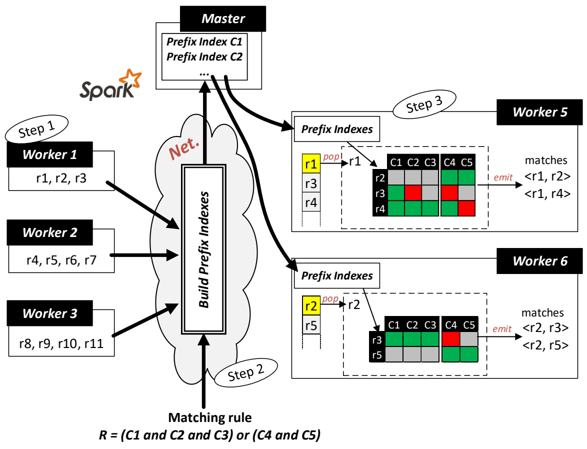


Figure 2: RulER execution model: green cells represents executed and passed rules; red cells executed that do not pass the rules; grey cells not executed rules.

algorithms. So, if two different similarity join conditions are considered (e.g., two different similarity measures on two different attributes), existing algorithms would create two different prefix indexes and generate two completely different parallelization strategies. Thus, given a fixed number of computational nodes, by employing existing algorithms, the only way to get the complete result of multiple similarity joins (i.e., the predicates of the matching rule) is to perform the joins in series and then combine the result sets.

2.2 The RulER execution model

The main intuition of RulER is to exploit the prefix indexes—one prefix index for each predicate of the matching rule—to build a graph structure, which is then employed to iterate over the records (the nodes of the graph), efficiently applying the rules and keep only the candidates (the edges of the graph) that pass the whole rule. In other words, RulER adopts a record-based parallelization approach; in contrast to the existing algorithms, which adopt a prefix-based parallelization approach on a single predicate at a time.

Example 2.1 (Distributed and parallel matching rule with RulER). *Given a matching rule $R = (C1 \wedge C2 \wedge C3) \vee (C4 \wedge C5)$, in which each Cx is a similarity join predicate (e.g., Jaccard Similarity title ≥ 0.8). An example of how RulER executes R is outlined in Figure 2. First, a prefix index is built on the basis of the record-level matching rules expressed in the main matching rule R. Then, the index is distributed to each worker. Each worker iterates over each record in its partition extracting the possible candidates from the prefix index. The rules are applied to each candidate. If more rules are in or it is possible to avoid computing the other rules when one of them is verified, e.g., with r1-r2 the rule $(C1 \wedge C2 \wedge C3)$ is not verified since the pair passes the rule $(C4 \wedge C5)$. Otherwise, if more rules are in and, it is possible to avoid the computation when one of them fails, for example for the pair r1-r3 C2 fails, so C3 has not to be computed.*

2.2.1 The algorithm. The RulER matching rule algorithm is outlined in Algorithm 1. The presented algorithm is the self-join version for sake of the presentation; adapting it for joining two different data sets is straightforward. The algorithm takes as input a collection of records and a record-level matching rule $\mathcal{M}$ and gives as output the set of record pairs that satisfy $\mathcal{M}$. Recall that $\mathcal{M}$ is in DNF, i.e., it is composed of sets of predicates P_j in *logical or*, each set P_j contains predicates p_k in *logical and*. First of all, the values of attributes are converted into sets of elements (Line 1) according to the matching rule requirements (e.g., n-grams, trigrams, tokens, etc.); then the prefix indexes are built to find the candidate pairs (line 2)—one prefix index is needed for each predicate p_k of the matching rule. The prefix indexes are sent in broadcast to each node (line 3) to be available to each computational node (called *worker*). Then, each worker iterates over its portion of records (lines 5-6), and performs the following operations for each record r_i. First, a set of candidates for r_i is initialized as an empty set C_{r_i} (line 7). Second, for each set P_j, a set of candidates C_{P_j} is initialized as an empty set (lines 8-9) and for each $p_k \in P_j$ the candidates C_{r_i,p_k} that can match with r_i are extracted using the prefix indexes (lines 10-11). Third, the candidates C_{r_i,p_k} are pruned by removing those that already passed one of the previous P_j set of predicates (line 14), and those that did not passed previous $p_k \in P_j$ predicates (lines 15-16). Fourth, the retained candidates are probed with other filters that further improve the efficiency of the overall process (e.g., length filter, position filter, etc. [10, 11]) according to the rule (line 18). Since p_k is in *logical and* with the previous predicates, only the candidates that pass the filters are kept. Finally, the resulting candidates from P_j are added to C_{r_i} (line 20).

2.2.2 Difference operator. RulER implements a method to perform the difference between the result pairs generated by two matching rules, i.e., given two matching rules $\mathcal{M}_1$, $\mathcal{M}_2$ it efficiently performs the difference of the result sets $res(\mathcal{M}_1) \setminus res(\mathcal{M}_2)$. Since the algorithm works at record level it is possible to perform a fine control on the application of the rules: when a record r_i is processed first $\mathcal{M}_1$ is checked, so to get $res(\mathcal{M}_1, r_i)$; then, $\mathcal{M}_2$ is applied only on the records $r_j \in res(\mathcal{M}_1, r_i)$, avoiding to compute it on the whole pairs again, retaining only the records r_j that do not satisfy $\mathcal{M}_2$ (i.e., $res(\mathcal{M}_1) \setminus res(\mathcal{M}_2)$).

Algorithm 1 RulER core

```
Input: $R$ collection of records to join
Input: $\mathcal{M}$ matching rule in DNF
Output: $C$, the pairs of records that can satisfy $\mathcal{M}$
1: $R_T \leftarrow getElements(R, \mathcal{M})$
2: $I \leftarrow buildPrefixIndexes(R_T, \mathcal{M})$
3: broadcast($I$)
4: $C \leftarrow \{\}$ //Candidate pairs
5: foreach partition $part \in R_T$
6:   for each $r_i \in part$ do
7:     $C_{r_i} \leftarrow \{\}$ //Candidates for $r_i$
8:     for each $P_j \in \mathcal{M}$ do //For each set of predicates in logical or
9:       $C_{P_j} \leftarrow \{\}$ //Candidates that satisfy $P_j$
10:      for each $p_k \in P_j$ do //For each predicate in logical and
11:        $C_{r_i,p_k} \leftarrow I(p_k, r_i)$ //Gets the candidates from the prefix index
12:        /*Removes candidates that already passed previous predicates in or
13:        and those that did not pass previous predicates in and*/
14:        $C_{r_i,p_k} \leftarrow C_{r_i,p_k} - C_{r_i}$
15:        if $C_{P_j} \neq \emptyset$ then
16:          $C_{r_i,p_k} \leftarrow C_{r_i,p_k} \cap C_{P_j}$
17:        /*Applies filters (length, positional, ...)*/
18:        $C_{P_j} \leftarrow applyFilters(r_i, C_{r_i,p_k}, p_k)$
19:      $C_{r_i} \leftarrow C_{r_i} \cup C_{P_j}$
20:    $C.append(C_{r_i})$
```

3 DEMONSTRATION SCENARIO

During the demonstration, participants will try RulER[2] on several scenarios and data sets by means of Jupyter Notebooks[3]. Users will be guided in: defining custom matching rules for a practical data preparation task (Section 3.1); debugging a matching rule (Section 3.2); and compare the efficiency of RulER w.r.t. existing state-of-the-art similarity joins (Section 3.3). Multiple scenarios and data sets (e.g., products, movies, books, finance, etc.) on which it is possible to try our RulER are available, but for sake of the presentation we describe only some of them in the following.

3.1 Data preparation scenario

In this scenario we use two movies data sets `Rotten Tomatoes`[4] and `Roger Erbert`[5]. The former gather users' ratings about movies, the latter critics' ratings. There is no foreign key between the two data sets. We ask the attendee to discover if there is a correlation between the ratings given by the users with the ones given by the critics. To do that, we need to define a matching rule to integrate the two data sets, then we compute the Pearson correlation on the obtained results. We ask to the attendee to write a record-level matching rule, for instance: $(\text{"}movie\_name\text{"}, \text{"}Title\text{"}, JS, 0.8) \wedge (\text{"}actors\text{"}, \text{"}Cast\text{"}, JS, 0.5)$, That means that the JS between the names of the movies must be greater or equal than 0.8 and the JS between the actors of the movies must be greater or equal than 0.5. After the matching, it is possible to obtain a scatter plot of the ratings, and compute the Pearsons correlation rating given by critics and the rating given by users on the same movie.

Figure 3 shows how simple is to use RulER. The user has just to: include RulER library (line 1), load the data as Spark Dataframe (lines 3-4), define the rules (lines 6-7) and combine them to obtain the final matching rule (line 9). Finally, the matching rule can be used to join the two dataframes with the *joinWithRules* method (line 11).

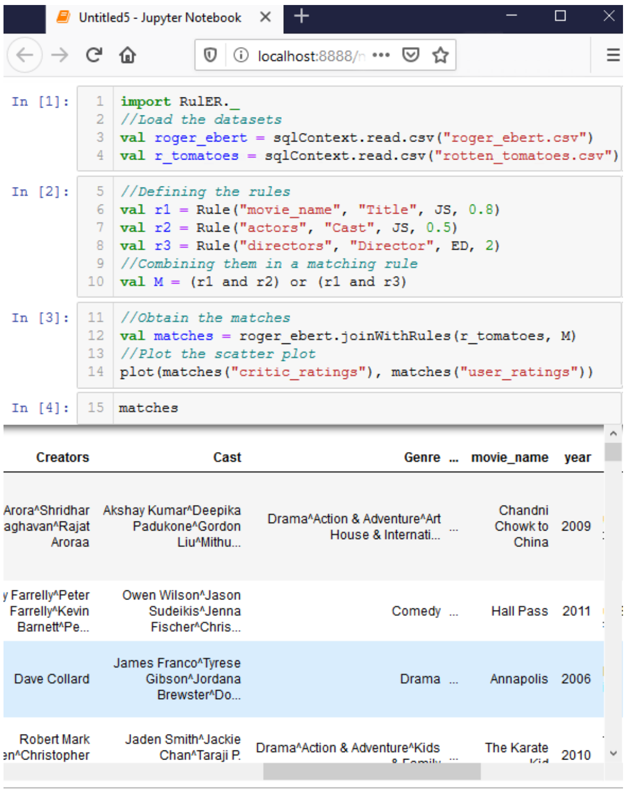
3.2 Matching rule debugging scenario

In this scenario we show how RulER can be used to efficiently debug different matching rules by using the difference operator. In particular, given two matching rules $\mathcal{M}_1$, $\mathcal{M}_2$ we will show how to use RulER to find the matches provided by the first rule that are not present in the matches provided by the second one, i.e. how to perform the difference between the two result sets $res(\mathcal{M}_1) \setminus res(\mathcal{M}_2)$. An example is shown in Figure 4. First, two matching rules are defined: in the first one the records are aligned by using the *title* and the *director* of the movies, while in the latter by using the *title* and the *cast*. Then, a new debugging rule is created using the difference operator defined in RulER. Finally, the debugging rule is applied to the dataset, obtaining the matches generated by $m1$ that are not generated by $m2$.

3.3 RulER efficiency demonstration

In this scenario we connect to a Spark cluster and use a subset of the IMDB data set[6] that contains records about movies, providing different fields that can be used to generate matching rules (e.g. movie title, cast, director, plot, etc.).

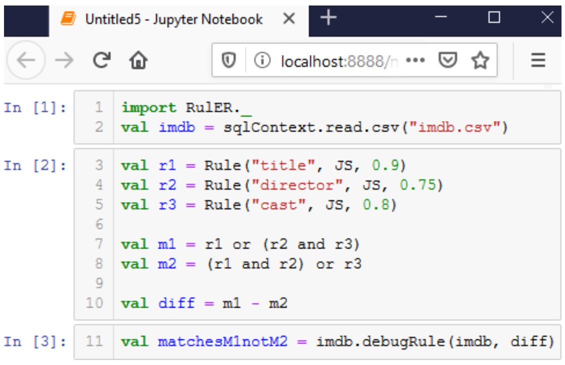
[2] We implemented RulER in Scala and made it open source: https://github.com/Gaglia88/ruler
[3] https://jupyter.org
[4] https://www.rottentomatoes.com
[5] https://www.rogerebert.com
[6] https://www.imdb.com



```
In [1]:  import RulER._
         //Load the datasets
         val roger_ebert = sqlContext.read.csv("roger_ebert.csv")
         val r_tomatoes = sqlContext.read.csv("rotten_tomatoes.csv")

In [2]:  //Defining the rules
         val r1 = Rule("movie_name", "Title", JS, 0.8)
         val r2 = Rule("actors", "Cast", JS, 0.5)
         val r3 = Rule("directors", "Director", ED, 2)
         //Combining them in a matching rule
         val M = (r1 and r2) or (r1 and r3)

In [3]:  //Obtain the matches
         val matches = roger_ebert.joinWithRules(r_tomatoes, M)
         //Plot the scatter plot
         plot(matches("critic_ratings"), matches("user_ratings"))

In [4]:  matches
```

Creators	Cast	Genre	...	movie_name	year
Arora^Shridhar aghavan^Rajat Aroraa	Akshay Kumar^Deepika Padukone^Gordon Liu^Mithu...	Drama^Action & Adventure^Art House & Internati...	...	Chandni Chowk to China	2009
y Farrelly^Peter Farrelly^Kevin Barnett^Pe...	Owen Wilson^Jason Sudeikis^Jenna Fischer^Chris...	Comedy	...	Hall Pass	2011
Dave Collard	James Franco^Tyrese Gibson^Jordana Brewster^Do...	Drama	...	Annapolis	2006
Robert Mark en^Christopher	Jaden Smith^Jackie Chan^Taraji P.	Drama^Action & Adventure^Kids	...	The Karate	2010

Figure 3: RulER usage example.



```
In [1]:  import RulER._
         val imdb = sqlContext.read.csv("imdb.csv")

In [2]:  val r1 = Rule("title", JS, 0.9)
         val r2 = Rule("director", JS, 0.75)
         val r3 = Rule("cast", JS, 0.8)

         val m1 = r1 or (r2 and r3)
         val m2 = (r1 and r2) or r3

         val diff = m1 - m2

In [3]:  val matchesM1notM2 = imdb.debugRule(imdb, diff)
```

Figure 4: Rule's debugging example.

We show how difficult is to write a complex rule by using existing similarity join algorithms w.r.t. the use of RulER. Moreover, RulER is much more faster. Figure 6 presents this scenario. To execute the rule as a similarity join chain, we use EDJoin [10] to perform a similarity join based on Edit Distance and PPJoin [11] to perform a similarity join based on Jaccard Similarity. The matching rule written by chaining similarity joins is expressed in lines 13-16. Note that, each rule provides a partial result and then the partial results of each rule have to be combined (line 16). If two rules are in *and* the partial results have to be intersected; if they are in *or* they have to be merged. To merge two candidate sets and avoid duplicates, a *distinct* operation has to be performed. The *distinct* and the *intersection* are very expensive operations in Spark because they require a *shuffling* since the same pairs have to be computed by the same worker.

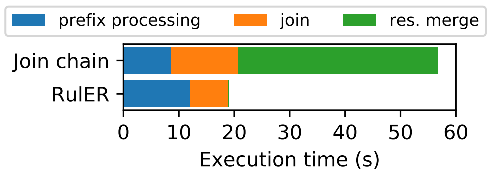
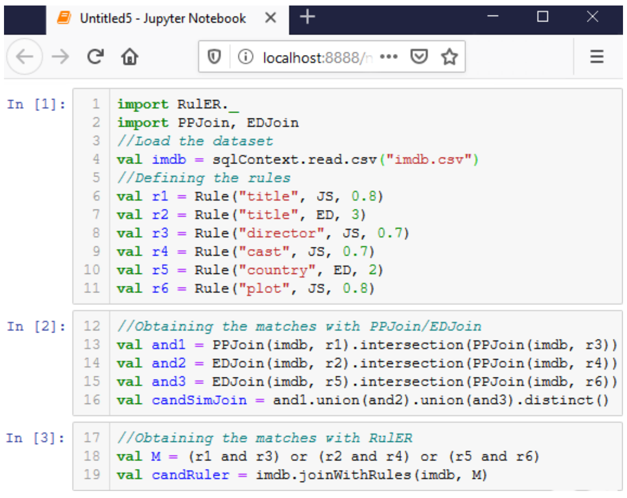


Figure 5: Execution time of RulER vs the execution time of the join chain in Figure 6.

Figure 5 shows the execution time of both the solutions. For the join chain, the prefix indexing and join times are computed as the sum of each join. The RulER indexing time is higher due to the time requested to broadcast the index, but the join time is faster. Moreover, RulER does not need to merge the partial results, that is the costly task of the join chain, which makes it highly inefficient.



```
In [1]:   1  import RulER._
          2  import PPJoin, EDJoin
          3  //Load the dataset
          4  val imdb = sqlContext.read.csv("imdb.csv")
          5  //Defining the rules
          6  val r1 = Rule("title", JS, 0.8)
          7  val r2 = Rule("title", ED, 3)
          8  val r3 = Rule("director", JS, 0.7)
          9  val r4 = Rule("cast", JS, 0.7)
         10  val r5 = Rule("country", ED, 2)
         11  val r6 = Rule("plot", JS, 0.8)

In [2]:  12  //Obtaining the matches with PPJoin/EDJoin
         13  val and1 = PPJoin(imdb, r1).intersection(PPJoin(imdb, r3))
         14  val and2 = EDJoin(imdb, r2).intersection(PPJoin(imdb, r4))
         15  val and3 = EDJoin(imdb, r5).intersection(PPJoin(imdb, r6))
         16  val candSimJoin = and1.union(and2).union(and3).distinct()

In [3]:  17  //Obtaining the matches with RulER
         18  val M = (r1 and r3) or (r2 and r4) or (r5 and r6)
         19  val candRuler = imdb.joinWithRules(imdb, M)
```

Figure 6: Join chain vs RulER execution.

REFERENCES

[1] Adel Ardalan, AnHai Doan, Aditya Akella, et al. 2018. Smurf: self-service string matching using random forests. *PVLDB* 12, 3 (2018), 278–291.
[2] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A primitive operator for similarity joins in data cleaning. In *ICDE*. 5–5.
[3] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. 2018. Set similarity joins on mapreduce: An experimental survey. *PVLDB* 11, 10 (2018), 1110–1122.
[4] Luca Gagliardelli, Giovanni Simonini, Domenico Beneventano, and Sonia Bergamaschi. 2019. SparkER: Scaling Entity Resolution in Spark. In *EDBT 2019*. 602–605.
[5] Guoliang Li, Jian He, Dong Deng, and Jian Li. 2015. Efficient similarity join and search on multi-attribute data. In *SIGMOD*. 1137–1151.
[6] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An empirical evaluation of set similarity join techniques. *PVLDB* 9, 9 (2016), 636–647.
[7] Fatemeh Nargesian, Erkang Zhu, Renée J Miller, Ken Q Pu, and Patricia C Arocena. 2019. Data lake management: challenges and opportunities. *PVLDB* 12, 12 (2019), 1986–1989.
[8] Giovanni Simonini, Luca Gagliardelli, Sonia Bergamaschi, and H. V. Jagadish. 2019. Scaling entity resolution: A loosely schema-aware approach. *Inf. Syst.* 83 (2019), 145–165.
[9] Giovanni Simonini, George Papadakis, Themis Palpanas, and Sonia Bergamaschi. 2019. Schema-Agnostic Progressive Entity Resolution. *IEEE TKDE* 31, 6 (2019), 1208–1221.
[10] Chuan Xiao, Wei Wang, and Xuemin Lin. 2008. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB* 1, 1 (2008), 933–944.
[11] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient similarity joins for near-duplicate detection. *ACM TODS* 36, 3 (2011), 15.
[12] Song Zhu, Giuseppe Fiameni, Giovanni Simonini, and Sonia Bergamaschi. 2017. SOPJ: A Scalable Online Provenance Join for Data Integration. In *HPCS 2017*. 79–85.

Schema Mapping Generation in the Wild: A Demonstration with Open Government Data

Lacramioara Mazilu
University of Manchester, United Kingdom
lara.mazilu@manchester.ac.uk

Nikolaos Konstantinou
University of Manchester, United Kingdom
nikolaos.konstantinou@manchester.ac.uk

Norman W. Paton
University of Manchester, United Kingdom
norman.paton@manchester.ac.uk

Alvaro A.A. Fernandes
University of Manchester, United Kingdom
alvaro.a.fernandes@manchester.ac.uk

ABSTRACT

Schema mapping generation identifies how data sets can be combined to create views that are relevant to an application. Where the data sets to be combined lack declared relationships, such as foreign keys, schema mapping generation can be considered to be *in the wild*. In this paper, we describe an approach to schema mapping generation in the context of open government data, in particular, the London Datastore. Mapping generation is informed by inferred profiling data about the data sets and their relationships, where the data sets are made available as csv files. We outline the mapping generation algorithm, and describe a demonstration of the approach, in which the user can: *(i)* specify the target to be populated by the generated mappings over a collection of sources from The London Datastore; *(ii)* browse the generated candidate mappings and the evidence that informed their creation; and *(iii)* steer the mapping generation process, to make use of preferred sources and dependable profiling results.

1 INTRODUCTION

Given a collection of source datasets, some metadata about them, and a target schema, schema mapping generation produces a collection of views that provide ways of populating the target from the sources. Mapping generation is important because the data of relevance to an application or an analysis is often not immediately available in a single, suitable, integrated form.

Most work on mapping generation has assumed that the source and the target benefit from declared constraints, for example in the form of primary and foreign keys (e.g., as in the seminal work on Clio and its descendents, as reviewed in [5]). However, with the growing availability of open data sets, and the emergence of data lakes, mapping generation over independently produced data sets, with minimal explicit metadata, is arguably even more necessary than for well-defined schemas.

We refer to mapping generation over data sets without declared relationships as *in the wild*. Mapping generation must, among other things, take into account relationships between data sources, and, in this paper, we assume that candidate keys and (partial) inclusion dependencies have been inferred through data profiling [1]. Then, to deploy schema mapping generation in the wild, the following are required:

(1) *A way of exploring the space of candidate mappings.* We use a dynamic programming algorithm to identify promising mappings, referred to as *Dynamap* [8].
(2) *A way of displaying to the user these mappings, their properties, and the evidence on which they build.* As (1) builds on necessarily speculative profiling data, the results of mapping generation must be able to be reviewed by users, for example to ensure that joins are building on inclusion dependencies that represent valid real-world relationships.
(3) *A way to enable the user to steer the mapping generation process.* As (2) may identify issues with generated mappings, users must be able to steer the mapping generation process away from unsuitable decisions, for example by ruling out the use of certain inclusion dependencies.

To show (1) to (3) in practice, we demonstrate our mapping generation algorithm, and its associated user interface, in use with data from The London Datastore[1], which provides hundreds of data sets providing diverse information about London.

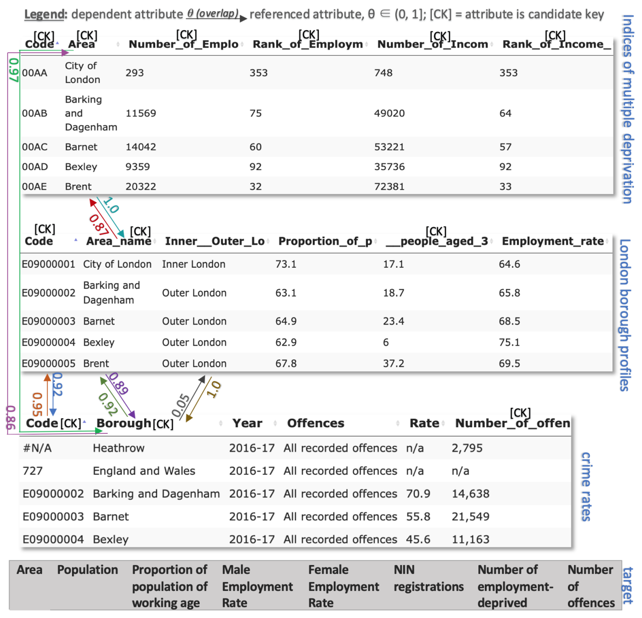
The remainder of the paper is structured as follows. Section 2 provides some details on The London Datastore. Our mapping generation approach is reviewed in Section 3. The demonstration in Section 4 shows an example of viewing a generated mapping and understanding it based on its properties and the evidence based on which it was created. The user can steer mapping generation based on the presented information. Section 5 concludes.

2 OPEN DATA CASE STUDY: THE LONDON DATASTORE

Open government data is published in a collection of national, regional, city or topic-based portals, with a view to increasing transparency and informing decision making [3]. The London Datastore is a representive example of a city data repository, providing data sets across a range of topic areas, including transport, employment, housing, health and education. These datasets come from a variety of publishers, including local and national government departments, and many of the data sets use consistent, generous licenses. The London Datastore supports both search and browse interfaces, and allows data sets to be downloaded in a variety of formats.

The demonstration uses comma-separated-value file data sets, released under the UK Open Government License[2]. Typically files contain from a few tens of rows (e.g., there are numerous data sets that have one row for each London Borough, of which there are *33*), to a few thousand rows (e.g., there are around 5000 rows in a data set of modelled household income estimates at a particular, rather fine, area granularity). There may be few (e.g., 2) to many columns in each table (e.g., there are hundreds of columns in a ward atlas table, describing different properties of an electoral ward).

[1]https://data.london.gov.uk
[2]http://www.nationalarchives.gov.uk/doc/open-government-licence/version/3/


Legend: dependent attribute θ (overlap) referenced attribute, θ ∈ (0, 1]; [CK] = attribute is candidate key
Indices of multiple deprivation
London borough profiles
crime rates
target
0.97
1.0
0.87
0.95
0.92
0.92
0.89
0.05
1.0
0.86

Code [CK]	Area [CK]	Number_of_Emplo [CK]	Rank_of_Employm [CK]	Number_of_Incom [CK]	Rank_of_Income_ [CK]
00AA	City of London	293	353	748	353
00AB	Barking and Dagenham	11569	75	49020	64
00AC	Barnet	14042	60	53221	57
00AD	Bexley	9359	92	35736	92
00AE	Brent	20322	32	72381	33

Code [CK]	Area_name [CK]	Inner__Outer_Lo	Proportion_of_p	__people_aged_3 [CK]	Employment_rate
E09000001	City of London	Inner London	73.1	17.1	64.6
E09000002	Barking and Dagenham	Outer London	63.1	18.7	65.8
E09000003	Barnet	Outer London	64.9	23.4	68.5
E09000004	Bexley	Outer London	62.9	6	75.1
E09000005	Brent	Outer London	67.8	37.2	69.5

Code [CK]	Borough [CK]	Year	Offences	Rate	Number_of_offen [CK]
#N/A	Heathrow	2016-17	All recorded offences	n/a	2,795
727	England and Wales	2016-17	All recorded offences	n/a	n/a
E09000002	Barking and Dagenham	2016-17	All recorded offences	70.9	14,638
E09000003	Barnet	2016-17	All recorded offences	55.8	21,549
E09000004	Bexley	2016-17	All recorded offences	45.6	11,163

Area	Population	Proportion of population of working age	Male Employment Rate	Female Employment Rate	NIN registrations	Number of employment-deprived	Number of offences

Figure 1: London datasets example

Figure 1 contains fragments (i.e., shows subsets of both tuples and attributes of the tables) from three tables in The London Datastore that are used in our running example. The tables in Figure 1 show population statistics for different London areas and (possibly their corresponding) crime rates[3].

3 MAPPING GENERATION IN DYNAMAP

In this section, we illustrate mapping generation in Dynamap with an example over The London Datastore.

The problem and some evidence. The *problem* is that of generating a set of mappings that populate a *target* table from a set of *source* tables. The target table definition provides only the names of the columns, and the source table definitions contain only column names and their associated values. Mapping generation techniques have been developed that use different types of evidence, including declared constraints [5], data examples [2] and feedback [4]. In Dynamap, the *evidence* used to inform mapping generation is *matches* between *source* and *target* attributes, and (partial) *inclusion dependencies*, both of which can be inferred using standard profiling algorithms [1]. In Figure 1, the *target* requires information about population and employment statistics per area (first 7 attributes), together with the corresponding number of offences in each area (last attribute). Also, Figure 1 shows the detected profile data on the represented sources: 10 inclusion dependencies (represented by arrows, where their direction is from the dependent to the referenced source) and 12 candidate keys (annotated with *[CK]*). The *overlap* (θ) is the fraction of the distinct values in the dependent attribute that are found in the referenced attribute. In the example, eight inclusion dependencies are partial ($\theta \in (0, 1)$) and two are full ($\theta = 1.0$).

Exploring the search space. Mapping generation is typically a search problem, either using generic (e.g., [6]) or bespoke (e.g., [5]) algorithms. In Dynamap, the space of candidate mappings is every way of combining the sources using *union* or *join* operations, where union compatibility is detected by comparing matches with the target and join operations are based on full or partial inclusion dependencies. We explore the search space using dynamic programming, but with pruning to ensure that only promising parts of the space are actually visited. As a result, mappings are constructed bottom-up, from pairs of tables, then from tables with intermediate mappings containing two tables, etc. For example, in Figure 1, a first step could be to merge *Indices of deprivation* with *London borough profiles* by joining on *Area* attributes, and then, their result intermediate mapping could be merged further with the *crime rates* table through join on either *Code* or *Area/Borough* attributes.

Annotating candidate mappings. As the search must combine pairs of intermediate mappings, we need to know the candidate keys of candidate mappings and their (partial) inclusion dependencies. These need to be derived during the search process, as it is not computationally viable to evaluate (i.e., materialize) every candidate mapping and run a profiler on it. The derived profile information is also used to compute a fitness, which represents the predicted fraction of complete rows. Formulas have been developed for propagating profiling data through unions and joins [8]. For example, in Figure 1, after joining *Indices of deprivation* with *London borough profiles* on *Area*, their corresponding inclusion dependencies and candidate keys are propagated such that it shows the relationship between the newly created intermediate mapping and the *crime rates* table. Based on this propagated profiling data, the mapping generation system can detect that the new mapping will produce *Area* and *Code* values that are overlapping with *Borough* and *Code* in *crime rates*, thus, it concludes that they can be joined on one of the overlapping attribute pairs.

4 DEMONSTRATION

Mapping generation in the wild will be demonstrated through hands-on experience generating and examining mappings over datasets from The London Datastore, in the context of the Data Preparer[4] data integration and cleaning platform, a descendent of the VADA Architecture [7]. Users interact with the system via a web interface; our goals are to demonstrate: *(i)* how data profiling can be used to provide information about the properties of the data sets; *(ii)* how this evidence can be used by Dynamap to combine sources in plausible ways; *(iii)* how users can review the candidate mappings, and the evidence on which they are based; and *(iv)* how users can steer the mapping generation process, for example by excluding schema elements or profiling data that are not relevant to the result.

Figure 2 shows an example that extends the one in Figure 1 by adding to it three other sources containing data about house and population density, national insurance registrations, and statistics on the population from the local authorities. Although the demonstration will support different ways of using the system, one approach would involve the following steps.

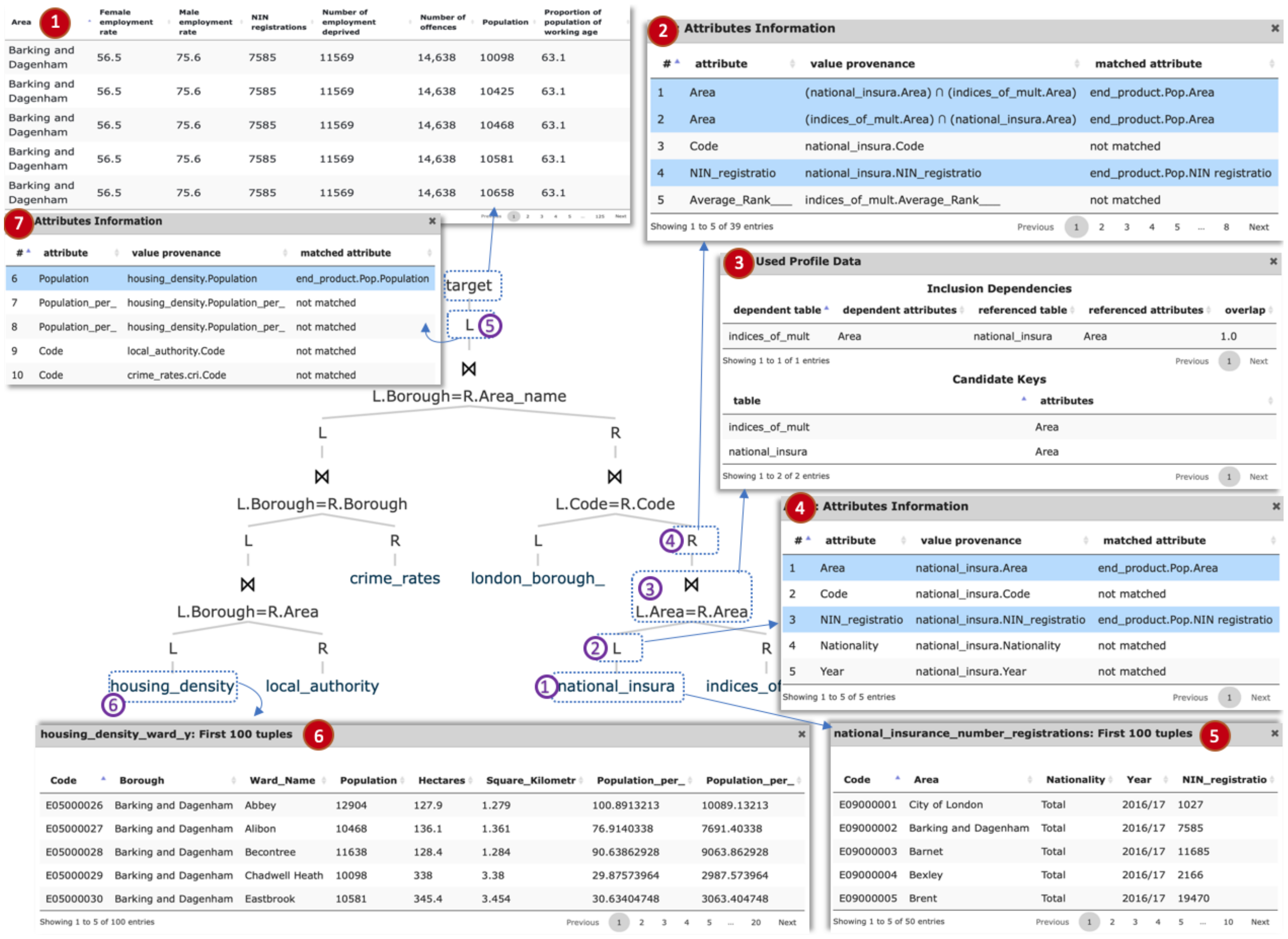
Browse the available sources. The interface supports search and browse over the sources that are input to the system.

Define a target. The user can interactively name a target table and its attributes, or edit an existing target to add/remove attributes. The target table used throughout is that from Figure 1.

Generate mappings and view the result. Given some source tables and a target, the user can ask the system to generate a result,

[3]Due to space limitations, throughout the rest of this paper, the figures contain attribute and table names capped at 15 characters.

[4]https://thedatavaluefactory.com



Area ❶	Female employment rate	Male employment rate	NIN registrations	Number of employment deprived	Number of offences	Population	Proportion of population of working age
Barking and Dagenham	56.5	75.6	7585	11569	14,638	10098	63.1
Barking and Dagenham	56.5	75.6	7585	11569	14,638	10425	63.1
Barking and Dagenham	56.5	75.6	7585	11569	14,638	10468	63.1
Barking and Dagenham	56.5	75.6	7585	11569	14,638	10581	63.1
Barking and Dagenham	56.5	75.6	7585	11569	14,638	10658	63.1

Previous 1 2 3 4 5 ... 125 Next

❷ Attributes Information

#	attribute	value provenance	matched attribute
1	Area	(national_insura.Area) ∩ (indices_of_mult.Area)	end_product.Pop.Area
2	Area	(indices_of_mult.Area) ∩ (national_insura.Area)	end_product.Pop.Area
3	Code	national_insura.Code	not matched
4	NIN_registratio	national_insura.NIN_registratio	end_product.Pop.NIN registratio
5	Average_Rank___	indices_of_mult.Average_Rank___	not matched

Showing 1 to 5 of 39 entries Previous 1 2 3 4 5 ... 8 Next

❼ Attributes Information

#	attribute	value provenance	matched attribute
6	Population	housing_density.Population	end_product.Pop.Population
7	Population_per_	housing_density.Population_per_	not matched
8	Population_per_	housing_density.Population_per_	not matched
9	Code	local_authority.Code	not matched
10	Code	crime_rates.cri.Code	not matched

❸ Used Profile Data

Inclusion Dependencies

dependent table	dependent attributes	referenced table	referenced attributes	overlap
indices_of_mult	Area	national_insura	Area	1.0

Showing 1 to 1 of 1 entries Previous 1 Next

Candidate Keys

table	attributes
indices_of_mult	Area
national_insura	Area

Showing 1 to 2 of 2 entries Previous 1 Next

❹ : Attributes Information

#	attribute	value provenance	matched attribute
1	Area	national_insura.Area	end_product.Pop.Area
2	Code	national_insura.Code	not matched
3	NIN_registratio	national_insura.NIN_registratio	end_product.Pop.NIN registratio
4	Nationality	national_insura.Nationality	not matched
5	Year	national_insura.Year	not matched

Showing 1 to 5 of 5 entries Previous 1 Next

housing_density_ward_y: First 100 tuples ❻

Code	Borough	Ward_Name	Population	Hectares	Square_Kilometr	Population_per_	Population_per_
E05000026	Barking and Dagenham	Abbey	12904	127.9	1.279	100.8913213	10089.13213
E05000027	Barking and Dagenham	Alibon	10468	136.1	1.361	76.9140338	7691.40338
E05000028	Barking and Dagenham	Becontree	11638	128.4	1.284	90.63862928	9063.862928
E05000029	Barking and Dagenham	Chadwell Heath	10098	338	3.38	29.87573964	2987.573964
E05000030	Barking and Dagenham	Eastbrook	10581	345.4	3.454	30.63404748	3063.404748

Showing 1 to 5 of 100 entries Previous 1 2 3 4 5 ... 20 Next

national_insurance_number_registrations: First 100 tuples ❺

Code	Area	Nationality	Year	NIN_registratio
E09000001	City of London	Total	2016/17	1027
E09000002	Barking and Dagenham	Total	2016/17	7585
E09000003	Barnet	Total	2016/17	11685
E09000004	Bexley	Total	2016/17	2166
E09000005	Brent	Total	2016/17	19470

Showing 1 to 5 of 50 entries Previous 1 2 3 4 5 ... 10 Next

Figure 2: Mapping explanation tree and associated information

which means running a matcher, a data profiler (for detecting candidate keys and partial/full inclusion dependencies), mapping generation and result display.

View a candidate mapping. From the result, the user can ask the system to explain how the generated mappings were produced. Each generated mapping has *(i)* a view that has the schema of the chosen target table and that shows how the target is populated once the mapping is executed (e.g., ❶ in Figure 2), and *(ii)* a *mapping explanation*. Each mapping is *explained* through the help of a tree view where the user can understand in which order the sources were merged and which operators were used to merge them. Figure 2 shows how six sources are merged through five joins by a mapping. The *root* of the tree is the chosen target table and the leaves of the tree are the input sources merged by the mapping. The intermediate nodes represent the incremental build-up of the explained mapping, starting with the merge of the leaves (the sources) and reaching the target when all sources were merged or when no other merge opportunities could be identified. As mentioned in Section 3, the mappings are constructed in a bottom-up fashion, and this is shown by the tree representation, where each intermediate mapping has two corresponding nodes in the tree: *(i)* an *operation* node (e.g., ③ in Figure 2), and *(ii)* a *L/R* node (e.g., ②, ④, ⑤). The *operation* node is used to show which was the chosen operation between the two operands, and (if present) the join condition. As explained in [8], the operations considered by Dynamap are union, (equi) join or full outer join; different combinations of profiling data lead to different operations, e.g., a candidate key whose attributes share a partial inclusion dependency can lead to a full outer join, while a full inclusion dependency can lead to a join. The *L/R* node shows whether the newly created intermediate mapping is the **l**eft or **r**ight operand in the next operation (if any). For example, in Figure 2, the *national insurance* table (①) is represented as the left operand (②) for the join operation represented in node ③.

Understand a mapping. For understanding mappings, the user can interact with the *mapping explanation tree* and drill down on the evidence that informed its creation. The interaction is done by clicking on the three types of nodes, i.e., *leaf*, *L/R*, and *operation* nodes, each revealing evidence for that merge step:
Leaf node. Clicking on a leaf node leads to a snippet of the initial source that it represents. This helps the user understand the dataset information and whether it was semantically correct to merge it with the other sources in the mapping. This helps to decide whether to keep or discard a source from the search. For instance, after clicking on *national insurance* (①) a snippet of the source appears (❺). It can be observed from it that the source contains data about national insurance registrations, which is relevant to the chosen target that requires employment data.
L/R node. Clicking on a *L/R* node shows information about each attribute of that intermediate mapping: *(i)* the target attribute

that it matches (if any), and *(ii)* the provenance of the distinct values in the attribute. The information about the value provenance is shown as set operations based on how the parent attributes were merged, e.g., the result of joined attributes contains the intersection of their values. This helps to understand if a match is inappropriate or if source attributes were incorrectly merged. In Figure 2, to understand how the *national insurance* source contributes to the target, the user needs to click on its corresponding *L* node (②), for which the information about the attributes appears (❹) so they learn that this source contributes with both *Area* and *NIN registration* values to the target (highlighted in blue in ❹). Node ④ is another example of *L/R* node that corresponds to the result of the join on *Area* between *national insurance* and *indices of deprivation* datasets (③). The attribute information for this intermediate mapping shows that its *Area* attribute contains the intersection of the values from *Area* attributes in *national insurance* and *indices of deprivation*, while the rest of the attributes are not merged.

Operation node. Clicking on an *operation* node shows the profiling data that was used to choose a certain merge operation. The information shown contains a table listing inclusion dependencies and another table for candidate keys. This helps decide whether the used profiling data is meaningful. In Figure 2, to understand how the *national insurance* dataset is merged with *indices of deprivation*, the user accesses the information about their merge by clicking on the corresponding *operation* node (③). This shows the profiling data that was used to decide their join on *Area* (❸), which comprises two candidate keys on *Area* attributes in both sources that share one full inclusion dependency.

Steer mapping generation. Understanding the generated mappings can be crucial in the selection of proper mappings to populate a target. Thus, a user can understand if the chosen sources should be merged or not, or if Dynamap was misled by any faulty matches or profiling data. After the user explores the mapping explanation and pins down any wrong decisions, they can alter through the interface the input that previously led to incorrect merges. The steering can be done through *(i)* adding/deleting source-to-target matches, e.g., remove those between a source attribute that represents a different concept than the one required by the target, *(ii)* removing misleading inclusion dependencies, e.g., remove those between semantically different source attributes that have common values, *(iii)* removing candidate keys on attributes which should not be keys, but were detected due to the (possibly) scarce data in the source, or *(iv)* removing unnecessary datasets from the search space. For example, in Figure 2, the merge opportunities that Dynamap finds seem correct as all joined attributes are suitable pairs in terms of their meaning and value overlap. However, the snippet of the target (❶) shows that many tuples have the same attribute values, for which only the *Population* values differ, indicating there might be an incorrect join with the source(s) that contributes to the *Population* attribute. By accessing the attribute information (❼) corresponding to the last merge in the mapping (⑤) it is shown that the value provenance of *Population* is from the *housing density* source. The next step is to analyze the data in the source by clicking on the corresponding leaf node (⑥) for which a snippet of its extent (❻) is shown. Analyzing the dataset fragment, it can be observed that the match is incorrect although the *Population* attribute in *housing density* has exactly the same name as the target, which makes it seem a correct match. The two attributes represent different types of population: in the source, the population information is per ward, not per area as required by the target. Thus, it can be concluded that the match is inaccurate. In such situations, it is not straightforward to differentiate between two (possibly close) semantic meanings, especially when the attributes contain numerical values. Learning about this incorrect match, the user steers the mapping generation process by discarding the *Population* match. After a new rerun of the mapping generation process with the updated input, the new resulting mappings produce sensible results in terms of aligned data per area. However, given that the *housing density* source does not contribute anything else to the target, it is automatically eliminated from the search space, thus, the new mappings do not involve it.

5 CONCLUSIONS

We have illustrated mapping generation in the wild with open data. Not only do we produce useful mappings, but also the system is transparent – the user can see what has been done, and steer the production of future mappings by altering and/or correcting the three types of input: inaccurate matches, misleading profiling data (candidate keys and inclusion dependencies), and unnecessary sources.

The problem of automated mapping generation might seem as an overly-engineered approach when the number of input sources is small and an expert user can hand-craft the mappings. However, the problem becomes increasingly complicated when tens or hundreds of sources are involved and they have a plethora of merge opportunities. Thus, automating mapping generation becomes an essential component for data integration. Moreover, given autonomous input sources, finding correct mappings completely automatically might become an unfeasible task. This gives rise to the problem of understanding complex mappings that involve numerous, autonomous sources. Helping the user understand the mappings leads to steering the mapping generation process by providing proper input, based on which the system can make correct decisions.

Acknowledgements. We are pleased to acknowledge the support of the UK Engineering and Physical Sciences Research Council, through the VADA Programme Grant (EP/M025268/1).

REFERENCES

[1] Z. Abedjan, L. Golab, F. Naumann, and T. Papenbrock. 2018. *Data Profiling*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00878ED1V01Y201810DTM052

[2] B. Alexe, B. ten Cate, P. G. Kolaitis, and W. C. Tan. 2011. Designing and refining schema mappings via data examples. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*. 133–144. https://doi.org/10.1145/1989323.1989338

[3] J. Attard, F. Orlandi, S. Scerri, and S. Auer. 2015. A systematic review of open government data initiatives. *Government Information Quarterly* 32, 4 (2015), 399–418. https://doi.org/10.1016/j.giq.2015.07.006

[4] A. Bonifati, R. Ciucanu, and S. Staworko. 2014. Interactive Inference of Join Queries. In *17th International Conference on Extending Database Technology, EDBT*. 451–462. https://doi.org/10.5441/002/edbt.2014.41

[5] R. Fagin, L. M. Haas, M. Hernandez, R. J. Miller, L. Popa, and Y. Velegrakis. 2009. Clio: Schema Mapping Creation and Data Exchange. In *Conceptual Modeling: Foundations and Applications*. LNCS, Vol. 5600. Springer Berlin Heidelberg, 198–236. https://doi.org/10.1007/978-3-642-02463-4_12

[6] G. H. L. Fletcher and C. M. Wyss. 2006. Data Mapping as Search. In *Advances in Database Technology - EDBT*. 95–111. https://doi.org/10.1007/11687238_9

[7] N. Konstantinou, E. Abel, L. Bellomarini, A. Bogatu, C. Civili, E. Irfanie, M. Koehler, L. Mazilu, E. Sallinger, A. A. A. Fernandes, G. Gottlob, J. A. Keane, and N. W. Paton. 2019. VADA: an architecture for end user informed data preparation. *J. Big Data* 6 (2019), 74. https://doi.org/10.1186/s40537-019-0237-9

[8] L. Mazilu, N. W. Paton, A. A. A. Fernandes, and M. Koehler. 2019. Dynamap: Schema Mapping Generation in the Wild. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management, SSDBM*. 37–48. https://doi.org/10.1145/3335783.3335785

EPIQUE: Extracting Meaningful Science Evolution Patterns from Large Document Archives

Ke Li
LIP6, CNRS, Sorbonne Université
Paris, France
ke.li@lip6.fr

Hubert Naacke
LIP6, CNRS, Sorbonne Université
Paris, France
hubert.naacke@lip6.fr

Bernd Amann
LIP6, CNRS, Sorbonne Université
Paris, France
bernd.amann@lip6.fr

ABSTRACT

There is an increasing demand from domain experts for tools that assist them to extract information about the scientific progress and technological innovations from bibliographic archives such as the Web of Science, arXiv, PubMed, etc. Topic evolution graphs track the evolution of science by identifying and analyzing science evolution patterns like the emergence and decay of research topics or the split of one research topic into several subtopics, etc. Building such topic evolution networks for extracting meaningful evolution patterns is still a difficult task requiring the tuning of several technical parameters. In our demonstration, we present our prototype implementation of a generic topic evolution model for representing and filtering evolution patterns extracted from very large document archives.

1 INTRODUCTION

Revealing meaningful evolution patterns from document archives has many applications and can be used to synthetize narratives from datasets accross multiple domains, including new stories, research papers, legal cases and works of literature [12]. The study of science evolution can help *philosophers and historians* of science [10] to test their theories with data, *researchers* to position their work in its scientific context, *policy makers* to foster innovation and get key indicators for decision-making processes, *industry* to evaluate the potential for innovation and technological transfer, *librarians* to classify scientific documents, etc.

Scientific evolution can broadly be studied by adopting a cognitive view or a social view on evolution dynamics. The cognitive view emphasizes the shared knowledge and the change of ideas (Kuhn's approach [10]), whereas the social view takes account of authorship and social interaction (e.g., citation graphs) [7, 13]. Bibliographic archives often include both kinds of information and there also exist methods which also combine both views to study science evolution [8]. In the interdisciplinary EPIQUE project[1] we adopt the cognitive view for modeling science evolution and assume that the evolution only depends on the content of the documents. Whereas this choice clearly reduces the expressivity of our evolution model it also decreases the "social" bias and detects more easily possible interactions between scientific ideas and contributions independently of any particular scientific community.

The goal of topic evolution networks is to track complex temporal changes by epoch-wise topic discovery and temporal similarity graphs aligning topics of different epochs. Existing evolution network based frameworks mainly can be distinguished by the chosen topic extraction and alignment methods. [4] comes up with a method to enable a bottom-up reconstruction of the dynamics of scientific fields. They generate topics by word co-occurrence graphs and align inter-temporal topics by Jaccard similarity [9]. [1] generates topics by a Hierarchical Dirichlet Process (HDP) [14] and uses Bhattacharyya similarity [2], representing the gradual speciation and convergence similar to biologic evolution, for identifying topic alignments. The alignment process also applies (asymmetric) Kullback-Leibler divergence (KLD) for detecting topic split and merge. [11] introduces a novel approach to the early detection of research topics by using the Computer Science Ontology[2] to model research topics in the Rexplore system. They apply a Clique Percolation Method (ACPM) for analyzing the dynamics between existent topics. Other examples of science evolution studies explore how "cognitive science" as a field has changed over the last three decades [6] or analyze topic evolution patterns (split, merge and knowledge transfer) in the field of Information retrieval (IR) [5].

The goal of our work is to develop a general framework which is easier to use by domain experts who can ignore the details of the underlying topic analysis methods. The contributions presented in our demonstration can be summarized as follows:

- We propose a generic topic evolution model enabling the specification and extraction of meaningful topic evolution patterns independently of a particular topic extraction method.
- We define high-level measures for estimating the quality of the topic extraction process and for characterizing the structural and quantitative evolution of topics during a time period. This enables the experts to tune the topic extraction process and explore large topic evolution graphs by defining complex topic evolution patterns.
- We implemented a scalable prototype on top of Apache Spark for processing large scientific corpora containing millions of documents and finding meaningful topic evolution graphs for both stable topics and highly evolving ones.

2 TOPIC EVOLUTION MODEL

Topic evolution graphs: We consider a *corpus* C of time-stamped documents, a set of *periods* P and a set of terms V (*vocabulary*). Let $\mathcal{M} : 2^C \rightarrow 2^{\mathbb{R}^{|V|}}$ be a *topic extraction method* generating for a subset of documents $C' \subseteq C$ a set of (sparse) weighted term vectors $\mathcal{M}(C') \subseteq \mathbb{R}^{|V|}$. We denote by $C_p \subseteq C$ the corpus of documents with timestamp $p \in P$ and by $T_p = \mathcal{M}(C_p)$ the *topic descriptions* extracted from the documents C_p of period p using topic extraction method $\mathcal{M}$. A *topic* $t \in T_p$ is then defined by a couple $t = (d, p)$ where $d \in \mathcal{M}(C_p)$. We will denote by $t.d$ the *topic description* and by $t.p$ the *topic period*. Observe that topics from different periods may share the same description. For example in Figure 1, P has 3 periods: $p1$="2000 – 2002", $p2$="2002 – 2004" $p3$="2004 – 2006", T_{p1} contains topics 54 to 92, and topic $92 = (d, p1)$, where d is a weighted vector containing

[1]This work was funded by French ANR-16-CE38-0002-01 project EPIQUE.

[2]http://cso.kmi.open.ac.uk/

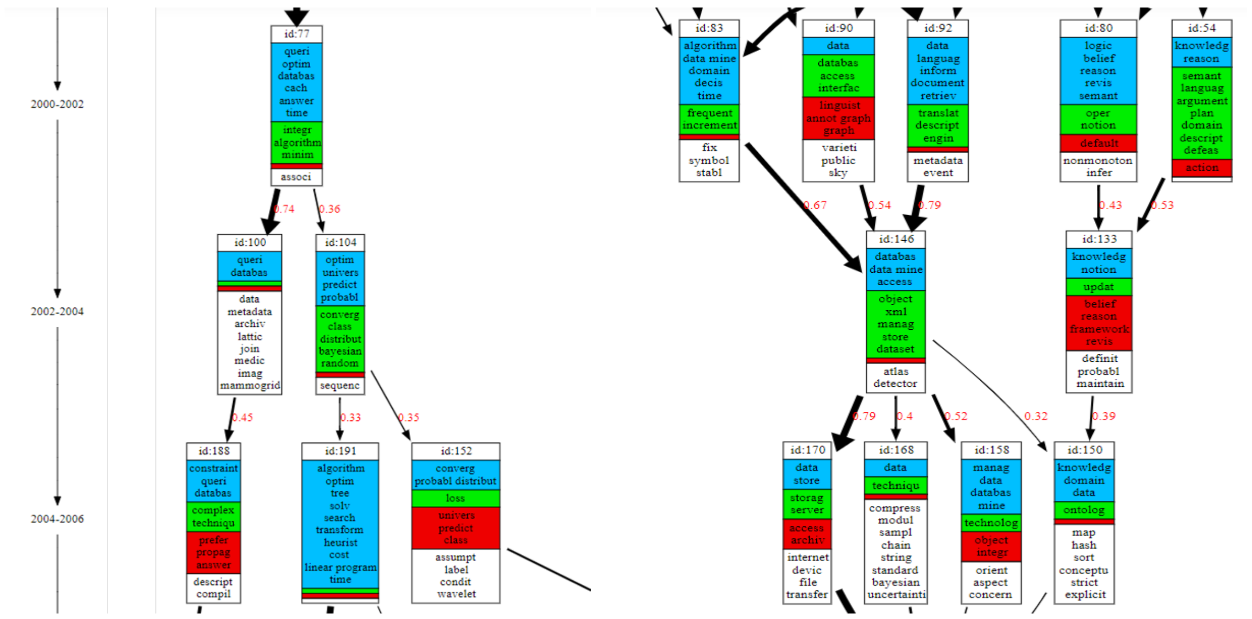


Figure 1: Pivot topics containing term "database" extracted from arXiv, green = emerging terms, blue = stable terms, red = decaying terms

terms "*queri*", "*optim*", "*databas*"... We define a *topic evolution function sim* : $T \times T \longrightarrow [0,1]$ estimating the *similarity* between topics in T. For example in Figure 1, the similarity measure depends on the topic description and estimates their semantic proximity using *cosine* similarity. The similarity between topic 77 and topic 100 is $sim(77,100) = 0.74$.

Based on the topic evolution function, we define a *topic evolution graph* as a directed labeled *multistage* graph $G_\beta = (T, E, sim, \beta)$ over topics T where the edges E connect all topics from consecutive periods with similarity higher or equal to some threshold β: $E = \{(t_i, t_j) \in T | sim(t_i, t_j) \geq \beta \wedge t_j.p = t_i.p + 1\}$.

Topic labeling: For visualization, we assume that all topics t of some evolution graph G_β are labeled by the top-k highest weighted terms in the topic description $t.d$. Let $t.l$ be the top-k highest weighted terms in $t.d$ and $t.l_p \subseteq t.l$ and $t.l_f \subseteq t.l$ be the subsets of *past* and *future* terms which appear, respectively, in the ancestor topics and in the descendant topics of t. Then, the terms in some topic vector $t.l$ are partitioned into the following four subsets of :

- *emerging* future terms $t.l_e = t.l_f - t.l_p$ which do not exist in past topics,
- *decaying* past terms $t.l_d = t.l_p - t.l_f$ which do not exist in future topics,
- *stable* terms $t.l_g = t.l_p \cap t.l_f$ which exist in the past and the future topics of t, and
- *specific* terms $t.l_s = t.l - (t.l_p \cup t.l_f)$ which neither exist in the past nor in the future topics of t.

The quadruple $[t.l_e, t.l_d, t.l_g, t.l_s]$ is called the *term label* of t.

Figure 1 shows two snippets of a single topic evolution graph extracted from the arXiv[3] corpus for the category DB (databases). Although the number of documents of category DB are limited, the generated graphs still generate meaningful evolution patterns. Emerging terms are shown in green boxes, decaying term boxes are colored in red, stable terms which exist both in ancestor topics and in descendant topics are grouped in blue boxes and specific terms which appear only in current topic are in white boxes. The thickness of edges reflects the similarity between topics. Several topics in both subgraphs contain the term "database" and we can observe different evolution patterns. The left hand graph shows that in period 2002-2004, topic 77 ("databases, queries, optimization, integration") split into topics 100 and 188 ("databases, queries and constraints") and topics 104, 191, 152 ("prediction, probability, random"). The right subgraph covers the same period with topics related to "data mining" (83), "data access interfaces" (90), "information retrieval" (92), "logics, semantics" (80) and "knowledge, reasoning" (54). The first three topics converge in 2002-2004 into a single topic on "object, xml, store, data mining" (146) which splits in the period of 2004-2006 into "storage servers" (170), "data mining and management" (158) and "knowledge and ontologies" (150).

Pivot evolution graphs: Threshold β strongly influences the complexity of the obtained evolution graphs. It is easy to see that $G_{\beta'}$ is a subgraph of G_β for all $\beta' \geq \beta$ and G_0 is the complete graph connecting all topics of two consecutive periods. More exactly, higher β values generate more "linear" graphs with many isolated topics, whereas lower values generate more complex graphs containing a variety of potentially interesting structures. Observe also that, whereas the pivot graph complexity of the same topic increases with decreasing β, high β thresholds might still generate complex pivot graphs and vice-versa. Analyzing science evolution by using topic evolution graphs then becomes a complex task which consist in computing and visually exploring multiple graphs for different β values. To solve this problem, we propose a different approach which allows users to formulate *filtering queries* for selecting interesting *subgraphs* with meaningful measures from a set of evolution graphs defined by a set of β thresholds. For this, we decompose topic evolution graphs into the set of all connected subgraphs defined by all paths containing a given topic t (one graph per topic). More formally, a *pivot evolution graph* $G_\beta(t) = (T', E', sim, \beta)$ *of topic* t in G_β is the subgraph of G_β which contains t and all *ancestors* and *descendants* of t. The subgraph of $G_\beta(t)$ containing all nodes which are reachable

[3]https://arxiv.org/

from t by a path is called the *future* of t, denoted by $F_\beta(t)$, and the subgraph of nodes which can reach t through a path is called the *past* of t, denoted by $P_\beta(t)$. The couple (t, β) is called a *pivot topic* with pivot graph $G_\beta(t)$, future $F_\beta(t)$ and past $P_\beta(t)$. It is easy to see that if t_1 appears in the future (past) of t_2, then the future (past) of t_1 is a subgraph of the future (past) of t_2 and t_2 appears in the past (future) of t_1. This property can be exploited to filter topics *wrt.* future and past topics (see the definition of Connection Filters below).

The evolution of topics within their evolution graphs can be characterized by the following metrics:

- The *liveliness* $live(G_\beta(t))$, of a pivot topic (t, β) is defined by the diameter (longest path length) of its pivot graph $G_\beta(t)$. A high liveliness value describes a long living topic, whereas a value equal to 0 corresponds to an isolated topic without ancestors and descendants. The liveliness $live(P_\beta(t))$ of t in its past estimates the "age" of t *wrt.* the first period, whereas $live(F_\beta(t))$ returns the "life expectation" of t (in its future).
- The *relative evolution degree* $revol(G_\beta(t))$ of a pivot topic (t, β) is defined by the average topic dissimilarity (edge) weight in $G_\beta(t)$. A low relative evolution degree states that most topics are connected to very similar topics, *i.e.*, most topics in $G_\beta(t)$ evolve slowly. On the other hand, a high value signifies that most topics have an important "semantic gap". By definition, $revol(G_\beta(t)) \geq \beta$.
- The *pivot evolution degree* $pevol(G_\beta(t))$ of a pivot topic (t, β) is defined by the average dissimilarity of all topics in $G_\beta(t)$ with respect to the pivot topic t. A low pivot evolution degree signifies that the pivot topic does not evolve a lot (all other topics are similar), whereas a high value indicates that the pivot topic evolves rapidly .
- The *split degree* $split(G_\beta(t))$ of a pivot topic (t, β) is defined by the average outdegree of $G_\beta(t)$. A low value signifies that the topics evolve along linear paths and a high value signifies that the topics split into several future sub-topics.
- The *convergence degree* $conv(G_\beta(t))$ of a pivot topic (t, β) is defined by the average indegree of $G_\beta(t)$. A low value signifies that many topics depend on a single parent topic and a high value signifies that many topics are the result of the fusion of past topics.

Evolution Pattern Filters: The previous evolution metrics characterize the the evolution of a topic in some evolution graph G_β. Combined with other filters on the topic labels and the graph structure, it is possible to filter pivot topics satisfying rich evolution patterns within a set of evolution graphs G_{β_i}, $1 \leq i \leq n$.

Term Filters select pivot graphs with respect to the pivot topic labels. In particular, they can be applied to filter pivot graphs *wrt.* to their emerging, decaying, stable, and specific terms.

Temporal Filters allow experts to filter the pivot topics situated within a certain time period.

Pattern Filters can filter topics by their pivot graph structure along their liveliness, split degree and convergence degree.

Evolution Filters are applied to filter topics by their relative and pivot evolution degrees.

The previous filters are applied to sets of pivot topics and can be combined with the following other kinds of operators:

Connection Filters are binary operators which select all pivot topics that are connected to at least one pivot topic in some other set of topics.

Temporal Projection allows to restrict structural, evolution and connection filters to the past or the future of the pivot topics.

Set Operators allow to combine two sets of topics by union, intersection and difference.

Ordering Operators sort pivot topics by their attributes, such as the topic period, its liveliness, evolution degree etc.

3 WORKFLOW AND IMPLEMENTATION

Figure 2 illustrates the overall workflow which takes as input a corpus of documents split into several, possibly overlapping time periods (the same document might appear in two periods).

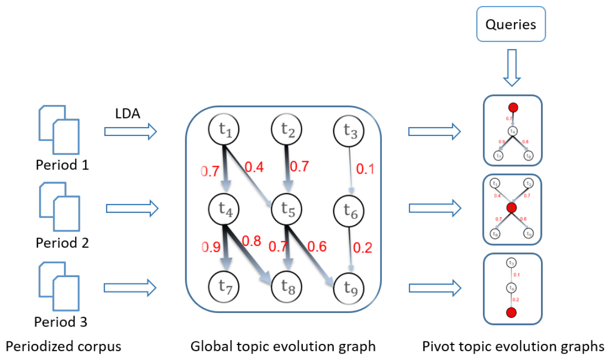


Figure 2: Topic evolution model of EPIQUE

All documents within a period are processed by LDA [3] to generate a set of topics which are aligned to produce a single topic evolution graph G_{β_0} for some small alignment threshold β_0. This global evolution graph is then transformed into n families of pivot evolution graphs defined by a set of alignment thresholds $\beta_i > \beta_0$, $1 \leq i \leq n$. Each family contains the pivot graphs $G_{\beta_i}(t)$ of all pivot topics (t, β_i). The final database contains $n \times |T|$ pivot graphs where $|T|$ is the number of topics in G_{β_0}. These graphs can then be queried using the filters defined in Section 2.

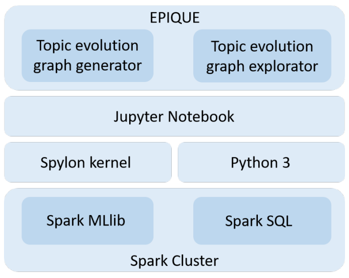


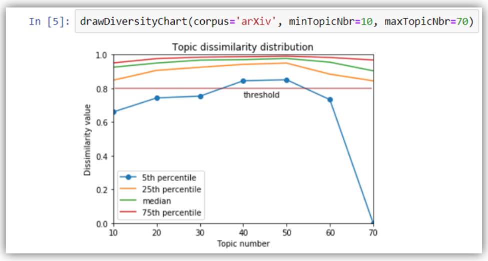
Figure 3: EPIQUE web application architecture

Figure 3 gives an overview of the architecture of our web application implemented on top of Apache Spark and Jupyter Notebook. The entire process to study science evolution over a corpus is split into two steps for building the pivot evolution graphs and for interactively exploring these graphs. Each step corresponds to a separate user interface. The evolution graph generation is implemented in Scala and exectued through the Spylon[4] kernel. Evolution graph exploration uses a standard Python kernel to take advantage of advanced Python 3 graphical user interface libraries for facilitating user interaction.

[4]https://github.com/Valassis-Digital-Media/spylon-kernel

4 DEMONSTRATION

Our EPIQUE prototype allows the audience to easily and intuitively generate high-quality evolution graphs and explore them. Among the corpora we have prepared in several domains, our demonstration focuses on the evolution of computer science based on the ArXiv corpus. We propose two interactive demonstration scenarios[5].



```
In [5]: drawDiversityChart(corpus='arXiv', minTopicNbr=10, maxTopicNbr=70)
```

Figure 4: Screenshot: topic diversity evaluation

Scenario 1 The audience selects or uploads a corpus of documents with a vocabulary of terms pre-processed by an on-line text-mining tool Gargantext[6] and specifies the time periods through sliding window over a global time period. Then, the LDA topic model is generated for each period. LDA requires a vocabulary and a number of topics to be generated. This number obviously influences the diversity of the resulting topics. Therefore, the application first generates a set of topic models for different topic numbers per period. The user can then visualize the diversity of the extracted topic models (topic dissimilairty distribution) and choose the model with the highest diversity for each period. A topic diversity distribution for different topic numbers is reported as shown in Figure 4 and, for example, by observing the 5th percentile values (blue line), the user can retain one of the two models (40 or 50 topics per period) that achieves 95% of pairwise dissimilarities above 0.8.

Then, the topics of consecutive periods are aligned and all pivot topic evolution graphs are generated along with their main temporal, structural and evolution indicators: *liveliness*, *split degree*, etc. All topic labels are also generated automatically in this step.

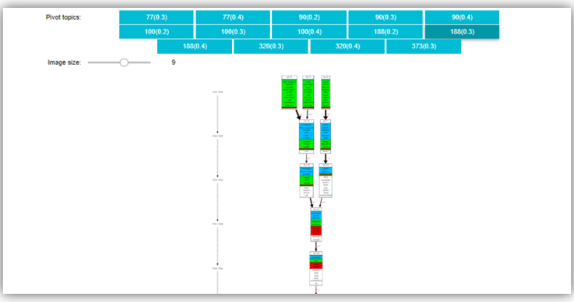


Figure 5: Screenshot: pivot topic evolution graph visualization

In the next step, the user specifies its exploration goal through an intuitive declarative query-by-example interface (as shown in the demonstration video[5]) and visualizes pivot topic evolution graphs as shown in Figure 5. These graphs are pre-computed in the last step to ensure fast query answer display.

We showcase a search for topic graphs containing a given term (*e.g.*, database) or set of terms suggested by the audience. Besides the topic content, the audience can search for topic graphs on their shape as well. We also prepared 10 predefined query templates for a typical shapes of high interest such as (i) topics that split in distinct 5+ years long branches, (ii) topic graphs with low *relative evolution degree* and high end-to-end *pivot evolution degree*. We also demonstrate more complex queries combining several query templates to build, for example, *concept drift* queries looking for pivot topics that contain emerging terms originating from other, "older" topics which are not part of their past pivot subgraph.

Scenario 2 In the second scenario, we will provide the audience with the possibility to prepare their own corpus using the Gargantext service which is also part of the EPIQUE project. Gargantext includes a number of bibliographic archives like Pubmed, Web of Science, etc. and allows to create domain specific document collections and vocabularies which are then processed by the same workflow as in Scenario 1.

5 FUTURE WORK

In the next step, we intend to optimize the computation of pivot topic evolution graphs and exploit the LDA document-topic matrix for enriching the analysis. Additionally, we plan to integrate other topic extraction methods than LDA. This prototype will also be used to validate our evolution model with philosophers of science to define and extract complex evolution patterns from different scientific domains.

REFERENCES

[1] Victor Andrei and Ognjen Arandjelović. 2016. Complex temporal topic evolution modelling using the Kullback-Leibler divergence and the Bhattacharyya distance. *EURASIP Journal on Bioinformatics and Systems Biology* 2016, 1 (2016), 16.

[2] A. Bhattacharyya. 1943. On a measure of divergence between two statistical populations defined by their probability distributions. *Bulletin of the Calcutta Mathematical Society* 35 (1943), 99–109.

[3] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.

[4] David Chavalarias and Jean-Philippe Philippe Cointet. 2013. Phylomemetic patterns in science evolution—the rise and fall of scientific fields. *PloS one* 8, 2 (2013), e54847.

[5] Baitong Chen, Satoshi Tsutsui, Ying Ding, and Feicheng Ma. 2017. Understanding the topic evolution in a scientific domain: An exploratory study for the field of information retrieval. *Journal of Informetrics* 11, 4 (2017), 1175–1189.

[6] Uriel Cohen Priva and Joseph L. Austerweil. 2015. Analyzing the history of Cognition using Topic Models. *Cognition* 135 (2015), 4–9.

[7] Eugene Garfield. 1955. Citation Indexes for Science: A New Dimension in Documentation through Association of Ideas. *Science* 122, 3159 (1955), 108–111.

[8] Qi He, Bi Chen, Jian Pei, Baojun Qiu, Prasenjit Mitra, and Lee Giles. 2009. Detecting Topic Evolution in Scientific Literature: How Can Citations Help?. In *ACM Conference on Information and Knowledge Management*. ACM, 957–966.

[9] Paul Jaccard. 1912. The Distribution of the Flora in the Alpine Zone.1. *New Phytologist* 11, 2 (1912), 37–50.

[10] Thomas S. Kuhn, Otto Neurath, and Thomas Samuel Kuhn. 1994. *The Structure of scientific revolutions* (2nd ed., enlarged ed.). Number ed.-in-chief: Otto Neurath ; Vol. 2 No. 2 in International encyclopedia of unified science Foundations of the unity of science. Chicago Univ. Press, Chicago, Ill.

[11] Angelo A. Salatino, Francesco Osborne, and Enrico Motta. 2018. AUGUR: Forecasting the Emergence of New Research Topics. In *ACM/IEEE on Joint Conference on Digital Libraries*. ACM, New York, NY, 303–312.

[12] Dafna Shahaf, Carlos Guestrin, Eric Horvitz, and Jure Leskovec. 2015. Information Cartography. *Commun. ACM* 58, 11 (2015), 62–73.

[13] Xiaoling Sun, Jasleen Kaur, Staša Milojević, Alessandro Flammini, and Filippo Menczer. 2013. Social Dynamics of Science. *Scientific Reports* 3 (2013), 1069.

[14] Yee W Teh, Michael I Jordan, Matthew J Beal, and David M Blei. 2005. Sharing clusters among related groups: Hierarchical Dirichlet processes. In *Advances in neural information processing systems*. 1385–1392.

[5] see http://www-bd.lip6.fr/wiki/site/recherche/projets/epique/demo/start for a video demonstration
[6] https://gargantext.org/

Task-Tuning in Privacy-Preserving Crowdsourcing Platforms

Joris Duguépéroux
Univ Rennes, CNRS, IRISA
Rennes, France
joris.dugueperoux@irisa.fr

Antonin Voyez
Univ Rennes, CNRS, IRISA
Rennes, France
antonin.voyez@irisa.fr

Tristan Allard
Univ Rennes, CNRS, IRISA
Rennes, France
tristan.allard@irisa.fr

ABSTRACT

Specialized worker profiles of crowdsourcing platforms may contain a large amount of identifying and possibly sensitive personal information (e.g., personal preferences, skills, available slots, available devices) raising strong privacy concerns. This led to the design of privacy-preserving crowdsourcing platforms, that aim at enabling efficient crowdsourcing processes while providing strong privacy guarantees even when the platform is not fully trusted. We propose a demonstration of the PKD algorithm, a privacy-preserving space partitioning algorithm dedicated to enabling secondary usages of worker profiles within privacy-preserving crowdsourcing platforms by combining differentially private perturbation with additively-homomorphic encryption. The demonstration scenario showcases the PKD algorithm by illustrating its use for enabling requesters tune their tasks according to the actual distribution of worker profiles while providing sound privacy guarantees.

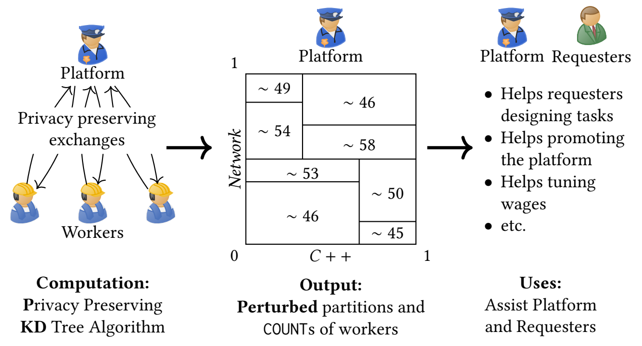


Figure 1: Overview of the PKD algorithm: supporting secondary usages of worker profiles with privacy guarantees

1 INTRODUCTION

Crowdsourcing platforms are online intermediates between requesters and workers: workers have skills and look for tasks, while requesters propose tasks that require specific skills. Crowdsourcing platforms are used in various application domains such as micro-tasks[1] or specialized software engineering[2]. Their efficiency, either for matching tasks to profiles (the primary usage of profiles) or for giving to requesters insights about the distribution of skills available within the population in order, e.g., to attract new requesters or to let requesters fine-tune their tasks according to the actual population of workers[3] (secondary usage of profiles), depends especially on the detailed information contained within worker profiles. A profile may indeed contain an arbitrary amount of information: professional or personal skills, daily availabilities, minimum wages, diplomas, professional experiences, centers of interest and personal preferences, devices owned and available, *etc.*

However fine grain worker profiles can be highly identifying or sensitive and privacy scandals have shown that those platforms are not immune to negligence or misbehaviours[4]. In a context where users expect crowdsourcing platforms to protect their personal data [Xia et al. 2017] and laws firmly require businesses and public organizations to safeguard the privacy of individuals (such as the European GDPR[5] or the California Consumer Privacy Act[6]) , designing and implementing sound privacy-preserving crowdsourcing processes is of utmost importance.

In this demonstration, we present the PKD algorithm [Duguépéroux and Allard 2019], a privacy-preserving space partitioning algorithm dedicated to enabling a wide range of secondary usages of worker profiles within privacy-preserving crowdsourcing platforms (see Figure 1). The PKD algorithm is distributed between a set of distrustful workers and an untrusted platform and builds on differentially private perturbation and additively-homomorphic encryption in order to compute a hierarchical partitioning of the skills of workers together with the approximate number of workers per partition. No raw worker profile is ever communicated to any other participant during the computation. The output of the PKD algorithm can be used for computing multi-dimensional COUNTs over worker profiles. The security of the PKD algorithm relies on composable security models in order to integrate well with privacy-preserving solutions to primary usages [Béziaud et al. 2017; Kajino 2015] without jeopardizing the privacy guarantees.

The demonstration scenario showcases the use of the PKD algorithm for letting requesters tune their tasks according to the actual population of workers, all this with sound privacy guarantees. The demonstration scenario essentially illustrates the impact of knowing the distribution of workers when tuning a task, and sheds the light on the tradeoff between privacy and utility within privacy-preserving crowdsourcing platforms, showing that a high privacy level can be guaranteed while still allowing high-standard secondary usages.

[1]https://www.mturk.com
[2]https://tara.ai
[3]For example, if functional language gurus are rare, it might be worth awarding more money for the task or updating the task such that it fits more common profiles.
[4]See, e.g., [Lease et al. 2013] or https://www.theverge.com/2014/11/19/7245447/uber-allegedly-tracked-journalist-with-internal-tool-called-god-view
[5]https://eur-lex.europa.eu/eli/reg/2016/679/oj
[6]https://www.caprivacy.org/

2 PRIVACY-PRESERVING INFORMED TASK-TUNING: AN OVERVIEW

2.1 Preliminaries

2.1.1 Participants. Three types of participants collaborate together during our crowdsourcing process. Workers are interested in solving tasks relevant to their profile, requesters propose tasks to be solved by appropriate workers, and the platform supports the intermediation. The number of skills n is fixed. A worker profile $p \in P$ is represented by an n-dimensional vector of floats where each value $p[j] \in [0, 1]$ represents the degree of competency of the profile p with respect to skill j. A task $t \in T$ is made of two parts. The first part is the meta-data containing the requirements needed to perform the task. We model it as an n-dimensional vector of ranges over skills (*i.e.,* a subspace of the space of profiles). The second part is the detailed task description provided by the requester (an arbitrary bitstring). In this work, we focus on the metadata part.

We assume that the participants are equipped with today's commodity hardware (i.e., the typical CPU/bandwidth/storage resources of a personal computer). However, we expect the platform to be available 24/7, similarly to a traditional client/server setting.

2.1.2 Security. We assume that all participants follow the *honest-but-curious* attack model: they may use any information disclosed along the algorithm to infer information about profiles, but they do not step outside the protocol.

As stated in Definition 1, the privacy model satisfied by the PKD algorithm is a computational variant of the well-known *differential privacy* model [Dwork 2006] called ϵ_κ-SIM-CDP [Mironov et al. 2009] (see the proofs in the technical report [Duguépéroux and Allard 2019]).

Definition 1 (ϵ_κ-SIM-CDP privacy [Mironov et al. 2009] (simplified)). *The randomized function f_κ provides ϵ_κ-SIM-CDP if there exists a function F_κ that satisfies ϵ-differential privacy and a negligible function $negl(\cdot)$, such that for every set of worker profiles $\mathcal{P}$, every probabilistic polynomial time adversary A_κ, every auxiliary background knowledge $\zeta_\kappa \in \{0, 1\}^*$, it holds that:*

$$|\Pr[\mathsf{A}_k(\mathsf{f}_\kappa(\mathcal{P}, \zeta_\kappa)) = 1] - \Pr[\mathsf{A}_k(\mathsf{F}_\kappa(\mathcal{P}, \zeta_\kappa)) = 1]| \le negl(\kappa)$$

The original ϵ-differential privacy model applies to a randomized function f and aims at hiding the impact of any possible individual value on the possible outputs of f, often by adding random noise to it. Computational variants of differential privacy are especially relevant when differentially private perturbation and semantically secure encryption are used jointly, as done within the PKD algorithm. First, the differentially private perturbation scheme used by the PKD algorithm is the Geometric mechanism [Ghosh et al. 2012]. It consists essentially in sampling a two-sided geometric distribution parameterized by the differential privacy parameter ϵ and by the aggregate to be perturbed. It benefits from the following nice properties: it is designed for perturbing integers, and the sampling can be easily distributed over workers (infinite divisibility of the two-sided geometric distribution [Duguépéroux and Allard 2019]). Second, the PKD algorithm makes use of an *additively homomorphic encryption scheme.* Additively-homomorphic encryption schemes essentially allow to perform addition operations over encrypted data. Any additively-homomorphic encryption scheme fits our approach as long as it provides *semantic security guarantees* (usual security guarantees), *additively-homomorphic encryption* (possibility to perform additively homomorphic sums) and *non-interactive threshold decryption* (allows the decryption key to be split in K key-shares, such that a complete decryption requires to perform independently $T \le K$ partial decryption by distinct key-shares). The Damgard-Jurik cryptosystem [Damgård and Jurik 2001], a generalization of Paillier [Paillier 1999], is an instance of encryption scheme that provides the desired properties. We refer the interested reader to the original paper for details [Damgård and Jurik 2001].

2.1.3 Quality. Roughly speaking, we evaluate the quality achieved by the outputs of the PKD algorithm by measuring the average absolute error between (approximate) counts estimated through its outputs and the corresponding exact counts computed on the raw, non-protected, worker profiles. We refer the interested reader to the technical report [Duguépéroux and Allard 2019] for more details on the quality definition and on the experimental results.

2.2 The PKD algorithm

The PKD algorithm is an adaptation of the well-known centralized KD-Tree construction algorithm [Bentley 1975], to a context where no central server is trusted, and the result is private. The resulting KD-Tree is used to estimate the underlying *multidimensional* distribution of workers (e.g., to let requesters tune their tasks accordingly). Each worker holds his profile locally and engages with the platform and other workers in the execution of the PKD algorithm. The PKD algorithm consists essentially in splitting recursively the space of skills in two and stops when a termination criterion is met (e.g., fixed number of splits). The split is performed by choosing one dimension d at each iteration (e.g., considering on dimension after the other), projecting the set of skills on d, and forming two partitions around the median. The PKD algorithm outputs a binary tree where each node is a partition of the space of skills with the (perturbed) number of worker profiles it contains. The key operation of the PKD algorithm is the distributed privacy-preserving computation of medians. It is implemented by building at each iteration the (perturbed) histogram of the dimension being split and using the histogram in order to estimate the median. This histogram results from the privacy-preserving aggregation, on the platform, of the local histograms of workers. The following execution steps synthesize the computation of a perturbed histogram over the dimension d:

(1) Each worker locally instantiates his local histogram over the dimension d such that all the bins are set to 0 except the bin within which the worker's skill degree on d falls, set to 1.
(2) Each worker locally adds a *noise-share* to each bin, where a noise-share is a random variable such that the sum of a fixed number of noise-shares follows the two-sided geometric distribution. Recall (1) that the two-sided geometric distribution is infinitely divisible, and (2) that the addition to an integer of a random variable sampled from a two-sided geometric distribution well parameterized satisfies differential privacy (see above).
(3) Each worker encrypts each of its bins with the additively-homomorphic encryption scheme and sends it to the platform.

(4) The platform sums up all the encrypted histograms received from workers (bin per bin).
(5) The workers and the platform collaboratively decrypt the resulting encrypted histogram building on the threshold decryption feature of the encryption scheme (see above). The platform thus obtains the perturbed "summed up" histogram.
(6) The platform estimates the median based on the histogram, splits the dimension d around it, and either iterates on each of the two resulting partitions, or stops the algorithm if a termination criteria is met (e.g., sufficient number of splits).

The PKD algorithm further improves the quality of the counts of the hierarchy of partitions by post-processing them based on constrained inference techniques (see [Cormode et al. 2012; Hay et al. 2010] for details). A complete description of the PKD algorithm together with thorough experimental results are available in [Duguépéroux and Allard 2019].

2.3 Informed Task Tuning

The partitioning of the space of skills output by the PKD algorithm enables the computation of multi-dimensional COUNTs over the space of skills of the actual population of workers. A large variety of usages can be envisioned. We focus in this paper on a precise illustration that consists in publishing the partitioning of the space to requesters, and letting them tune their tasks according to the actual distribution of skills. For example, through an appropriate *task tuning helper*, provided by the platform or implemented on behalf of the requester, the latter could define wages according to the scarcity of a profile, or tune the skills required by a task such that they fit the profiles of a sufficiently high number of workers which results in lower pickup times.

3 DEMONSTRATION

This demonstration illustrates the PKD algorithm by (1) allowing its execution on a wide variety of parameters (e.g., various populations of workers, different numbers of iteration, different values of the ϵ privacy parameter) and (2) allowing the audience to create tasks matching the population of workers through a simple *task tuning* helper. The demonstration platform is centralized and simulates the distributed components of the PKD algorithm. We present below the technicals details of the demonstration platform, the parameters that can be set up by the audience (called *mutable* parameter) , the parameters that are fixed, and the demonstration scenario.

3.1 Platform

Figure 2 depicts the demonstration platform. The demonstration is implemented as a web application running through a single docker-compose file. In this file, several services handle every aspect of the application without any configuration or installation (except for docker and docker-compose, which are not specific to this demonstration). The first service and the core of the application is a Python Django web server used to serve the web interface and handle the commands issued by the demonstrator. The user interface, served by Django consists in simple HTML pages using the CSS framework Bootstrap for the design and ViewJS scripts for the dynamic components. A second service handles the long tasks in the background that cannot reasonably

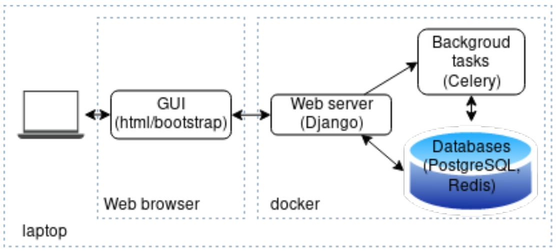


Figure 2: The demonstration runs on a single laptop executing the demonstration platform: the web server (Python Django), the background tasks handler (Celery) and the databases (PostgreSQL and Redis). The demonstration is accessible through a web interface (e.g., on the browser of the demonstration laptop).

be handled by Django without causing a loss of the user experience (i.e., tasks that last more than a few seconds such as the PKD algorithm and the CSV import of the workers). This service is written in Python with the Celery framework. Databases required by the application (PostgreSQL and Redis), to handle data persistency, are directly embedded in the docker-compose file. The homomorphic encryption features are disabled in order to reduce computation time for the demonstration. Indeed, for the sake of simplicity, all distributed operations, normally done by distinct workers, are done locally by the demonstration platform[7]. In particular, worker profiles are stored locally rather than being hold by individual workers, and the encrypted sum of histograms is replaced by a cleartext sum. These simplifications have no consequence on the output of the PKD algorithm. The source code of the demonstration is available publicly [8] and can be executed on a laptop where docker is installed (no configuration is required).

3.2 Parameters

In this demonstration we let the audience set various parameters, while others are fixed to default values. Default parameters are chosen to reflect plausible real-life settings while keeping the computation time reasonable. In particular, we limit the number of skills (e.g., 2 or 3 skills chosen by the audience) and the number of workers (e.g., a few hundreds instead of a few thousands in a real-life system).

The audience is able to set the skills, the worker profiles (either manually or automatically) and the PKD algorithm parameters. For these parameters, we also provide default values to help the audience: the differential privacy security parameter ($\epsilon = 0.1$), the number of splits for the partitioning ($splits = 7$) and the number of bins of histograms ($bins = 10$). Finally, the audience can tune tasks to fit with the previously defined workers.

Note that the termination criteria must be chosen carefully because it limits the number of splits of the space of skills[9]. The dimensions that are not part of the sequence will simply be ignored. The number of dimensions in workers profiles, and their respective priorities, is closely related to the application domain (e.g., How specific does the crowdsourcing process need to be?). In this paper, we make no assumption on the relative importance of dimensions.

[7]In a real-life scenario, encrypted operations would be performed in parallel by workers and the platform itself, which greatly reduces the costs
[8]https://gitlab.inria.fr/crowdguard-public/implems/pkd-demo
[9]It also impacts the overall computation time and quality of the estimation

3.3 Datasets

Three populations of workers are available by default: two populations are generated synthetically (i.e., through our `UNIF` worker generator that samples skill levels uniformly at random and our `ONESPE` worker generator that choses one strong skill uniformly at random for each worker and sets a low skill level to all others skills - see [Duguépéroux and Allard 2019] for details), and one population is computed from the public *Stackoverflow* dataset[10]. Additionally, the audience can instantiate a set of workers manually. Additional arbitrary populations of workers defined by the audience can be imported. Our platform accepts `CSV` files, such that each line is defined by three columns, as shown in Figure 3.

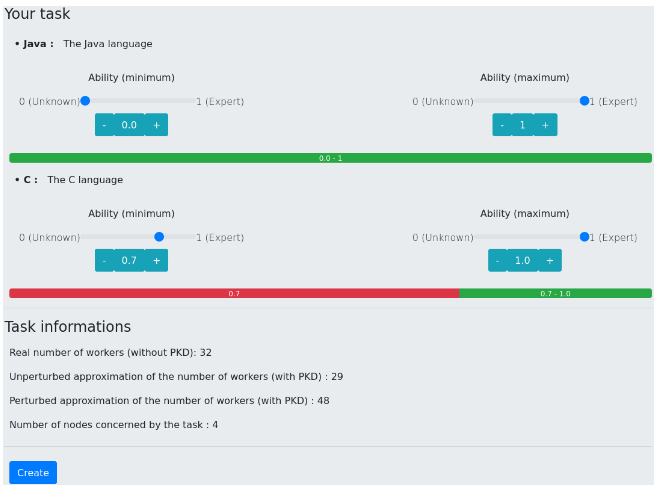
`UserID` (int)	`SkillID` (int)	`SkillLevel` (float in [0; 1])
12	3	0.4

Figure 3: Format of a worker dataset and illustration on a single worker.

3.4 Scenario

The demonstration scenario presents a simple execution sequence allowing the audience to observe the different steps of the PKD algorithm and to use the task tuning module. It concentrates on the task design, only the necessary information about the distribution of workers is displayed. A strong focus has been put on the simplicity and the clarity of the GUI which includes all the explanations needed to understand intuitively each step of the demonstration. The GUI is divided into a sequence of screens, where each screen is dedicated to a specific step of the execution sequence. First, an introduction screen presents the demonstration and its objective. Second, the audience choses the set of skills to consider. Third, the audience can launch the workers import (according to the various methods described above, including the import of a `CSV` file from the audience). Additional information about the distribution of skills within the dataset chosen is displayed through a Notebook document and commented. Fourth, the `PKD` algorithm is executed on the population of workers defined and outputs the space partitioning computed. Finally and most importantly, the audience uses our simple task tuning helper in order to tune a few tasks (1) without any information on the underlying population (default method within privacy-preserving crowdsourcing platforms) and (2) with the hierarchy of partitions computed by the `PKD` algorithm. Figure 4 shows the screen dedicated to tuning the task with information from the hierarchy of partitions. Optionnally, in order to observe the privacy/utility tradeoff, the audience is invited to explore the hierarchy of partitions and inspect the impact of the differentially private perturbation by comparing the perturbed counts to the exact unperturbed number of workers within each partition.

[10] We consider that a user is a worker, tags of posts are skills, and skill levels are a simple popularity score computed from the number of up-votes of each post. See https://gitlab.inria.fr/crowdguard-public/data/workers-stackoverflow for more details (i.e., description of the method and pre-processing scripts).



Your task

- **Java :** The Java language

Ability (minimum): 0 (Unknown) – 1 (Expert): 0.0

Ability (maximum): 0 (Unknown) – 1 (Expert): 1

0.0 - 1

- **C :** The C language

Ability (minimum): 0 (Unknown) – 1 (Expert): 0.7

Ability (maximum): 0 (Unknown) – 1 (Expert): 1.0

0.7 | 0.7 - 1.0

Task informations

Real number of workers (without PKD): 32

Unperturbed approximation of the number of workers (with PKD) : 29

Perturbed approximation of the number of workers (with PKD) : 48

Number of nodes concerned by the task : 4

Figure 4: Tuning the task with information from the hierarchy of partitions. On the first half of the screen (top), the skills requirements of a task are being tuned over the Java and `C` programming skills (the union of the leaf partitions appears on the green lines, and neighboring nodes appear on the red lines). On the second half of the screen (bottom), the screen displays information about the perturbed and actual number of workers corresponding to the task requirements.

ACKNOWLEDGMENTS

This work is partially supported by the French National Research Agency (Crowdguard project, ANR grant number `ANR-16-CE23-0004`).

REFERENCES

Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

Louis Béziaud, Tristan Allard, and David Gross-Amblard. 2017. Lightweight privacy-preserving task assignment in skill-aware crowdsourcing. In *Proc. of DEXA '28.* 18–26.

Graham Cormode, Cecilia Procopiuc, Divesh Srivastava, Entong Shen, and Ting Yu. 2012. Differentially private spatial decompositions. In *Proc. of IEEE ICDE '12.* 20–31.

Ivan Damgård and Mads Jurik. 2001. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *International Workshop on PKC.* 119–136.

Joris Duguépéroux and Tristan Allard. 2019. *A Space Partitioning Algorithm for Privacy-Preserving Crowdsourcing.* Technical Report. (available on demand).

Cynthia Dwork. 2006. Differential privacy. In *Proc. of ICALP '06.* 1–12.

Arpita Ghosh, Tim Roughgarden, and Mukund Sundararajan. 2012. Universally utility-maximizing privacy mechanisms. *SIAM J. Comput.* 41, 6 (2012), 1673–1693.

Michael Hay, Vibhor Rastogi, Gerome Miklau, and Dan Suciu. 2010. Boosting the accuracy of differentially private histograms through consistency. *Proc. of the VLDB Endow.* 3, 1-2 (2010), 1021–1032.

Hiroshi Kajino. 2015. *Privacy-Preserving Crowdsourcing.* Ph.D. Dissertation. Univ. of Tokyo.

Matthew Lease, Jessica Hullman, Jeffrey P. Bigham, Michael S. Bernstein, Juho Kim, Walter Lasecki, Saeideh Bakhshi, Tanushree Mitra, and Robert C. Miller. 2013. Mechanical Turk is Not Anonymous. *SSRN Electronic Journal* (2013).

Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. 2009. Computational Differential Privacy. In *Proc. of CRYPTO '29.* 126–142.

Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. of EUROCRYPT '99.* 223–238.

Huichuan Xia, Yang Wang, Yun Huang, and Anuj Shah. 2017. Our Privacy Needs to be Protected at All Costs: Crowd Workers' Privacy Experiences on Amazon Mechanical Turk. *Proc. of ACM HCI'17* 1 (2017), 113.

Scaling a Public Transport Monitoring System to Internet of Things Infrastructures

Haralampos Gavriilidis[1] Adrian Michalke[2] Laura Mons[2] Steffen Zeuch[1,2] Volker Markl[1,2]
[1]Technische Universität Berlin [2]DFKI GmbH
{gavriilidis,volker.markl}@tu-berlin.de,{adrian.michalke,laura.mons,steffen.zeuch}@dfki.de

ABSTRACT

Applications for the Internet of Things (IoT) face several challenges when it comes to exploiting the underlying infrastructure for data management operations efficiently. IoT infrastructures consist of heterogeneous compute nodes and geographically distributed network topologies. Today's IoT applications offload data management to cloud-based stream processing engines (SPEs). However, this offloading represents a severe bottleneck that might hinder upcoming large-scale IoT applications in the future. In our demonstration, we showcase this problem using a public transport application as a potential large-scale IoT application. Our application consists of an interactive map for monitoring public transport vehicles and current demand. We implement this application on top of NebulaStream (NES), a new data management system that is designed for the IoT. In contrast to common cloud-based SPEs, NES answers queries by unifying cloud, fog, and sensor nodes under one system. Thus, NES minimizes network traffic and avoids resource over-utilization by considering the physical network topology and available compute nodes. The goal of this demonstration is to reveal the shortcomings of current system designs for large-scale IoT applications. Furthermore, we showcase how NES addresses these shortcomings and thus enables future large-scale IoT applications.

1 INTRODUCTION

Applications for the IoT, such as reporting and monitoring dashboards, consist of real-time data preprocessing and data mining tasks. Such applications visualize high-velocity data streams, resulting from large sensor networks, which flow through heterogeneous hardware and network topologies to the cloud.

Sensor streams naturally match the stream processing programming abstractions provided by cloud-based SPEs. Thus today's IoT applications use such systems to offload data management tasks. SPEs exploit the on-demand scalability of cloud resources to process compute-intensive data management workloads efficiently. However, state-of-the-art SPEs, such as Apache Flink [4], were designed for cloud environments composed of homogeneous high-performance hardware, where nodes are interconnected through high-speed network connections.

In contrast, IoT infrastructures have different characteristics regarding compute nodes and network connections [3]. In this new type of infrastructure, nodes are heterogeneous, geographically distributed, and sparsely interconnected through unstable networks. Geographically distributed sensor nodes continuously generate data, resulting in a large number of data streams with small-sized records. Intermediate nodes, also called fog nodes, provide network resources to route sensor data to the cloud. Certain nodes provide compute resources that support executing data management tasks. In particular, intermediate nodes range from low-end devices, such as system-on-a-chip devices, to high-end nodes, e.g., desktop computers and server racks. Cloud-based SPEs ignore intermediate nodes when distributing their load, hindering IoT applications from scaling beyond the cloud.

To scale data management tasks on all participating devices of an IoT infrastructure, the design of data management systems must be revisited. Recent work points out that to exploit resources of every node in an IoT infrastructure, data management systems must employ infrastructure-aware execution strategies [11]. A data management system for the IoT should leverage the scale-out capabilities of the cloud, and at the same time, exploit the resources of intermediate nodes. In particular, cloud nodes can scale resources for compute-intensive tasks within the cloud, while intermediate nodes apply fog computing techniques [6–8] to reduce intermediate results. As a result, network traffic and cloud resources are minimized. However, cloud-based approaches utilize intermediate nodes only to forward data.

NebulaStream [11] is an application and data management platform designed for the upcoming IoT era. NES addresses the mentioned IoT challenges by unifying sensor, fog, and cloud nodes into a single system. To this end, NES combines research from sensor network, distributed, and database system communities. This allows NES to transparently optimize and efficiently execute data management workloads across IoT infrastructures. The unified sensor-fog-cloud approach enables scaling the number of sensors in data management and visualization applications.

In our demonstration, we simulate an IoT infrastructure with Raspberry PIs and showcase a visualization application for a public transport system. Our application aims to provide real-time monitoring for public transport systems and suggestions for vehicle rescheduling. Therefore, it detects critical geographical areas, i.e., underserved areas with high demand. Our GUI consists of an interactive map, which visualizes real-time public transport vehicles and potential passengers. During the demonstration, visitors will interact with the map by filtering objects and configuring the clustering algorithm employed for critical area detection.

Furthermore, visitors will explore potential execution strategies for data management tasks on IoT infrastructures. We demonstrate how all participating nodes of a public transport system can be part of a query and how this influences performance and resource utilization. Our application demonstrates both centralized system designs and new designs enabled by NES. Overall, we showcase that in contrast to cloud-based SPEs, NES allows large-scale applications on IoT infrastructures, and thus enables a variety of upcoming IoT application use cases.

The rest of this paper is structured as follows. In Section 2, we discuss IoT application scenarios and challenges related to data management. In Section 3, we provide a brief overview of NES's design and architecture. After that, in Section 4, we present our demonstration scenario and finally conclude in Section 5.

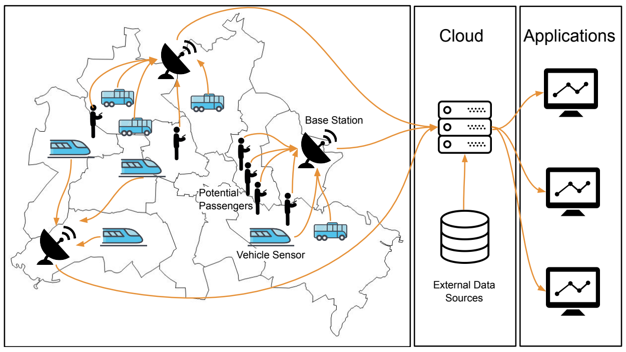


Figure 1: Exemplary IoT Infrastructure.

2 DATA MANAGEMENT IN THE IOT

In the following, we introduce a representative IoT scenario (Section 2.1) and discuss data management challenges (Section 2.2).

2.1 IoT Application Scenarios

In Figure 1, we illustrate a public transport system as a representative large-scale IoT scenario. In our scenario, the cloud hosts applications, i.e., a real-time dashboard that monitors the underlying IoT infrastructure. The infrastructure consists of geographically distributed and constantly moving sensors, base stations, and cloud nodes. In particular, potential passengers with mobile phones and vehicles with attached sensors move within a city and transmit measurements to base stations in regular time intervals. Base stations are geographically distributed nodes that collect and forward sensor streams to the cloud. Cloud nodes receive the data from the base stations and enrich it with external sources, e.g., databases with weather or air pollution data. Finally, applications consume the preprocessed sensor stream, apply additional processing, and visualize the result for users.

Public transport agencies could employ such applications to enable various smart city optimizations. Examples include rescheduling vehicles based on the demand of potential passengers, air pollution reduction by traffic light control mechanisms, and ad-hoc route planning to cope with traffic jams.

2.2 IoT Infrastructure Challenges

Today's IoT applications use data acquisition systems for data ingestion [7, 9, 10] and cloud-based SPEs for data management operations. IoT infrastructures differ significantly from cloud infrastructures, and thus pose several challenges for data management operations. Zeuch et al. [11] point out that cloud-based SPEs rely on assumptions that IoT infrastructures violate. First, fog and cloud paradigms assume different network topologies. In particular, processing nodes in cloud-based SPEs are densely connected, i.e., each node has stable connections with all other nodes. In contrast, the physical IoT topology predefines the network paths from data sources (sensors) to data sinks (cloud). Therefore, every node accesses only the subset of data that is routed through it. For example, in the IoT infrastructure depicted in Figure 1, base stations located in the west of the city are not able to directly access sensor streams that are generated on the east of the city.

Second, both paradigms expect different types of input streams. In an IoT infrastructure, millions of sensors constantly produce data streams. Those streams consist of small records that possibly capture physical phenomena, such as earthquakes, and might produce data at infrequent intervals. In contrast, cloud-based SPEs expect a few large-volume data streams at constant producing intervals. Furthermore, to ingest the sensor streams, cloud-based systems utilize message brokers, such as Apache Kafka.

In sum, IoT applications relying on cloud-based data management paradigms do not exploit intermediate nodes. The assumptions of cloud-based SPEs regarding physical topologies and types of incoming streams do not hold in IoT infrastructures.

3 NEBULASTREAM PLATFORM OVERVIEW

In the following, we present specific aspects of NES. We refer the reader to our previous work [11] for a detailed description of NES. First, in Section 3.1, we outline the limitations of cloud-based SPEs that prevent applications from exploiting IoT infrastructures. After that, we describe NES and its architecture in Section 3.2.

3.1 Limitations of State-of-the-art SPEs

In the following, we discuss two important features that hinder cloud-based SPEs to efficiently support future IoT scenarios.

Exploiting Intermediate Nodes: Applications relying on cloud-based SPEs do not exploit all participating heterogeneous intermediate and sensor nodes, since data management tasks are executed only in the cloud layer. For example, consider a simple aggregation task, e.g., counting the number of vehicles per geographical area. To execute this task, cloud-based systems would have to wait until intermediate nodes propagate data to the cloud. However, in the described IoT infrastructure, intermediate nodes can execute this task at an earlier stage and forward pre-aggregated intermediate results.

Minimizing Network Resources: If a set of running queries requires only a subset of the sensor data, acquisitional data processing avoids redundant sensor reads [7]. Another technique to further reduce network traffic between sensors and cloud is to adapt sensor sampling frequencies based on the query requirements [9]. For example, a vehicle could acquire and send its data only if it is located in a certain area [11]. These techniques are not available in cloud-based SPEs, but allow for scaling IoT data processing by minimizing redundant data traffic.

3.2 Architecture

In the following, we describe NES's architecture, illustrated in Figure 2. We focus only on the components that are related to our application scenario and omit components that are out of the scope of this demonstration, e.g., fault-tolerance mechanisms.

Optimization: NES exposes APIs for common data processing operations, as found in cloud-bases SPEs, extending those with fog-specific abstractions. IoT applications, e.g., our visualization application, submit queries to the underlying data streams, both ad-hoc and long-running. To allow for multi-query optimization, the *Query Manager* maintains a catalog of submitted queries. Query optimization and execution in NES proceeds as follows. First, NES translates user queries to logical query plans. After that, the plan is handed over to the optimizer. The *NES Optimizer* consults the *NES Topology Manager*, which provides information about the infrastructure status and the performance statistics. After optimizing the execution plan, it is handed over to the *NES Deployment Manager*.

Deployment and Execution: NES's execution plan maps segmented sub-plans to participating intermediate, or cloud nodes. Each segment contains processing instructions (tasks), as well as information about I/O operations. The NES Deployment Manager is responsible for transmitting the sub-plans to each node. After receiving a sub-plan, a node initializes the necessary network connections and starts the execution utilizing its task scheduler.

Monitoring: During runtime, the NES Topology Manager monitors the execution, and reacts to changes incrementally, e.g., by transitioning smoothly between execution plans. To this end, the NES Topology Manager collects hardware statistics, such as CPU and main memory utilization, and additional metadata, such as selectivities and data distributions. NES nodes are designed to handle several scenarios autonomously, e.g., transient network failures. Once failed network connections are restored, topology updates are propagated to the Topology Manager.

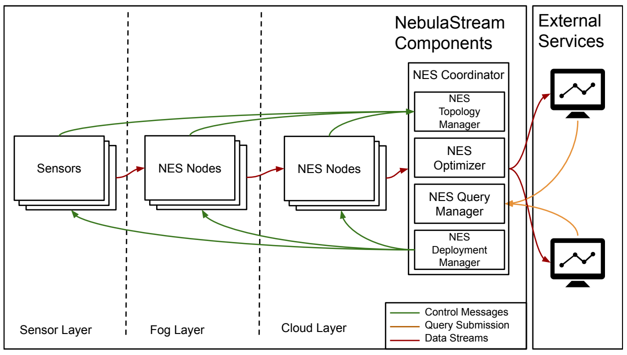


Figure 2: Simplified NES architecture overview.

The holistic view of the underlying infrastructure and the submitted queries allow NES to optimize for specific combinations of queries and topologies. In contrast, cloud-based SPEs handle incoming queries independently and schedule queries with external cluster resource management frameworks. Its characteristics allow NES to overcome limitations of cloud-based SPEs in IoT infrastructures, and to offer scalable data management for the upcoming IoT application scenarios.

4 DEMONSTRATION

In the following, we describe our user interface (Section 4.1), demonstration setup (Section 4.2), and application (Section 4.3).

4.1 User Interface

Visitors interact with our application through a GUI, as shown in Figure 3. Our GUI consists of an interactive map, which includes moving objects that are either potential passengers or public transport vehicles (trains, buses, etc.). Our application aims at detecting crowded geographical areas, that public transport vehicles underserve. The application clusters potential passengers according to their geolocation. When public transport vehicles underserve a cluster of potential passengers, our application notifies the visitor by highlighting the respective clusters on the map. The resulting notifications are useful for public transport agencies, e.g., to schedule vehicles for crowded areas.

Additionally, our GUI allows filtering by moving the visible map area and by selecting vehicle types. The visitor configures the visible objects and the clustering algorithm by changing parameters, e.g., object distance and cluster size, as shown in Figure 3 ①. A main feature is choosing between processing modes ② that represent the solution space for IoT applications. To show the implications of each mode, we provide performance metrics, such as ad-hoc application statistics ③ and resource utilization ④. Our demonstration aims at showcasing the strengths and weaknesses of each solution in a hands-on experience.

4.2 Demo Setup

In the following, we describe the setup of our demonstration.

Hardware: For our demonstration scenario, we assume a topology where each public transport vehicle carries a sensor transmitting information to base stations at regular time intervals. Potential passengers send their geolocation to base stations, e.g., through a mobile application. We use Raspberry PIs as base stations (fog nodes), and a mobile workstation as a cloud node.

Software: Our application consists of a frontend and a backend component. The backend uses NES to offload data management operations. The backend acts as a sink for NES, i.e., it is an intermediator between multiple user frontends and NES. It coordinates query transmission to NES, forwarding results to connected frontends. Our application backend is implemented in Python and utilizes the Flask microframework. The frontend is implemented in Javascript, utilizes websockets for communication, Leaflet for its map, and Grafana for monitoring.

Dataset: We simulated two sensor streams to compose our dataset. First, we simulated vehicle sensor data using real-world *General Transit Feed Specification* datasets [1] (enhanced with vehicle occupancy). Second, we simulated potential passengers using the *Simulation of Urban Mobility* (SUMO) generator [2]. We merge the two resulting datasets, and partition them by geolocation, to resemble geographically distributed base stations. We utilize sensor data from the city of Berlin, however, our application supports all sensor streams following the GTFS schema.

4.3 Application

We highlight two aspects of an IoT data management application scenario. First, we describe our deployment process, during which our application transforms user queries to execution plans and deploys them on the underlying nodes. Second, we demonstrate several execution strategies by deploying different execution plans and revealing their implications on resource utilization.

4.3.1 Query and Deployment. Every time a visitor interacts with the map and its options, our application sends NES a new query with the following parametrized operations.

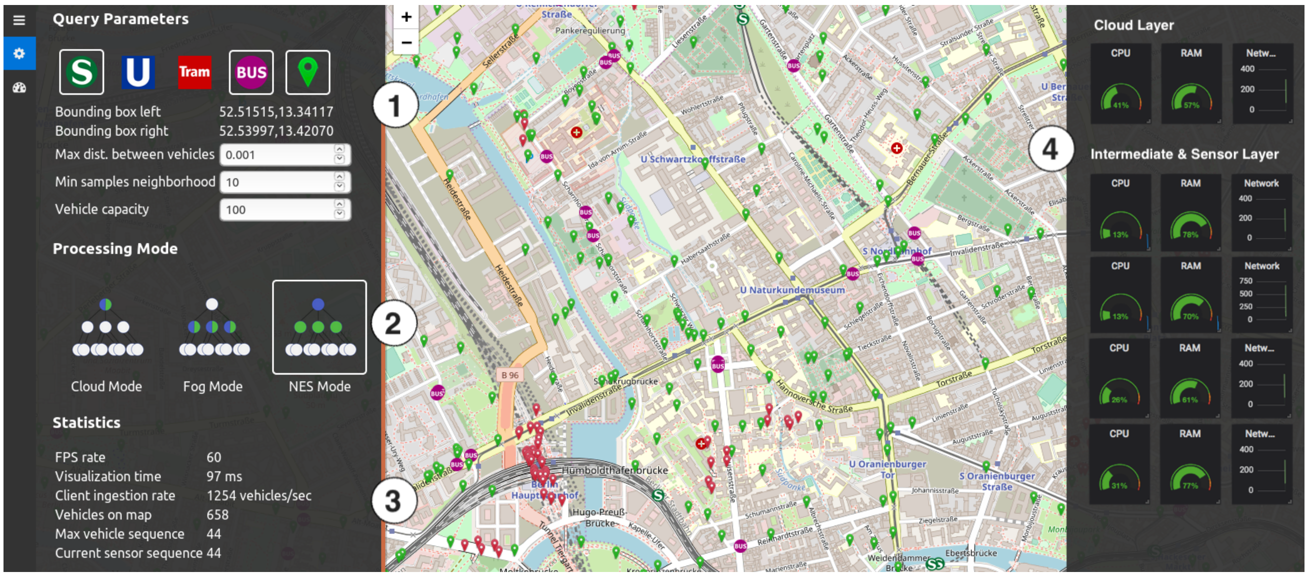
Map Bounding Box: These parameters filter vehicles and potential passengers located within a bounding box. The bounding box is defined by two coordinates and is automatically computed by the map area currently viewed by the visitor.

Vehicles and Passengers: These parameters filter potential passengers and selected vehicle types, e.g., bus, train, or subway.

Clustering: These parameters, e.g., object distance and cluster size, are related to our clustering algorithm (DBSCAN [5]).

Overall, our application composes and deploys queries that filter sensor records, cluster them by geolocation, calculate the average vehicle capacity within each cluster, and yield critical areas depending on user-defined thresholds.

In this application, NES allows us to reduce data as early as possible in the IoT infrastructure. Especially in visualization scenarios, data reduction is a key performance factor, and naturally occurs because users are seldomly interested in all sensor data. To this end, NES provides the option to filter data needed for visualization purposes already in the intermediate (fog) layer. The resulting data reduction is two-fold. First, NES uses on-demand data acquisition techniques to only gather sensor data that is currently required to answer a query. Second, NES uses intermediate nodes to evict unnecessary data close to the sensors. Both data reductions minimize the overall network traffic within the IoT infrastructure, as well as data that has to be sent to applications.


Query Parameters

Bounding box left 52.51515,13.34117
Bounding box right 52.53997,13.42070
Max dist. between vehicles 0.001
Min samples neighborhood 10
Vehicle capacity 100

Processing Mode

Cloud Mode | Fog Mode | NES Mode

Statistics

FPS rate 60
Visualization time 97 ms
Client ingestion rate 1254 vehicles/sec
Vehicles on map 658
Max vehicle sequence 44
Current sensor sequence 44

Figure 3: Demo GUI with potential passengers, buses, and trains. The application clusters potential passengers (green) by area density, and marks clusters red for insufficiently covered areas. The visitor may configure the query parameters (1), the processing modes (2), and observe query information (3) and runtime statistics about resource utilization (4).

4.3.2 Query Execution. In Section 3, we introduced the NES Optimizer, which produces, and evaluates potential execution plans. Each execution plan contains a mapping between NES operators and nodes in the IoT infrastructure. The optimizer would come up with three execution plans that resemble cloud-based, fog-based, and unified approaches, which we refer to as processing modes. Note that in our demonstration, we use NES to reproduce all processing modes.

The graphs in Figure 3 (2) illustrate the processing modes. Vertices represent sensor, fog and cloud nodes, while edges define the network connections between them. We color filter operations green, and clustering operations blue. In our demonstration, visitors can choose between the following three processing modes.

Cloud Mode: NES places all operations on the cloud, and utilizes the remaining nodes only for data forwarding. Performance metrics will reveal that the main workload gathers in the cloud layer, while resources of intermediate fog nodes remain unused.

Fog Mode: NES places all operations on the intermediate layer. Filtering on the fog reduces network traffic. CPU, and RAM utilization on the fog nodes increases significantly.

NES Mode: NES places filter operations on fog nodes and clustering operations on the cloud. Performance metrics show that network traffic, CPU and RAM utilization remain at moderate levels, since operations are evenly distributed.

In sum, our demonstration showcases how future data management systems allow exploiting all resources of an IoT infrastructure, instead of relying solely on the cloud. Our public transport monitoring system and its underlying topology resemble common IoT application scenarios.

5 CONCLUSION

In this paper, we highlighted data processing challenges in the IoT domain, and described a representative IoT scenario, a public transport system. Our application, a public transport monitoring system, visualizes vehicles and potential passengers on an interactive map, and detects underserved areas. Our application translates user actions on its GUI to NES queries. We offload data management tasks on the IoT infrastructure in different ways, using potential NES execution plans. The visitor can explore these execution plans and observe their implications on resource utilization. Our demonstration shows that existing systems do not address scaling data management on IoT infrastructures.

ACKNOWLEDGMENTS

This work was funded by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref 01IS18037A). The authors would like to thank the NebulaStream team for their insightful comments and fruitful discussions.

REFERENCES

[1] 2019. General Transit Feed Specification. https://gtfs.org/. Accessed: 2019-11-22.
[2] 2019. SUMO - Simulation of Urban Mobility. http://sumo.dlr.de/index.html. Accessed: 2019-11-14.
[3] Flavio Bonomi et al. 2012. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 13–16.
[4] Paris Carbone et al. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).
[5] Martin Ester et al. 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.
[6] Alberto Lerner et al. 2019. The Case for Network Accelerated Query Processing. In *CIDR*.
[7] Samuel R Madden et al. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)* 30, 1 (2005), 122–173.
[8] Dan O'Keeffe et al. 2018. Frontier: Resilient edge processing for the internet of things. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1178–1191.
[9] Jonas Traub et al. 2017. Optimized on-demand data streaming from sensor nodes. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 586–597.
[10] Jonas Traub et al. 2019. SENSE: Scalable Data Acquisition from Distributed Sensors with Guaranteed Time Coherence. *arXiv preprint arXiv:1912.04648* (2019).
[11] Steffen Zeuch et al. 2020. The NebulaStream Platform: Data and Application Management for the Internet of Things. In *CIDR*.

Governor: Operator Placement for a Unified Fog-Cloud Environment

Ankit Chaudhary
TU Berlin
ankit.chaudhary@tu-berlin.de

Steffen Zeuch
DFKI GmbH, TU Berlin
steffen.zeuch@dfki.de

Volker Markl
DFKI GmbH, TU Berlin
volker.markl@tu-berlin.de

ABSTRACT

The processing of geo-distributed data streams is a key challenge for many Internet of Things (IoT) applications. Cloud-based SPEs process data centrally and thus require all data to be present in the cloud before processing. However, this centralized approach becomes a bottleneck for processing data from millions of geo-distributed sensors on a large scale IoT infrastructure. A new line of research extends the centralized cloud with decentralized fog devices to mitigate this bottleneck. One major challenge for an SPE in this unified fog-cloud environment is to fulfill user requirements by placing operators on fog or cloud nodes.

In this demonstration, we introduce *Governor*, an operator placement approach for a unified fog-cloud environment. Our approach consists of the *Governor placement process* and *Governor policies* (GPs). The Governor placement process utilizes heuristic-based GPs to optimize operator placement for a user query. Using GPs, administrators can control the operator placement process to fulfill specific Service-Level-Agreement (SLA). We implement Governor in the NebulaStream Platform (NES), a data and application management system for the IoT. We showcase the impact of GPs on operator placement for different example queries. Our demonstration invites participants to simulate the operator placement of queries and discover their characteristics. This demonstration represents a first step towards an efficient operator placement approach for upcoming IoT infrastructures with millions of sensors and thousands of queries.

1 INTRODUCTION

Over the last decade, the adoption of IoT devices has increased significantly [7]. Processing IoT data in real-time enables a wide range of new opportunities for businesses (e.g., smart homes, connected cars, health-care) [13]. Many IoT applications today are implemented using a cloud-based infrastructure. To perform the data processing, a continuous transfer of geo-distributed IoT data to a centralized data-center is required. Data processing frameworks such as Flink [2] and Spark [12] are designed for cloud-based environments and support efficient data analytics and virtually unlimited scaling. However, cloud-based processing of geo-distributed IoT data presents challenges for real-time and latency-sensitive applications. These challenges include high data transmission costs, delayed data processing, and high demand for cloud-resources [13].

Fog computing addresses these shortcomings by processing data closer to the source devices [1]. In particular, fog computing leverages intermediate compute nodes to perform data processing and thus minimizes data transfer between source IoT devices and cloud servers. However, in contrast to the robust and elastic cloud resources, the fog consists of limited, unreliable, and low-end heterogeneous nodes. On the one hand, the fog can apply in-network processing to reduce large volumes of data. On the other hand, the fog has to cope with unreliable nodes and intermittent failures [11, 13].

The NebulaStream Platform provides data analytics capabilities in a unified fog-cloud environment [13]. The proximity of computing resources to IoT devices in the fog, combined with nearly unlimited computing resources in the cloud, presents novel opportunities for IoT data processing and holistic optimizations. One major challenge in such an environment is the placement of query operators on compute nodes with regards to the unique infrastructure characteristics and specific SLA requirements. Recent work on operator placement in the cloud focuses mainly on network and compute resource efficiency but does not take the volatility and heterogeneity of the fog infrastructure into account [5, 8, 10]. In contrast, approaches for unified fog-cloud environments consider volatility and heterogeneity but only optimize for a specific goal, e.g., network efficiency or fault-tolerance [3, 4, 6]. Furthermore, current approaches do not allow administrators to specify SLA objectives (e.g., high-throughput, low resource consumption) for operator placement.

In this demonstration, we introduce *Governor*, a new fog-cloud operator placement approach that allows specifying custom SLA objectives. Our approach consists of a set of heuristic-based rules called *Governor policies* and a two-phase *Governor placement process*. Using GPs, we enable administrators to tune the Governor placement process for specific SLAs. The two phases of the Governor placement process are the following. First, the *path selection* phase identifies a set of paths between sensors, intermediate compute nodes, and the cloud. Second, the *operator assignment* phase assigns operators to the compute nodes residing on the identified paths. In this demonstration, we showcase our placement approach and the impact of different GPs using five example scenarios. Attendees of our demonstration can compare operator placements of custom queries for different GPs. Overall, our approach allows administrators to guide operator placement using GPs and finds effective operator mappings for large query plans over millions of sensors. In summary, our contributions are as follows:

(1) We introduce Governor for performing operator placement in a unified fog-cloud environment.
(2) We present five Governor Policies for example scenarios.
(3) We present a user interface that allows attendees to study the impact of different GPs on operator placement.

The rest of this paper is structured as follows. In Section 2, we introduce our Governor approach. In Section 3, we present our demonstration scenario and describe the overall system design. We discuss related work in Section 4 and conclude in Section 5.

2 GOVERNOR

Governor consists of Governor Policies and the Governor placement process. We introduce GPs in Section 2.1 and the Governor placement process in Section 2.2. After that, we showcase the

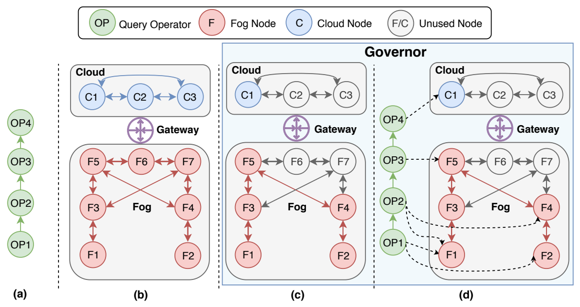


Figure 1: Governor placement process.

application of the Governor approach using example scenarios (see Section 2.3).

2.1 Governor Polices

Governor Policies are a fixed set of heuristic-based rules that guide the operator placement of a query. For example, to place a query with a SLA-objective high fault-tolerance SLA-objective, a GP selects all available network paths between source and sink nodes, and places replicated operators on different nodes along the selected paths. In case of an operator failure, another replica operator will take over the failed operator's workload, thereby achieving the fault-tolerance objective. Using GPs, the operator placement process will optimize the query execution for a specific SLA-objective. An administrator prepares the GP by selecting rules from a predefined catalog, which is maintained and updated by a domain expert. The rules are classified into two categories: *path selection* and *operator assignment*. The path selection rules guide the selection of a subset of all available paths. In contrast, the operator assignment rules guide the placement of operators on nodes on the selected paths. In general, a GP contains at least one rule from each category. However, an administrator can define several GPs, each aiming to optimize query placement for a unique SLA-objective using its distinct set of rules. Additionally, while different GPs can share individual rules, the combination of rules within a GP is unique.

2.2 Governor Placement Process

In this section, we describe how the Governor placement process performs the operator assignments while taking GPs into account. Figure 1 shows the steps necessary to place the operators of a query on a physical infrastructure. First, NES transforms a user query into a directed acyclic graph (DAG) that is composed of query operators and directed links among them, as shown in Figure 1(a). The query DAG Q is represented by $Q = \{O, L\}$ where O and L represent operators and directed links respectively. In the background, NES maintains an infrastructure graph containing compute nodes that are interconnected by network links (see Figure 1(b)). The infrastructure graph G is represented as $G = \{V, E\}$ where V and E represent vertices and edges respectively. Second, NES provides the query DAG, the infrastructure graph, and a GP to Governor for operator placement. Governor uses its placement process to assign the DAG operators on the infrastructure nodes while following the rules from GP.

The operator placement process consists of two phases: the *path selection phase* and the *operator assignment phase*. In the path selection phase, shown in Figure 1(c), Governor identifies the path between sources (IoT device) and the sink (cloud servers) node using the heuristics defined in the GP. The number of paths that are selected in the path selection phase ranges from all paths to just a specific path, and thus reduces the search space for the operator assignment phase. The path selection phase is crucial for large-scale IoT infrastructures containing thousands of heterogeneous nodes and millions of data sources. We make use of depth first search (DFS) algorithm for finding paths. The maximum number of source nodes in G is given by total number of vertices $|V|$. The worst-case runtime complexity for identifying paths between $|V|$ sources and sink node is represented by $O(|V|^3)$.

In the operator assignment phase, shown in Figure 1(d), Governor performs the operator placement on nodes along the selected paths. During the operator assignment phase, Governor distinguishes between *pinned* and *unpinned* operators. Pinned operators reside on a specific location, e.g., source operators on IoT devices and sink operators on cloud servers. In contrast, unpinned operators can be placed on any node along the selected paths (if resource constraints permit this). Common strategies would place non-blocking operators (e.g., filter, source, sink) close to the IoT devices and blocking operators with state (e.g., window, aggregation, join) close to the cloud servers. For a query DAG Q with $|O|$ operators and $|P|$ selected paths with average $|N|$ nodes, the worst-case runtime complexity of operator assignment phase is given by $O(|O| * |N| * |P|)$.

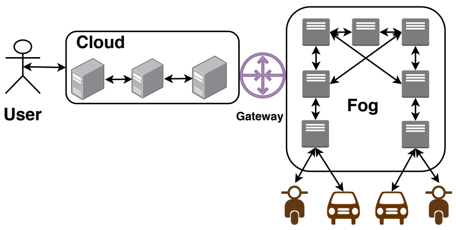


Figure 2: Infrastructure for a ride sharing services.

2.3 Governor in Action

In Figure 2, we show a representative IoT infrastructure for ride-sharing services such as ShareNow[1], WeShare[2], or Coup[3]. Vehicles with on-board computing units communicate with fog computing devices and transmit the operational information (e.g., location, user, or time information) to the fog infrastructure. While data is flowing through the fog into the cloud, fog nodes can apply processing. Users interact with the cloud-based system using a mobile application for locating, renting, or accessing various other services. Using this infrastructure, we present five example application scenarios:

(1) **Fast-response:** Realtime tracking of vehicle fleet.
(2) **Fail-safety:** Billing at the end of a trip.
(3) **Bursty-data:** Monitor vehicle statistics during usage.
(4) **Save-resource:** Health checks on IoT infrastructure.
(5) **Save-energy:** Save energy on battery-operated vehicles.

The presented scenarios have different SLA requirements, which guide their respective placement of operators. In Table 1, we present five example GPs for these scenarios. The *Low-Latency*, *Fault-Tolerance*, and *High-Throughput* GPs are focused primarily on the performance of queries. In contrast, the *Minimum Resource Consumption* and *Minimum Energy Consumption* GPs are focused on the performance of the infrastructure nodes. In the following, we use GPs from Table 1 and briefly discuss the placement process for the five application scenarios.

Fast-response requires fast event processing and delivery by performing early data computation and using low latency network links. The *Low-Latency* GP from Table 1 satisfies this SLA requirement. In the path selection phase, Governor selects distinct paths with low link latencies between source and sink

[1] www.sharenow.com [2] www.weshare.com [3] www.coup.com

	Low-Latency	**Fault-Tolerance**	**High-Throughput**	**Minimum Resource Consumption**	**Minimum Energy Consumption**
Path Selection Phase	• Distinct paths with low link latency	• All paths between sources and sink	• Distinct paths with high bandwidth capacity	• Common path between sources and sink	• Common path between sources and sink
Operator Assignment Phase	• Non-blocking operators closer to source • Replicate operators when possible	• Use shared nodes among selected paths • Replicate operators when possible	• Non-blocking operators closer to source • Blocking operators closer to sink	• Share intermediate operators among different sources • Avoid operator replication	• Non-blocking operators closer to source • Share intermediate operators among different sources • Avoid operator replication

Table 1: Five example Governor policies.

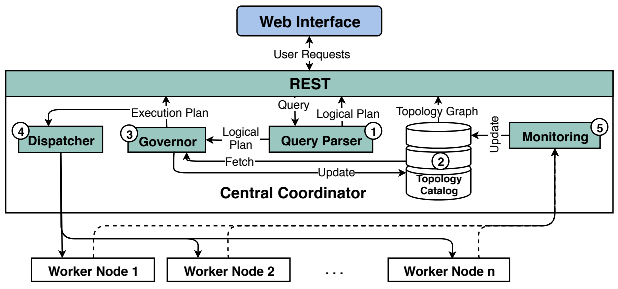


Figure 3: Demonstration system architecture.

nodes. In the operator assignment phase, Governor places all non-blocking operators close to the source operators and replicates the operators wherever possible to achieve high data parallelism [9].

Fail-safety requires events to be delivered irrespective of intermittent network or node failures by using alternative nodes or links for processing and data transfer. The *Fault-Tolerance* GP from Table 1 satisfies this SLA. In the path selection phase, Governor selects all possible paths between the sources and the sink nodes. In the operator assignment phase, Governor places operators on the nodes shared across multiple paths such that in the event of a path failure, another network path can be used for data delivery.

Bursty-data requires the query to handle a sudden burst of events by transmitting data using a high bandwidth link. The *High-Throughput* GP from Table 1 satisfies this SLA. In the path selection phase, Governor selects distinct paths with high bandwidth capacity between source and sink nodes. In the operator assignment phase, Governor places all non-blocking operators close to the source operator and all blocking operators close to the sink operator.

Save-resource requires the compute node to save resources by sharing query operators and preventing replications. The *Minimum Resource Consumption* GP from Table 1 satisfies this SLA. In the path selection phase, Governor selects a common path between source and sink nodes. In the operator assignment phase, Governor ensures a high degree of operator sharing among multiple sources contributing to the query.

Save-energy requires the compute nodes to save power either by reusing the existing query operators or by reducing the amount of data transmitted over the network. The *Minimum Energy Consumption* GP from Table 1 satisfies this SLA. Like for **Save-resource**, Governor selects a common path between source and sink nodes. However, in the operator assignment phase, Governor tries to place non-blocking operators closer to the source to reduce downstream data traffic and thus saving energy for transmission.

Governor uses a greedy approach for the operator placement, which can return a solution in a reasonable amount of time at the cost of sub-optimal placement decisions. In the future work, we will examine the response time-optimality trade-off in detail. Although Governor uses one query plan as an input, this query plan can be the outcome of a multi-query optimization process and thus may contain operators from multiple queries.

3 DEMONSTRATION

In Section 3.1, we introduce the architecture of our demonstration. After that, we present the web interface and describe the functions available for the attendees to explore in Section 3.2.

3.1 Demonstration System Architecture

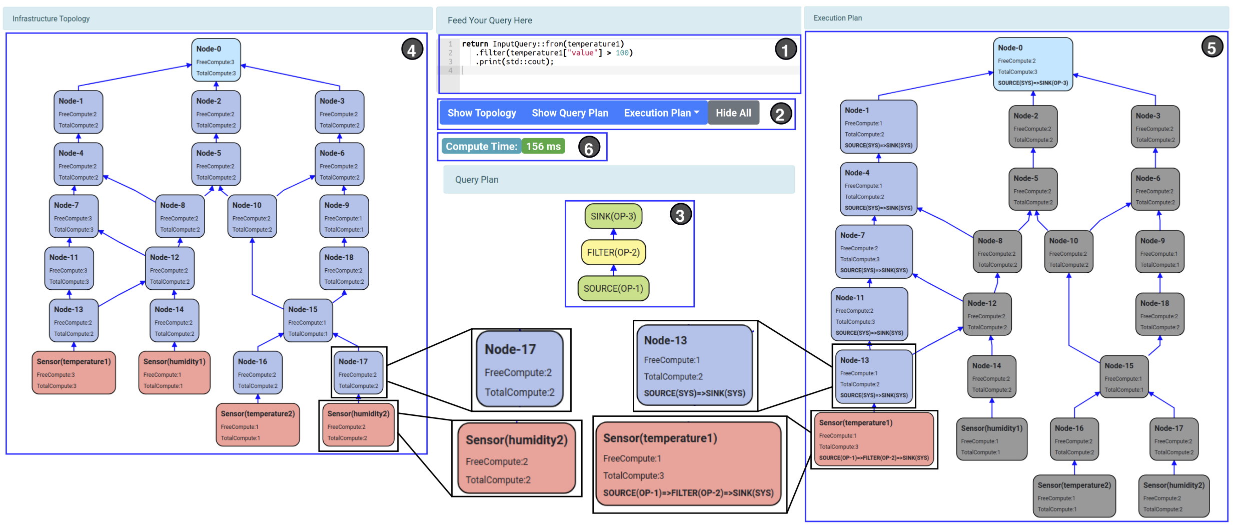
In Figure 3, we present the interaction among the web interface, the central coordinator, and the worker nodes. The web interface submits queries and retrieves information from NES over a REST interface, e.g., the DAG for currently deployed query or the infrastructure topology. The central coordinator node manages both the query placement and the deployment process. The worker nodes manage the execution of operators from different queries. We refer the reader to relevant literature for a detailed design overview of NES [13]. In this section, we will only cover aspects that affect the operator placement process of NES.

Inside the coordinator node, the queries are first validated and parsed into DAGs using the *Query Parser* ①. After that, Governor receives the logical query plan and fetches the information from the *Topology Catalog* ②. The topology catalog maintains the latest infrastructure graph, including information on resource availability and the set of deployed query operators on individual infrastructure nodes. Based on this information, *Governor* ③ creates the execution plan, which contains the operator placement information. Finally, the *Dispatcher* ④ receives the overall execution plan and forwards it to the respective worker nodes.

In an asynchronous second feedback loop, Governor updates the topology catalog with the operator placement information. Furthermore, worker nodes send regular updates to the *Monitoring* ⑤ system, which in turn updates the topology catalog.

3.2 Web Interface

Figure 4 shows the web interface that attendees can use to explore operator placement with Governor. Attendees have to write their queries into the text panel ① to explore different functionalities of the web interface. The central coordinator system from Figure 3 is responsible for parsing, validating, and returning the DAG for the input query. The button-panel ② on the web interface presents various options for interacting with the components of the demonstration system. First, attendees can click on the *Show Topology* button to fetch the infrastructure graph representing the latest state of compute nodes and network connections among them. The display panel ④ shows the returned infrastructure graph. Second, the button *Show Query Plan* triggers the query parser component, which returns a DAG of interconnected operators for the submitted user query in the text panel ①. The display panel ③ shows the query DAG. Third, the drop-down button *Execution Plan* presents a set of GPs for triggering Governor. Attendees can choose between any of the available GPs shown



```
return InputQuery::from(temperature1)
    .filter(temperature1["value"] > 100)
    .print(std::cout);
```

Figure 4: Demonstration system web interface.

in Table 1, i.e., Low-Latency, Fault-Tolerance, High-Throughput, Low Energy Consumption, and Low Resource Consumption. The display panel ⑤ will show the operator placement plan returned by Governor. The operator placement plan contains pinned, unpinned, as well as system-generated operators (e.g., forward operators) and their corresponding mappings on infrastructure nodes. Additionally, the time taken for the plan computation using the selected GP is shown in the UI ⑥.

Overall, attendees of our demonstration can explore the operator placement in a unified fog-cloud infrastructure. We will point out different options and their impact on particular characteristics. Using this demonstration, we present a first step towards an operator placement designed for upcoming IoT infrastructures with millions of sensors and thousands of queries.

4 RELATED WORK

In this section, we categorize related work from different research areas. One line of research covers operator placement approaches for a centralized computing infrastructure. Huang et al. offer a heuristic-based operator placement approach for optimizing with latency and throughput [8]. Kafil et al. consider heterogeneity within a distributed environment to find one optimum compute node to place all operators together [10]. Chatzistergiou et al. optimize the operator placement for inter-node network bottlenecks by reducing the number of nodes required to running a query [5]. In contrast, Governor allows a flexible and controlled operator placement strategy by enabling administrators to specify different optimization objectives within the same infrastructure.

Another line of research examines placement approaches for a unified fog-cloud environment. The approach presented by Veith et al. focuses mainly on minimizing total data processing latency [6]. However, this approach does not consider the node and link volatility within the fog as a factor that can significantly impact query execution. Cardellini et al. present a multi-objective operator placement approach [4]. However, their approach is not optimized for computing operator placements for a large scale IoT infrastructure. In contrast, Governor considers various characteristics of the fog-cloud infrastructure, including the high volatility that is commonly present in a fog infrastructure. Furthermore, Governor's two-phase approach using heuristics reduces the computation time for operator placement.

Overall, none of the previous approaches support defining flexible operator placement objectives. In contrast, Governor allows the administrator to specify different optimization goals using GPs for each query submission.

5 CONCLUSION

In this demonstration, we present Governor, a first step towards operator placement on IoT infrastructures with millions of sensors and thousands of queries. We demonstrate Governor's ability to accept custom Governor Policies with different optimization objectives for operator placement. In addition, we present five example policies and showcase their placement for five example application scenarios. In our demonstration, we will present the challenges as well as early solutions for operator placement in the future IoT era.

6 ACKNOWLEDGEMENT

This work was funded by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref 01IS18037A).

REFERENCES

[1] Bonomi et al. 2012. Fog computing and its role in the internet of things. In *MCC*. ACM.
[2] Carbone et al. 2015. Apache flink: Stream and batch processing in a single engine. *IEEE Computer Society TCDE* 36, 4.
[3] Cardellini et al. 2016. Optimal operator placement for distributed stream processing applications. In *DEBS*. ACM.
[4] Cardellini et al. 2017. Optimal operator replication and placement for distributed stream processing systems. *SIGMETRICS* 44, 4 (2017).
[5] Chatzistergiou et al. 2014. Fast heuristics for near-optimal task allocation in data stream processing over clusters. In *CIKM*. ACM.
[6] da Silva Veith et al. 2018. Latency-aware placement of data stream analytics on edge computing. In *ICSOC*. Springer.
[7] de Assuncao et al. 2018. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications* 103 (2018).
[8] Huang et al. 2011. Operator placement with QoS constraints for distributed stream processing. In *CNSM*. IEEE.
[9] Jaspal et al. 1993. Exploiting task and data parallelism on a multicomputer. In *ACM SIGPLAN Notices*, Vol. 28. ACM.
[10] Kafil et al. 1998. Optimal task assignment in heterogeneous distributed computing systems. *IEEE concurrency* 6, 3 (1998).
[11] O'Keeffe et al. 2018. Frontier: Resilient edge processing for the internet of things. In *VLDB*.
[12] Zaharia et al. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX Association.
[13] Zeuch et al. 2020. The NebulaStream Platform: Data and Application Management for the Internet of Things. In *CIDR*.

Human-in-the-Loop Schema Inference for Massive JSON Datasets

Mohamed-Amine Baazizi
Sorbonne Université, LIP6 UMR 7606
baazizi@ia.lip6.fr

Clément Berti
Sorbonne Université
clement.berti.upmc@gmail.com

Dario Colazzo
Université Paris-Dauphine, PSL Research University
dario.colazzo@dauphine.fr

Giorgio Ghelli
Dipartimento di Informatica, Università di Pisa
ghelli@di.unipi.it

Carlo Sartiani
DIMIE, Università della Basilicata
carlo.sartiani@unibas.it

ABSTRACT

JSON established itself as a popular data format for representing data whose structure is irregular or unknown a priori. JSON collections are usually massive and schema-less. Inferring a schema describing the structure of these collections is crucial for formulating meaningful queries and for adopting schema-based optimizations.

In a recent work, we proposed a Map/Reduce schema inference approach that either infers a compact representation of the input collection or a precise description of every possible shape in the data. Since no level of precision is ideal, it is more appealing to give the analyst the freedom of choosing between different levels of precisions in an interactive fashion. In this paper we describe a schema inference system offering this important functionality.

1 INTRODUCTION

Borrowing flexibility from semistructured data models and simplicity from nested relational ones, JSON affirmed as a convenient and widely adopted data format for exchanging data between applications as well as for exporting data through Web API and/or public repositories. JSON datasets are usually retrieved from remote, uncontrolled sources, with partial, incomplete, or no schema information about the data. In these contexts, however, having a precise description of the structure of the data is of paramount importance, in order to design effective and efficient data processing pipelines. Schema inference, therefore, becomes a crucial operation enabling the formulation of meaningful queries and the adoption of well-known schema-based optimization techniques.

Several approaches and tools exist for inferring structural information from JSON data collections [13–15]. As pointed out in [10, 11], the common aspect of all these approaches is the extraction of some structural description with a precision that is fixed a priori, by the approach itself. While this methodology has the advantage of simplicity, it is in practice not satisfactory, since a JSON dataset can be rather (oftentimes highly) irregular in structure, and for this reason it can be typically described at different precision levels by a schema, while there exists no "best" precision level that can be fixed a priori. In general, one is interested in a description that is compact, easy to read even if it hides lots of details, typically in the first exploration steps, while in subsequent steps he/she is likely to be interested in a more precise, and therefore less succinct, schema description, where more details about the alternative shapes that can be found in the data are provided.

We believe that leaving the user the ability of tuning the level of precision of the inferred schema, by trying different possibilities and changing the level of details at different times, is an important feature, that existing techniques do not provide. With such a motivation in mind, in two recent works [9, 12], we devised, respectively, i) a Map/Reduce-based schema inference technique for massive JSON data that enables the user to choose, a priori, the level of precision of the inferred schema, and ii) a formal system which provides the user with mechanisms to interactively refine/expand the inferred schema, even locally, without the need of re-processing the data multiple times.

The goal of this demonstration is to showcase results and mechanisms provided by these two works, by means of an implementation of the parametric schema inference system [9] which is based on Spark and which interacts with a Web interface that the user can exploit to choose or submit a dataset of interest, and to play with the interactive schema inference process [12].

The user interacts with the system by choosing an existing, already analyzed, dataset, or by submitting a new one. The system initially returns to the user a succinct, but not very precise, schema, and the user then can explore it in order to decide where to get more precision, at several nesting levels: indeed, the user can choose to get a more detailed schema description at a given nesting level, while leaving the inner levels described in a more succinct fashion, hence at a lower degree of precision.

In the remainder of this article, Sections 2 and 3 introduce the parametric [9] and the interactive schema [12] inference techniques, while Section 4 details the architecture supporting our system and the demonstration scenario.

2 PARAMETRIC SCHEMA INFERENCE

The schema inference technique proposed in [9] is based on a Map/Reduce algorithm to ensure scalability. During the *map* phase, an input collection of JSON objects is processed by inferring a schema for each object in the collection. The *reduce* phase produces the final schema by invoking a commutative and associative function whose role is to *merge* the object schemas that are *equivalent*. Deciding whether two schemas are equivalent is a crucial aspect of our approach, as this allows one to choose between different precision levels. We rely on two main equivalence relations (kind equivalence and label equivalence), which we identified to be useful in practice, but our system, which is parametric, allows for using other equivalences defined by the user (see [9] for details).

When using the *kind* equivalence (K), every record type is equivalent to any other record type, and every array type is equivalent to any array type. Hence, this equivalence leads to merging all record types into a single one while indicating for each field whether it is optional or mandatory.

To illustrate, consider the following heterogeneous collection containing three JSON records and one array.

o_1 $\{a : 1, b : 2, d : \{e : 3, f : 4\}\}$
o_2 $\{a : 1, c : 2, d : \{g : 3, h : 4\}\}$
o_3 $\{a : 1, c : 2, d : \{e : 3, f : 4\}\}$
o_4 $[123, \text{"}abc\text{"}, \{a : 10, b : 20\}]$

The *map* phase yields for each value a corresponding schema. Essentially, atomic values are mapped to their corresponding atomic types (numbers to Num, etc), while complex constructs are processed recursively. The potentially heterogeneous content of arrays is concisely represented using the union (+) operator.

$o_1 \longrightarrow s_1 = \{a : \mathsf{Num}, b : \mathsf{Num}, d : \{e : \mathsf{Num}, f : \mathsf{Num}\}\}$
$o_2 \longrightarrow s_2 = \{a : \mathsf{Num}, c : \mathsf{Num}, d : \{g : \mathsf{Num}, h : \mathsf{Num}\}\}$
$o_3 \longrightarrow s_3 = \{a : \mathsf{Num}, c : \mathsf{Num}, d : \{e : \mathsf{Num}, f : \mathsf{Num}\}\}$
$o_4 \longrightarrow s_4 = [\mathsf{Num} + \mathsf{Str} + \{a : \mathsf{Num}, b : \mathsf{Num}\}]$

During the *reduce* phase, equivalent types are merged based on the chosen equivalence relation. The K equivalence merges all record types and yields a union of a record and array type, as follows:

$$\begin{aligned} S_3 = \{\ & a : \mathsf{Num}, b : \mathsf{Num?}, c : \mathsf{Num?}, \\ & d : \{e : \mathsf{Num?}, f : \mathsf{Num?}, g : \mathsf{Num?}, h : \mathsf{Num?}\} \\ \}\ & \\ +\ [\ & \mathsf{Num} + \mathsf{Str} + \{a : \mathsf{Num}, b : \mathsf{Num}\}] \end{aligned}$$

The record type reports all fields appearing in the merged record types from the *map* phase, while indicating whether they are mandatory or optional (this latter fact being indicated by decorating the fields with ?). For instance, a is a mandatory field of type Num, while b, c, and d are optional fields of type Num; furthermore, d values are objects whose fields are all optional.

Notation 2.1 *In the following, when a union schema $s_1 + \ldots + s_n$ is inferred by means of an equivalence $\mathcal{E}$, we will use the prefix notation $+_{\mathcal{E}}(s_1, \ldots, s_n)$, so that the inferred schema indicates which equivalence has been used in order to decide what schemas to merge in the inference process. For readability, we omit the $+_{\mathcal{E}}$ prefix for atomic types when they appear as a singleton.*

So for instance, we note S_3 as

$$\begin{aligned} S_3 = +_K(\ \{\ & a : \mathsf{Num}, b : \mathsf{Num?}, c : \mathsf{Num?}, \\ & d : +_K(\{e : \mathsf{Num?}, f : \mathsf{Num?}, g : \mathsf{Num?}, h : \mathsf{Num?}\}) \\ \}, & \\ [\ \ & +_K(\mathsf{Num},\ \mathsf{Str},\ \{a : \mathsf{Num}, b : \mathsf{Num}\})] \\ ) \end{aligned}$$

Concerning precision, it is worth observing that the above schema hides important correlation information like the fact that b and c never co-occur or the fact that fields e and f always occur together.

To derive a more precise schema, where records having different labels are kept separated, we use the *label* equivalence (L), according to which record types are equivalent only if they share the same top-level field labels. So, by means of the L equivalence only s_2 and s_3 are merged, thus obtaining:

$$\begin{aligned} S_4 = +_L(\ & \{\ a : \mathsf{Num},\ b : \mathsf{Num},\ d : \{e : \mathsf{Num}, f : \mathsf{Num}\}\}, \\ & \{\ a : \mathsf{Num}, c : \mathsf{Num}, \\ & \quad d : +_L(\{e : \mathsf{Num}, f : \mathsf{Num}\},\ \{g : \mathsf{Num}, h : \mathsf{Num}\}\ )\ \}, \\ & [\ +_L(\mathsf{Num}, \mathsf{Str}, \{a : \mathsf{Num}, b : \mathsf{Num}\}\ )] \\ ) \end{aligned}$$

The resulting inferred schema S_4 gives now a very detailed description of the records in the data by sacrificing conciseness.

However, in general, schemas are much larger than those in the above example, and this could be a complication for an analyst who wants a precise description of a specific part of the dataset/schema (for instance one specific record type) without being overwhelmed by a too large schema describing all the rest. In order to overcome this limitation, we show in the next section how inferred schemas can be interactively manipulated by the analyst, by preserving soundness (schemas obtained in the interaction all describe the dataset at hand).

3 INTERACTIVE SCHEMA INFERENCE

The interactive schema inference is very useful when it allows for describing the same data with different levels of precision-succinctness, so that parts of greater interest to the user are described with the finest precision, while parts with lower interest are described in a succinct way. The interactive schema inference proposed in [12] goes into this direction, and to show its effectiveness we illustrate below a possible interaction that the user can perform on the schema inferred from a real-life dataset, crawled from the official NYTimes API [5] and consisting in metadata about articles of the newspaper. This dataset is interesting for our problem since it features many irregularities at several levels, and discovering these irregularities, using a traditional type inference mechanism, may simply become unpractical.

A simplified version of the K type inferred from this dataset is depicted in Figure 1. This schema focuses on the *byline* part which describes the authors of the articles. By examining this schema, the user realizes that almost all the fields are optional and hence, she/he may want to dig deeper to investigate for a potential correlation between the different fields.

$$\begin{aligned} & +_K(\{\ \ docs : \\ & \quad +_K(\{\ \ byline : \\ & \quad\quad +_K(\{\ \ contributor : \mathsf{Str?} \\ & \quad\quad\quad organization : \mathsf{Str?} \\ & \quad\quad\quad original : \mathsf{Str?} \\ & \quad\quad\quad person : [+_K(\{fn : \mathsf{Str?}, \\ & \quad\quad\quad\quad ln : \mathsf{Str?}, \\ & \quad\quad\quad\quad mn : \mathsf{Str?}, \\ & \quad\quad\quad\quad org : \mathsf{Str?}\})\ \ ] \\ & \quad\quad\}) \\ & \quad\}) \\ & \}) \end{aligned}$$

Figure 1: The NYTimes K type.

The type resulting from refining the content of *byline* is depicted in Figure 2 and shows four possible situations which correspond to different combinations of the *contributor*, *organization*, and *original* fields, but, more interestingly, it reveals that the occurrence of *organization* implies that *person* has an empty array, while its absence coincides with the case where *person* contains an array with a record type. This observation can be explained by the fact that, when an article is written by an organization, the *person* field is not relevant and hence it contains an empty array and, conversely, when it is written by persons, the *organization* field is irrelevant and, hence, it just does not appear in the *byline* field.

Now that the user has gained some knowledge about the structure of the *byline* field, she/he may want to explore the *person*

$$
\begin{array}{l}
+_K(\{\quad docs: \\
\qquad +_K(\{\quad byline: \\
\qquad\qquad +_L(\{\quad contributor: \mathtt{Str} \\
\qquad\qquad\qquad organization: \mathtt{Str} \\
\qquad\qquad\qquad original: \mathtt{Str} \\
\qquad\qquad\qquad person: [\,] \\
\qquad\qquad\qquad \}, \\
\qquad\qquad\quad \{\quad contributor: \mathtt{Str} \\
\qquad\qquad\qquad original: \mathtt{Str?} \\
\qquad\qquad\qquad person: [+_K(\{fn: \mathtt{Str?}, \ldots \\
\qquad\qquad\qquad\qquad\qquad \ldots, org: \mathtt{Str?}\quad\})\quad] \\
\qquad\qquad\qquad \}, \\
\qquad\qquad\quad \{\quad organization: \mathtt{Str} \\
\qquad\qquad\qquad original: \mathtt{Str} \\
\qquad\qquad\qquad person: [\,] \\
\qquad\qquad\qquad \}, \\
\qquad\qquad\quad \{\quad original: \mathtt{Str} \\
\qquad\qquad\qquad person: [+_K(\{fn: \mathtt{Str?}, \ldots \\
\qquad\qquad\qquad\qquad\qquad \ldots, org: \mathtt{Str?}\quad\})\quad] \\
\qquad\qquad\qquad \}) \\
\qquad\qquad \}) \\
\qquad \})
\end{array}
$$

Figure 2: The L refinement of the content of *byline*.

field which also contains records with optional fields. The user can recover the original type depicted in Figure 1, then expand the record inside the array resulting in the type that is partially depicted in Figure 3. This type shows different situations whose relevance is left to the discretion of the user.

$$
\begin{array}{l}
[+_L(\\
\qquad \{fn: \mathtt{Str}, ln: \mathtt{Str}, mn: \mathtt{Str}, org: \mathtt{Str}\}, \\
\qquad \{fn: \mathtt{Str}, ln: \mathtt{Str}, org: \mathtt{Str}\}, \\
\qquad \{fn: \mathtt{Str}, org: \mathtt{Str}\} \\
\quad)\ ]
\end{array}
$$

Figure 3: The L refinement of the content of *person*.

4 DEMONSTRATION OVERVIEW

The objective of this demonstration is to help the attendees understand the features and the goals of our schema inference system. To this aim, attendees will be able to:

(i) infer schemas for real-life JSON datasets according to K and L;

(ii) explore the inferred schemas and interactively fine-tune their precision;

(iii) get a concrete representation of the inferred schemas in JSON Schema.

We first describe the architecture of our system as well as the setup of the demo, and then illustrate the demonstration scenarios we propose.

4.1 System Architecture and Setup Details

Our system is based on a web application implemented following the client-server architecture depicted in Figure 4. The web client is used for loading the JSON collection and for performing the interactive schema inference, while the web server is dedicated to storing the collection and to inferring the initial schema. The storage is supported by HDFS while the computation is ensured by Spark, as presented in [9]. The web client and the remote inference engine communicate through a REST API implemented in Python 3 using the Flask [2] library. The API requests from the client are processed by an orchestrator that coordinates between the storage and the inference modules in the server side. This coordination is ensured by API calls using two open source libraries: webHDFS [8] for communicating with the HDFS storage system, and livy [4] for submitting jobs to the Spark engine.

Upon receiving the input collection in JSONLines format [3], through the client, the server will store the collection on the HDFS then infers the L schema, using the Spark engine. The L schema is then sent to the client and used for inferring the K schema. The *schema visualizer* displays the K schema and translates the user actions into corresponding schema operations that are processed by the *schema manager*. These two modules coordinate during all the interaction session to fulfill the user requests.

The web client is implemented in Typescript [7] using the Angular 6 platform [1], which offers many advantages like modularity and the clean separation between the content of a web page and the program modifying its content; the schema inference module of the server is implemented in Scala and is fully described in [9].

A lightweight system showcasing the core features of the full-system is available online at [6]. Differently from the full-system that will be demonstrated at the conference, the lightweight system performs schema inference on the client side and hence, it is limited to processing small size documents.

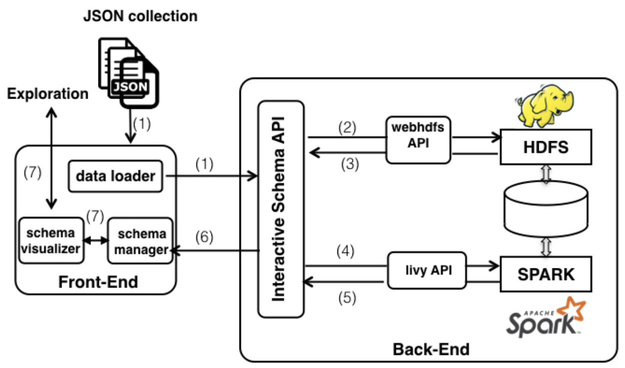


Figure 4: System architecture.

4.2 Demonstration Scenario

Our demonstration enables attendees to infer schemas for preloaded JSON datasets, to provide their own datasets, to explore the extracted schemas, and to fine-tune their precision, as well as to convert them in more popular schema languages like JSON Schema. The datasets we plan to use in our demo are described below.

The GitHub dataset corresponds to metadata generated upon pull requests issued by users willing to commit a new version of code. It takes 14GB of storage and contains 1 million JSON objects sharing the same top-level schema and only varying in their lower-level schema. All objects of this dataset consist exclusively of records nested up to four levels of nesting. Arrays are not used at all.

The Twitter dataset corresponds to metadata that are attached to the tweets shared by Twitter users. It takes 23 GB of storage and contains nearly 10 million records corresponding, in most cases, to tweet entities. A tiny fraction of these records corresponds to a specific API call meant to delete tweets using their ids.

Finally, the NYTimes dataset, which was partly described in Section 3, contains approximately 1.2 million records and reaches the size of 22GB. Its records feature both nested records and arrays, and are nested up to 7 levels. Most of the fields in records are associated to text data which explains the large size of this dataset compared to the previous ones.

Schema Inference. The attendee will start the demo by connecting the web interface to the remote engine, and by selecting a pre-loaded dataset for schema inference; alternatively, the attendee will request the system to load an external dataset by providing a URI. Once selected a dataset, the attendee will choose an inference algorithm to be used by the remote engine. While the focus of this demonstration is on the interactive refinement of K-based schemas, the attendee will also have the opportunity to directly infer a schema according to the L-based approach as well as to get basic statistics about the data (average object size, AST height, etc). Once the inference system has completed the schema extraction process, it will upload the inferred schema to the web interface.

Schema Exploration. After the inference of the K schema, the attendee will likely start to explore this schema through the web interface which allows the user to change the precision level of the schema without any further intervention of the remote inference engine.

Schema Translation. After having explored the inferred schema and (possibly) fine-tuned its precision level, the attendee will be able to translate the schema in a JSON Schema representation; by relying on this feature, the attendee will be able to exploit the schema in any system or application supporting this language, without the need to manually rephrase the schema.

5 RELATED WORK

The problem of inferring structural information from JSON received some attention as reviewed in our recent paper [9], where we outlined the improvements of our approach over state-of-the-art approaches for JSON schema inference, while the topic of interactive JSON schema inference was only recently addressed [12].

In the context of XML, the only work about interactive inference we are aware of relies on user intervention for recognizing regular expressions that are similar enough to be merged and for deriving sophisticated XML schemas expressing complex constructs like inheritance and derivation [16].

REFERENCES

[1] Angular. Available at https://angular.io.
[2] Flask. https://www.flaskapi.org.
[3] JSONLines. http://jsonlines.org.
[4] livy REST API. Available at https://livy.incubator.apache.org.
[5] NYTimes API. https://developer.nytimes.com/.
[6] Online demo. http://132.227.204.195:4200/host.
[7] Typescript. Available at https://www.typescriptlang.org.
[8] webHDFS REST API. Available at https://hadoop.apache.org/.
[9] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Parametric schema inference for massive JSON datasets. *VLDB J.* 28, 4 (2019), 497–521. https://doi.org/10.1007/s00778-018-0532-7
[10] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Schemas And Types For JSON Data. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019.* 437–439. https://doi.org/10.5441/002/edbt.2019.39
[11] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Schemas and Types for JSON Data: From Theory to Practice. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* 2060–2063. https://doi.org/10.1145/3299869.3314032
[12] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. A Type System for Interactive JSON Schema Inference (Extended Abstract). In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece (LIPIcs)*, Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi (Eds.), Vol. 132. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 101:1–101:13. https://doi.org/10.4230/LIPIcs.ICALP.2019.101
[13] Stefanie Scherzinger, Eduardo Cunha de Almeida, Thomas Cerqueus, Leandro Batista de Almeida, and Pedro Holanda. 2016. Finding and Fixing Type Mismatches in the Evolution of Object-NoSQL Mappings. In *Proceedings of the Workshops of the EDBT/ICDT 2016.* http://ceur-ws.org/Vol-1558/paper10.pdf
[14] Peter Schmidt. 2017. mongodb-schema. Available at `https://github.com/mongodb-js/mongodb-schema`.
[15] scrapinghub. 2015. Skinfer. Available at `https://github.com/scrapinghub/skinfer`.
[16] Julie Vyhnanovska and Irena Mlynkova. 2010. Interactive Inference of XML Schemas. In *Proceedings of the Fourth IEEE International Conference on Research Challenges in Information Science, RCIS 2010, Nice, France, May 19-21, 2010.* 191–202.

SQL Query Processing Using an Integrated FPGA-based Near-Data Accelerator in ReProVide

Demo Paper

Lekshmi B.G., Andreas Becher, Klaus Meyer-Wegener, Stefan Wildermann, Jürgen Teich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Erlangen, Germany
{lekshmi.bg.nair,andreas.becher,klaus.meyer-wegener,stefan.wildermann,juergen.teich}@fau.de

ABSTRACT

In this demo, we explain the working of ReProVide, a framework that integrates an on-the-fly reconfigurable FPGA-based SoC architecture with a DBMS for accelerated query processing. For this, a capability-aware optimization that can optimize and partition the queries is demonstrated. This optimization for the hardware is based on its capabilities. Our hardware can also generate statistics of the data while executing a query. In contrast to the existing approaches, this does not have any additional costs in terms of execution time. We will demonstrate how these statistics are later used by the DBMS to select the best plan from the search space using accurate cost values.

1 INTRODUCTION

Current trends in hardware technologies such as manycores, GPUs and field-programmable gate arrays (FPGAs) for data processing, NVRAM and open-channel SSDs for storage solutions and RDMA-capable high-speed network solutions are interesting candidates for the acceleration of database query processing on Big Data. In this scenario, the German DFG Priority Program no. 2037[1] on "Scalable Data Management for Future Hardware", funds research projects to develop new architectural concepts for data management systems that support both current as well as future hardware technologies. Our Reconfigurable data provider (ReProVide) project is one of them. The goal of the ReProVide project is to provide a new FPGA-based smart storage solution together with query optimization techniques, which considers the capabilities of the hardware for the scalable and powerful near-data processing of Big Data. The benefits of FPGA technology are pipelined processing of data at line rate (at least for non-blocking operators), energy-efficiency and speedup through parallelization, as well as the option of dynamic adaptation of the hardware through reconfiguration. In the ReProVide project, a generic FPGA architecture named ReProVide processing unit (RPU) for the efficient processing of database queries is developed. The goal of such an architecture is to serve as intelligent storage system and reconfigurable data (pre-) processing interface between diverse data sources and host systems requesting data from these sources. This can reduce network utilization and host workload as well as saving power to a great extent.

Integrating smart storage like RPUs into a database management system (DBMS) requires novel optimization techniques to (a) decide which operations are worthwhile to assign to the RPU, and which are not (*query partitioning* [2]) and (b) to determine how to map and execute the assigned (sub-)queries or database operators on the RPU (*query placement* [1]). For query partitioning, the DBMS query optimizer must take the characteristics of an RPU system into account. This involves considering which particular operators are actually supported by the RPU as not all operators and data types can be accelerated at line rate on FPGAs. It also includes using novel cost models for estimating the RPU performance which consider hardware reconfiguration overheads and make use of hardware-generated statistics of the data it currently stores. We therefore also investigate a novel hierarchical (multi-level) query-optimization technique where the global optimizer of the DBMS and the architecture-specific local optimizer of the RPU work together to obtain scalable query processing [2]. A bidirectional hint interface (see Fig. 1b) exchanging hints between global and local optimizer enables scalable optimization in both directions. i. e., globally and locally.

In this demo, we show how DBMS and RPU work together to process database queries with the goal of minimizing the overall execution time. We will demonstrate how our techniques abstract away the complex decision space of how to partition and execute queries on heterogeneous hardware. We furthermore will explain the underlying hardware concepts and optimization techniques.

2 SYSTEM OVERVIEW

2.1 ReProVide

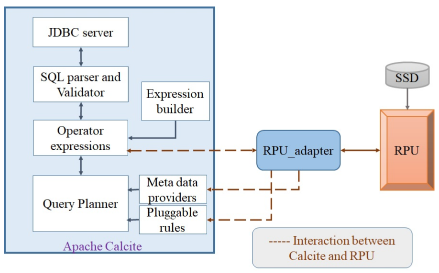
The ReProVide system is a distributed/federated data-processing system which includes Apache Calcite as the DBMS as well as FPGA-based hardware (RPU) as the co-processor and the remote data storage (see Fig. 1a). ReProVide follows the idea of near-data processing, which means the RPU has the query-processing capabilities (near to data storage), that can be used for data (pre-)processing.

Apache Calcite[2] is a dynamic data-management framework that provides query processing, optimization, and query-language support to many data-processing system [5]. It is a complete query-processing system, except for data storage. The framework that Calcite provides for building databases allows to extend its set of rules, cost models, relational expressions, and user-defined functions. This property of Calcite has encouraged us to choose it as our DBMS for this project. Calcite provides data-provider federation through adapters. Hence it is possible to connect any number of databases to Calcite using their own adapter.

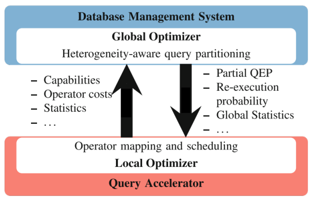
An RPU is implemented on a programmable SoC (multi-core CPU + FPGA) that serves as a data provider accessible via an Ethernet network. The data is stored in non-volatile storage such as solid-state disks (SSDs) and in volatile memory such as DDR-SDRAM, which is directly attached to and managed by the platform and can be processed by accelerators implemented

[1]https://www.dfg-spp2037.de/

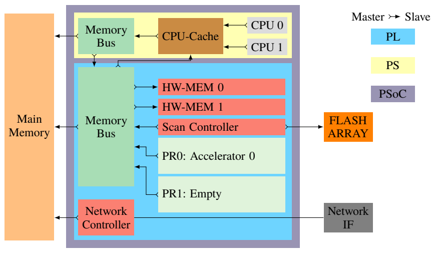
[2]https://calcite.apache.org/



(a) ReProVide architecture



(b) Interplay of Global and Local Optimizer



(c) The architecture of an RPU [1]

Figure 1: (1a) shows the architecture diagram of ReProVide system. (1b) gives the interaction between co-operating optimizers in ReProVide. (1c) illustrates the RPU with the FPGA part (programmable logic (PL)) of the system on chip (SoC) colored blue and the processing system (PS) with its dual core processor colored yellow. Two partially reconfigurable regions (PRs) are available with *PR0* being configured with *Accelerator 0*.

in the Programmable Logic (PL/FPGA) part of the system. This design also enables an RPU to locally optimize the storage layout of the data with regards to availability, access latency, power consumption, and the access patterns of the hardware accelerators.

The interaction of the DBMS and ReProVide is sketched in Fig. 1b. Global optimization applies rules to the query-evaluation plan (QEP), with the major goal of partitioning the QEP into sub-trees and assigning some of them to the RPU (query partitioning). The local optimizer is responsible for finding the best implementation of a sub-tree on the RPU (query placement [1]).

The execution of a query mainly consists of three phases (see Fig. 2) : In the first phase, the local optimizer provides information on its capabilities. The global optimizer uses this information to decide, which operations of a QEP can be assigned to the RPU. Based on this the query will be partitioned. For finding the best partitioning, statistics of the data stored in the RPU can be requested. Generation of such statistics on streaming hardware comes at almost no cost [3] and can even be refined when subsequently accessing the same data (see section 4.2).

In the second phase, the sub-tree of QEP (partition) is passed from the global optimizer to the local optimizer, including some hints (e.g. re-execution probability, upper latency bound, a time budget for running the local optimizer). The local optimizer then selects the required accelerators, maps query operations to these accelerators, and schedules the reconfiguration of the accelerators specific to the operations that it receives. Based on the hints, the local optimizer can adjust buffer sizes for higher throughput but decreased latency until the first data is sent.

In the third and final phase, the query is executed and the results are returned to the host system.

2.2 RPU

The RPU constitutes a heterogeneous hardware/software system, as shown in Figure 1c. The table management is executed on one core of the processing system (PS). While the processing system could also process the full variety of operators and types, its performance may be limited. Thus, handling of the data is mostly dealt with within the programmable logic (PL). Requests are received via a high-speed network interface and forwarded to the management software for further processing. To relieve the processing system from the task of result transmission, a specialized Network Controller implemented in the static system allows sending the resulting data to the requesting host with minimum intervention from the management software.

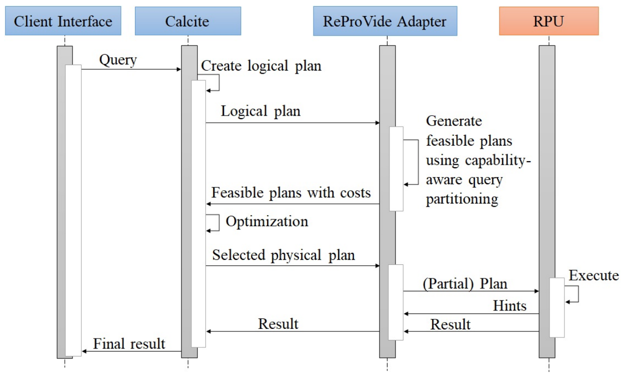


Figure 2: Sequence Diagram for the demonstrated query processing

RPUs include partially reconfigurable regions (PR) into which hardware accelerators can be dynamically loaded and which can be configured to process operations on streaming data. By means of hardware reconfiguration, different hardware accelerators can be loaded onto the FPGA for processing the locally-stored data before transmitting it to the requesting DBMS.

A dedicated and parameterizable hardware component called *Scan Controller* is responsible for data loading. The controller also has direct access to the attached FLASH storage to stream the relevant data to the accelerators. RPUs make use of a library of pre-synthesized reconfigurable hardware accelerators for query placement as the dynamic synthesis of a new hardware accelerator (query compilation) can take from minutes up to multiple hours. As such, the capabilities of a RPU are determined by the accelerators in the library: Operators for which no accelerators are available in the library cannot be executed on the RPU. Operators available span from simple filters on integer values to more complex filters such as regular expressions. Also accelerators for hash-joins have been developed. Please see [4, 15].

2.3 RPU-enabled DBMS

The implemented RPU-enabled DBMS is prototyped using Apache Calcite. We have implemented the *Global Optimizer* as follows.

Calcite allows to extend its rules for query optimization via adapters. The global optimizer must be tuned to act as a capability-aware optimizer by defining some additional rules (push-down

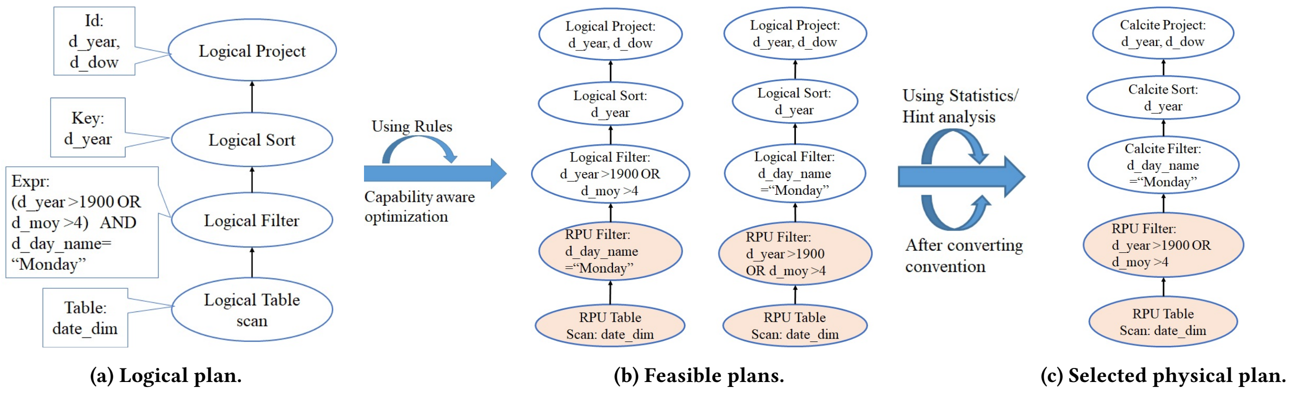

Id: d_year, d_dow
Logical Project
Key: d_year
Logical Sort
Expr: (d_year>1900 OR d_moy>4) AND d_day_name= "Monday"
Logical Filter
Table: date_dim
Logical Table scan
Using Rules
Capability aware optimization
Logical Project: d_year, d_dow
Logical Sort: d_year
Logical Filter: d_year>1900 OR d_moy>4
RPU Filter: d_day_name ="Monday"
RPU Table Scan: date_dim
Logical Filter: d_day_name ="Monday"
RPU Filter: d_year>1900 OR d_moy>4
Using Statistics/ Hint analysis
After converting convention
Calcite Project: d_year, d_dow
Calcite Sort: d_year
Calcite Filter: d_day_name ="Monday"
(a) Logical plan.
(b) Feasible plans.
(c) Selected physical plan.

Figure 3: Query optimization by global optimizer.

rules) to consider the capabilities of RPU during the query optimization. These push-down rules define required operations and attributes which can be pushed down to RPU. The main purpose of the attribute push-down is to avoid unnecessary data transfer over the network and thus to reduce network traffic as well as host workload. These push-down rules are not RPU specific, and can be applied with any hardware configuration connected with Calcite but require the capabilities of the connected co-processor. As an example, consider the query below:

```
SELECT d_year, d_dow
FROM date_dim
WHERE d_day_name="Monday" AND
(d_year > 1900 OR d_moy > 4 )
ORDER BY d_year
```

Fig. 3a. shows the logical plan of the given query. The global optimizer (GO) partitions the query using push-down rules. In this example, the RPU is only capable (a) of doing Filter operations in the Where clause, so the Sort operation cannot be assigned to the RPU, and (b) it can either do String comparison or Integer comparison at a time. Hence, the Filter node is partitioned into a String comparison and an Integer comparison node. Only one can be pushed down to the RPU, so two plans are generated acc. to Fig.3b. Based on cost models and statistics, the GO now selects one alternative and converts it into a physical plan (see Fig. 3c).

3 EVALUATION

Initial measurements of our system were performed on Intel Core i9-7920x processor with CPU 2.90GHz × 24 and Memory 64 GiB. For comparison, PostgreSQL executed on the same system is used as a baseline for comparison. Please note that for the PostgreSQL baseline the data is available locally and, in contrast to our setup, has no network overhead involved. For the ReProVide setup, the RPU is connected via a 1Gb/s Ethernet. Both, the RPU and PostgreSQL are connected to Calcite using their respective software drivers. To evaluate the performance of our proposed system, we chose two different SELECT statements and varied the WHERE clause to test the influence of selectivity. As Table we used the date dimension of the TPC-DS benchmark suite. Figure 4 depicts the relative execution time difference of ReProVide versus PostgreSQL. Due to the run-to-completion implementation of this early version of the RPUs, we can observe a slowdown for the query (blue) with bigger tuple sizes of the result table: Overhead for allocating and synchronizing buffers depends on their sizes.

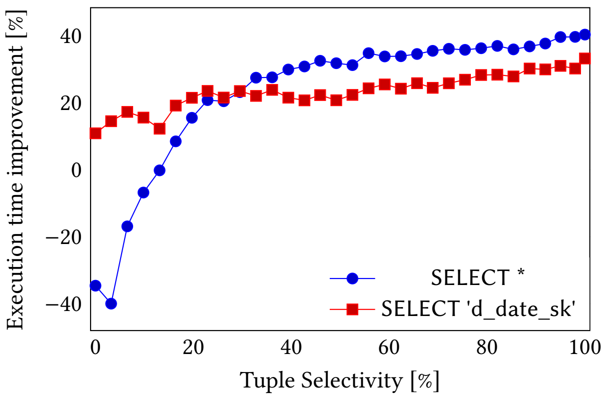


Figure 4: Relative execution time improvement vs. Tuple Selectivity. The improvement is calculated as $\frac{time_{PostgreSQL} - time_{ReProVide}}{time_{PostgreSQL}}$. One query (blue) selects all available attributes of the TPC-DS date dimension while the second query (red) only selects a single attribute.

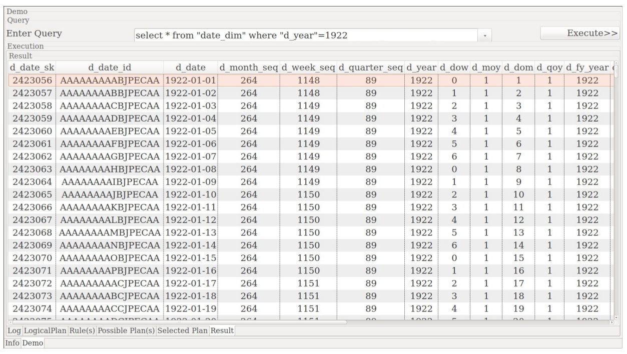
This overhead is greatly reduced if only a small part (columns) will be present in the intermediate result (red) or the selectivity rises and the buffers become less and less oversized. Furthermore, as the RPU is network connected there is an initial latency to send the request and receive the results which PostgreSQL running in the same host doesn't suffer from. As one can see, even with PostgreSQL having these benefits and higher data transfer rates, our approach already shows benefits in terms of execution time reduction of up to 40%.

4 DEMONSTRATION WALKTHROUGH

For the demonstration, we will present all the above mentioned concepts to the conference attendees. They will be explained how to use our framework for executing their own queries. They do not require any prerequisite understanding about any of the hardware or software that we are using for demonstration. The demo includes a laptop where the DBMS (particularly Calcite) is running which will be connected with an FPGA board (RPU) using an Ethernet cable.

4.1 Query partition and execution

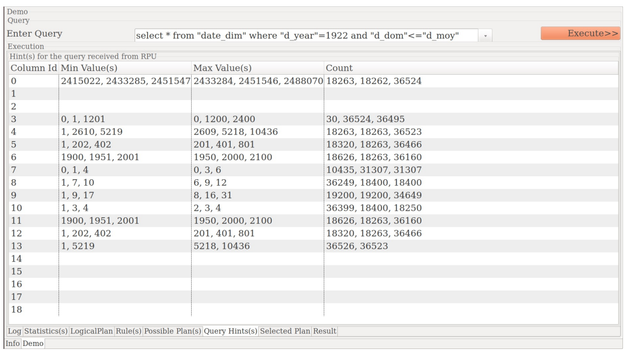
The conference attendees can select/ frame a query that they want to execute in our framework. When the attendee types in a query and presses the execute button, the query execution log will be displayed. The push-down rules are listed and possible execution plans generated by the global optimizer using those push-down rules are visualized. Furthermore, the selected plan for the execution and, finally, the result of the query (as depicted by Fig 5) are shown.



select * from "date_dim" where "d_year"=1922

d_date_sk	d_date_id	d_date	d_month_seq	d_week_seq	d_quarter_seq	d_year	d_dow	d_moy	d_dom	d_qoy	d_fy_year
2423056	AAAAAAAAABJPECAA	1922-01-01	264	1148	89	1922	0	1	1	1	1922
2423057	AAAAAAAABBJPECAA	1922-01-02	264	1148	89	1922	1	1	2	1	1922
2423058	AAAAAAAACBJPECAA	1922-01-03	264	1149	89	1922	2	1	3	1	1922
2423059	AAAAAAAADBJPECAA	1922-01-04	264	1149	89	1922	3	1	4	1	1922
2423060	AAAAAAAAEBJPECAA	1922-01-05	264	1149	89	1922	4	1	5	1	1922
2423061	AAAAAAAAFBJPECAA	1922-01-06	264	1149	89	1922	5	1	6	1	1922
2423062	AAAAAAAAGBJPECAA	1922-01-07	264	1149	89	1922	6	1	7	1	1922
2423063	AAAAAAAAHBJPECAA	1922-01-08	264	1149	89	1922	0	1	8	1	1922
2423064	AAAAAAAAIBJPECAA	1922-01-09	264	1149	89	1922	1	1	9	1	1922
2423065	AAAAAAAAJBJPECAA	1922-01-10	264	1150	89	1922	2	1	10	1	1922
2423066	AAAAAAAAKBJPECAA	1922-01-11	264	1150	89	1922	3	1	11	1	1922
2423067	AAAAAAAALBJPECAA	1922-01-12	264	1150	89	1922	4	1	12	1	1922
2423068	AAAAAAAAMBJPECAA	1922-01-13	264	1150	89	1922	5	1	13	1	1922
2423069	AAAAAAAANBJPECAA	1922-01-14	264	1150	89	1922	6	1	14	1	1922
2423070	AAAAAAAAOBJPECAA	1922-01-15	264	1150	89	1922	0	1	15	1	1922
2423071	AAAAAAAAPBJPECAA	1922-01-16	264	1150	89	1922	1	1	16	1	1922
2423072	AAAAAAAAACJPECAA	1922-01-17	264	1151	89	1922	2	1	17	1	1922
2423073	AAAAAAAABCJPECAA	1922-01-18	264	1151	89	1922	3	1	18	1	1922
2423074	AAAAAAAACCJPECAA	1922-01-19	264	1151	89	1922	4	1	19	1	1922

Figure 5: Final results



select * from "date_dim" where "d_year"=1922 and "d_dom"<="d_moy"

Column Id	Min Value(s)	Max Value(s)	Count
0	2415022, 2433285, 2451547	2433284, 2451546, 2488070	18263, 18262, 36524
1			
2			
3	0, 1, 1201	0, 1200, 2400	30, 36524, 36495
4	1, 2610, 5219	2609, 5218, 10436	18263, 18263, 36523
5	1, 202, 402	201, 401, 801	18320, 18263, 36466
6	1900, 1951, 2001	1950, 2000, 2100	18626, 18263, 36160
7	0, 1, 4	0, 3, 6	10435, 31307, 31307
8	1, 7, 10	6, 9, 12	36249, 18400, 18400
9	1, 9, 17	8, 16, 31	19200, 19200, 34649
10	1, 3, 4	2, 3, 4	36399, 18400, 18250
11	1900, 1951, 2001	1950, 2000, 2100	18626, 18263, 36160
12	1, 202, 402	201, 401, 801	18320, 18263, 36466
13	1, 5219	5218, 10436	36526, 36523
14			
15			
16			
17			
18			

Figure 6: Statistics received from RPU

4.2 Online Statistics Visualization

One speciality of the RPU is the possibility to generate statistics such as histograms at almost zero cost during regular query execution runs [3]. The demo will present the refined histograms after new information is available in an event driven way to demonstrate that statistics can be refined without additional cost (see Fig 6). The statistics can then guide the optimizer to select the better plans with more accurate cost estimations.

5 RELATED WORKS

Research about the benefits of modern hardware, especially FPGAs and GPUs, for query accelerations is well exploited in recent years. [11–14] are studying the performance improvement when integrating FPGAs into CPU systems for query acceleration. Contrary, we are introducing a co-operating optimizer, where FPGA and CPU are interacting. Based on this interaction, a capability-aware optimization is implemented.

In [7] the statistics (histogram) have been generated as a side effect of query processing but with additional hardware resources while in our system, statistics are gathered during the execution of partial query[3].

Heterogeneity-aware query optimization has been studied extensively in [6, 8–10]. But other than the approaches they have implemented, a capability-aware optimization using the hints from the attached co-processor is used in our approach to find the best operations for them.

6 CONCLUSION AND FUTURE WORK

A reconfigurable near-data accelerator and a capability-aware global optimizer for a scalable and energy efficient execution of an SQL query processing is shown in this demo. In the current version, we are focusing on the query partition and on-the-fly reconfiguration of accelerators. In the future work, we would like to address multi-query optimization and common sub-expression evaluation.

Acknowledgements. This work has been supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) as part of the SPP 2037.

REFERENCES

[1] Andreas Becher, Achim Herrmann, Stefan Wildermann, and Jürgen Teich. 2019. ReProVide: Towards Utilizing Heterogeneous Partially Reconfigurable Architectures for Near-Memory Data Processing. In *BTW 2019 – Workshopband*, Holger Meyer, Norbert Ritter, Andreas Thor, Daniela Nicklas, Andreas Heuer, and Meike Klettke (Eds.). Gesellschaft für Informatik, Bonn, 51–70. https://doi.org/10.18420/btw2019-ws-04

[2] Andreas Becher, BG Lekshmi, David Broneske, Tobias Drewes, Bala Gurumurthy, Klaus Meyer-Wegener, Thilo Pionteck, Gunter Saake, Jürgen Teich, and Stefan Wildermann. 2018. Integration of FPGAs in Database Management Systems: Challenges and Opportunities. *Datenbank-Spektrum* 18, 3 (2018), 145–156.

[3] Andreas Becher and Jürgen Teich. 2019. In situ Statistics Generation within partially reconfigurable Hardware Accelerators for Query Processing. In *15th International Workshop on Data Management on New Hardware (DaMoN) Held with ACM SIGMOD/PODS 2019* (2019-07-01/2019-07-01).

[4] Andreas Becher, Daniel Ziener, Klaus Meyer-Wegener, and Jürgen Teich. 2015. A co-design approach for accelerated SQL query processing via FPGA-based data filtering. In *2015 International Conference on Field Programmable Technology, FPT 2015, Queenstown, New Zealand, December 7-9, 2015.* 192–195. https://doi.org/10.1109/FPT.2015.7393148

[5] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 221–230.

[6] Sebastian Breß, Bastian Köcher, Max Heimel, Volker Markl, Michael Saecker, and Gunter Saake. 2014. Ocelot/HyPE: optimized data processing on heterogeneous hardware. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1609–1612.

[7] Zsolt Istvan, Louis Woods, and Gustavo Alonso. 2014. Histograms as a side effect of data movement for big data. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 1567–1578.

[8] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2015. Local vs. Global Optimization: Operator Placement Strategies in Heterogeneous Environments.. In *EDBT/ICDT Workshops*. 48–55.

[9] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2017. Adaptive work placement for query processing on heterogeneous computing resources. *Proceedings of the VLDB Endowment* 10, 7 (2017), 733–744.

[10] Tomas Karnagel, Dirk Habich, Benjamin Schlegel, and Wolfgang Lehner. 2014. Heterogeneity-aware operator placement in column-store DBMS. *Datenbank-Spektrum* 14, 3 (2014), 211–221.

[11] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2013. Flexible query processor on FPGAs. *Proceedings of the VLDB Endowment* 6, 12 (2013), 1310–1313.

[12] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. 2017. Centaur: A framework for hybrid CPU-FPGA databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 211–218.

[13] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 403–415.

[14] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: an intelligent storage engine with support for advanced SQL offloading. *Proceedings of the VLDB Endowment* 7, 11 (2014), 963–974.

[15] Daniel Ziener, Helmut Weber, Jörg-Stephan Vogt, Ute Schürfeld, Klaus Meyer-Wegener, Jürgen Teich, Christopher Dennl, Andreas Becher, and Florian Bauer. 2016. FPGA-Based Dynamically Reconfigurable SQL Query Processing. *ACM Transactions on Reconfigurable Technology and Systems* 9 (2016), 25:1–25:24. https://doi.org/10.1145/2845087

Declarative Languages for Big Streaming Data

A database Perspective

Riccardo Tommasini
University of Tartu
riccardo.tommasini@ut.ee

Sherif Sakr
Unversity of Tartu
sherif.sakr@ut.ee

Emanuele Della Valle
Politecnico di Milano
emanuele.dellavalle@polimi.it

Hojjat Jafarpour
Confluent Inc.
hojjat@confluent.com

ABSTRACT

The Big Data movement proposes data streaming systems to tame *velocity* and to enable *reactive* decision making. However, approaching such systems is still too complex due to the paradigm shift they require, i.e., moving from scalable batch processing to continuous data analysis and pattern detection.

Recently, declarative Languages are playing a crucial role in fostering the adoption of Stream Processing solutions. In particular, several key players introduce SQL extensions for stream processing. These new languages are currently playing a central role in fostering the stream processing paradigm shift. In this tutorial, we give an overview of the various languages for declarative querying interfaces big streaming data. To this extent, we discuss how the different Big Stream Processing Engines (BigSPE) interpret, execute, and optimize continuous queries expressed with SQL-like languages such as `KSQL`, `Flink-SQL`, and `Spark SQL`. Finally, we present the open research challenges in the domain.

KEYWORDS

Stream Processing, Data Stream Management Systems, Complex Event Processing, Streaming SQL

1 GOALS & OBJECTIVES

The world is accelerating; every day, hour, minute, and second the amount of data that we produce grows quicker. Initially, Big Data systems focused on scalable batch processing, and `MapReduce` [12] led the analytics market for more than a decade.

Recently, with the growing interest for (near) real-time insights, we observed a paradigm-shift, i.e., from *data at rest* and post-hoc analyses, to *data-in-motion* and continuous analyses. Thus, Big Data systems evolved to process streams with low latency and high throughput. Nowadays, the state of the art includes many alternative solutions, such as `Storm`, `Flink`, `Spark Structured Streaming`, and `Kafka Streams`) to name the most prominent ones.

Models for continuous data processing have been around for decades [15]. However, the advent of the Big-Data dramatically increased the popularity of data streams in several application domains. For example, developers aim at implementing efficient and effective analytics on massive flows of data for realizing the `Smart X` phenomena (e.g., Smart Home, Smart Hospital, Smart City). Stream processing systems are designed to support a large class of applications in which data are generated from multiple sources and are pushed asynchronously to servers which are responsible for processing them [13].

To facilitate the adoption, initially, most of the big stream processing systems provided their users with a set of API for implementing their applications. However, recently, the need for declarative stream processing languages has emerged to simplify common coding tasks; making code more readable and maintainable, and fostering the development of more complex applications. Thus, Big Data frameworks (e.g., `Flink` [9], `Spark` [3], `Kafka Streams`[1], and `Storm` [19]) are starting to develop their own SQL-like approaches (e.g., `Flink SQL`[2], `Beam SQL`[3], `KSQL`[4]) to declaratively tame data velocity.

In general, declarative languages are extremely useful when writing the optimal solution is harder than solving the problem itself. Indeed, they leverage compilers to catch programming mistakes and automatically transform and optimize code. They fill the gap between expert and normal users. They also increase the acceptance and usability of the system for the end-users. The aim of this tutorial is to provide an overview of the state-of-the-art of the ongoing research and development efforts in the domain of declarative languages for big streaming data processing. In particular, the goals of the tutorial are:

- providing an overview of the fundamental notions of processing streams with declarative languages;
- outlining the process of developing and deploying stream processing applications;
- offering an overview of state of the art for streaming query languages, with a deep-dive into optimization techniques and examples from prominent systems; and
- presenting open research challenges and future research directions in the field.

The content of this tutorial is highly relevant for EBDT-2020 attendees, as it focuses on database-related aspects that concern SQL-like domain-specific languages for stream processing.

2 TUTORIAL PROGRAM OUTLINE

The tutorial introduces the various approaches for declarative stream processing for Big Data. It is centered on providing motivations, models, and optimization techniques related to declarative Stream Processing languages, e.g., `EPL`, `KSQL`, `Flink-SQL` and `Spark Structured Streaming`. In the following, we provide an overview of program followed by a detailed description of the different lectures. Notably, we aim at preparing also practical examples and exercises for the audience to interact with the presented tools.

[1]https://kafka.apache.org/documentation/streams/
[2]https://ci.apache.org/projects/flink/flink-docs-stable/dev/table/sql.html
[3]https://beam.apache.org/documentation/dsls/sql/overview/
[4]https://www.confluent.io/product/ksql/

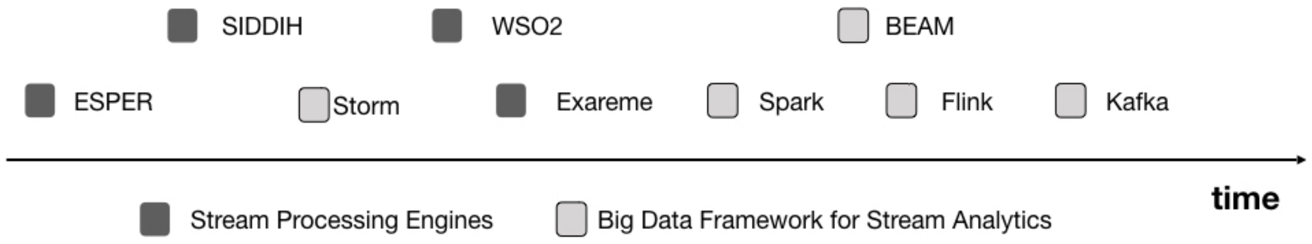

SIDDIH
WSO2
BEAM
ESPER
Storm
Exareme
Spark
Flink
Kafka
time
Stream Processing Engines
Big Data Framework for Stream Analytics

Figure 1: Adoption of Declarative Interfaces for Stream Processing.

(1) A Brief History of Stream Processing and Its Models.
(2) Big Stream Processing Engines
    - Apache Flink [9]
    - Apache Spark [22]
    - Apache Kafka and Kafka Streams [6]
    - Esper[5]
(3) Declarative Languages for Stream Processing
    - The Continuous Query Language [2]
    - Flink SQL (i.e.,Apache Calcite [7])
    - Spark SQL [3]
    - KSQL [14]
(4) Conclusion and Research Directions

2.1 A Brief History of Stream Processing

During this part, we will focus on describing the motivations that led to the development of stream processing [20]. Besides, we present the state of the art, surveying from seminal models [5, 8] to the more recent ones [1, 17].

`SECRET` [8] is a model to explain the operational semantics of window-based SPEs. SECRET focuses on modeling stream-to-relation operators using four abstractions:

- the *Scope* function maps an event time instant to the time interval over which the computation will occur.
- the *Content* function maps a processing time instant to the subset of stream elements that occur during the event-time interval identified by the scope function.
- the *Report* dimension identifies the set of conditions under which the window content become visible to downstream operators.
- the *Tick* dimension shows the conditions that cause a report strategy to be evaluated.

The `Dataflow Model` [1] presented a fundamental shift on the approach of stream processing, leaving the user the choice of the appropriate trade-off between correctness, latency and cost. The model describes the processing model of Google Cloud Dataflow. In particular, it operates on streams of *(key, value)* pairs using the following primitives.

- *ParDo* for generic element-wise parallel processing producing zero or more output elements per input.
- *GroupByKey* for collecting data for a given key before sending them downstream for reduction.

Being data unbounded *GroupByKey* needs windowing in order to be able to decide when it can output. Windowing is usually treated as key modifier, this way GroupByKey will group also by window. The model addresses the problem of window completeness, claiming the Watermarking an insufficient mechanism to regulate out-of-order arrival. Therefore, the Dataflow Model introduces *Triggers* to provide multiple answers for any given window. Windows and Triggers are complementary operators. The former determines *when* data are grouped together for processing using event time; the latter determines *where* the results of groupings are emitted in processing time.

The `Stream and Table Duality` [17] includes three notions, i.e., *table*, *table changelog stream* and a *record* stream. The static view of an operator's state is a *table*, updated for each input record and has a primary key attribute. *Record* streams and *changelog* streams are special cases of streams. A *changelog* stream is the dynamic view of the result of an update operation on a table. The semantics of an update is defined over both keys and timestamps. Replaying a table changelog stream allows to materialize the operator result as a table. On the other hand, a *record stream* represents facts instead of updates. A record streams model immutable items and, thus, each record has a unique key. Stream processing operators are divided in *stateless* and *stateful* ones, may have one or multiple input streams and might be defined over special types of input streams only.

2.2 Big Stream Processing Engines

During this section of the tutorial, we provide an overview of the following Big Stream Processing Engines (BigSPE). Figure 1 reports the system publication timeline, indicating in light-grey to that are in the scope of the tutorial.

`Apache Flink` [9] is an open source platform for distributed stream and batch processing. Flink uses a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations. Apache Flink features two relational APIs - the Table API and SQL - for unified stream and batch processing. Flink's Streaming SQL support is based on Apache Calcite which implements the SQL standard.

`Apache Spark` [3] is a general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. Spark's main abstraction are resilient distributed datasets (RDDs). An RDD is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel.

`Apache Kafka` [21] is a distributed streaming platform. Kafka is run as a cluster on one or more servers, called brokers, that can span multiple datacenters. The Kafka cluster stores streams of records in unbounded append only logs called topics. Each record consists of a key, a value, and a timestamp. *Kafka Streams* is a stream processing library built on top of Apache Kafka producer/consumer APIs. It is based on the Stream/Table duality model, which combines the Dataflow model and CQL.

[5]http://www.espertech.com/

2.3 Declarative Stream Processing Languages

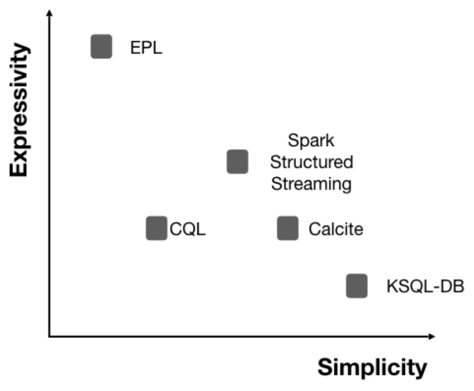


Figure 2: Expressiveness vs Simplicity w.r.t. Declarative Stream Processing Languages

In the context of Big Data Streams, declarative languages suffer from the absence of a shared formal framework that clarifies execution models and time-management approaches. Therefore, existing systems have developed their declarative languages with a primary focus on meeting specific industrial needs.

The `Continuous Query Language` (CQL) [2] is the first attempt to extend relational algebra to process streams of data. CQL defines three families of operators, i.e., Stream-to-Relation (S2R) operators that produce a relation from a stream, Relation-to-Relation (R2R) operators that produce a relation from one or more other relations, and Relation-to-Stream (R2S) operators that produce a stream from a relation. Combining these operators is it possible to define stream-to-stream transformations.

During this section, we survey existing declarative languages for stream processing. Influenced by CQL, BigSPEs' languages try to be as syntactically-close as possible to SQL focusing on usability at the cost of solid design principles like Codd's ones [10]. Following Cugola and Margara's classification [11], we will explain and compare them in terms of expressiveness and simplicity, as shown in Figure 2.

Flink SQL is based on Apache Calcite [7], i.e., a system that provides query parsing, planning, optimization, and execution of SQL queries on top of any data management system. Calcite leaves data storage and management to specialized engines. Flink relies on Calcite to offer a streaming-compliant SQL interface and an advanced query optimization layer. Any language construct used in Flink is also compliant to the SQL standard syntax. Listing 1 shows an example of Flink SQL query that counts the number of views per nation every hour and reporting every minute.

```
SELECT nation, COUNT(*)
FROM pageviews
GROUP BY HOP(rowtime,INTERVAL 1H,INTERVAL 1M), nation
```

Listing 1: Counting Page Views using Flink SQL.

Spark SQL [4] a Spark module for structured data processing. It provides a Dataframe API that can perform relational operations. Moreover, it relies on Catalyst, i.e., an extensible optimizer that allows custom optimization rules. The Dataframe API provides to Catalyst source metadata that allows it to perform optimizations. Structured Streaming [3] ports to Spark some of the Google DataFlow ideas, e.g., the separation between event-time and processing-time. Structured Streaming reuses the Spark SQL execution engine, Catalyst, and the code generator. To support streaming, Structured Streaming add the features to Spark SQL:(1) Triggers to control result reporting; (2) Time extractors to mark a column for event time;(3) Stateful operators to implement complex aggregations. Listing 2 shows the same query of Listing 1, but using Spark SQL.

```
val df = pageviews.groupBy(
  window($"timestamp", "1 hour", "1 minute"),$"nation"
).count()
```

Listing 2: Counting Page Views using Spark SQL.

KSQL [14] is a streaming SQL engine implemented on top of the Kafka Streams API. KSQL is directly based on the Stream Duality model [17]. Thus, it relies on two first-class constructs, i.e., Streams and Tables. To move between these abstractions, KSQL provides powerful stream processing capabilities such as joins, aggregations, event-time windowing, and many more. Listing **??** shows our example pageviews query using KSQL syntax.

```
CREATE TABLE analysis AS SELECT nation, COUNT(*)
FROM pageviews
WINDOW HOPPING (SIZE 1 HOUR, ADVANCE BY 1 MINUTE)
GROUP BY nation;
```

Listing 3: Counting Page Views using KSQL label

The presenters go in-depth regarding the languages above, focusing in particular on the following constructs [18]: (i) Filters and stateless operations; (ii) table-stream and stream-to-stream joins; (iii) windowing, aggregates, and stateful computations.

Finally, presenters will comment on the design of the languages above w.r.t. the Codd's principles:

- *Minimality*, i.e., a language should provide only a small set of needed language constructs so that different language constructs cannot express the same meaning;
- *Symmetry*, i.e., a language should ensure that the same language construct always expresses the same semantics regardless of the context it is used in; and
- *Orthogonality*, i.e., a language should guarantee that every meaningful combination its constructs is applicable.

2.4 Conclusion and Research Directions

This section closes the tutorial indicating ongoing research and industrial works. Moreover, we will ask the audience about the fact that SQL was not intended to be used with unbounded streams of data nor with the continuous semantics required to process them. Nevertheless, existing BigSPE are adopting it and a question naturally raises: *Can we consider BigSPE as databases?*

3 LEARNING OUTCOMES

The tutorial targets researchers, knowledge workers, and practitioners who want to understand the current state-of-the-art as well as the future directions of stream processing.

This tutorial includes relevant technologies and topics for people from IoT, as well as social media, pervasive health, ans oil industry, who have to analyze in massive amounts of streaming data. After attending this tutorial, the audience will have:

- Good understanding of the fundamental and main concepts of stream processing its models;

- An overview and detailed insight into declarative stream processing languages and the ongoing developments by big data streaming system in this domain;
- An overview of research directions in which the state-of-the-art can be improved.

The material of the tutorial will cover some of the work of the presenters on the topics of the tutorial, including [13, 16, 18]

4 PREVIOUS EDITIONS

The first version of this tutorial was presented at the DEBS 2019 conference[6]. It was a full-day tutorial, and it focused on the processing models behind declarative stream processing languages. With the tutorial, the presenters were invited to publish a companion short paper [18] and a website[7].

5 PRESENTERS & ORGANIZERS

Riccardo Tommasini is a research fellow at the University of Tartu, Estonia. Riccardo did his PhD at the Department of Electronics and Information of the Politecnico di Milano. His thesis, titled "Velocity on the Web", investigates the velocity aspects that concern the Web environment. His research interests span Stream Processing, Knowledge Graphs, Logics and Programming Languages. Riccardo's tutorial activities comprise Stream Reasoning Tutorials at ISWC 2017, ICWE 2018, ESWC 2019, and TheWebConf 2019, and DEBS 2019.

Sherif Sakr is the Head of Data Systems Group at the Institute of Computer Science, University of Tartu, Estonia. He received his PhD degree in Computer and Information Science from Konstanz University, Germany in 2007. He is currently the Editor-in-Chief of the Springer Encyclopedia of Big Data Technologies. His research interest include data and information management, big data processing systems, big data analytics and data science. Prof. Sakr has published more than 150 research papers in international journals and conferences. He delivered several tutorials in various conferences including WWW'12, IC2E'14, CAiSE'14, EDBT Summer School 2015, . The 2nd ScaDS International Summer School on Big Data 2016, The 3rd Keystone Training School on Keyword search in Big Linked Data 2017, DEBS 2019 and ISWC 2019.

Emanuele Della Valle is an Assistant Professor at the Department of Electronics and Information of the Politecnico di Milano. His research interests covered Big Data, Stream Processing, Semantic technologies, Data Science, Web Information Retrieval, and Service Oriented Architectures. His work on Stream Reasoning research filed was applied in analysing Social Media, Mobile Telecom and IoT data streams in collaboration with Telecom Italia, IBM, Siemens, Oracle, Indra, and Statoil. Emanuele presented several Stream Reasoning related tutorials at SemTech 2011, ESWC 2011, ISWC 2013, ESWC 2014, ISWC 2014, ISWC 2015, ISWC 2016, DEBS 2016, ISWC 2017 and KR 2018.

Hojjat Jafarpour is a Software Engineer and the creator of KSQL at Confluent. Before joining Confluent he has worked at NEC Labs, Informatica, Quantcast and Tidemark on various big data management projects. Hojjiat earned his PhD in computer science from UC Irvine, where he worked on scalable stream processing and publish/subscribe systems.

[6]http://debs2019.org/

[7]http://streaminglangs.io/

Acknowledgements.

The work of Sherif Sakr is funded by the European Regional Development Funds via the Mobilitas Plus programme (grant MOBTT75).

The work of Riccardo Tommasini is funded by the European Regional Funds through IT Academy programme.

REFERENCES

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[3] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *SIGMOD*, 2018.

[4] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in spark. In *SIGMOD Conference*, pages 1383–1394. ACM, 2015.

[5] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS*, pages 1–16. ACM, 2002.

[6] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. One SQL to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables. In *SIGMOD Conference*, pages 1757–1772. ACM, 2019.

[7] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD Conference*, pages 221–230. ACM, 2018.

[8] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura M. Haas, Renée J. Miller, and Nesime Tatbul. SECRET: A model for analysis of the execution semantics of stream processing systems. *PVLDB*, 3(1):232–243, 2010.

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[10] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[11] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, 2012.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Martin Hirzel, Guillaume Baudart, Angela Bonifati, Emanuele Della Valle, Sherif Sakr, and Akrivi Vlachou. Stream processing languages in the big data era. *SIGMOD Record*, 47(2):29–40, 2018.

[14] Hojjat Jafarpour and Rohan Desai. KSQL: streaming SQL engine for apache kafka. In *EDBT*, pages 524–533. OpenProceedings.org, 2019.

[15] Shanmugavelayutham Muthukrishnan et al. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2):117–236, 2005.

[16] Sherif Sakr. *Big data 2.0 processing systems: a survey*. Springer, 2016.

[17] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. Streams and tables: Two sides of the same coin. In *BIRTE*, pages 1:1–1:10. ACM, 2018.

[18] Riccardo Tommasini, Sherif Sakr, Marco Balduini, and Emanuele Della Valle. An outlook to declarative languages for big streaming data. In *DEBS*, pages 199–202. ACM, 2019.

[19] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[20] Emanuele Della Valle, Stefano Ceri, Frank van Harmelen, and Dieter Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6):83–89, 2009.

[21] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with apache kafka. *PVLDB*, 8(12):1654–1655, 2015.

[22] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.

Entity Resolution: Past, Present and Yet-to-Come
From Structured to Heterogeneous, to Crowd-sourced, to Deep Learned

George Papadakis
National and Kapodistrian
University of Athens, Greece
gpapadis@di.uoa.gr

Ekaterini Ioannou
University of Tilburg, Netherlands
Ekaterini.Ioannou@uvt.nl

Themis Palpanas
University of Paris, France
themis@mi.parisdescartes.fr

ABSTRACT

Entity Resolution (ER) lies at the core of data integration, with a bulk of research focusing on its effectiveness and its time efficiency. Most past relevant works were crafted for addressing Veracity over structured (relational) data. They typically rely on schema, expert and external knowledge to maximize accuracy. Part of these methods have been recently extended to process large volumes of data through massive parallelization techniques, such as the MapReduce paradigm. With the present advent of Big Web Data, the scope moved towards Variety, aiming to handle semi-structured data collections, with noisy and highly heterogeneous information. Relevant works adopt a novel, loosely schema-aware functionality that emphasizes scalability and robustness to noise. Another line of present research focuses on Velocity, i.e., processing data collections of a continuously increasing volume.

In this tutorial, we present the ER generations by discussing past, present, and yet-to-come mechanisms. For each generation, we outline the corresponding ER workflow along with the state-of-the-art methods per workflow step. Thus, we provide the participants with a deep understanding of the broad field of ER, highlighting the recent advances in crowd-sourcing and deep learning applications in this active research domain. We also equip them with practical skills in applying ER workflows through a hands-on session that involves our publicly available ER toolbox and data.

1 GOALS AND OBJECTIVES

Entity profiles assemble valuable information about real-world objects. Hence, entities constitute the core organizational unit of *structured* (e.g., relational databases) as well as *semi-structured data* (e.g., knowledge bases, such as DBPedia and Geonames). Various data management applications, such as query answering [47], are based on entity semantics and connections in order to improve their performance. Typically, these applications require the integration of different profiles that pertain to the same real-world object [11, 18]. The task of inter-linking and deduplicating (i.e., canonicalizing) data instances that describe the same real-world objects is called *Entity Resolution* (ER) [12].

ER is a relatively old problem that was mainly crafted for structured data, which were described by schemata of known semantics and quality [11]. This schema knowledge allowed experts to develop customized solutions that effectively addressed **Veracity**, i.e., the various forms of inconsistencies, noise or errors in entity profiles, which are introduced during manual data entry, or by the limitations of the automatic extraction techniques [23]. For even higher effectiveness, labelled instances are also typically used in order to automatically learn matching rules that simultaneously maximize precision and recall [48, 60, 61].

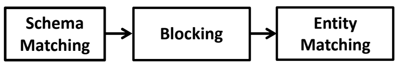


Figure 1: The workflow of the 1st and 2nd ER generations.

The *end-to-end* workflow implemented by **the 1st generation** of ER solutions is depicted in Figure 1 [11]. The first step, *Schema Matching*, creates mappings between the attributes of the input entities based on their relatedness, as inferred from the similarity of their structure, name and/or values [6, 45]. By identifying semantically identical attributes (e.g., “profession” and “job”), it facilitates the schema-aware functionality of the subsequent workflow steps.

The second step, which is called *Blocking*, addresses the quadratic time complexity, $O(n^2)$, of brute-force ER, which compares every entity profile with all others [11]. Blocking reduces the executed comparisons to a significant extent by sacrificing recall to a minor extent. It restricts the computational cost by comparing only the most similar entity profiles, as they are determined by signatures that are composed of (combinations of) parts of values that correspond to the most informative attribute names [11]. E.g., two person entities are likely matches if their addresses have the same zip code.

The entities that co-occur in at least one block are compared during the third step, which is called *Entity Matching*. This applies a combination of string similarity measures to the values of selected attribute names. The resulting degree of similarity is then used to assign the entity pairs into one of the three possible categories, i.e., `match`, `non-match` or `uncertain` [11]. In case of collective approaches, the latest decision is propagated to *neighboring entities*, i.e., entities connected with important relationships to the compared pair, so as to refine their matching likelihood [7, 16].

Note that each step accommodates both *learning-based* and *non-learning methods* [41]; the former methods leverage labelled instances to extract effective rules through a Machine Learning algorithm, while the latter methods rely on heuristics that capture expert or domain knowledge.

The same workflow lies at the core of **the 2nd generation**, which additionally targets **Volume**, i.e., the cases where the input data comprise (tens of) millions of entity profiles. Typically, this challenge is addressed through the new paradigm for *massive parallelization*, i.e., Map/Reduce [14]. Several techniques for Blocking [38] and Entity Matching [9] have been adapted to MapReduce so that they scale to voluminous datasets. Special care is also taken to avoid underutilization of the computational resources through Load Balancing techniques [39, 77].

A shift was marked by **the 3rd generation** of the ER end-to-end workflow, which is depicted in Figure 2. In addition to Veracity and Volume, its goal is to address **Variety**, which is caused by the unprecedented levels of schema heterogeneity and noise as well as the loose schema binding of unclear semantics [12, 17]. Instead of a database-like schema, there is a rich diversity of

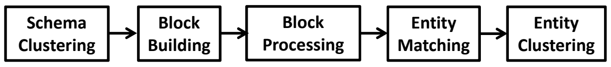


Figure 2: The workflow of the 3rd ER generation.

the domains. For example, there are ∼2,600 different vocabularies in the LOD cloud but only 109 from them are shared by more than one entity collection [22]. This results in hundreds, or even thousands, of different attributes with high entity frequency, rendering inapplicable the schema-aware methods of the first two generations [31, 54].

The first step in the new workflow is *Schema Clustering*, which clusters together attributes with similar values, regardless of their semantics. The goal is to improve the performance of the subsequent steps. E.g., Blocking uses the created schema clusters and the associated signatures (i.e., blocking keys) to split large blocks into smaller ones. This significantly enhances precision for a negligible (if any) impact on recall. This idea has been successfully applied to Blocking via Attribute Clustering [52] and to Meta-blocking via BLAST [62].

The second step, which is called *Block Building*, creates a set of blocks by disregarding schema knowledge and the ensuing human intervention completely. Through a schema-agnostic approach that leverages redundancy, it is inherently crafted for tackling the unprecedented levels of schema heterogeneity in semi-structured data. In this way, it yields blocks of very high recall, but very low precision, independently of human intervention and domain/expert knowledge [12, 54].

The third step of the workflow is *Block Processing*, which enhances precision to a significant extent at a limited, if any, cost in recall [52, 54, 56]. To this end, it refines the original blocks by efficiently removing comparisons that are repeated or involve non-matching entities. Its techniques are distinguished into two categories: the *Block Cleaning* ones operate at the coarse-grained level of entire blocks (e.g., Block Clustering [26]), while the *Comparison Cleaning* ones operate at the fine-grained level of individual comparisons (e.g., Meta-blocking [18, 53] and Blast [62]). In both cases, all techniques are generic and schema-agnostic by definition, thus applying naturally to both structured and semi-structured data [56].

Subsequently, *Entity Matching* executes all comparisons contained in the final set of blocks. Typically, this process depends heavily on neighbor similarity, using the entity relations in the semi-structured data. This is done through an iterative process that discovers duplicate entities gradually and propagates the latest matches to related entities that could benefit from them [42, 44, 66]. This step can also consider probabilistic matching of the entities, e.g., [1, 36].

The end result of Entity Matching is a *similarity graph*, which conveys a node for every entity and a weighted edge for every pair of entities that have been compared. This intermediate model is transformed into the final outcome of ER by *Entity Clustering* [34], which partitions the graph nodes into equivalence clusters - every cluster contains all duplicate entity profiles that actually correspond to the same real-world object. These techniques are schema-agnostic by default, as they exclusively consider the information contained in the similarity graph.

The 4th generation of ER goes beyond the previous ones, by also addressing **Velocity**. This pertains to the continuously increasing volume of available data that imposes special ER challenges, e.g., the data set can never be considered as final, and incoming data might alter the existing ones. To address them,

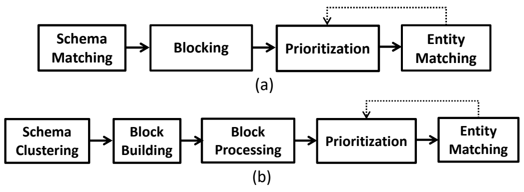


Figure 3: The end-to-end workflow of the 4th ER generation for (a) structured and (b) semi-structured data.

Progressive ER produces useful results in a pay-as-you-go manner before the full completion of ER. Figures 3(a) and (b) illustrate the schema-aware [58] and the schema-agnostic [63] workflows. *Incremental ER* [33] is another mechanism, which minimizes the cost for updating the existing results when new evidence becomes available. Some methods also consider that the new evidence might be conflicting with already processed data [36]. Another mechanism is *Query-driven ER* [4], which gradually resolves entities that are returned as results to incoming queries. A different mechanism is supporting queries for obtaining aggregate, statistical insights about the collection of resulting entities [35].

Note that all generations can be upgraded by exploiting **external knowledge** to achieve higher performance. To this category fall *Deep Learning* techniques for ER [20, 48], which incorporate contextual information in the form of word- or character-embeddings, and *Crowd-sourced ER* [13, 15, 24, 30, 32, 67–70, 72, 75], which relies on human feedback. These two approaches lie at the focus of the latest breakthroughs in ER.

After examining the four ER generations, our tutorial proceeds with a hands-on session that focuses on the state-of-the-art tools for end-to-end Entity Resolution, like Magellan [40]. We then present *JedAI* [57], which constitutes a comprehensive, open-source toolkit that implements most of the state-of-the-art methods for every step of the 3rd and 4th ER generations. Thus, it enables users to build versatile workflows on-the-fly and can be readily used both for experimentation and for integration in Entity Resolution applications. It is distributed under the Apache License 2.0 and, thus, it is suitable for both the academic and the commercial domain.

Overall, our tutorial provides researchers with a complete coverage of the state-of-the-art ER methods along with a discussion of the main open research problems. Practitioners get a good overview of the benefits of the primary ER methods and learn how to use them to improve the productivity of their businesses. They also learn to identify the methods or products that are more suitable for a particular task at hand, or better fit their general needs. Additionally, the audience and especially the developers of information integration tools benefit from the hands-on session, learning how to integrate (parts of) the JedAI Toolkit into their applications. Developers also become acquainted with novel ideas that could well improve their existing products.

Related Tutorials. Our tutorial provides for the first time a novel holistic and systematic view of the evolution of ER, stressing the current state-of-the-art in deep learning and crowd-sourcing applications. We categorize the main ER methods into four generations, going from those crafted for maximizing Veracity over structured data, all the way to those tackling Veracity, Volume, Variety and Velocity over semi-structured data. No other tutorial covers comprehensively large-scale, end-to-end ER for both structured and semi-structured data. Past tutorials on the subject [17, 27, 29, 65] focus either on one of these data types, or cover partially the end-to-end ER workflow.

2 SCOPE AND COVERAGE

Our tutorial aims to provide an overview of the state-of-the-art techniques for all generations of End-to-End ER, analyzing each one in a different session of ∼10 minutes. More emphasis is devoted to the approaches leveraging external knowledge in order to upgrade any workflow step in any generation (∼30 minutes), while a hands-on session discusses the main ER tools and demonstrates the latest version of JedAI (∼10 minutes). Together with the introduction, 5 minutes for questions and the conclusions, the intended duration of the tutorial is *1.5 hours.* The content of the individual sessions is outlined below:

I. Introduction and motivation
- Preliminaries on Entity Resolution [12, 18]
- Fundamental Assumptions, Principles and Definitions [23]

II. The 1st ER Generation: Tackling Veracity
- Schema Matching [6, 19]
- Blocking [11, 37, 61]
- Entity Matching [5, 16, 60]

III. The 2nd ER Generation: Tackling Volume and Veracity
- Parallel Blocking [38]
- Parallel Entity Matching [59]
- Load Balancing [39, 77]

IV. The 3rd ER Generation: Tackling Variety, Volume and Veracity
- Schema Clustering [52, 62]
- Block Building [50–52]: Parallel Methods [12]
- Block Processing [8, 26, 55, 56, 62]: Parallel Methods [21]
- Entity Matching [42, 44, 66]: Parallel Methods [9, 22]
- Entity Clustering [34]

V. The 4th ER Generation: Tackling Velocity, Variety, Volume and Veracity
- Progressive ER for (Semi-)Structured Data [58, 63, 76]
- Incremental Entity Resolution [33, 74]
- Query-Driven Entity Resolution [2–4, 71]
- Query Analytics for Entity Resolution [35, 64].

VI. Entity Resolution Revisited: Leveraging External Knowledge
- Deep Learning for Entity Resolution [20, 48]
- Crowd-sourced Entity Resolution:
  - Generating HITs [15, 43, 70]
  - Formulating HIts [25, 67–69, 72, 75]
  - Balancing accuracy and monetary cost [10, 28, 73, 78]
  - Restrict the labour cost [13, 30]

VII. Hands-on Session: ER tools
- The state-of-the-art end-to-end ER tools [40]
- The JedAI Open Source Toolkit [57]

VIII. Challenges and Final Remarks
- Automatic Parameter Configuration [46, 49]
- Multi-modal Entity Resolution
- Conclusions

3 INTENDED AUDIENCE AND MATERIAL

Our tutorial is example-driven, avoiding excessive technical details and proofs. As a result, there is no prerequisite knowledge, apart from a basic understanding of data management technology. This renders it suitable for a broad audience, covering not only students and researchers, but also practitioners and developers. In other words, it is intended for anyone with an interest in understanding the main techniques for scalable and robust end-to-end Entity Resolution over structured and semi-structured data, using both non-learning and learning-based techniques.

In addition to the theoretical background in the state-of-the-art in the field, the tutorial also presents available entity-related resources, enabling the participants to directly work on the particular domain. Discussed resources include available data as well as the state-of-the-art tools for performing end-to-end Entity Resolution, like Magellan [40] and JedAI [57], which can be readily used to tackle ER problems via numerous combinations of the most prominent methods.

Tutorial Material. The material of the tutorial is distributed through the conference website[1] as well as through a dedicated website[2]. In both locations, we also give pointers and guidelines for the ER toolkit that is used during the hands-on session. All relevant code is publicly released through the Apache License 2.0, which supports both academic and commercial uses.

4 PRESENTERS

The tutorial is given by three presenters:

(1) *George Papadakis* is a Research Fellow at the Department of Informatics of the University of Athens, Greece, and an Internal Auditor of Information Systems at the Public Power Company, the main electricity company in Greece.
(2) *Ekaterini Ioannou* is an Assistant Professor at the University of Tilburg, Netherlands.
(3) *Themis Palpanas* is a Senior Member of the French University Insitute (IUF), and a Professor of Computer Science at the University of Paris, France.

All authors have published papers related to Entity Resolution, focusing on the efficient management of large data collections as well as on addressing various challenges, such as uncertainty, volatility, and correlations.

Acknowledgements. This work was partially funded by the EU H2020 project ExtremeEarth (825258).

REFERENCES

[1] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha U. Nabar, Tomoe Sugihara, and Jennifer Widom. 2006. Trio: A System for Data, Uncertainty, and Lineage. In *VLDB*. 1151–1154.
[2] Hotham Altwaijry, Dmitri Kalashnikov, and Sharad Mehrotra. 2013. Query-Driven Approach to Entity Resolution. *PVLDB* 6, 14 (2013), 1846–1857.
[3] Hotham Altwaijry, Dmitri Kalashnikov, and Sharad Mehrotra. 2017. QDA: A Query-Driven Approach to Entity Resolution. *TKDE* (2017).
[4] Hotham Altwaijry, Sharad Mehrotra, and Dmitri V. Kalashnikov. 2015. QuERy: A Framework for Integrating Entity Resolution with Query Processing. *PVLDB* 9, 3 (2015), 120–131.
[5] Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven Euijong Whang, and Jennifer Widom. 2009. Swoosh: a generic approach to entity resolution. *VLDB J.* 18, 1 (2009), 255–276.
[6] Philip A. Bernstein, Jayant Madhavan, and Erhard Rahm. 2011. Generic Schema Matching, Ten Years Later. *PVLDB* 4, 11 (2011), 695–701.
[7] Indrajit Bhattacharya and Lise Getoor. 2007. Collective entity resolution in relational data. *TKDD* 1, 1 (2007), 5.
[8] Guilherme Dal Bianco, Marcos André Gonçalves, and Denio Duarte. 2018. BLOSS: Effective meta-blocking with almost no effort. *Inf. Syst.* 75 (2018), 75–89.
[9] Christoph Böhm, Gerard de Melo, Felix Naumann, and Gerhard Weikum. 2012. LINDA: distributed web-of-data-scale entity matching. In *CIKM*. 2104–2108.
[10] Chengliang Chai, Guoliang Li, Jian Li, Dong Deng, and Jianhua Feng. 2018. A partial-order-based framework for cost-effective crowdsourced entity resolution. *VLDB J.* 27, 6 (2018), 745–770.
[11] Peter Christen. 2012. *Data Matching.* Springer.
[12] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. 2015. *Entity Resolution in the Web of Data.* Morgan & Claypool Publishers.
[13] Sanjib Das, Paul Suganthan G. C., AnHai Doan, Jeffrey F. Naughton, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, Vijay Raghavendra, and Youngchoon Park. 2017. Falcon: Scaling Up Hands-Off Crowdsourced Entity Matching to Build Cloud Services. In *SIGMOD.* 1431–1446.
[14] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[1]https://diku-dk.github.io/edbticdt2020
[2]https://research.tilburguniversity.edu/en/projects/4ger

[15] Gianluca Demartini, Djellel Eddine Difallah, and Philippe Cudré-Mauroux. 2013. Large-scale linked data integration using probabilistic reasoning and crowdsourcing. *VLDB J.* 22, 5 (2013), 665–687.

[16] Xin Dong, Alon Y. Halevy, and Jayant Madhavan. 2005. Reference Reconciliation in Complex Information Spaces. In *SIGMOD*. 85–96.

[17] Xin Luna Dong and Divesh Srivastava. 2013. Big Data Integration. *PVLDB* 6, 11 (2013), 1188–1189.

[18] Xin Luna Dong and Divesh Srivastava. 2015. *Big Data Integration.* Morgan & Claypool Publishers.

[19] Songyun Duan, Achille Fokoue, Oktie Hassanzadeh, Anastasios Kementsietsidis, Kavitha Srinivas, and Michael J. Ward. 2012. Instance-Based Matching of Large Ontologies Using Locality-Sensitive Hashing. In *ISWC*. 49–64.

[20] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq R. Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed Representations of Tuples for Entity Resolution. *PVLDB* 11, 11 (2018), 1454–1467.

[21] Vasilis Efthymiou, George Papadakis, George Papastefanatos, Kostas Stefanidis, and Themis Palpanas. 2017. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Inf. Syst.* (2017).

[22] Vasilis Efthymiou, George Papadakis, Kostas Stefanidis, and Vassilis Christophides. 2019. MinoanER: Schema-Agnostic, Non-Iterative, Massively Parallel Resolution of Web Entities. In *EDBT*. 373–384.

[23] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate Record Detection: A Survey. *TKDE* 19, 1 (2007), 1–16.

[24] Donatella Firmani, Sainyam Galhotra, Barna Saha, and Divesh Srivastava. 2018. Robust Entity Resolution Using a CrowdOracle. *IEEE Data Eng. Bull.* 41, 2 (2018), 91–103.

[25] Donatella Firmani, Barna Saha, and Divesh Srivastava. 2016. Online Entity Resolution Using an Oracle. *PVLDB* 9, 5 (2016), 384–395.

[26] Jeffrey Fisher, Peter Christen, Qing Wang, and Erhard Rahm. 2015. A Clustering-Based Framework to Control Block Sizes for Entity Resolution. In *KDD*. 279–288.

[27] Avigdor Gal. 2014. Tutorial: Uncertain Entity Resolution. *PVLDB* 7, 13 (2014), 1711–1712.

[28] Sainyam Galhotra, Donatella Firmani, Barna Saha, and Divesh Srivastava. 2018. Robust Entity Resolution using Random Graphs. In *SIGMOD*. 3–18.

[29] Lise Getoor and Ashwin Machanavajjhala. 2012. Entity Resolution: Theory, Practice & Open Challenges. *PVLDB* 5, 12 (2012), 2018–2019.

[30] Chaitanya Gokhale, Sanjib Das, AnHai Doan, Jeffrey F. Naughton, Narasimhan Rampalli, Jude W. Shavlik, and Xiaojin Zhu. 2014. Corleone: hands-off crowdsourcing for entity matching. In *SIGMOD*. 601–612.

[31] Behzad Golshan, Alon Halevy, George Mihaila, and Wang-Chiew Tan. 2017. Data Integration: After the Teenage Years. In *PODS*. 101–106.

[32] Yash Govind, Erik Paulson, Palaniappan Nagarajan, Paul Suganthan G. C., AnHai Doan, Youngchoon Park, Glenn Fung, Devin Conathan, Marshall Carter, and Mingju Sun. 2018. CloudMatcher: A Hands-Off Cloud/Crowd Service for Entity Matching. *PVLDB* 11, 12 (2018), 2042–2045.

[33] Anja Gruenheid, Xin Luna Dong, and Divesh Srivastava. 2014. Incremental Record Linkage. *PVLDB* 7, 9 (2014), 697–708.

[34] Oktie Hassanzadeh, Fei Chiang, Renée J. Miller, and Hyun Chul Lee. 2009. Framework for Evaluating Clustering Algorithms in Duplicate Detection. *PVLDB* 2, 1 (2009), 1282–1293.

[35] Ekaterini Ioannou and Minos Garofalakis. 2015. Query Analytics over Probabilistic Databases with Unmerged Duplicates. *TKDE* 27, 8 (2015), 2245–2260.

[36] Ekaterini Ioannou, Wolfgang Nejdl, Claudia Niederée, and Yannis Velegrakis. 2010. On-the-Fly Entity-Aware Query Processing in the Presence of Linkage. *PVLDB* 3, 1 (2010), 429–438.

[37] Mayank Kejriwal and Daniel P. Miranker. 2013. An Unsupervised Algorithm for Learning Blocking Schemes. In *ICDM*. 340–349.

[38] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Dedoop: Efficient Deduplication with Hadoop. *PVLDB* 5, 12 (2012), 1878–1881.

[39] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Load Balancing for MapReduce-based Entity Resolution. In *ICDE*. 618–629.

[40] Pradap Konda, Sanjib Das, Paul Suganthan G. C., AnHai Doan, Adel Ardalan, Jeffrey R. Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeffrey F. Naughton, Shishir Prasad, Ganesh Krishnan, Rohit Deep, and Vijay Raghavendra. 2016. Magellan: Toward Building Entity Matching Management Systems. *PVLDB* 9, 12 (2016), 1197–1208.

[41] Hanna Köpcke, Andreas Thor, and Erhard Rahm. 2010. Evaluation of entity resolution approaches on real-world match problems. *PVLDB* 3, 1 (2010), 484–493.

[42] Simon Lacoste-Julien, Konstantina Palla, Alex Davies, Gjergji Kasneci, Thore Graepel, and Zoubin Ghahramani. 2013. SIGMa: simple greedy matching for aligning large knowledge bases. In *KDD*. 572–580.

[43] Guoliang Li, Yudian Zheng, Ju Fan, Jiannan Wang, and Reynold Cheng. 2017. Crowdsourced Data Management: Overview and Challenges. In *SIGMOD*. 1711–1716.

[44] Juanzi Li, Jie Tang, Yi Li, and Qiong Luo. 2009. RiMOM: A Dynamic Multistrategy Ontology Alignment Framework. *TKDE* 21, 8 (2009), 1218–1232.

[45] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. 2001. Generic Schema Matching with Cupid. In *VLDB*. 49–58.

[46] Ruhaila Maskat, Norman W. Paton, and Suzanne M. Embury. 2016. Pay-as-you-go Configuration of Entity Resolution. *T. Large-Scale Data- and Knowledge-Centered Systems* (2016), 40–65.

[47] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. 2016. Exemplar queries: a new way of searching. *VLDB J.* 25, 6 (2016), 741–765.

[48] 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*. 19–34.

[49] Kevin O'Hare, Anna Jurek, and Cassio de Campos. 2018. A new technique of selecting an optimal blocking method for better record linkage. *Inf. Syst.* 77 (2018), 151–166.

[50] George Papadakis, George Alexiou, George Papastefanatos, and Georgia Koutrika. 2015. Schema-agnostic vs Schema-based Configurations for Blocking Methods on Homogeneous Data. *PVLDB* 9, 4 (2015), 312–323.

[51] George Papadakis, Ekaterini Ioannou, Claudia Niederée, Themis Palpanas, and Wolfgang Nejdl. 2012. Beyond 100 million entities: large-scale blocking-based resolution for heterogeneous data. In *WSDM*. 53–62.

[52] George Papadakis, Ekaterini Ioannou, Themis Palpanas, Claudia Niederée, and Wolfgang Nejdl. 2013. A Blocking Framework for Entity Resolution in Highly Heterogeneous Information Spaces. *TKDE* 25, 12 (2013), 2665–2682.

[53] George Papadakis, Georgia Koutrika, Themis Palpanas, and Wolfgang Nejdl. 2014. Meta-Blocking: Taking Entity Resolutionto the Next Level. *TKDE* 26, 8 (2014), 1946–1960.

[54] George Papadakis and Wolfgang Nejdl. 2011. Efficient entity resolution methods for heterogeneous information spaces. In *ICDE Workshops*. 304–307.

[55] George Papadakis, George Papastefanatos, and Georgia Koutrika. 2014. Supervised Meta-blocking. *PVLDB* 7, 14 (2014), 1929–1940.

[56] George Papadakis, Jonathan Svirsky, Avigdor Gal, and Themis Palpanas. 2016. Comparative Analysis of Approximate Blocking Techniques for Entity Resolution. *PVLDB* 9, 9 (2016), 684–695.

[57] George Papadakis, Leonidas Tsekouras, Emmanouil Thanos, George Giannakopoulos, Themis Palpanas, and Manolis Koubarakis. 2018. The return of JedAI: End-to-End Entity Resolution for Structured and Semi-Structured Data. *PVLDB* 11, 12 (2018), 1950–1953.

[58] Thorsten Papenbrock, Arvid Heise, and Felix Naumann. 2015. Progressive Duplicate Detection. *TKDE* 27, 5 (2015), 1316–1329.

[59] Vibhor Rastogi, Nilesh N. Dalvi, and Minos N. Garofalakis. 2011. Large-Scale Collective Entity Matching. *PVLDB* 4, 4 (2011), 208–218.

[60] Orion Fausto Reyes-Galaviz, Witold Pedrycz, Ziyue He, and Nick J. Pizzi. 2017. A supervised gradient-based learning algorithm for optimized entity resolution. *DKE* (2017).

[61] Anish Das Sarma, Ankur Jain, Ashwin Machanavajjhala, and Philip Bohannon. 2012. An automatic blocking mechanism for large-scale de-duplication tasks. In *CIKM*. 1055–1064.

[62] Giovanni Simonini, Sonia Bergamaschi, and H. V. Jagadish. 2016. BLAST: a Loosely Schema-aware Meta-blocking Approach for Entity Resolution. *PVLDB* 9, 12 (2016), 1173–1184.

[63] Giovanni Simonini, George Papadakis, Themis Palpanas, and Sonia Bergamaschi. 2019. Schema-Agnostic Progressive Entity Resolution. *IEEE Trans. Knowl. Data Eng.* 31, 6, 1208–1221.

[64] Yannis Sismanis, Ling Wang, Ariel Fuxman, Peter J. Haas, and Berthold Reinwald. 2009. Resolution-Aware Query Answering for Business Intelligence. In *ICDE*. 976–987.

[65] Kostas Stefanidis, Vasilis Efthymiou, Melanie Herschel, and Vassilis Christophides. 2014. Entity resolution in the web of data. In *WWW*.

[66] Fabian M. Suchanek, Serge Abiteboul, and Pierre Senellart. 2011. PARIS: Probabilistic Alignment of Relations, Instances, and Schema. *PVLDB* 5, 3 (2011), 157–168.

[67] Vasilis Verroios and Hector Garcia-Molina. 2015. Entity Resolution with crowd errors. In *ICDE*. 219–230.

[68] Vasilis Verroios, Hector Garcia-Molina, and Yannis Papakonstantinou. 2017. Waldo: An Adaptive Human Interface for Crowd Entity Resolution. In *SIGMOD*. 1133–1148.

[69] Norases Vesdapunt, Kedar Bellare, and Nilesh N. Dalvi. 2014. Crowdsourcing Algorithms for Entity Resolution. *PVLDB* 7, 12 (2014), 1071–1082.

[70] Jiannan Wang, Tim Kraska, Michael J. Franklin, and Jianhua Feng. 2012. CrowdER: Crowdsourcing Entity Resolution. *PVLDB* 5, 11 (2012), 1483–1494.

[71] Jiannan Wang, Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, Tim Kraska, and Tova Milo. 2014. A sample-and-clean framework for fast and accurate query processing on dirty data. In *SIGMOD*. 469–480.

[72] Jiannan Wang, Guoliang Li, Tim Kraska, Michael J. Franklin, and Jianhua Feng. 2013. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*. 229–240.

[73] Sibo Wang, Xiaokui Xiao, and Chun-Hee Lee. 2015. Crowd-Based Deduplication: An Adaptive Approach. In *SIGMOD*. 1263–1277.

[74] Steven Euijong Whang and Hector Garcia-Molina. 2014. Incremental entity resolution on rules and data. *VLDB J.* 23, 1 (2014), 77–102.

[75] Steven Euijong Whang, Peter Lofgren, and Hector Garcia-Molina. 2013. Question Selection for Crowd Entity Resolution. *PVLDB* 6, 6 (2013), 349–360.

[76] Steven Euijong Whang, David Marmaros, and Hector Garcia-Molina. 2013. Pay-As-You-Go Entity Resolution. *TKDE* 25, 5 (2013), 1111–1124.

[77] Wei Yan, Yuan Xue, and Bradley Malin. 2013. Scalable load balancing for mapreduce-based record linkage. In *IPCCC*. 1–10.

[78] Chen Jason Zhang, Rui Meng, Lei Chen, and Feida Zhu. 2015. CrowdLink: An Error-Tolerant Model for Linking Complex Records. In *ExploreDB*. 15–20.

Fairness in Rankings and Recommenders

Evaggelia Pitoura
pitoura@cs.uoi.gr
University of Ioannina
Ioannina, Greece

Georgia Koutrika
georgia@athenarc.gr
Athena Research Center
Athens, Greece

Kostas Stefanidis
konstantinos.stefanidis@tuni.fi
Tampere University
Tampere, Finland

ABSTRACT

With the growing complexity of the available online information, search engines via rankings and recommender systems come to the rescue, providing suggestions to users about items of potential interest, from movies and products to news articles and even potential friends. Such results and suggestions aim at covering the user information needs and play an important role in guiding users' decisions and in forming their opinions.

However, the same technology, if not used responsibly, may lead to discrimination, amplify potential biases in the original data, restrict transparency and strengthen unfairness. For example, consider scenarios in which models based on biased data produce results that abet violence, decrease diversity, or have an adverse impact on economic policies.

While the potential benefits of rankings and recommenders are well-accepted and understood, the importance of using such systems in a fair manner has only recently attracted attention. In this tutorial, we cover recent advancements and highlight future research directions in this increasingly relevant research area.

1 INTRODUCTION

Currently, algorithmic systems driven by large amounts of data are increasingly being used in all aspects of society. Such systems offer enormous opportunities. They accelerate scientific discovery in all domains, including personalized medicine and smart weather forecasting, they automate tasks, they help in improving our life through personal assistants and recommendations, they have the potential of transforming society through open government, to name just a few of their benefits.

Often, such systems are being used to assist, or, even replace human decision making in diverse domains. Examples include software systems used in school admissions, housing, pricing of goods, credit score estimation, job applicant selection, and sentencing decisions in courts and surveillance. A prominent case is the COMPAS software used in courts in the US to assist bail and sentencing decisions through a risk assessment algorithm that predicts future crime.

The ubiquitous use of such systems may create possible threats of economic loss, social stigmatization, or even loss of liberty. For instance, a known study by ProPublica found that in COMPAS, the false positive rate for African American defendants, namely people labelled "high-risk" who did not re-offend, was nearly twice as high as that for white defendants [11]. Another well-known study shows that names used predominantly by men and women of colour are much more likely to generate ads related to arrest records [34].

Data-driven systems are also being employed by search and recommendation engines, social media tools, and news outlets, among others. Recent studies report that social media has become the main source of online news with more than 2.4 billion internet users, of which nearly 64.5% receive breaking news from social media instead of traditional sources [22]. Thus, to a great extent, such systems play a central role in shaping our experiences and influencing our perception of the world. Again, there are many reports questioning the output of such systems. For instance, a known study on search results showed evidence for stereotype exaggeration in images returned when people search for professional careers [16].

Fairness in rankings and recommenders. In this tutorial, we pay special attention to the concept of fairness in rankings and recommender systems. By fairness, we typically mean lack of discrimination. It is not correct to assume that insights achieved via computations on data are unbiased simply because data was collected automatically or processing was performed algorithmically. Bias may come from the algorithm, reflecting, for example, commercial or other preferences of its designers, or even from the actual data, for example, if a survey contains biased questions, or, if some specific population is misrepresented in the input data.

In this tutorial, we review a number of definitions of fairness that aim at addressing discrimination, bias amplification, and ensure fair treatment. We organize these definitions around the notions of individual and group fairness. We also present methods for achieving fairness in rankings and recommendations, taking a cross-type view, distinguishing them between pre-processing, in-processing and post-processing approaches. We conclude with a discussion of the new research directions that arise.

2 TUTORIAL OBJECTIVES

This tutorial aims at presenting a toolkit of definitions, models and methods used for ensuring fairness in rankings and recommendations. Our objectives are three-fold: (*a*) to provide a solid framework on a novel, quickly evolving, and impactful domain, (*b*) to highlight challenges and research paths for researchers and practitioners that work on problems in the intersection of recommender systems and databases, and (*c*) to show how fairness challenges manifest in other areas (e.g., cloud computation and job scheduling) and transfer findings from existing works in these areas.

For this purpose, we organize our tutorial along the following main axes: (i) Motivation and background for the need for fair rankings and recommendations, (ii) Modeling fairness in rankings and recommendations, (iii) Ensuring fair rankings and recommendations, and (iv) Fairness in computations, algorithms and systems, and open research challenges.

3 MOTIVATION AND BACKGROUND

Fairness has emerged as an important category of research for machine learning systems in many application areas. Extending this concept to rankings and recommendations is tricky. First, there is an essential tension between the goals of fairness and those of personalization. Inherent in the idea of personalization is that the best items for one user may be different than those for another. However, there are contexts in which equity across rankings and recommendation outcomes is a desirable goal. Furthermore, fairness is a multi-sided concept (e.g., [7, 8]), in which the impacts on multiple groups of individuals must be considered.

In this tutorial, we start by presenting motivating examples for the need for fair rankings and recommendations from several domains, including justice, ads, image search and others. We highlight possible causes of unfairness, such as biased or incomplete data, and algorithmic inefficiencies. We point out potential harms, such as filter bubbles, polarization, loss of opportunity, and discrimination.

We consider a number of different dimensions based on which we classify existing models and approaches. Firstly, we distinguish between the multiple viewpoints that fairness can have in recommendation systems, namely (*a*) fairness for the recommended items (e.g., [31]), (*b*) fairness for the users (e.g., [19, 38]), (*c*) fairness for groups of users (e.g., [1, 24, 29]) and (*d*) fairness for the item providers, and the recommendation platform (e.g., [25]). Furthermore, we distinguish the existing methods for achieving fairness in rankings and recommendations as: (*a*) pre-processing (e.g., [31]), (*b*) in-processing (e.g., [13]) and (*c*) post-processing approaches (e.g., [15]).

4 MODELING FAIRNESS

Fairness is a general term and coming up with a single definition or model is tricky. We start this part of the tutorial by reviewing definitions of fairness which, in general, ask for nondiscrimination of users or items, based on the values of one or more sensitive or protected attributes, such as gender or race. We organize the definitions with respect to the notions of *individual fairness*, i.e., treating similar individuals similarly [10, 18], and *group fairness*, i.e., treating different groups equally (e.g., nondiscrimination of sensitive groups) [2, 35].

We present a number of widely used models and definitions for fairness [23, 36], including:

- *Demographic (or statistical) parity* (e.g., [35]), stating that the proportion of each part of a protected class (e.g., gender) should take the positive outcome at equal rates.
- *Conditional statistical parity* (e.g., [36]), which defines statistical parity given a set of legitimate factors.
- *Equalized odds* (e.g., [2]), stating that the protected and unprotected groups should have equal rates for true positives and false positives.
- *Fairness through awareness* (e.g., [10]), stating that any two similar individuals should receive a similar outcome.
- *Counterfactual fairness* (e.g., [18]), stating that a decision for an individual is fair, if it is the same in both the actual world and a counterfactual world where the individual belongs to a different demographic group.
- *Calibration-based fairness* (e.g., [26]), stating that if a group receives a predicted probability p, at least a fraction p of its members should belong to the predicted class.

Next, we review how these models of fairness have been *extended in the case of ranked outputs*, including attention-based and probability-based approaches [3] as well as approaches based on pair-wise comparisons [4, 37]. Then, we look at how definitions of algorithmic fairness and fair ranking have been *adopted in recommender systems* (e.g., [31, 39]). Given that fairness is a multi-sided concept, we extend our taxonomy under the umbrella of recommender systems, considering that fairness can refer to suggested data items [31], users [19, 38], group of users [27, 29] or item providers. Finally, we investigate the notion of fairness in sequential and multi-round recommenders [5, 6, 25, 33], where the goal is to ensure fairness in a number of interactions between the users and the system. We also discuss fairness in the case of link recommendations in networks and related concepts of homogeneity, echo chambers and polarity [12].

This part of the tutorial concludes with a discourse on other related concepts, such as the relationship between fairness and diversity [9], recommendation independence, transparency [15] and feedback loops.

5 ENSURING FAIRNESS

In this section, we present methods for achieving fairness in rankings and recommendations. We first discuss the trade-offs among fairness, personalization and accuracy.

Taking a cross-type view, approaches can be distinguished as pre-processing, in-processing and post-processing.

- Pre-processing approaches target at transforming the data so that any underlying bias or discrimination is removed.
- In-processing approaches target at modifying existing or introducing new algorithms that result in fair rankings and recommendations, e.g., by removing bias.
- Post-processing approaches treat the algorithms for producing rankings and recommendations as black boxes, without changing their inner workings. To ensure fairness, they modify the output of the algorithm.

5.1 Recommenders

We first study fairness in systems that produce recommendations for individuals. These comprise the majority of existing recommender systems. We start by presenting *pre-processing approaches* that work on modifying the input to the recommender, for example, by appropriate sampling (e.g., [9]), by adding more data to the input (e.g., [31]), or by performing database repair [28]. Then, we focus on approaches for designing *fairness-aware algorithms*, that is, recommendation algorithms that produce fair recommendations. We will present algorithms for fairness-aware matrix factorization [7, 38], multi-armed bandits [13, 21] and deep learning recommenders (e.g., [6, 44]). For instance, we show that when fairness with respect to both consumers and to item providers is important, variants of the well-known sparse linear method (SLIM) can be used to negotiate the trade-off between fairness and accuracy and improve the balance of user and item neighborhoods [7]. Alternatively, we can augment the learning objective in matrix factorization by adding a smoothed variation of a fairness metric [38]. As another example, we present methods that mitigate bias to increase fairness by incorporating randomness in variational autoencoders recommenders (e.g., [6]). Finally, we present *post-processing approaches that modify the output of the recommenders* to ensure fairness (e.g., [15]).

Moving from individuals to groups, group recommendations have attracted significant research efforts for their importance in benefiting a group of users. However, maximizing the satisfaction of each group member while minimizing the unfairness between them is very challenging [20]. We study different *fair-aware algorithms for group recommenders* [20, 27, 29, 32].

5.2 Rankings

In order to guarantee fair rankings, *in-processing approaches* work with result generation procedures that allow the systematic control of the degree of unfairness in the output, by exploiting learning techniques, satisfying statistical parity, while preserving relevance [37, 43]. The work in [30] formulates fairness constraints on rankings, targeting at relevance maximization, in terms of exposure allocation. A learning-based in-processing approach is also used in [41] to reduce discrimination and inequality of opportunity in rankings, Here, the method learns a ranking function with an additional objective that reduces disparate exposure. A recent learning to rank approach, DELTR, looks at the average probability of items from a protected group to be ranked at the top position [42].

The *post-processing approach* of [40] aims at satisfying statistical tests of representativeness, when ranking items in a certain order, so as to ensure that the ratio of protected individuals that appear within a prefix of the ranking (namely, top-k) must be above a given proportion. The attention received by the items in different positions in the ranking is also not the same: items ranked in first positions are exposed to much more attention than the lower ones. [5] tackles the problem of having a ranking to be presented as a query result, where the items in the first positions have the same or very similar relevance. When it happens, there is a decision to be made of which items are being top-ranked and which are not. A solution to this situation, called amortized fairness, considers that the position index is a proxy for the level of attention an item is exposed, while the output of the prediction algorithm corresponds to the item relevance. Accumulated attention across a series of rankings should be proportional to accumulated relevance, as indicating long term ranking fairness.

6 OPEN ISSUES AND RESEARCH DIRECTIONS

In this section, we present a critical comparison of the existing work on ensuring fair rankings and recommendations, and the lessons learnt in these areas. Furthermore, we discuss open issues and new research directions that arise.

First, we present fairness concepts studied in different areas of computer science. Fairness is often a ubiquitous property of computations, algorithms, and systems beyond recommender systems. For instance, in federated stream processing systems, it is an open challenge how to ensure global fairness on processing quality experienced by queries [14]. Systems for processing big data such as Hadoop, Spark, and massively parallel databases, need to run workloads on behalf of multiple tenants simultaneously. The abundant disk-based storage in these systems is usually complemented by a smaller, but much faster, cache. Cache allocation strategies are required that speed up the overall workload while being fair to each tenant [17].

Then, we highlight a number of possible research directions. We start with the observation that even if there exist several definitions and models for representing fairness, coming from different research perspectives, these definitions and models are many times somewhat incomparable, hindering consistent understanding and treatment. Compiling existing definitions to produce new ones and evaluating their suitability in different domains and applications appears to be an open topic for further research. Fairness in recommendations is multi-sided, achieving fairness for all parties involved is also a topic that needs to be investigated further.

While the potential benefits of fairness are well-accepted nowadays, we still need to study the actual impact of fairness-enhancing algorithms. For example, extensive user studies are needed to evaluate the level of acceptance of the fairness-enhanced results by the users and the long term effect of these results on their own perceptions and preferences. Extensive studies that exploit feedback loops, should also be performed in this line of work, so as to investigate deeper the connections between the concepts of fairness, explainability and personalization. Moreover, it will be very advantageous to study comparatively the notions of equality, that ensures equal treatment, over equity, that ensures treatment based on needs. Operationalizing equity is a difficult task that often depends on the domain under study.

7 TUTORIAL INFORMATION

Motivation and Target Audience: The tutorial's topic lies in the core of the conference interests. The tutorial aims at researchers and students, as well as IT professionals and developers in searching, ranking and recommender systems, and the general data management community. Researchers and students will get a good introduction to the topic and get inspired by challenging research problems. Furthermore, IT professionals and developers will learn appropriate fairness-aware techniques to promote fairness in their systems. All the materials that will be used for the tutorial will be publicly available.
Prerequisites: The tutorial is carefully structured to accommodate both attendees unfamiliar with the topic and more experienced participants by providing required background knowledge, shared terminology and common understanding of the basic fairness-related concepts.
Intended Duration: We are aiming for a 90-minute tutorial.
Link to Tutorial Resources:
https://sites.google.com/view/fair-ranking-recommend

8 PRESENTERS

Evaggelia Pitoura is a Prof. at the Univ. of Ioannina, Greece, where she also leads the Distributed Management of Data Laboratory. She received her PhD degree from Purdue Univ., USA. Her research interests are in the area of data management systems with a recent emphasis on social networks and responsible data management. Her publications include more than 150 articles in international journals (including TODS, TKDE, PVLDB) and conferences (including SIGMOD, ICDE, WWW) and a highly-cited book on mobile computing. Her research has been funded by the EC and national sources. She has served or serves on the editorial board of ACM TODS, VLDBJ, TKDE, DAPD and as a group leader, senior PC member, or co-chair of many international conferences (including PC chair of EDBT 2016 and ICDE 2012). She has more than 20 years experience in teaching. **Prior tutorials:** Temporal Graphs [eBISS'17], Social Graphs [BigDat'15], Data Graphs [SummerSOC'14], Personalization [ICDE'10], Mobile Computing [ICDE'03], Pervasive Computing [ICDE'00].

Georgia Koutrika is 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, building innovative solutions for recommendations, data analytics and exploration. Her work has been incorporated in commercial products, described in 9 granted patents and 18 patent applications in the US and worldwide, and published in more than 80 papers in top-tier conferences and journals. She is an ACM Distinguished Speaker and associate editor for TKDE and PVLDB. She has served or serves as PC member or co-chair of many conferences (including Demo PC chair of ACM SIGMOD 2018 and General Chair of ACM SIGMOD 2016). **Prior tutorials:** Recommender Systems [SIGMOD'18, EDBT'18, ICDE'15], Personalization [ICDE'10, ICDE'07, VLDB'05].

Kostas Stefanidis is an Assoc. Professor on Data Science at the Tampere University, Finland. He got his PhD in personalized data management from the Univ. of Ioannina, Greece. His research interests lie in the intersection of databases, information retrieval, data mining and the Web, and include personalization and recommender systems, and large-scale entity resolution and information integration. His publications include more than 80 papers in peer-reviewed conferences and journals, including SIGMOD, ICDE, and ACM TODS, and a book on entity resolution in the Web of data. He has 8 years experience in teaching. **Prior tutorials:** Recommender Systems [MUMIA Training School'14], Personalization [ICDE'10], Entity Resolution [ICDE'17, ESWC'16, WWW'14, CIKM'13].

REFERENCES

[1] Sihem Amer-Yahia, Senjuti Basu Roy, Ashish Chawla, Gautam Das, and Cong Yu. 2009. Group Recommendation: Semantics and Efficiency. *PVLDB* 2, 1 (2009), 754–765.

[2] Pranjal Awasthi, Matthäus Kleindessner, and Jamie Morgenstern. 2019. Effectiveness of Equalized Odds for Fair Classification under Imperfect Group Information. *CoRR* abs/1906.03284 (2019).

[3] Richard Berk, Hoda Heidari, Shahin Jabbari, Matthew Joseph, Michael J. Kearns, Jamie Morgenstern, Seth Neel, and Aaron Roth. 2017. A Convex Framework for Fair Regression. *CoRR* abs/1706.02409 (2017).

[4] Alex Beutel, Jilin Chen, Tulsee Doshi, Hai Qian, Li Wei, Yi Wu, Lukasz Heldt, Zhe Zhao, Lichan Hong, Ed H. Chi, and Cristos Goodrow. 2019. Fairness in Recommendation Ranking through Pairwise Comparisons. In *KDD*. 2212–2220.

[5] Asia J. Biega, Krishna P. Gummadi, and Gerhard Weikum. 2018. Equity of Attention: Amortizing Individual Fairness in Rankings. In *SIGIR*. 405–414.

[6] Rodrigo Borges and Kostas Stefanidis. 2019. Enhancing Long Term Fairness in Recommendations with Variational Autoencoders. In *MEDES*. 95–102.

[7] Robin Burke. 2017. Multisided Fairness for Recommendation. *CoRR* abs/1707.00093 (2017).

[8] Robin Burke, Nasim Sonboli, and Aldo Ordonez-Gauger. 2018. Balanced Neighborhoods for Multi-sided Fairness in Recommendation. In *FAT*. 202–214.

[9] L. Elisa Celis, Amit Deshpande, Tarun Kathuria, and Nisheeth K. Vishnoi. 2016. How to be Fair and Diverse? *CoRR* abs/1610.07183 (2016).

[10] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard S. Zemel. 2012. Fairness through awareness. In *Innovations in Theoretical Computer Science*. 214–226.

[11] J. Angwin et al. 2016. Machine Bias. *ProPublica* (2016). https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing

[12] Kiran Garimella, Gianmarco De Francisci Morales, Aristides Gionis, and Michael Mathioudakis. 2018. Reducing Controversy by Connecting Opposing Views. In *IJCAI*. 5249–5253.

[13] Matthew Joseph, Michael J. Kearns, Jamie H. Morgenstern, and Aaron Roth. 2016. Fairness in Learning: Classic and Contextual Bandits. In *NIPS*. 325–333.

[14] Evangelia Kalyvianaki, Marco Fiscato, Theodoros Salonidis, and Peter Pietzuch. 2016. THEMIS: Fairness in Federated Stream Processing Under Overload. In *SIGMOD*. 541–553.

[15] Toshihiro Kamishima, Shotaro Akaho, Hideki Asoh, and Jun Sakuma. 2018. Recommendation Independence. In *FAT*. 187–201.

[16] Matthew Kay, Cynthia Matuszek, and Sean A. Munson. 2015. Unequal Representation and Gender Stereotypes in Image Search Results for Occupations. In *CHI*. 3819–3828.

[17] Mayuresh Kunjir, Brandon Fain, Kamesh Munagala, and Shivnath Babu. 2017. ROBUS: Fair Cache Allocation for Data-parallel Workloads. In *SIGMOD*. 219–234.

[18] Matt J. Kusner, Joshua R. Loftus, Chris Russell, and Ricardo Silva. 2017. Counterfactual Fairness. In *NIPS*. 4066–4076.

[19] Jurek Leonhardt, Avishek Anand, and Megha Khosla. 2018. User Fairness in Recommender Systems. In *WWW*. 101–102.

[20] Xiao Lin, Min Zhang, Yongfeng Zhang, Zhaoquan Gu, Yiqun Liu, and Shaoping Ma. 2017. Fairness-Aware Group Recommendation with Pareto-Efficiency. In *RecSys*. 107–115.

[21] Yang Liu, Goran Radanovic, Christos Dimitrakakis, Debmalya Mandal, and David C. Parkes. 2017. Calibrated Fairness in Bandits. *CoRR* abs/1707.01875 (2017).

[22] N. Martin. 2018. How Social Media Has Changed How We Consume News. *Forbes* (2018). https://www.forbes.com/sites/nicolemartin1/2018/11/30/how-social-media-has-changed-how-we-consume-news/#18ae4c093c3c

[23] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. 2019. A Survey on Bias and Fairness in Machine Learning. *CoRR* abs/1908.09635 (2019).

[24] Eirini Ntoutsi, Kostas Stefanidis, Kjetil Nørvåg, and Hans-Peter Kriegel. 2012. Fast Group Recommendations by Applying User Clustering. In *ER*. 126–140.

[25] Gourab K Patro, Abhijnan Chakraborty, Niloy Ganguly, and Krishna P Gummadi. 2020. Incremental Fairness in Two-Sided Market Platforms: On Smoothly Updating Recommendations. In *AAAI*.

[26] Geoff Pleiss, Manish Raghavan, Felix Wu, Jon Kleinberg, and Kilian Q Weinberger. 2017. On Fairness and Calibration. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 5680–5689.

[27] Dimitris Sacharidis. 2019. Top-N group recommendations with fairness. In *SAC*. 1663–1670.

[28] Babak Salimi, Luke Rodriguez, Bill Howe, and Dan Suciu. 2019. Interventional Fairness: Causal Database Repair for Algorithmic Fairness. In *SIGMOD*. 793–810.

[29] Dimitris Serbos, Shuyao Qi, Nikos Mamoulis, Evaggelia Pitoura, and Panayiotis Tsaparas. 2017. Fairness in Package-to-Group Recommendations. In *WWW*. 371–379.

[30] Ashudeep Singh and Thorsten Joachims. 2018. Fairness of Exposure in Rankings. In *KDD*. 2219–2228.

[31] Harald Steck. 2018. Calibrated recommendations. In *RecSys*. 154–162.

[32] Maria Stratigi, Haridimos Kondylakis, and Kostas Stefanidis. 2018. FairGRecs: Fair Group Recommendations by Exploiting Personal Health Information. In *DEXA*. 147–155.

[33] Maria Stratigi, Jyrki Nummenmaa, Evaggelia Pitoura, and Kostas Stefanidis. 2020. Fair Sequential Group Recommendations. In *SAC*.

[34] Latanya Sweeney. 2013. Discrimination in online ad delivery. *Commun. ACM* 56, 5 (2013), 44–54.

[35] Virginia Tsintzou, Evaggelia Pitoura, and Panayiotis Tsaparas. 2019. Bias Disparity in Recommendation Systems. In *RMSE*.

[36] Sahil Verma and Julia Rubin. 2018. Fairness definitions explained. In *FairWare*. 1–7.

[37] Ke Yang and Julia Stoyanovich. 2017. Measuring Fairness in Ranked Outputs. In *SSDM*. 22:1–22:6.

[38] Sirui Yao and Bert Huang. 2017. Beyond Parity: Fairness Objectives for Collaborative Filtering. In *NIPS*. 2921–2930.

[39] Sirui Yao and Bert Huang. 2017. New Fairness Metrics for Recommendation that Embrace Differences. *FAT/ML* (2017).

[40] Meike Zehlike, Francesco Bonchi, Carlos Castillo, Sara Hajian, Mohamed Megahed, and Ricardo Baeza-Yates. 2017. FA*IR: A Fair Top-k Ranking Algorithm. In *CIKM*. 1569–1578.

[41] Meike Zehlike and Carlos Castillo. 2018. Reducing Disparate Exposure in Ranking: A Learning To Rank Approach. *CoRR* abs/1805.08716 (2018).

[42] Meike Zehlike, Gina-Theresa Diehn, and Carlos Castillo. 2020. Reducing Disparate Exposure in Ranking: A Learning to Rank Approach. In *WWW*.

[43] Richard S. Zemel, Yu Wu, Kevin Swersky, Toniann Pitassi, and Cynthia Dwork. 2013. Learning Fair Representations. In *ICML*. 325–333.

[44] Ziwei Zhu, Xia Hu, and James Caverlee. 2018. Fairness-Aware Tensor-Based Recommendation. In *CIKM*. 1153–1162.

NoSQL Schema Evolution and Data Migration: State-of-the-Art and Opportunities

Uta Störl
Darmstadt Univ. of Applied Sciences
Germany
uta.stoerl@h-da.de

Meike Klettke
University of Rostock
Germany
meike.klettke@uni-rostock.de

Stefanie Scherzinger
OTH Regensburg
Germany
stefanie.scherzinger@oth-r.de

ABSTRACT

NoSQL database systems are very popular in agile software development. Naturally, agile deployment goes hand-in-hand with database schema evolution. The main aim of this tutorial is to present to the audience the current state-of-the-art in continuous NoSQL schema evolution and data migration: (1) We present case studies on schema evolution in NoSQL databases; (2) we survey existing approaches to schema management and schema inference, as implemented in popular NoSQL database products, and also as proposed in academic research; (3) we present approaches for extracting schema versions; (4) we analyze different methods for efficient NoSQL data migration; and (5) we give an outlook on further research opportunities.

Duration: 1.5 hours

1 INTRODUCTION

Recent position papers demand more schema flexibility, such as the ability to handle variational data [3, 42]. Many agile software developers have long since turned towards NoSQL database systems such as *MongoDB*[1], *Couchbase*[2], or *ArangoDB*[3] which are schema-flexible, or even altogether schema-free. They allow to store datasets in different structural versions to co-exist.

Yet even when the database management system does not maintain an explicit schema, there is commonly an implicit schema, as the application code makes assumptions about the structure of the stored data. For instance, in Figure 1, the Java code in lines 1 through 9 implies a schema: An entity for person `Jo Bloggs` is created, and then persisted in the `people` collection.

```
1  List<Integer> books = Arrays.asList(27464, 747854);
2  DBObject person = new BasicDBObject("_id", "jo")
3                    .append("name", "Jo Bloggs")
4                    .append("address",
5                      new BasicDBObject("street", "123 Fake St")
6                        .append("city", "Faketon")
7                        .append("state", "MA")
8                        .append("zip", 12345))
9                    .append("books", books);
10 DBCollection collection = database.getCollection("people");
11 collection.insert(person);
```

Figure 1: Storing a person entity in MongoDB, using Java.[4]

[1]https://docs.mongodb.com/
[2]https://www.couchbase.com/products/server
[3]https://www.arangodb.com/
[4]From "Getting Started with MongoDB and Java", https://www.mongodb.com/blog/post/getting-started-with-mongodb-and-java-part-i, published August 2014.

With each new release of the software, the application code evolves. Eventually, so will the implicit schema declarations. Then, data stored in the production system will have to be migrated accordingly. Yet writing custom migration scripts — which seems to be the common practice today — is error-prone and expensive. Thus, there is a dire need for well-principled tool support for long-term maintenance of NoSQL database schemas.

With the schema declared within the application layer, the burden of schema maintenance is shifted into the domain of the application developers. Accordingly, we observe various grass-roots efforts from the developer community to tackle schema evolution. Unfortunately, these solutions do not build upon the existing state-of-the-art in research. Overall, we see it as an opportunity for the database research community to contribute well-founded and practical solutions to real-world problems.

In this tutorial, we give an overview of schema management in agile development with NoSQL database systems. The authors proposing this tutorial have been publishing in this domain for over 6 years. We present the current state-of-the-art in research, as well as in practice. We cover inferring schema-on-read with outlier detection, deriving schema versions, the corresponding schema evolution operations matching between them, as well as the resulting data migration operations. Different migration strategies and their impact, such as the overall migration costs and the latency upon accessing an entity, are also discussed.

A strong point of this tutorial is that we motivate the problem domain by presenting an empirical study on NoSQL schema evolution in real-world applications. Moreover, we demo existing tools for NoSQL schema management, e.g., for schema design and schema extraction. Incorporating small, live demos, we put together a diverse and diverting tutorial.

2 OUTLINE

This 1.5-hour tutorial is split into five parts:

(1) **Case Studies (~15 min.)**. We present an empirical study on the schema imposed on NoSQL databases by applications, as well as the dynamics of NoSQL schema evolution.
(2) **NoSQL Schema Management (~20 min.)**. In this part we discuss different architectures and existing solutions for NoSQL schema management. Here, we present research approaches as well as first products.
(3) **NoSQL Evolution Management (~25 min.)**. We present solutions for NoSQL evolution management. Beside a language for declaring NoSQL schema evolution operations, we focus on approaches for extracting schema versions.
(4) **Data Migration (~20 min.)**. Based on the previous parts, we present different data migration strategies and discuss their quantitative assessment.
(5) **Future Opportunities (~10 min.)**. Finally, we outline open research problems as potential directions for further research, as well as current development in this area.

3 GOALS AND OBJECTIVES

3.1 Case Studies

For our introduction, we present an empirical study on real-world database applications, each backed by a schema-flexible NoSQL data store. We investigate whether developers denormalize their schema, as is the recommended practice in data modeling for NoSQL data stores, and also a research subject [4, 25]. Further, we analyze the entire project history, and with it, the evolution of the NoSQL schema. By looking at real-world evidence, we pinpoint characteristic problems (such as the increased frequency of schema changes). We discuss how existing solutions do not fully transfer (e.g., since they rely on the schema being declared explicitly, rather than being implicit in the code). Thus, existing solutions cannot address the needs of application developers. Finally, we state the desiderata for tackling NoSQL schema evolution, in preparation to the subsequent tutorial.

3.2 NoSQL Schema Management

NoSQL database systems differ with regard to schema support: There are *schema-free* systems without any native schema support (e.g. *Couchbase*, *CouchDB*[5], *Neo4j*[6]). Other systems offer optional schema support (e.g. *MongoDB*, *OrientDB*[7]; some of these systems support different schema modes: *schema-full* or *schema-flexible*). There are also schema-full systems, with a mandatory schema (e.g. *Cassandra*[8]). In addition, there are proposals for vendor-independent middleware that manages the NoSQL schema (e.g. the *Darwin* schema management component [45]).

3.2.1 *Capturing the NoSQL Schema.*

In our tutorial, we present three strategies how the NoSQL schema can be captured from a schema-free or schema-flexible data store: (a) In the tradition of textbook schema design, the NoSQL schema can be derived top-down from some conceptual model. (b) The schema may be extracted posthumously, given a data instance. (c) The schema may be also derived by static analysis of the application code. We now briefly discuss these options.

a) Forward Engineering/Schema-First. In forward engineering, or schema-first approaches, the schema is explicitly defined – for example with modeling tools such as *Hackolade*[9] or *erwin*[10]: Users of these tools draw extended entity relationship models or graphical visualizations of JSON Schema, which they can compile for a given NoSQL database system. The principles behind NoSQL schema design are being explored in academic research: In [1], a model-driven approach for designing NoSQL databases has been developed. The authors of [4] propose an abstract data model for NoSQL databases, which exploits the commonalities of various NoSQL systems. Another project [8] extends the schema-first approach and generates object-oriented class hierarchies. The class declarations actually represent entity collections, using Object-NoSQL mapper libraries such as Mongoose and Morphia.

Yet in agile development, the schema is often not fixed up front. Therefore, we next discuss schema reverse engineering.

b) Reverse Engineering from Data/Schema-on-Read. For processing data without explicit schema information, *reverse engineering* can be necessary. In the following we will refer to this approach as

[5]https://couchdb.apache.org/
[6]https://neo4j.com/
[7]https://orientdb.org/
[8]http://cassandra.apache.org/
[9]https://hackolade.com/
[10]https://erwin.com/products/erwin-dm-nosql/

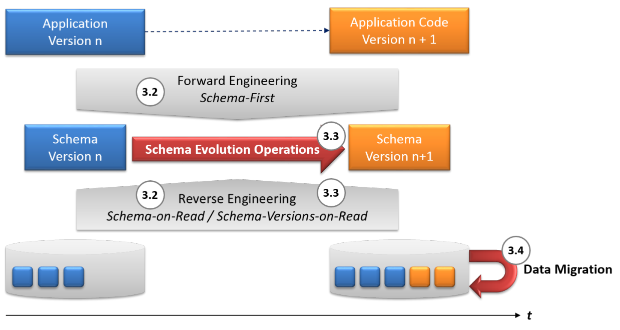


Figure 2: The Big Picture: Moving from schema version n (blue, shown to the left) to version $n + 1$ (orange, shown to the right), the persisted entities (blue) may not be immediately migrated. Rather, the data store now holds entities in both schema versions (blue and orange).

schema-on-read (this term is currently used differently in various Big Data/NoSQL application areas).

The general idea has been introduced in [22], based on an earlier approach for XML schema extraction in [26]: The implicit structural information from all entities is combined into a graph, from which the schema and statistics can be derived. The schema inference approach delivers a *schema overview*. The algorithm and its optimizations will be introduced.

Similar approaches [2, 13, 20, 33, 36] also infer a schema or generate conceptual models. The authors of [43] consider the challenge of designing schemas for existing JSON datasets as an interactive problem and present a roll-up/drill-down style interface for exploring collections of JSON records. In [5], schemas are inferred from datasets by typing the input according to a type system. This approach is designed for MapReduce-based, and thus highly scalable, execution. In [48], a descriptive schema is inferred, and documents are indexed by their structural properties, so that they may even be queried accordingly.

Certain (NoSQL) database design tools, such as *Hackolade* and *erwin*, also implement schema reverse engineering, and certain NoSQL database systems (for example MongoDB) come with similar features built-in.

Schema inference from existing data is also one of the subtasks in data lake analytics [29]. In data lakes, the "load-first, schema-later" paradigm requires a combination of schema inference with the inference of integrity constraints [12, 14, 30, 31], and further descriptions of the data content, such as the semantic data type [19]. Data cleaning methods are also based on reverse engineering of structure and frequencies of occurrence [32, 38].

c) Reverse Engineering from Code. Since the application code implicitly declares a schema, this schema may be extracted. This task is straightforward when applications use Object-NoSQL mapper libraries, since class declarations then map to collections of persisted entities. This approach is followed in [34, 39]. For application code without mapper libraries, we need to resort to more involved code analysis. For instance, in programmatically extracting a schema for collection `people` from the code in Figure 1, we might employ data flow analysis, as done in [49], to detect which statements characterize a collection.

3.2.2 *The Big Picture.*

Figure 2 gives an overview of NoSQL schema (evolution) management and data migration, which will accompany us through the entire tutorial. The left part of the figure visualizes the approaches of forward engineering and reverse engineering discussed so far (top-down vs. bottom-up).

If we want to consider not only a static view of the schema, but also its evolution along with the application code (illustrated by application versions n and $n + 1$ in the upper part, and two different versions of the database instance at two different times, in the lower part of Figure 2), new challenges arise. In our tutorial, we next discuss challenges and solutions for handling data in co-existing schema versions, especially schema evolution operations (represented by the right arrow in the center of the figure), as well as approaches for extracting not only a schema, but to also identifying different schema versions.

In the fourth part of the tutorial we then discuss various data migration strategies (symbolized by the self-referencing arrow on the bottom right of Figure 2).

3.3 NoSQL Schema Evolution Management

Handling different versions of data in a single database is becoming more and more important – in relational databases [3, 16, 42] as well as for different types of NoSQL databases [6, 40]. In this tutorial, we survey different approaches.

NoSQL Evolution Language. Most NoSQL database management systems do not provide a language capturing schema evolution operations. There are several proposals for such a language. For instance, the authors of [41] define a language originally for transformation between different NoSQL databases that can also be used for data migration within the same store [15]. In this tutorial, we present the NoSQL schema evolution language introduced in [40] for different types of NoSQL database systems, implemented in [45] and extended in [18] for multi-model data. The language contains operations that affect only one entity-type, or synonymously in MongoDB terminology, one collection (*single-type* operations are *add*, *delete*, and *rename*). For complex refactoring, operations that affect the entities of more than one entity type (*multi-type* operations *copy*, *move*, *split*, and *merge*) are available. Via these schema evolution operations (or schema modification operations by the terminology of [9]), the schema changes between consecutive versions of the application code can be declared. This mapping is sketched by the red arrow in the center of Figure 2.

Heterogeneity Classes. We capture the degree of structural heterogeneity in a data instance via the notion of heterogeneity classes, introduced in [27]. For example, persisted entities may be very homogeneous in their structure, or very heterogeneous. We outline the implications, present illustrative examples, and define the class-specific semantics of evolution operations.

Schema-Versions-on-Read. Whereas the *schema overview* presented in the second part of the tutorial delivers information on structures, data types, nesting of information, as well as information on required and optional parts, it does not capture the structural changes over time. For this reason, we present a further method, developed for inferring schema versions, as well as the changes between consecutive versions [21, 44]. Approaches for extracting *schema-versions-on-read* are based on a partial order of the data, e.g. based on timestamps. Besides schema inference, we can use the partial order to find out when and how structural information has changed and derive schema versions accordingly. Applying additional information on integrity constraints, we also can suggest evolution operations that have caused the schema changes [21]. To our knowledge, this functionality is not yet offered by any commercial products, but primarily implemented in research prototypes, e.g., [7, 35] for representing UML class model versions, and [21, 45] for generating JSON Schema versions as well as the matching evolution operations.

3.4 Data Migration

After identifying the schema evolution operations, the data can be migrated. There are various strategies, with different impacts on latency and migration effort. Traditionally, data migration is carried out *eagerly*, i.e., all data is migrated immediately when the schema changes. Since this can be expensive, especially in a cloud environment [11, 17], data can be migrated *lazily* [23, 37], so data is not migrated until it is actually accessed. This approach is preferable in databases where the share of "hot" data is relatively small compared to the total amount of data. The downside is that lazy migration adds latency to database reads and writes, since the data might still have to be migrated. Figure 3 visualizes the conflicting goals of monetary data migration costs (e.g., charged by a cloud provider) and latency overhead upon read or write access, for different migration strategies.

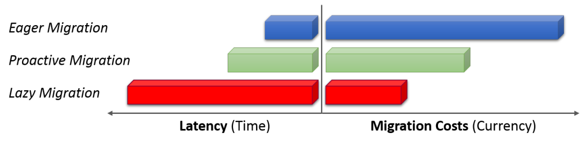


Figure 3: Tradeoffs in data migration (from [17]).

A compromise between the two competing goals is to migrate hot data *proactively* (one of the options in Figure 3). In [23] and [17], different proactive strategies are presented. Depending on the characteristics of the data, the workload, and the schema evolution operations, predictively migrating data promises good latency at moderate costs. In the tutorial, we give a detailed overview over such strategies and their tradeoffs. Further we discuss optimizations, such as the version jumps suggested in [23, 46]. In [17], we propose a first tool providing decision support for the challenge of choosing between migration strategies.

In addition to the impact on costs and latency, it should be noted that for all strategies except eager, the database system or the external schema management component must support *query rewriting*, since data may exist in previous (older) versions and therefore with a different structure than expected by the query [16, 28]. Depending on the heterogeneity class (c.f. Section 3.3), there are additional challenges to query rewriting [27].

In some cases, a *no migration* approach may be necessary, when auditing requires that datasets are preserved in their original version. Then, when an entity in a legacy version is accessed, we may migrate it lazily, but we preserve the original entity.

3.5 Future Opportunities

We finally discuss selected research opportunities related to NoSQL schema evolution and data migration.

Cost Models. For choosing an appropriate data migration strategy, appropriate cost models are needed. These cost models must take into account the characteristics of NoSQL database systems such as distribution, replication, the consistency concepts. A related challenge is the design of a suitable benchmark.

Multi-Model Data. Multi-model database systems like OrientDB, ArangoDB, and Cosmos DB[11] support more than one data model [24]. Similarly, polystores [10, 47] pose new challenges in our context. As we outline in [18], evolution in multi-model databases triggers further research questions, such as inter-model operations, the handling of global vs. local evolution operations, inference of multi-model schemas, and synchronizing migration over different models/systems.

4 INTENDED AUDIENCE AND MATERIAL

Our goal is to give EDBT attendees an overview of the challenges and the current state-of-the-art in both research and practice on NoSQL schema evolution and data migration. We will not assume any background in NoSQL database systems, making our tutorial appropriate for researchers, practitioners, and graduate students.

The material is available at https://tinyurl.com/evolving-nosql.

5 BIOGRAPHIES

Uta Störl is a professor at Darmstadt University of Applied Sciences. Her research focuses on database technologies for Big Data and Data Science. Before, she worked for Dresdner Bank. (uta.stoerl@h-da.de)
Meike Klettke is a professor for Data Science at the University of Rostock. She works on database evolution and reverse engineering of databases. (meike.klettke@uni-rostock.de)
Stefanie Scherzinger is a professor at OTH Regensburg. Her research is influenced by her experience as a software engineer at IBM and Google. (stefanie.scherzinger@oth-r.de)

Acknowledgements: Funded by the Deutsche Forschungsgemeinschaft (DFG) — Projektnummer 385808805.

REFERENCES

[1] F. Abdelhédi, A. A. Brahim, F. Atigui, and G. Zurfluh. MDA-Based Approach for NoSQL Databases Modelling. In *Proc. DaWaK'17*, volume 10440 of *LNCS*, pages 88–102. Springer, 2017.
[2] J. Akoka and I. Comyn-Wattiau. Roundtrip engineering of NoSQL databases. *Enterprise Modelling and Information Systems Architectures*, 13(Special):281–292, 2018.
[3] P. Ataei, A. Termehchy, and E. Walkingshaw. Variational databases. In *Proc. DBPL 2017*, 2017.
[4] P. Atzeni, F. Bugiotti, L. Cabibbo, and R. Torlone. Data modeling in the NoSQL world. *Computer Standards & Interfaces*, 67, 2020.
[5] M. A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani. Parametric schema inference for massive JSON datasets. *VLDB J.*, 28(4):497–521, 2019.
[6] A. Bonifati, P. Furniss, A. Green, R. Harmer, E. Oshurko, and H. Voigt. Schema Validation and Evolution for Graph Databases. In *Proc. ER'19*, volume 11788 of *LNCS*, pages 448–456. Springer, 2019.
[7] A. H. Chillón, S. F. Morales, D. Sevilla, and J. G. Molina. Exploring the visualization of schemas for aggregate-oriented nosql databases. In *Proc. ER'17*, pages 72–85, 2017.
[8] A. H. Chillón, D. S. Ruiz, J. G. Molina, and S. F. Morales. A model-driven approach to generate schemas for object-document mappers. *IEEE Access*, 7:59126–59142, 2019.
[9] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo. Schema Evolution in Wikipedia - Toward a Web Information System Benchmark. In *Proc. ICEIS'08*, pages 323–332, 2008.
[10] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik. The BigDAWG Polystore System. *SIGMOD Record*, 44(2):11–16, 2015.
[11] M. Ellison, R. Calinescu, and R. F. Paige. Evaluating cloud database migration options using workload models. *J. Cloud Computing*, 7:6, 2018.
[12] M. H. Farid, A. Roatis, I. F. Ilyas, H. Hoffmann, and X. Chu. CLAMS: Bringing Quality to Data Lakes. In *Proc. SIGMOD '16*, pages 2089–2092. ACM, 2016.
[13] E. Gallinucci, M. Golfarelli, and S. Rizzi. Schema Profiling of Document Stores. In *Proc. SEBD'17*, 2017.
[14] R. Hai, C. Quix, and D. Wang. Relaxed Functional Dependency Discovery in Heterogeneous Data Lakes. In *Proc. ER 2019*, pages 225–239, 2019.
[15] F. Haubold, J. Schildgen, S. Scherzinger, and S. Deßloch. ControVol Flex: Flexible Schema Evolution for NoSQL Application Development. In *Proc. BTW'17*, pages 601–604, 2017.
[16] K. Herrmann, H. Voigt, A. Behrend, J. Rausch, and W. Lehner. Living in Parallel Realities: Co-Existing Schema Versions with a Bidirectional Database Evolution Language. In *Proc. SIGMOD'17*, pages 1101–1116. ACM, 2017.
[17] A. Hillenbrand, M. Levchenko, U. Störl, S. Scherzinger, and M. Klettke. MigCast: Putting a Price Tag on Data Model Evolution in NoSQL Data Stores. In *Proc. SIGMOD'19*, pages 1925–1928. ACM, 2019.
[18] I. Holubová, M. Klettke, and U. Störl. Evolution Management of Multi-model Data - (Position Paper). In *Proc. Poly and DMAH Workshop @ VLDB'19*, volume 11721 of *Lecture Notes in Computer Science*, pages 139–153. Springer, 2019.
[19] 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 *Proc. KDD'19*, pages 1500–1508. ACM, 2019.
[20] J. L. C. Izquierdo and J. Cabot. JSONDiscoverer: Visualizing the schema lurking behind JSON documents. *Knowl.-Based Syst.*, 103:52–55, 2016.
[21] M. Klettke, H. Awolin, U. Störl, D. Müller, and S. Scherzinger. Uncovering the evolution history of data lakes. In *Proc. SCDM'17*, pages 2462–2471. IEEE, 2017.
[22] M. Klettke, U. Störl, and S. Scherzinger. Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In *Proc. BTW'15*, LNI, pages 425–444, 2015.
[23] M. Klettke, U. Störl, M. Shenavai, and S. Scherzinger. NoSQL schema evolution and big data migration at scale. In *Proc. SCDM'16*, pages 2764–2774, 2016.
[24] J. Lu and I. Holubová. Multi-model Databases: A New Journey to Handle the Variety of Data. *ACM Comput. Surv.*, 52(3):55:1–55:38, 2019.
[25] M. J. Mior and K. Salem. Renormalization of NoSQL Database Schemas. In *Proc. ER 2018*, pages 479–487, 2018.
[26] C. Moh, E. Lim, and W. K. Ng. DTD-Miner: A Tool for Mining DTD from XML Documents. In *Proc. WECWIS '00*, pages 144–151. IEEE, 2000.
[27] M. L. Möller, M. Klettke, A. Hillenbrand, and U. Störl. Query Rewriting for Continuously Evolving NoSQL Databases. In *Proc. ER'19*, volume 11788 of *LNCS*, pages 213–221. Springer, 2019.
[28] H. J. Moon, C. Curino, and C. Zaniolo. Scalable architecture and query optimization fortransaction-time DBs with evolving schemas. In *Proc. SIGMOD'10*, pages 207–218. ACM, 2010.
[29] 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.
[30] T. Papenbrock, S. Kruse, J. Quiané-Ruiz, and F. Naumann. Divide & Conquer-based Inclusion Dependency Discovery. *PVLDB*, 8(7):774–785, 2015.
[31] T. Papenbrock and F. Naumann. A Hybrid Approach to Functional Dependency Discovery. In *Proc. SIGMOD'16*, pages 821–833. ACM, 2016.
[32] N. Prokoshyna, J. Szlichta, F. Chiang, R. J. Miller, and D. Srivastava. Combining Quantitative and Logical Data Cleaning. *PVLDB*, 9(4):300–311, 2015.
[33] C. Quix, R. Hai, and I. Vatov. Metadata Extraction and Management in Data Lakes With GEMMS. *CSIMQ*, 9:67–83, 2016.
[34] A. Ringlstetter, S. Scherzinger, and T. F. Bissyandé. Data model evolution using object-NoSQL mappers: Folklore or state-of-the-art? In *Proc. BIGDSE'16*, pages 33–36. ACM, 2016.
[35] D. S. Ruiz, S. F. Morales, and J. G. Molina. Inferring Versioned Schemas from NoSQL Databases and Its Applications. In *Proc. ER'15*, pages 467–480, 2015.
[36] F. J. B. Ruiz, J. G. Molina, and O. D. García. On the application of model-driven engineering in data reengineering. *Inf. Syst.*, 72:136–160, 2017.
[37] K. Saur, T. Dumitras, and M. W. Hicks. Evolving NoSQL Databases without Downtime. In *Proc. ICSME'16*, pages 166–176. IEEE, 2016.
[38] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Bießmann, and A. Grafberger. Automating Large-Scale Data Quality Verification. *PVLDB*, 11(12):1781–1794, 2018.
[39] S. Scherzinger, T. Cerqueus, and E. Cunha de Almeida. ControVol: A framework for controlled schema evolution in NoSQL application development. In *Proc. ICDE 2015*, pages 1464–1467, 2015.
[40] S. Scherzinger, M. Klettke, and U. Störl. Managing Schema Evolution in NoSQL Data Stores. In *Proc. DBPL'13*, 2013.
[41] J. Schildgen, T. Lottermann, and S. Deßloch. Cross-system NoSQL data transformations with NotaQL. In *Proc. BeyondMR@SIGMOD 2016*. ACM, 2016.
[42] W. Spoth, B. S. Arab, E. S. Chan, D. Gawlick, A. Ghoneimy, B. Glavic, B. C. Hammerschmidt, O. Kennedy, S. Lee, Z. H. Liu, X. Niu, and Y. Yang. Adaptive Schema Databases. In *Proc. CIDR 2017*, 2017.
[43] W. Spoth, T. Xie, O. Kennedy, Y. Yang, B. C. Hammerschmidt, Z. H. Liu, and D. Gawlick. SchemaDrill: Interactive Semi-Structured Schema Design. In *Proc. HILDA@SIGMOD'18*, pages 11:1–11:7. ACM, 2018.
[44] U. Störl, D. Müller, M. Klettke, and S. Scherzinger. Enabling Efficient Agile Software Development of NoSQL-backed Applications. In *Proc. BTW'17*, pages 611–614, 2017.
[45] U. Störl, D. Müller, A. Tekleab, S. Tolale, J. Stenzel, M. Klettke, and S. Scherzinger. Curating Variational Data in Application Development. In *Proc. ICDE'18*, pages 1605–1608, 2018.
[46] U. Störl, A. Tekleab, M. Klettke, and S. Scherzinger. In for a Surprise When Migrating NoSQL Data. In *Proc. ICDE'18*, page 1662. IEEE, 2018. (ICDE Lightning Talk).
[47] M. Vogt, A. Stiemer, and H. Schuldt. Polypheny-DB: Towards a Distributed and Self-Adaptive Polystore. In *Proc. SCDM'18*, pages 3364–3373. IEEE, 2018.
[48] L. Wang, S. Zhang, J. Shi, L. Jiao, et al. Schema Management for Document Stores. *Proc. VLDB Endow.*, 8(9), May 2015.
[49] S. Wu and I. Neamtiu. Schema Evolution Analysis for Embedded Databases. In *Proc. ICDEW'11*, 2011.

[11] https://docs.microsoft.com/azure/cosmos-db/

Erratum for Discovering Order Dependencies through Order Compatibility

Jaroslaw Szlichta
Ontario Tech University
Canada
jarek@ontariotechu.ca

Parke Godfrey
York University
Canada
godfrey@yorku.ca

Lukasz Golab
University of Waterloo
Canada
lgolab@uwaterloo.ca

Mehdi Kargar
Ryerson University
Canada
kargar@ryerson.ca

Divesh Srivastava
AT&T Labs-Research
United States
divesh@research.att.com

ABSTRACT

A number of extensions to the classical notion of functional dependencies have been proposed to express and enforce application semantics. One of these extensions is that of order dependencies (ODs), which express rules involving order. The article entitled "Discovering Order Dependencies through Order Compatibility" by Consonni et al., published in the EDBT conference proceedings in March 2019, investigates the OD discovery problem. The authors claim to prove that their OD discovery algorithm, OCDDISCOVER, is *complete*, as well as being significantly more efficient in practice than the state-of-the-art. They further claim that the implementation of the existing FASTOD algorithm (ours)—we shared our code base with the authors—which they benchmark against is flawed, as OCDDISCOVER and FASTOD report different sets of ODs over the same data sets.

In this rebuttal, we show that their claim of completeness is, in fact, *not* true. OCDDISCOVER's pruning rules are overly aggressive, and prune parts of the search space that contain legitimate ODs. This is the reason their approach appears to be "faster" in practice. Finally, we show that Consonni et al. misinterpret our set-based canonical form for ODs, leading to an incorrect claim that our FASTOD implementation has an error.

1 INTRODUCTION

Integrity constraints specify the intended semantics of dataset attributes. They are commonly used in a number of application areas, such as schema design, data integration, data cleaning, and query optimization [2]. Past work focused primarily on *functional dependencies* (FDs). In recent years, several extensions to the notion of an FD have been studied, including that of *order dependencies* (ODs) [3, 5–8, 10]. FDs cannot capture relationships among attributes with naturally *ordered* domains, such as over timestamps, numbers, and strings, which are common in business data [9]. For example, consider Table 1, which shows employee tax records in which `tax` is calculated as a `percentage` of `salary`. Both `tax` and `percentage` are non-decreasing with `salary`.

Order dependencies naturally express such semantics. For a second example from Table 1, the OD <`salary` *orders* `group`, `subgroup`> holds. When the table is sorted by `salary`, it is also then sorted by `group` (with ties broken by `subgroup`). However, <`salary` *orders* `subgroup`, `group`> does *not* hold. This illustrates that the *order* in which attributes appear in the OD matters.

© 2020 Copyright held by the owner/author(s). Published in Proceedings of the 23rd International Conference on Extending Database Technology (EDBT), March 30-April 2, 2020, ISBN 978-3-89318-083-7 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

Table 1: Table with employee information.

#	ID	yr	posit	bin	sal	perc	tax	grp	subg
t1	10	19	secr	1	5K	20%	1K	A	III
t2	11	19	mngr	2	8K	25%	2K	C	II
t3	12	19	direct	3	10K	30%	3K	D	I
t4	10	18	secr	1	4.5K	20%	0.9K	A	III
t5	11	18	mngr	2	6K	25%	1.5K	C	I
t6	12	18	direct	3	8K	25%	2K	C	II

The theory of order dependencies subsumes that of functional dependencies. Any FD can be *mapped* to an equivalent OD by prefixing the left-hand-side attributes onto the right-hand side [8, 10]. For example, if `salary` *functionally determines* `tax`, then `salary` *orders* `salary, tax`.

The purpose of this article is to refute the following claims in Consonni et al. [3].

(1) The authors present a definition of *minimality* for order compatibility dependencies (OCDs). An OCD is a more specific form of order dependency in which two lists of attributes order each other, when taken together [8]. They claim that their definition of minimality is *complete*; that is, from it, one can recover all valid OCDs that hold over a given table.
(2) Given their definition of minimal OCDs, Consonni et al. [3] propose an algorithm to *discover* ODs via OCDs, which has factorial complexity in the number of attributes. They claim to prove that their algorithm produces a canonically *complete* set of ODs. (That is, a *minimal* set of ODs with respect to their definition, from which all the ODs which hold over the data could purportedly be inferred.)
(3) The authors claim that their experimental evaluation illustrates an *error* in our *implementation* of OD discovery algorithm (FASTOD) [6, 7], which leads to discovering many additional—and, purportedly, incorrect—dependencies. In spite of this claim of an "implementation error" in the FASTOD implementation that we provided them, they support via benchmark experiments that their algorithm, OCDDISCOVER, outperforms our algorithm, FASTOD.

We show that each of these three claims is incorrect, in turn.

(1) The definition of minimality in Consonni et al. [3]—insofar as its intended purpose is a *canonical* form—is incorrect. Their "canonical" form does not allow for the inference of *all* OCDs. It misses an important subclass of OCDs (and, respectively, ODs), any dependency which has a common prefix on the *left* and *right* (that is, repeated attributes at the beginning of the dependency).

(2) The claim of completeness of the OD discovery algorithm in Consonni et al. [3] is incorrect, as it relies upon their incorrect notion of "minimal" OCDs. Their conjecture that their algorithm is complete is incorrect; it is incomplete.
(3) Consonni et al. [3] misinterpret our set-based canonical form for ODs [6, 7] (which is equivalent to the list-based canonical form for ODs). This leads the authors to confuse set-based OCDs with ODs. Their claim that our implementation has an error arises from this, and their belief that their approach is complete. Consonni et al. [3] conclude that their algorithm is faster in practice, despite being significantly worse in asymptotic complexity. This arises in their benchmark experiments, however, due to the fact that their algorithm is incomplete.

In Section 2, we provide basic definitions and canonical forms for ODs. In Section 3, we analyze the completeness of OD discovery. In Section 4, we discuss the experimental evaluation conducted by Consonni et al. [3]. We conclude in Section 5.

2 FOUNDATIONS

2.1 Background

We use the following notational conventions.

Table 2: Notational conventions.

- **Relations. R** denotes a *relation schema* and **r** denotes a specific *table* instance. Letters from the beginning of the alphabet, A, B and C, denote single *attributes*. Additionally, t and s denote *tuples*, and t_{A} denotes the value of an attribute A in a tuple t.
- **Sets.** Letters from the end of the alphabet, $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$, denote *sets* of attributes. Also, $t_{\mathcal{X}}$ denotes the *projection* of a tuple t on $\mathcal{X}$. $\mathcal{XY}$ is shorthand for $\mathcal{X} \cup \mathcal{Y}$. The empty set of attributes is denoted as $\{\}$.
- **Lists. X**, **Y** and **Z** denote *lists*. The empty list of attributes is represented as [ ]. List $[\mathsf{A}, \mathsf{B}, \mathsf{C}]$ denotes an explicit list. $[\mathsf{A} \mid \mathbf{T}]$ denotes a list with the *head* A and the *tail* **T**. **XY** is shorthand for **X** concatenate **Y**. Set $\mathcal{X}$ denotes the *set* of elements in *list* **X**. $\mathbf{X}^p$ denotes any arbitrary permutation of list **X** or set $\mathcal{X}$. Given a set of attributes $\mathcal{X}$, for brevity, we state $\forall i$, X_i to indicate indices $[1, ..., i]$ that have valid ranges ($i \leq |\mathcal{X}|$).

We provide a summary of the relevant definitions. The operator '$\preceq_{\mathbf{X}}$' defines a *weak total order* over any set of tuples, where **X** denotes a list of attributes. Unless otherwise specified, numbers are ordered numerically, strings are ordered lexicographically and dates are ordered chronologically.

Definition 2.1. [6, 7] Let **X** be a list of attributes. For two tuples t and s, $\mathcal{X} \in \mathbf{R}$, $t \preceq_{\mathbf{X}} s$ if[1]
– $\mathbf{X} = [\,]$; or
– $\mathbf{X} = [\mathsf{A} \mid \mathbf{T}]$ and $t_{\mathsf{A}} < s_{\mathsf{A}}$; or
– $\mathbf{X} = [\mathsf{A} \mid \mathbf{T}]$, $t_{\mathsf{A}} = s_{\mathsf{A}}$, and $t \preceq_{\mathbf{T}} s$.
Let $t \prec_{\mathbf{X}} s$ if $t \preceq_{\mathbf{X}} s$ but $s \not\preceq_{\mathbf{X}} t$.

Next, we define order dependencies.

Definition 2.2. [3, 5–8, 10] Let **X** and **Y** be lists of attributes over a relation schema **R**. Table **r** over **R** *satisfies* an OD $\mathbf{X} \mapsto \mathbf{Y}$ ($\mathbf{r} \models \mathbf{X} \mapsto \mathbf{Y}$), read as **X** *orders* **Y**, if for all $t, s \in \mathbf{r}$, $t \preceq_{\mathbf{X}} s$ implies $t \preceq_{\mathbf{Y}} s$. $\mathbf{X} \mapsto \mathbf{Y}$ is said to *hold* for **R** ($\mathbf{R} \models \mathbf{X} \mapsto \mathbf{Y}$) if, for each admissible table instance **r** of **R**, table **r** satisfies $\mathbf{X} \mapsto \mathbf{Y}$. $\mathbf{X} \mapsto \mathbf{Y}$ is *trivial* if, for all **r**, $\mathbf{r} \models \mathbf{X} \mapsto \mathbf{Y}$. $\mathbf{X} \leftrightarrow \mathbf{Y}$, read as **X** and **Y** are *order equivalent*, if $\mathbf{X} \mapsto \mathbf{Y}$ and $\mathbf{Y} \mapsto \mathbf{X}$.

[1] By some conventions, "*iff*"—"if and only if—would be used here. The intent, in any case, is that the use of "if" defines completely the notion.

The OD $\mathbf{X} \mapsto \mathbf{Y}$ means that **Y** values are monotonically non-decreasing wrt **X** values. Thus, if a list of tuples is ordered by **X**, then it is also ordered by **Y**, but not necessarily vice versa.

Example 2.3. Consider Table 1 in which tax is calculated as a percentage of salary, and tax groups and subgroups are based on salary. Tax, percentage and group are not decreasing with salary. Furthermore, within the same group, subgroup is not decreasing with salary. Finally, within the same year, bin increases with salary. Thus, the following order dependencies hold in that table: [salary] $\mapsto$ [tax], [salary] $\mapsto$ [percentage], [salary] $\mapsto$ [group, subgroup] and [year, salary] $\mapsto$ [year, bin].

Definition 2.4. [3, 5, 8, 10] Two order specifications **X** and **Y** are *order compatible*, denoted as $\mathbf{X} \sim \mathbf{Y}$, if $\mathbf{XY} \leftrightarrow \mathbf{YX}$. ODs in the form of $\mathbf{X} \sim \mathbf{Y}$ are called order compatible dependencies (OCDs)

The empty list of attributes (i.e., [ ]) is order compatible with *any* list of attributes. There is a strong relationship between ODs and FDs. Any OD implies an FD, modulo lists and sets, however, not vice versa.

LEMMA 2.5. [8, 10] If $\mathbf{R} \models \mathbf{X} \mapsto \mathbf{Y}$ (OD), then $\mathbf{R} \models \mathcal{X} \to \mathcal{Y}$ (FD).

Also, there is a correspondence between FDs and ODs.

THEOREM 2.6. [8, 10] $\mathbf{R} \models \mathcal{X} \to \mathcal{Y}$ *iff* $\mathbf{X} \mapsto \mathbf{XY}$, for any list **X** over the attributes of $\mathcal{X}$ and any list **Y** over the attributes of $\mathcal{Y}$.

ODs can be violated in two ways.

THEOREM 2.7. [8, 10] $\mathbf{R} \models \mathbf{X} \mapsto \mathbf{Y}$ (OD) *iff* $\mathbf{R} \models \mathbf{X} \mapsto \mathbf{XY}$ (FD) and $\mathbf{X} \sim \mathbf{Y}$ (OCD).

We are now ready to explain the two sources of OD violations: *splits* and *swaps* [8, 10]. An OD $\mathbf{X} \mapsto \mathbf{Y}$ can be violated in two ways, as per Theorem 2.7.

Definition 2.8. [8, 10] A *split* wrt an OD $\mathbf{X} \mapsto \mathbf{XY}$ (FD) is a pair of tuples s and t such that $s_{\mathcal{X}} = t_{\mathcal{X}}$ but $s_{\mathcal{Y}} \neq t_{\mathcal{Y}}$.

Definition 2.9. [8, 10] A *swap* wrt $\mathbf{X} \sim \mathbf{Y}$ (OCD) is a pair of tuples s and t such that $s \prec_{\mathbf{X}} t$, but $t \prec_{\mathbf{Y}} s$.

Example 2.10. In Table 1, there are three splits with respect to the OD [position] $\mapsto$ [position, salary] because position does not functionally determine salary. The violating tuple pairs are *t1* and *t4*, *t2* and *t5*, and *t3* and *t6*. There is a swap with respect to [salary] $\sim$ [subgroup], e.g., over the pair of tuples *t1* and *t2*.

2.2 Canonical Forms

Consonni et al. [3] use a native list-based canonical form, which is based on decomposing an OD into a FD and an OCD [8, 10]. Recall that based on Theorem 2.7 "OD = FD + OCD", as $\mathbf{X} \mapsto \mathbf{Y}$ *iff* $\mathbf{X} \mapsto \mathbf{XY}$ (FD) and $\mathbf{X} \sim \mathbf{Y}$ (OCD). The authors exploit this relationship to guide their discovery algorithm through order compatibility. Since they use a list-based representation for ODs, this leads to factorial complexity of OD discovery in the number of attributes.

Expressing ODs in a natural way relies on lists of attributes, as in the SQL order-by statement. One might well wonder whether lists are inherently necessary. We provide a polynomial *mapping* of list-based ODs into *equivalent* set-based canonical ODs [6, 7]. The mapping allows us to develop an OD discovery algorithm that traverses a much smaller set-containment lattice (to identify candidates for ODs) rather than the list-containment lattice used in Consonni et al. [3].

Two tuples, t and s, are *equivalent* over a set of attributes $\mathcal{X}$ if $t_{\mathcal{X}} = s_{\mathcal{X}}$. An attribute set $\mathcal{X}$ partitions tuples into *equivalence classes* [4]. We denote the *equivalence class* of a tuple $t \in \mathbf{r}$ over a set $\mathcal{X}$ as $\mathcal{E}(t_{\mathcal{X}})$, i.e., $\mathcal{E}(t_{\mathcal{X}}) = \{s \in \mathbf{r} \mid s_{\mathcal{X}} = t_{\mathcal{X}}\}$. A *partition* of $\mathbf{r}$ over $\mathcal{X}$ is the set of equivalence classes, $\Pi_{\mathcal{X}} = \{\mathcal{E}(t_{\mathcal{X}}) \mid t \in \mathbf{r}\}$. For instance, in Table 1, $\mathcal{E}(t1_{\{\text{year}\}}) = \mathcal{E}(t2_{\{\text{year}\}}) = \mathcal{E}(t3_{\{\text{year}\}}) = \{t1, t2, t3\}$ and $\Pi_{\text{year}} = \{\{t1, t2, t3\}, \{t4, t5, t6\}\}$.

We now present a set-based *canonical form* for ODs.

Definition 2.11. [6, 7] An attribute A is a *constant* within each equivalence class over $\mathcal{X}$, denoted as $\mathcal{X}$: $[\,] \mapsto \mathsf{A}$, if $\mathbf{X}^p \mapsto \mathbf{X}^p\mathsf{A}$. Furthermore, two attributes, A and B, are order-compatible within each equivalence class wrt $\mathcal{X}$, denoted as $\mathcal{X}$: $\mathsf{A} \sim \mathsf{B}$, if $\mathbf{X}^p\mathsf{A} \sim \mathbf{X}^p\mathsf{B}$. ODs of the form of $\mathcal{X}$: $[\,] \mapsto \mathsf{A}$ and $\mathcal{X}$: $\mathsf{A} \sim \mathsf{B}$ are called *(set-based) canonical* ODs, and the set $\mathcal{X}$ is called a *context*.

Example 2.12. In Table 1, the attribute `bin` is a constant in the context of `position` (`posit`) written as {`position`}: $[\,] \mapsto$ `bin`, since $\mathcal{E}(t1_{\{\text{position}\}}) \models [\,] \mapsto$ `bin`, $\mathcal{E}(t2_{\{\text{position}\}}) \models [\,] \mapsto$ `bin` and $\mathcal{E}(t3_{\{\text{position}\}}) \models [\,] \mapsto$ `bin`. Also, there is no swap between `bin` and `salary` in the context of `year`, i.e., {`year`}: `bin` $\sim$ `salary`, since $\mathcal{E}(t1_{\{\text{year}\}}) \models$ `bin` $\sim$ `salary` and $\mathcal{E}(t4_{\{\text{year}\}}) \models$ `bin` $\sim$ `salary`.

Based on Theorem 2.13 and Theorem 2.14, list-based ODs in the form of FDs and OCDs, respectively, can be mapped into equivalent set-based ODs.

THEOREM 2.13. [6, 7] $\mathbf{R} \models \mathbf{X} \mapsto \mathbf{XY}$ *iff* $\forall \mathsf{A} \in \mathbf{Y}, \mathbf{R} \models \mathcal{X}$: $[\,] \mapsto \mathsf{A}$.

THEOREM 2.14. [6, 7] $\mathbf{R} \models \mathbf{X} \sim \mathbf{Y}$ *iff* $\forall i, j, \mathbf{R} \models \{\mathsf{X}_1, .., \mathsf{X}_{i-1}, \mathsf{Y}_1, .., \mathsf{Y}_{j-1}\}$: $\mathsf{X}_i \sim \mathsf{Y}_j$.

A list-based OD can be mapped into an equivalent set of set-based ODs via a polynomial mapping.

THEOREM 2.15. [6, 7] $\mathbf{R} \models \mathbf{X} \mapsto \mathbf{Y}$ *iff* $\forall \mathsf{A} \in \mathbf{Y}, \mathbf{R} \models \mathcal{X}$: $[\,] \mapsto \mathsf{A}$ and $\forall i, j, \mathbf{R} \models \{\mathsf{X}_1, .., \mathsf{X}_{i-1}, \mathsf{Y}_1, .., \mathsf{Y}_{j-1}\}$: $\mathsf{X}_i \sim \mathsf{Y}_j$.

Example 2.16. The OD [AB] $\mapsto$ [CD] can be mapped to the following equivalent canonical ODs: {A, B}: $[\,] \mapsto$ C, {A, B}: $[\,] \mapsto$ D, {}: A $\sim$ C, {A}: B $\sim$ C, {C}: A $\sim$ D, {A, C}: B $\sim$ D.

3 COMPLETENESS ANALYSIS

While the theoretical search space for FASTOD [6, 7] is $O(2^{|\mathbf{R}|})$, the search space for OCDDISCOVER [3] is $O(|\mathbf{R}|!)$, which is much larger as it traverses a lattice of attribute *permutations* (where $|\mathbf{R}|$ denotes the number of attributes over a relational schema $\mathbf{R}$). To mitigate the factorial complexity, the list-based algorithm in Consonni et al. [3] uses pruning rules. We show that, despite the authors' claim that their approach discovers a canonically complete set of ODs, their pruning rules lead to *incompleteness*.

Section 3 in Consonni et al. [3] addresses their completeness "proof" for their OD discovery algorithm. The authors introduce a notion of *minimality* of a set of dependencies which is incorrect. Herein, a set of dependencies is called *minimal*—as it is in previous work on FDs and ODs [4, 6, 7]—if *all* dependencies that logically hold over a relation schema $\mathbf{R}$ can be inferred from this minimal (canonical) set of dependencies.[2] That is, a set of dependencies $\mathcal{M}$ is *minimal* over a table $\mathbf{r}$, if $\{\mathbf{X} \mapsto \mathbf{Y} \mid \mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}\}$ is equivalent to $\{\mathbf{X} \mapsto \mathbf{Y} \mid \mathbf{r} \models \mathbf{X} \mapsto \mathbf{Y}\}$.

[2]In some previous work [1], minimal dependencies $\mathcal{M}$ also satisfy an additional condition that that no proper subset of $\mathcal{M}$ can be used to infer all dependencies.

Thus, one should be able to infer from a minimal set of dependencies via the inference rules (axioms), $\mathcal{I}$, *all* the dependencies that are valid over the given instance of the table. That is, $\{\mathbf{X} \mapsto \mathbf{Y} \mid \mathcal{M} \vdash_{\mathcal{I}} \mathbf{X} \mapsto \mathbf{Y}\}$ is equal to $\{\mathbf{X} \mapsto \mathbf{Y} \mid \mathbf{r} \models \mathbf{X} \mapsto \mathbf{Y}\}$. Consonni et al. [3] use the set of *sound* and *complete* OD inference rules, $\mathcal{I}$, from [9, 10].

Pruning applied by a dependency discovery algorithm, thus, must respect minimality. This allows for the *implicit* discovery of the full set of valid dependencies, and thus be deemed *complete*.

In [3], an attribute list is minimal if it has no embedded order dependency (the list of attributes is the shortest possible).

Definition 3.1. [3] An attribute list $\mathbf{X}$ is *minimal* if there is no other list of attributes $\mathbf{X}'$ such that:

- $\mathbf{X}'$ is smaller than $\mathbf{X}$, and
- $\mathbf{X} \leftrightarrow \mathbf{X}'$

Example 3.2. [A, B, A] is *not* minimal as [A, B, A] $\leftrightarrow$ [A, B].

It follows then that an OCD is minimal in [3] *if and only if* there are no repeated attributes in the OCD. That is, there are no repeated attributes within the *left* or within the *right* list of the minimal OCD, as each is a minimal attribute list, *and* there is no repeated attribute between *left* and *right*.

Definition 3.3. [3] An OCD $\mathbf{X} \sim \mathbf{Y}$ is *minimal* if

- $\mathbf{X}$ and $\mathbf{Y}$ are minimal attribute lists *and*
- $\mathcal{X} \cap \mathcal{Y} = \emptyset$.

Definition 3.3 of *minimality* with no permitted repeated attributes is at the heart of the incompleteness problem of [3], as it does not allow for the inference of all dependencies that are valid over the given table. Theorem 3.4 states this, that an OCD with a common prefix between *left* and *right* (repeated attributes) can hold over a table, while no OCD without repeated attributes holds. Our proof of Theorem 3.4 is by offering a counter-example to the completeness premise in [3].

THEOREM 3.4. $\mathbf{R} \not\models \mathbf{Y} \sim \mathbf{Z}$, $\mathbf{R} \not\models \mathbf{XY} \sim \mathbf{Z}$ and $\mathbf{R} \not\models \mathbf{Y} \sim \mathbf{XZ}$ do not imply $\mathbf{R} \not\models \mathbf{XY} \sim \mathbf{XZ}$

Proof

It suffices to construct a table in which the OCD of the form

- $\mathbf{XY} \sim \mathbf{XZ}$

holds, but OCDs

- $\mathbf{Y} \sim \mathbf{Z}$,
- $\mathbf{XY} \sim \mathbf{Z}$, and
- $\mathbf{Y} \sim \mathbf{XZ}$

do not.

Consider Table 3 constructed over attributes A, B and C. In Table 3, the OCD [A, B] $\sim$ [A, C] holds, but [B] $\sim$ [C], [AB] $\sim$ [C], and [B] $\sim$ [AC] do not. □

In [3], the authors only show—as is stated in Theorem 3.5 below—that OCDs of the form $\mathbf{XY} \sim \mathbf{XZ}$ can be derived from $\mathbf{Y} \sim \mathbf{Z}$ (Theorem 3.5 via Theorem 3.10 in [3]).

THEOREM 3.5. [3] If $\mathbf{R} \models \mathbf{Y} \sim \mathbf{Z}$, then $\mathbf{R} \models \mathbf{XY} \sim \mathbf{XZ}$

Theorem 3.5 in [3] is *true*. The flaw in the authors' logic is that this theorem proves only *one* direction (the "if" of an intended "if and only if"). The "only if" (not proved by the theorem) is implicitly assumed as *true*, while it assuredly is not. It follows that their claim of canonical completeness for their definition of minimal OCDs is incorrect (Section 3.3 in [3]). OCDs with common prefixes between their *left* and *right* attribute lists are *not* redundant, by Theorem 3.4. This leads to an *incomplete* approach for OD

Table 3: Incompleteness of OCDDISCOVER [3].

#	A	B	C
t1	0	0	1
t2	1	1	0
t3	2	3	2
t4	3	2	3

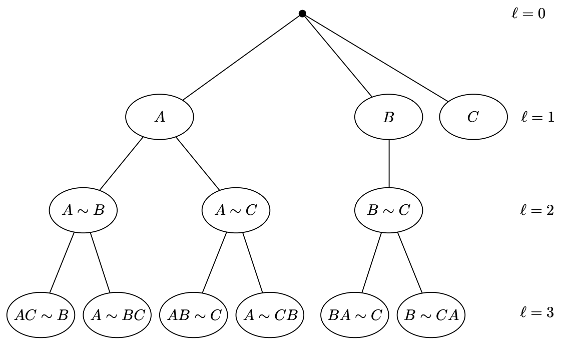


Figure 1: Lattice permutation tree for OCDDISCOVER [3].

discovery, as the recovery of the full set of valid dependencies is not possible.

Details of the OD discovery algorithm, OCDDISCOVER, by Consonni et al. [3] are presented in their Section 4. Let $\mathcal{U}$ be a set of attributes over a relation schema **R**. In the first level of the lattice, they generate candidates of the form $\mathsf{A} \sim \mathsf{B}$, where $\mathsf{A}, \mathsf{B} \in \mathcal{U}$ and $\mathsf{A} \neq \mathsf{B}$. (An OCD $\mathsf{B} \sim \mathsf{A}$ is not generated as it is equivalent to $\mathsf{A} \sim \mathsf{B}$.) At each level of the lattice (Fig. 1), if the candidate $\mathbf{X} \sim \mathbf{Y}$ is order compatible, they generate dependencies for the next level of the lattice. For each attribute not already present in the OCD, for each attribute $\mathsf{A} \in \mathcal{U} \setminus \{\mathcal{X} \cup \mathcal{Y}\}$, they add it to the right of each attribute list; i.e., $\mathbf{X}\mathsf{A} \sim \mathbf{Y}$ and $\mathbf{X} \sim \mathbf{Y}\mathsf{A}$. Thus, important OCDs with repeated attributes in a common prefix are never considered (as is consistent with their incorrect definition of minimality for OCDs). For example, an OCD [`year`, `month`] $\sim$ [`year`, `week`] would be missed. As a consequence, the authors do not discover ODs with repeated attributes, such as [`year`, `salary`] $\mapsto$ [`year`, `bin`] (recall Table 1).

In contrast, our FASTOD algorithm [6, 7] is *complete* for OD discovery. It does not miss dependencies with common prefixes. This is because the algorithm considers as candidates dependencies of the set-based form: OCD $\{\mathcal{X}\}$: $\mathsf{A} \sim \mathsf{B}$. This is built into the *context* of the set-based notation used in [6, 7], and cannot be missed when using this representation (see Theorem 2.14). Thus, dependencies with common prefixes are considered.

4 EXPERIMENTAL ANALYSIS

We demonstrate that the experimental analysis in Consonni et al. [3] that compares their OD discovery algorithm, OCDDISCOVER, with ours, FASTOD [6, 7], is incorrect. The authors misinterpret the set-based canonical representation for ODs as introduced in [6, 7] and as used in FASTOD. They conflate OCDs and ODs as we report them when evaluating the results. In [6, 7], we quantify the numbers of found FDs and OCDs. In [3], they incorrectly report these as the FDs and ODs, respectively, that we found. This occurs in their Table 6, where, for instance, they report 400 ODs and 89,571 FDs found by FASTOD, whereas this should be 400 OCDs and 89,571 FDs, respectively.

Table 4: Correctness of implementation for FASTOD [6].

#	A	B	C	D
t1	1	3	1	1
t2	2	3	3	2
t3	2	3	2	2
t4	2	5	2	2
t5	3	1	2	3
t6	4	4	4	2
t7	4	5	3	2

As a consequence of this misunderstanding of the set-based canonical representation for ODs [6, 7], the authors in [3] claim that the implementation of FASTOD finds ODs that are not present in the data. As an example of this, they provide the OD [B] $\mapsto$ [A, C] over Table 4 [3]. However, the FASTOD algorithm implementation in question finds the following ODs with respect to Table 4, where clearly the OD [B] $\mapsto$ [A, C] is not present.

(1) OCD {D}: A $\sim$ C
(2) OCD {C}: A $\sim$ D
(3) FD {A}:[ ] $\mapsto$ D
(4) OCD {B}: A $\sim$ D
(5) OCD {B}: C $\sim$ D
(6) OCD {B}: A $\sim$ C
(7) FD {B, C}:[ ] $\mapsto$ D
(8) FD {B, C}:[ ] $\mapsto$ A
(9) FD {A, B}:[ ] $\mapsto$ C
(10) OCD {C, D}: A $\sim$ B

The authors confuse the OCD {B}: A $\sim$ C with the OD [B] $\mapsto$ [A, C]. Consequently, they falsely assert that the reason the number of ODs found by OCDDISCOVER and FASTOD differ is due to an error in the implementation of FASTOD that we provided them.[3] The real reason that the number of reported dependencies differ, however, is, that OCDDISCOVER [3] is *incomplete*. The claim that they outperform the state-of-art despite a much worse asymptotic complexity, when tested in practice on real datasets, is invalid.

The authors in Consonni et al. [3] also state that FASTOD considers all columns to be of type string, while their code also considers real and integer numbers. While a minor point, we wish to clarify that the implementation we sent the authors does discover ODs over data types including real and integer numbers. The dependencies 1–10 reported in Table 4 remain the same, regardless of using numerical or string data type, given that the values are in the range of 1 to 5.

5 CONCLUSIONS

In this article, we have conducted a detailed analysis of the correctness of the results in the recent article by Consonni et al. [3] concerning the order dependency discovery problem. We have shown that, for the main claimed results related to the OD discovery problem, there are fundamental errors and omissions in the proof or experiments.

[3]While Consonni et al. [3] state that they were not able to isolate and resolve the root cause of what they felt was incorrect behavior in the implementation of FASTOD (which we had provided to them at their request for "ensuring fairness and reproducibility"), they never contacted us to help resolve it.

REFERENCES

[1] C. Beeri and P.A Bernstein. 1979. Computional Problems Related to the Design of Normal Form Relational Schemas. *TODS* 4, 1 (1979), 30–59.

[2] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. 2007. Improving Data Quality: Consistency and Accuracy. In *VLDB*. 315–326.

[3] C. Consonni, P. Sottovia, A. Montresor, and Y. Velegrakis. 2019. Discovering Order Dependencies through Order Compatibility. In *EDBT*. 409–420. https://doi.org/10.5441/002/edbt.2019.36

[4] Y. Huhtala, J. Karkkainen, P. Porkka, and H. Toivonen. 1998. Efficient Discovery of Functional and Approximate Dependencies Using Partitions. In *ICDE*. 392–401.

[5] P. Langer and F. Naumann. 2016. Efficient Order Dependency Detection. *VLDB J.* 25, 2 (2016), 223–241.

[6] 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.

[7] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. 2018. Effective and Complete Discovery of Bidirectional Order Dependencies via Set-based Axiomatization. *VLDB J.* 27, 4 (2018), 573–591.

[8] J. Szlichta, P. Godfrey, and J. Gryz. 2012. Fundamentals of Order Dependencies. *PVLDB, 5(11): 1220-1231* (2012).

[9] J. Szlichta, P. Godfrey, J. Gryz, W. Ma, W. Qiu, and C. Zuzarte. 2014. Business-Intelligence Queries with Order Dependencies in DB2. In *EDBT, 750-761*.

[10] J. Szlichta, P. Godfrey, J. Gryz, and C. Zuzarte. 2013. Expressiveness and Complexity of Order Dependencies. *PVLDB 6(14): 1858-1869* (2013).