

---

# Advances in Database Technology — EDBT 2016

19th International Conference  
on Extending Database Technology  
Bordeaux, France, March 15–18, 2016  
Proceedings

*Editors*

Evaggelia Pitoura  
Sofian Maabout  
Georgia Koutrika  
Amelie Marian  
Letizia Tanca  
Ioana Manolescu  
Kostas Stefanidis



Advances in Database Technology — EDBT 2016  
Proceedings of the 19th International Conference  
on Extending Database Technology  
Bordeaux, France, March 15–18, 2016

Series ISSN: 2367-2005

*Editors*

Evaggelia Pitoura, University of Ioannina, Greece  
Sofian Maabout, University of Bordeaux, France  
Georgia Koutrika, HP Labs, USA  
Amelie Marian, Rutgers University, USA  
Letizia Tanca, Politecnico di Milano, Italy  
Ioana Manolescu, INRIA, France  
Kostas Stefanidis, ICS-FORTH, Greece



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

COPYRIGHT NOTICE: Copyright © 2016 by the authors of the individual papers.

Distribution of all material contained in this volume is permitted under the terms of the Creative Commons license  
CC-by-nc-nd 4.0

OpenProceedings ISBN: 978-3-89318-070-7

DOI of this front matter: 10.5441/002/edbt.2016.01

# Foreword

Welcome to the 19th edition of the International Conference of Extending Database Technology (EDBT). Originally biennial, the EDBT conference has been held annually and jointly with ICDT (“International Conference on Database Theory”) since 2009. This year, EDBT is taking place in Bordeaux, France, on March 15–18, 2016, continuing its long tradition as a top venue for presenting and discussing recent advancements in data management.

This year we received 137 submissions to the research track, 18 submissions to the vision track, 24 submissions to the industrial/application track, 28 submissions to the demo track and 11 tutorial proposals. The high quality of these submissions made the job of selecting the best of them rather challenging. The various program committees after thorough reviewing and careful consideration selected 38 research papers, 5 vision papers, 9 industrial/application papers, 16 demos and 3 tutorials. The proceedings include these contributions. A new addition this year is the poster track for presenting novel ongoing work. There were 64 submissions from which the poster program committee selected 31 contributions included in this proceedings.

The proceedings also include an overview of the keynote talk by Elisa Bertino (Purdue), an overview of the keynote talk by Gustavo Alonso (ETHZ) and a laudation concerning the EDBT 2016 Test of Time Award that was given to the paper

“Bridging Physical and Virtual Worlds: Complex Event Processing for RFID Data Streams” by Fusheng Wang, Shaorong Liu, Peiya Liu, Yijian Bai, published in the EDBT 2006 proceedings.

The EDBT 2016 program is the result of the joint effort of many people that I would like to take this opportunity to thank. Ioana Manolescu (Vision Track Chair), Georgia Koutrika (Industrial/Application Track Chair), Letizia Tanca (Demo Track Chair) and Amelie Marian (Tutorial Chair), all did an excellent job, as Themis Palpanas with the workshops (whose proceedings appear in a companion volume). Thanks also to the members of the program committees of the various tracks that worked very hard to review each submission in detail and engaged in many discussions to create the best possible program.

Special mention should be made to the Test of Time Award committee members: Sihem Amer-Yahia, Yannis Ioannidis and Christian S. Jensen. The general chair, Sofian Maabout and the local organizers worked hard with all arrangements necessary for securing a successful event. Special thanks to Kostas Stefanidis, the proceedings chair, and Patrick Mary, the website chair, for their invaluable contribution to this event. Christine Collet and Norman Paton were instrumental in advising and coordinating with the EDBT Executive Board.

And lastly and most importantly, thanks to all the authors that submitted their work to EDBT 2016. Their contributions were what made this a strong program. I hope that you find the EDBT 2016 conference informative, enjoyable and thought-provoking!

Evaggelia Pitoura  
EDBT 2016 Program Chair

# Program Committee Members

## Research Program Committee

Bernd Amann (U Pierre et Marie Curie)      Wang-Chien Lee (Penn State U)  
Walid Aref (Purdue U)      Wolfgang Lehner (TU Dresden)  
Sourav S Bhowmick (Nanyang TU)      Hong-Va Leong (Hong Kong Polytechnik U)  
Michael Böhlen (U of Zurich)      Roy Levin (IBM Research)  
Klemens Böhm (KIT)      Feifei Li (U of Utah)  
Francesco Bonchi (Yahoo! Labs)      Xuemin Lin (U of New South Wales)  
Angela Bonifati (Lille 1 U)      Eric Lo (Honk Kong Polytechnik)  
Philippe Bonnet (ITU)      Norman May (SAP)  
Luc Bouganim (INRIA)      Sebastian Michel (TU Kaiserslautern)  
Nieves Brisaboa (U de La Coruna)      Kjetil Norvag (Norwegian U of Sc. & Tech.)  
Reynold Cheng (U of Hong Kong)      Ippokratis Pandis (Cloudera)  
Beng Chin Ooi (National U of Singapore)      Paolo Papotti (QCRI)  
Vassilis Christophides (INRIA Paris)      Marta Patino (Politecnico de Madrid)  
Panos K Chrysanthis (U of Pittsburgh)      Torben B Pedersen (U of Aalborg)  
Paolo Ciaccia (U of Bologna)      Peter Pietzuch (Imperial College)  
Philippe Cudre-Mauroux (U of Fribourg)      Maya Ramanath (IIT Delhi)  
Bin Cui (Peking U)      Matthias Renz (LMU)  
Alfredo Cuzzocrea (U of Trieste)      Rodolfo Resende (U Federal de Minas Gerais)  
Khuzaima Daudjee (U of Waterloo)      Tore Risch (Uppsala U)  
Antonios Deligiannakis (TU of Crete)      Pierangela Samarati (U Studi Milano)  
Elena Ferrari (U of Insubria)      Mohamed Sarwat (Arizona State U)  
Peter Fischer (U Freiburg)      Kai-Uwe Sattler (TU Ilmenau)  
Helena Galhardas (U of Lisbon)      Marc Scholl (U of Konstanz)  
Johann Gamper (Free U Bolzano)      Heiko Schuldt (U of Basel)  
Minos Garofalakis (TU of Crete)      Assaf Schuster (Technion)  
Floris Geerts (U of Antwerp)      Thomas Seidl (RWTH Aachen)  
Jiawei Han (UI Urbana Champaign)      Jianwen Su (UC Santa Barbara)  
Takahiro Hara (Osaka U)      Peter Triantafillou (U of Glasgow)  
Thomas Heinis (Imperial College)      Yannis Velegrakis (U of Trento)  
Arantza Illarramendi (U del Paes Vasco)      Stratis Viglas (U of Edinburgh)  
George Kollios (Boston U)      Jef Wijsen (U of Mons – UMONS)  
Georgia Koloniari (U of Macedonia)      Yoshitaka Yamamoto (U of Yamanashi)  
Yiannis Kotidis (Athens U of Bus. & Econ.)      Carlo Zaniolo (UCLA)  
Nick Koudas (U of Toronto)      Demetrios Zeinalipour-Yazti (U of Cyprus)  
Georg Lausen (U Freiburg)      Wenjie Zhang (U of New South Wales)

### **Vision Track Committee**

Nicolas Ancaux (INRIA Paris-Rocquencourt)  
Iovka Boneva (U Lille 1)  
Yanlei Diao (Ecole Polytechnique)  
Stratos Idreos (Harvard U)  
Yannis Ioannidis (U of Athens)  
Christian Jensen (Aalto U)  
Alekh Jindal (Microsoft)  
Zoi Kaoudi (QCRI)  
Giansalvatore Mecca (U della Basilicata)  
Leonid Libkin (U of Edinburgh)  
Neoklis Polyzotis (Google)  
Nicoleta Preda (U de Versailles)  
Eric Simon (SAP)  
Alessandro Solimando (INRIA)  
Fabian Suchanek (Télécom ParisTech)

### **Industrial Program Committee**

Andrey Balmin (Platfora)  
Fei Chen (HP Labs)  
Vuk Ercegovic (Google)  
Mohamed Eltabakh (Worcester PI)  
Irina Fundulaki (ICS-FORTH)  
Oktie Hassanzadeh (IBM Watson)  
Anastasios Kementsietsidis (Google)  
Lipyew Lim (U of Hawaii)  
Konstantinos Morfonios (Oracle)  
Lucian Popa (IBM Almaden Research)  
Lin Qiao (LinkedIn)  
Mohamed Sharaf (U of Queensland)  
Julia Stoyanovich (Drexel U)  
Nesime Tatbul (Intel Labs and MIT)  
Panayiotis Tsaparas (U of Ioannina)  
Steven (Euijong) Whang (Google)  
Kevin Wilkinson (HP)

### **Poster Track Committee**

Alberto Abelló (Politécnica de Catalunya)  
Nikolaus Augsten (U of Salzburg)  
Christos Doulkeridis (U of Piraeus)  
Ioana Giurgiu (IBM Research (Zurich))  
Aris Gkoulalas-Divanis (IBM Research)  
Sven Groppe (U of Lubeck)  
Katja Hose (Aalborg U)  
Verena Kantere (U of Geneva)  
Viktor Leis (Technische Ut Munchen)  
Paolo Missier (Newcastle U)  
Eirini Ntoutsis (LMU)  
Senjuti Basu Roy (U of Washington Tacoma)  
George Pallis (U of Cyprus)  
Shaoyu Song (Tsinghua U)

### **External Reviewers**

Daichi Amagata (Osaka U)  
Mohammad Amiri (UC Santa Barbara)  
Khaled Ammar (U of Waterloo )  
Christos Anagnostopoulos (U of Glasgow)

Carlos Andrade (U of Hawaii at Manoa)  
Ilaria Bartolini (U di Bologna)  
Dritan Bleco (AUEB)  
Carlos Bobed (U of Zaragoza)  
Douglas Burdick (IBM Research Almaden)  
Siarhei Bykau (Purdue U)  
Lijun Chang (UNSW)  
Georgios Chatzimilioudis (U of Cyprus)  
Sean Chester (NTNU)  
Pietro Colombo (U of Insubria)  
Camelia Constantin (U P&M Curie)  
Maria Daltayanni (U of San Francisco)  
Vasilis Efthymiou (U of Crete)  
Ioanna Filippidou (AUEB)  
George Fletcher (Eindhoven UT)  
Sara Foresti (U degli Studi di Milano)  
Daniele Foroni (U of Trento)  
Shi Gao (UCLA)  
Xiaoyu (Steve) Ge (U of Pittsburgh)  
Kostas Georgoulas (AUEB)  
Orestis Gkorgkas (NTNU)  
Alfredo Goni (Basque Country U)  
Zengfeng Huang (UNSW)  
Meng Jiang (UIUC)  
Julius Koepke (U of Klagenfurt)  
Mustafa Korkmaz (U of Waterloo)  
Zeynep Korkmaz (U of Waterloo)  
Christos Laoudias (U of Cyprus)  
Jialu Liu (UIUC)  
Giovanni Livraga (U Milano)  
Xiuli Ma (Peking U)  
Massimo Mazzeo (UCLA)  
Evica Milchevski (TU Kaiserslautern)  
Davide Mottin (U of Trento)  
Hubert Naacke (UPMC-LIP6)  
Nathan Rico Ong (U of Pittsburgh)  
Kiril Panev (TU Kaiserslautern)  
Marco Patella (U di Bologna)  
Fabio Petroni (Sapienza U of Rome)  
Yoann Pitarch (U Paul Sabatier)  
Donatello Santoro (U della Basilicata)  
Klaus Schmid (LMU)  
Konstantinos Semertzidis (U of Ioannina)  
Anatoli Shein (U of Pittsburgh)  
Masumi Shirakawa (Osaka U)  
Vasilis Spyropoulos (AUEB)  
Yan Tang (UC Santa Barbara)  
Io Taxidou (U of Freiburg)  
Cory Thoma (U of Pittsburgh)  
Sabrina De Capitani di Vimercati (U Milano)  
Xiaoyang Wang (UNSW)  
Doris Xin (UIUC)  
Mohan Yang (UCLA)  
Man Lung Yiu (Hong Kong Polytechnic U)  
Quan Yuan (UIUC)  
Roberto Yus (U of Zaragoza)  
Chao Zhang (UIUC)  
Andreas Zuefle (LMU)

# Test-of-Time Award

In 2014, EDBT began awarding the EDBT Test-of-Time (ToT) Award, with the goal of recognizing one paper, or a small number of papers, presented at EDBT earlier and that have best met the “test of time”, i.e., that has had the most impact in terms of research, methodology, conceptual contribution, or transfer to practice over the past decade(s). The EDBT ToT Award for 2016 will be presented during the EDBT/ICDT 2016 Joint Conference, March 15–18, 2016, in Bordeaux (France). The EDBT 2016 Test-of-Time Award committee was formed by Sihem Amer-Yahia (CNRS, Laboratoire d’Informatique de Grenoble, France), Yannis Ioannidis (University of Athens, Greece), Christian S. Jensen (Aalborg University, Denmark), and all PC chairs of former EDBT conferences including EDBT 2006.

The committee was asked to select a paper or a small number of papers from the EDBT 2006 (Munich) proceedings. After careful consideration, the committee and the EDBT Executive Board have decided to select the following paper as the EDBT ToT Award winner for 2016:

**Bridging Physical and Virtual Worlds:  
Complex Event Processing for RFID Data Streams**

by Fusheng Wang, Shaorong Liu, Peiya Liu, Yijian Bai

published in the EDBT 2006 proceedings, 588–607

The paper proposes an event-oriented approach to the processing of RFID data which makes it possible to automate the translation of RFID based application semantics through complex event detection. In particular, it demonstrates the ability to process complex events by capturing temporal constraints in an algebra. The resulting declarative event-based approach is shown to simplify RFID data processing and is shown to be scalable. The paper pioneers declarative event-based RFID processing. The simplicity and expressiveness of the proposed framework are admirable. For example, the framework makes it possible to express object tracking on historical data as well as to formulate real-time monitoring.

The committee and the EDBT Executive Board find that this paper stands out in terms of relevance, impact, and influence in databases. It has had substantial impact. In particular, it has impacted real systems, and the engine it proposes has been integrated into Siemens RFID Middleware. It is also the most cited EDBT 2006 paper, has spurred a significant amount of follow-up work, and remains relevant today.

# Table of Contents

Foreword . . . . .	i
Program Committee Members . . . . .	ii
Test-of-Time Award . . . . .	iv
Table of Contents . . . . .	v
<b>Invited Keynotes</b>	
Data Security and Privacy in the IoT <i>Elisa Bertino</i> . . . . .	1
Data Processing in Modern Hardware <i>Gustavo Alonso</i> . . . . .	4
<b>Research Papers</b>	
Finding Users of Interest in Micro-blogging Systems <i>Camelia Constantin, Ryadh Dahimene, Quentin Grossetti, Cedric Du Mouza</i> . . . . .	5
Slowing the Firehose: Multi-Dimensional Diversity on Social Post Streams <i>Shiwen Cheng, Marek Chrobak, Vagelis Hristidis</i> . . . . .	17
Social, Structured and Semantic Search <i>Raphaël Bonaque, Bogdan Cautis, François Goasdoué, Ioana Manolescu</i> . . . . .	29
Indexing Query Graphs to Speedup Graph Query Processing <i>Jing Wang, Nikos Ntarmos, Peter Triantafillou</i> . . . . .	41
GSCALER: Synthetically Scaling A Given Graph <i>J.W. Zhang, Y.C. Tay</i> . . . . .	53
Storing and Analyzing Historical Graph Data at Scale <i>Udayan Khurana, Amol Deshpande</i> . . . . .	65
Providing Serializability for Pregel-like Graph Processing Systems <i>Minyang Han, Khuzaima Daudjee</i> . . . . .	77
DBExplorer: Exploratory Search in Databases <i>Manish Singh, Michael Cafarella, Hosagrahar Visvesvar Jagadish</i> . . . . .	89
Refinement Driven Processing of Aggregation Constrained Queries <i>Manasi Vartak, Venkatesh Raghavan, Elke Rundensteiner, Samuel Madden</i> . . . . .	101
Reverse Engineering Top-k Database Queries with PALEO <i>Kiril Panev, Sebastian Michel</i> . . . . .	113
CrowdSky: Skyline Computation with Crowdsourcing <i>Jongwuk Lee, Dongwon Lee, Sang-Wook Kim</i> . . . . .	125
Cohesive Keyword Search on Tree Data <i>Aggeliki Dimitriou, Ananya Dass, Dimitri Theodoratos, Yannis Vassiliou</i> . . . . .	137
Generic Keyword Search over XML Data <i>Manoj Agarwal, Krithi Ramamritham, Prashant Agarwal</i> . . . . .	149
Answering Keyword Queries involving Aggregates and GROUPBY on Relational Databases <i>Zhong Zeng, Mong Li Lee, Tok Wang Ling</i> . . . . .	161

Finding All Maximal Cliques in Very Large Social Networks <i>Alessio Conte, Roberto De Virgilio, Antonio Maccioni, Maurizio Patrignani, Riccardo Torlone . . . . .</i>	173
RPM: Representative Pattern Mining for Efficient Time Series Classification <i>Xing Wang, Jessica Lin, Pavel Senin, Tim Oates, Sunil Gandhi, Arnold Boedihardjo, Crystal Chen, Susan Frankenstein . . . . .</i>	185
Interactive Temporal Association Analytics <i>Xiao Qin, Ramoza Ahsan, Xika Lin, Elke Rundensteiner, Matthew Ward . . . . .</i>	197
Efficient Record Linkage Using a Compact Hamming Space <i>Dimitrios Karapiperis, Dinusha Vatsalan, Vassilios Verykios, Peter Christen . . . . .</i>	209
Scaling Entity Resolution to Large, Heterogeneous Data with Enhanced Meta-blocking <i>George Papadakis, George Papastefanatos, Themis Palpanas, Manolis Koubarakis . . . . .</i>	221
Practical Query Answering in Data Exchange Under Inconsistency-Tolerant Semantics <i>Balder ten Cate, Richard Halpert, Phokion Kolaitis . . . . .</i>	233
Querying RDF Data Using A Multigraph-based Approach <i>Vijay Ingalalli, Dino Ienco, Pascal Poncelet, Serena Villata . . . . .</i>	245
Optimization of Complex SPARQL Analytical Queries <i>Padmashree Ravindra, HyeongSik Kim, Kemafor Anyanwu . . . . .</i>	257
RDF-TX: A Fast, User-Friendly System for Querying the History of RDF Knowledge Bases <i>Shi Gao, Jiaqi Gu, Carlo Zaniolo . . . . .</i>	269
Efficient Computation of Containment and Complementarity in RDF Data Cubes <i>Marios Meimaris, George Papastefanatos, Panos Vassiliadis, Ioannis Anagnostopoulos . . . . .</i>	281
Semi-automatic support for evolving functional dependencies <i>Mirjana Mazuran, Elisa Quintarelli, Letizia Tanca, Stefania Ugolini . . . . .</i>	293
Holistic Data Profiling: Simultaneous Discovery of Various Metadata <i>Jens Ehrlich, Mandy Roick, Lukas Schulze, Jakob Zwiener, Thorsten Papenbrock, Felix Naumann . . . . .</i>	305
Monitoring MaxRS in Spatial Data Streams <i>Daichi Amagata, Takahiro Hara . . . . .</i>	317
Similarity Search on Spatio-Textual Point Sets <i>Christodoulos Efstathiades, Alexandros Belesiotis, Dimitrios Skoutas, Dieter Pfoser . . . . .</i>	329
Nearest Window Cluster Queries <i>Chen-Che Huang, Jiun-Long Huang, Tsung-Ching Liang, Jun-Zhe Wang, Wen-Yuah Shih, Wang-Chien Lee . . . . .</i>	341
Adaptive query parallelization in multi-core column stores <i>Mrunal Gawade, Martin Kersten . . . . .</i>	353
PARAGON: Parallel Architecture-Aware Graph Partition Refinement Algorithm <i>Angen Zheng, Alexandros Labrinidis, Patrick Pisciuneri, Panos Chrysanthis, Peyman Givi . . . . .</i>	365
Query Workload-based RDF Graph Fragmentation and Allocation <i>Peng Peng, Lei Zou, Lei Chen, Dongyan Zhao . . . . .</i>	377
Efficient Query Processing using the Earth's Mover Distance in Video Databases <i>Merih Seran Uysal, Christian Becks, Daniel Sabinasz, Jochen Schmuecking, Thomas Seidl . . . . .</i>	389
Probabilistic Threshold Indexing for Uncertain Strings <i>Sudip Biswas, Manish Patil, Sharma Thankachan, Rahul Shah . . . . .</i>	401



Context-Aware Event Stream Analytics <i>Olga Poppe, Chuan Lei, Elke Rundensteiner, Dan Dougherty</i> . . . . .	413
Who Cares about Others' Privacy: Personalized Anonymization of Moving Object Trajectories <i>Despina Kopanaki, Vasilis Theodossopoulos, Nikos Pelekis, Ioannis Kopanakis, Yannis Theodoridis</i> . . .	425
Identifying and Describing Streets of Interest <i>Dimitrios Skoutas, Dimitris Sacharidis, Kostas Stamatoukos,</i> . . . . .	437
Finding Frequently Visited Indoor POIs Using Symbolic Indoor Tracking Data <i>Hua Lu, Chenjuan Guo, Bin Yang, Christian Jensen</i> . . . . .	449
<b>Visionary Papers</b>	
Designing Access Methods: The RUM Conjecture <i>Manos Athanassoulis, Michael Kester, Lukas Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, Mark Callaghan</i> . . . . .	461
Self-Curating Databases <i>Mohammad Sadoghi, Kavitha Srinivas, Oktie Hassanzadeh, Yuan-Chi Chang, Mustafa Canim, Achille Fokoue, Yishai Feldman</i> . . . . .	467
Data Wrangling for Big Data: Challenges and Opportunities <i>Tim Furché, Georg Gottlob, Leonid Libkin, Giorgio Orsi, Norman Paton</i> . . . . .	473
Road to Freedom in Big Data Analytics <i>Divy Agrawal, Sanjay Chawla, Ahmed Elmagarmid, Zoi Kaoudi, Mourad Ouzzani, Paolo Papotti, Jorge Quiane, Nan Tang, Mohammed Zaki</i> . . . . .	479
Data Management for Next Generation Genomic Computing <i>Stefano Ceri, Abdulrahman Kaitoua, Marco Masseroli, Pietro Pinoli, Francesco Venco</i> . . . . .	485
<b>Industrial and Applications Papers</b>	
Exploring Text Classification for Messy Data: An Industry Use Case for Domain-Specific Analytics <i>Laura Kassner, Bernhard Mitschang</i> . . . . .	491
Discovering Correlations in Annotated Databases <i>Xuebin He, Stephen Donohue, Mohamed Eltabakh</i> . . . . .	503
Query Performance Problem Determination with Knowledge Base in Semantic Web System OptImatch <i>Guilherme Damasio, Piotr Mierzejewski, Jaroslaw Szlichta, Calisto Zuzarte</i> . . . . .	515
Scalable Public Transportation Queries on the Database <i>Alexandros Efentakis</i> . . . . .	527
Characterizing Home Device Usage From Wireless Traffic Time Series <i>Katsiaryna Mirylenka, Vassilis Christophides, Themis Palpanas, Ioannis Pefkianakis, Martin May</i> . . .	539
Parallel Duplicate Detection in Adverse Drug Reaction Databases with Spark <i>Chen Wang, Sarvnaz Karimi</i> . . . . .	551
e#: Sharper Expertise Detection from Microblogs <i>Thibault Sellam, Martin Hentschel, Vasilis Kandylas, Omar Alonso</i> . . . . .	563
DECT: Distributed Evolving Context Tree for Mining Web Behavior Evolution <i>Xiaokui Shu, Nikolay Laptev, Danfeng Yao</i> . . . . .	573
Strudel: A Framework for Transaction Performance Analyses on SQL/NoSQL Systems <i>Junichi Tatemura, Oliver Po, Zheng Li, Hakan Hacigumus</i> . . . . .	580

## Demonstrations

GROM: a General Rewriter of Semantic Mappings <i>Giansalvatore Mecca, Guillem Rull, Donatello Santoro, Ernest Teniente</i> . . . . .	592
PowerQ: An Interactive Keyword Search Engine for Aggregate Queries on Relational Databases <i>Zhong Zeng, Mong Li Lee, Tok Wang Ling</i> . . . . .	596
Visualization Through Inductive Aggregation <i>Parke Godfrey, Jarek Gryz, Piotr Lasek, Nasim Razavi</i> . . . . .	600
Contextual Event Search: Finding Contextual Messages in Dynamic microblog Data Stream in Real Time <i>Manoj Agarwal, Divyam Bansal, Mridul Garg, Krithi Ramamritham</i> . . . . .	604
Answering Controlled Natural Language Questions on RDF Knowledge Bases <i>Giuseppe Mazzeo, Carlo Zaniolo</i> . . . . .	608
tPredictor: A Micro-blog Based System for Teenagers' Stress Prediction <i>Jing Huang, Qi Li, Zhuonan Feng, Yiping Li, Ling Feng</i> . . . . .	612
OSNI: Searching for Needles in a Haystack of Social Network Data <i>Shiwen Cheng, James Fang, Vagelis Hristidis, Harsha Madhyastha, Niluthpol Chowdhury Mithun, Dorian Perkins, Amit Roy-Chowdhury, Moloud Shahbazi, Vassilis Tsotras</i> . . . . .	616
PROX: Approximated Summarization of Data Provenance <i>Eleanor Ainy, Pierre Bourhis, Susan Davidson, Daniel Deutch, Tova Milo</i> . . . . .	620
PAW: A Platform for Analytics Workflows <i>Maxim Filatov, Verena Kantere</i> . . . . .	624
streamLoader: An Event-Driven ETL System for the On-line Processing of Heterogeneous Sensor Data <i>Marco Mesiti, Luca Ferrari, Stefano Valtolina, Giacomo Licari, Gianluca Galliani, Minh-San Dao, Koji Zettsu</i> . . . . .	628
TINTIN: a Tool for INcremental INtegrity checking of Assertions in SQL Server <i>Xavier Oriol, Ernest Teniente, Guillem Rull</i> . . . . .	632
Efficient regular path query evaluation using path indexes <i>George Fletcher, Jeroen Peters, Alexandra Poulouvassilis</i> . . . . .	636
Galaxy: A Platform for Explorative Analysis of Open Data Sources <i>Seyed-Mehdi-Reza Beheshti, Boualem Benatallah, Hamid Reza Motahari Nezhad,</i> . . . . .	640
OAPT: A Tool for Ontology Analysis and Partitioning <i>Alsayed Algergawy, Samira Babalou, Friederike Klan, Birgitta König-Ries</i> . . . . .	644
ShapeExplorer: Querying and Exploring Shapes using Visual Knowledge <i>Tong Ge, Yafang Wang, Gerard de Melo, Zengguang Hao, Andrei Sharf, Baoquan Chen</i> . . . . .	648
Distributed Secure Search in the Personal Cloud <i>Thu Le, Nicolas AnCIAUX, Sebastien Guilloton, Saliha Lallali, Philippe Pucheral, Iulian Sandu Popa, Chao Chen</i> . . . . .	652

## Poster Papers

Type-aware Web-search <i>Michael Gubanov, Anna Pyayt</i> . . . . .	656
Indexing and Querying A Large Database of Typed Intervals <i>Jianqiu Xu, Hua Lu, Bin Yao</i> . . . . .	658
Quantifying Likelihood of Change through Update Propagation across Top-k Rankings <i>Evica Milchevski, Sebastian Michel</i> . . . . .	660

Optimizing B+-Tree for PCM-Based Hybrid Memory <i>Lu Li, Peiquan Jin, Chengcheng Yang, Zhanglin Wu, Lihua Yue</i> . . . . .	662
A Data Mining Approach to Choosing Categorical Attributes for Ranked Lists <i>Koninika Pal, Sebastian Michel</i> . . . . .	664
Efficient Implementation of Joins over Cassandra DBs <i>Haridimos Kondylakis, Antonis Fountouris, Dimitris Plexousakis</i> . . . . .	666
Double Chain-Star: an RDF indexing scheme for fast processing of SPARQL joins <i>Marios Meimaris, George Papastefanatos</i> . . . . .	668
Minoan ER: Progressive Entity Resolution in the Web of Data <i>Vasilis Efthymiou, Kostas Stefanidis, Vassilis Christophides</i> . . . . .	670
Proposal of a Database Type and Aggregation Function for Accelerating Medical Genomics Study on RDBMS <i>Yoshifumi Ujibashi, Motoyuki Kawaba, Lilian Harada</i> . . . . .	672
The Best Bang for Your Bu(ck)g <i>Benjamin Dietrich, Tobias Müller, Torsten Grust</i> . . . . .	674
A Way to Automatically Enrich Biomedical Ontologies <i>Juan Antonio Lossio-Ventura, Mathieu Roche, Clement Jonquet, Maguelonne Teisseire</i> . . . . .	676
A Distributed Mining Framework for Influence in Evolving Entities <i>Tian Guo, Karl Aberer</i> . . . . .	678
Sweet KIWI: Statistics-Driven OLAP Acceleration using Query Column Sets <i>Sung-Soo Kim, Taewhi Lee, Moonyoung Chung, Jongho Won</i> . . . . .	680
On-Line Mobility Pattern Discovering using Trajectory Data <i>Ticiana Coelho da Silva, Karine Zeitouni, José Fernandes de Macêdo, Marco Casanova</i> . . . . .	682
Summarizing Linked Data RDF Graphs Using Approximate Graph Pattern Mining <i>Mussab Zneika, Claudio Lucchese, Dan Vodislav, Dimitris Kotzinos</i> . . . . .	684
Understanding Customer Attrition at an Individual Level: a New Model in Grocery Retail Context <i>Clément Gautrais, Peggy Cellier, Thomas Guyet, René Quiniou, Alexandre Termier</i> . . . . .	686
Towards an Efficient Ranking of Interval-Based Patterns <i>Marwan Hassani, Yifeng Lu, Thomas Seidl</i> . . . . .	688
SOFYA: Semantic on-the-fly Relation Alignment <i>Maria Koutraki, Nicoleta Preda, Dan Vodislav</i> . . . . .	690
Model Kit for Lightweight Data Compression Algorithms <i>Juliana Hildebrandt, Dirk Habich, Patrick Damme, Wolfgang Lehner</i> . . . . .	692
Revisiting DBMS Space Management for Native Flash <i>Sergey Hardock, Ilija Petrov, Robert Gottstein, Alejandro Buchmann</i> . . . . .	694
A Two Phase Deep Learning Model for Identifying Discrimination from Tweets <i>Shuhan Yuan, Xintao Wu, Yang Xiang</i> . . . . .	696
Top-k Dominating Queries, in Parallel, in Memory <i>Sean Chester, Orestis Gkorgkas, Kjetil Nørkvåg</i> . . . . .	698
Snapshot Isolation for Neo4j <i>Marta Patino, Ricardo Jimenez-PEris, Diego Burgos-Sancho, Ivan Brondino, Valerio Vianello, Rohit Dhamane</i> . . . . .	700

Maximum Coverage Representative Skyline <i>Malene S�holm, Sean Chester, Ira Assent</i> . . . . .	702
An On-Line Approximation Algorithm for Mining Frequent Closed Itemsets Based on Incremental Inter- section <i>Koji Iwanuma, Yoshitaka Yamamoto, Shoshi Fukuda</i> . . . . .	704
Extending Database Accelerators for Data Transformations and Predictive Analytics <i>Felix Beier, Knut Stolze, Daniel Martin</i> . . . . .	706
Privacy Protection through Query Rewriting in Smart Environments <i>Hannes Grunert, Andreas Heuer</i> . . . . .	708
DatShA :A Data Sharing Algebra for access control plans <i>Luc Bouganim, Athanasia Katsouraki, Benjamin Nguyen</i> . . . . .	710
Cluster-based Contextual Recommendations <i>Kostas Stefanidis, Eirini Ntoutsi</i> . . . . .	712
Empirical evaluation of guarded structural indexing <i>Erik Agterdenbos, George Fletcher, Chee-Yong Chan, Stijn Vansummeren</i> . . . . .	714
Context-Dependent Quality-Aware Source Selection for Live Queries on Linked Data <i>Barbara Catania, Giovanna Guerrini, Beyza Yaman</i> . . . . .	716
<b>Tutorials</b>	
Data Responsibly: Fairness, Neutrality and Transparency in Data Analysis <i>Julia Stoyanovich, Serge Abiteboul, Gerome Miklau</i> . . . . .	718
Core Decomposition in Graphs: Concepts, Algorithms and Applications <i>Fragkiskos D. Malliaros, Apostolos N. Papadopoulos, Michalis Vazirgiannis</i> . . . . .	720
Distance-based Multimedia Indexing <i>Christian Beecks, Merih Seran Uysal, Thomas Seidl</i> . . . . .	722

# Data Security and Privacy in the IoT

Elisa Bertino  
Department of Computer Science  
Purdue University  
West Lafayette, IN 47907  
bertino@purdue.edu

## ABSTRACT

Deploying existing data security solutions to the Internet of Things (IoT) is not straightforward because of device heterogeneity, highly dynamic and possibly unprotected environments, and large scale. In this paper, after outlining key challenges in data security and privacy, we summarize research directions for securing IoT data, including efficient and scalable encryption protocols, software protection techniques for small devices, and fine-grained data packet loss analysis for sensor networks.

## 1. INTRODUCTION

The Internet of Things (IoT) paradigm refers to the network of physical objects or “things” embedded with electronics, software, sensors, and connectivity to enable objects to exchange data with servers, centralized systems, and/or other connected devices based on a variety of communication infrastructures. IoT makes it possible to sense and control objects creating opportunities for more direct integration between the physical world and computer-based systems. When IoT is augmented with sensors and actuators, IoT is able to support cyber-physical applications by which networked objects can impact the physical environment by taking “physical” actions. IoT will usher automation in a large number of domains, ranging from manufacturing and energy management (e.g. SmartGrid), to healthcare management and urban life (e.g. SmartCity). Applications range from monitoring the moisture in a field of crops, to tracking the flow of products through a factory, to remotely monitoring patients with chronic illnesses and remotely managing medical devices, such as implanted devices and infusion pumps. Forecasts by McKinsey&Company estimate that the economic impact of IoT technology by year 2025 will range from 2.7 to 6.2 trillion dollars [7]. Gartner forecasts predict that by the year 2020 20.8 billions of IoT devices will be installed. Such staggering numbers show that IoT will have a major impact.

However, while on one side, IoT will make many novel

applications possible, on the other side IoT increases the risk of cyber security attacks. In addition, because of its fine-grained, continuous and pervasive data acquisition and control/actuation capabilities, IoT raises concerns about privacy and safety. A recent study by HP about the most popular devices in some of the most common IoT niches reveals an alarmingly high average number of vulnerabilities per device [10]. On average, 25 vulnerabilities were found per device. For example, 80% of devices failed to require passwords of sufficient complexity and length, 70% did not encrypt local and remote traffic communications, and 60% contained vulnerable user interfaces and/or vulnerable firmware [10]. Multiple attacks have already been reported in the past against different embedded devices [2], [16] and we can expect many more in the IoT domain.

## 2. SECURITY AND PRIVACY RISKS FOR IOT

IoT systems are at high security risks for several reasons. They do not have well defined perimeters, are highly dynamic, and continuously change because of mobility. In addition IoT systems are highly heterogeneous with respect to communication medium and protocols, platforms, and devices. IoT systems may also include “objects” not designed to be connected to the Internet. Finally, IoT systems, or portions of them, may be physically unprotected and/or controlled by different parties. Attacks, against which there are established defense techniques in the context of conventional information systems and mobile environments, are thus much more difficult to protect against in the IoT. The OWASP Internet of Things Project [1] has identified the most common IoT vulnerabilities and has shown that many such vulnerabilities arise because of the lack of adoption of well-known security techniques, such as encryption, authentication, access control and role-based access control. A reason for the lack of adoption may certainly be security unawareness by IT companies involved in the IoT space and by end-users. However another reason is that existing security techniques, tools, and products may not be easily deployed to IoT devices and systems, for reasons such as the variety of hardware platforms and limited computing resources on many types of IoT devices. Even well known encryption protocols, such as RSA, prove to be very expensive when running on devices with limited computing capabilities especially when multiple encryption operations have to be executed concurrently such as in the case of networked vehicles [12], and small drones [14].

Privacy is particularly critical in the context of IoT. As

medical and well-being devices are increasingly been adopted by users and personalized medicine and health care applications are being designed and deployed that rely on continuous fine-grained data acquisition from these devices, the human body is becoming a rich source of information. Such information is typically collected from devices and then uploaded to some cloud and/or transmitted to other devices, such as mobile phones, which in turn may forward the information to other parties. The collected information is typically very rich and often includes meta-data such as location, time, and context, thus making possible to easily infer personal habits, behaviors, and preferences of individuals. It is thus clear that on one side such information has to be carefully protected by all parties involved in its acquisition, management, and use, but also users should be provided with suitable, easy to use tools to protect their privacy and support anonymity depending on specific contexts [11].

### 3. RESEARCH DIRECTIONS

Developing comprehensive security and privacy solutions for IoT requires revisiting almost all security techniques we may think of. Encryption protocols need to be engineered so to be efficient and scalable for deployment on large-scale IoT systems and devices with limited computational resources. Benchmarks are needed to perform detailed assessments of such protocols [14]. In addition, as devices may be physically unprotected, attackers may have access to the state of the memory while encryption operations are being performed. Addressing such problems may require new techniques based, for example, on white-box cryptography [3]. White-box encryption techniques hide encryption keys by transforming them into large look-up tables in order to make harder for attackers to extract the keys. Such techniques are however very expensive and many of the proposed white-box encryption protocols have been cryptanalyzed. Introducing dynamics in the look-up tables by a shuffling approach [15] may help addressing such problem. In addition, scalability of such protocols is critical, in that in many safety-sensitive applications encryption operations must be very efficient. For example, in a vehicle network, a message from a vehicle informing other vehicles of a sudden break should be processed very quickly in order to give the other vehicles enough time to break. Carefully engineered approaches taking advantage of specialized hardware, such as GPUs, available on systems on chip must be designed and benchmarked [12].

Software running on the devices must also be secured. Major challenges here arise from the fact that many IoT devices are based on processors such the ARM processor, which have differences in the instruction sets with respect to other conventionally used processors. Such diversity has an implication for example on the techniques for protecting software from attacks, such as return-oriented programming attacks, as such techniques must be tailored to the specific instruction set of the platform of interest [6]. Other research issues concern how to protect at run-time software from memory vulnerabilities. Solutions to this problem may have to take into account the specific programming languages used on IoT devices, such the case of nesC used in TinyOS, and the resource limitations [8]. Also well-known software management practices, like remote software patching and firmware updates, may become difficult if at all possible in an IoT environment and may actually open the door to additional attacks [5], [4]. Communication protection and defense tech-

niques against novel botnet attacks that exploit IoT devices [8] are also critical.

Data security, availability, and quality are other critical areas for IoT. Data security requires, in addition to the use of encryption to secure the data while being transmitted and at rest, access control policies to govern access to data, by taking into account information on data provenance and meta-data concerning the data acquisition context, such as location and time [9]. Availability requires among other things to make sure that relevant data is not lost. Addressing such requirement entails designing protocols for data acquisition and transmission that have data loss minimization as a key security goal. Kinesis [13] is an example of a sensor network system designed to make it possible for sensors to automatically take response actions in the event of data transmission disruptions. Ensuring data quality is a major critical requirement in IoT as data acquired and transmitted by IoT devices may be of poor quality, because of several reasons such as bad device calibration, device faults, and deliberate attacks aiming at data deception attacks. Solutions like data fusion need to be revised and extended to deal with dynamic environments and large-scale heterogeneous data sources.

Finally privacy introduces new challenges, including how to prevent personal devices from acquiring and/or transmitting information depending on the user location and other context information, and how to allow users to understand risks and advantages in sharing their personal data.

### 4. CONCLUDING REMARKS

IoT technology introduces several exciting opportunities and new applications. However, it is critical that solutions be adopted to ensure security, privacy, and safety of IoT systems with minimal impact on performance, scalability, and usability. Even though the computer and network security area has offered over the years many important techniques and methods, revisiting and extending these techniques and methods in order to address the specificities of IoT systems entails many scientific and engineering challenges.

### 5. ACKNOWLEDGMENTS

The work reported in this paper has been partially supported by the Purdue Cyber Center and the National Science Foundation under grant CNS-1111512.

### 6. REFERENCES

- [1] [https://www.owasp.org/index.php/OWASP\\_Internet\\_of\\_Things\\_Project](https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project).
- [2] S. K. Bansal. Linux worm targets internet-enabled home appliances to mine cryptocurrencies. <http://thehackernews.com/2014/03/linux-worm-targets-internet-enabled.html>, March 2014.
- [3] A. Bogdanov and T. Isobe. White-box cryptography revisited: Space-hard ciphers. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1058–1069, 2015.
- [4] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 95–110, 2014.

- [5] A. Cui, M. Costello, and S. J. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.
- [6] J. Habibi, A. Panicker, A. Gupta, and E. Bertino. Disarm: Mitigating buffer overflow attacks on embedded devices. In *Network and System Security - 9th International Conference, NSS 2015, New York, NY, USA, November 3-5, 2015, Proceedings*, pages 112–129, 2015.
- [7] J. Manyika, M. Chui, J. Bughin, R. Dobbs, P. Bisson, and A. Marrs. Disruptive technologies: Advances that will transform life, business, and the global economy. [http://www.mckinsey.com/insights/business\\_technology/disruptive\\_technologies](http://www.mckinsey.com/insights/business_technology/disruptive_technologies), May 2013.
- [8] D. Midi, M. Payer, and E. Bertino. nesCheck: Static analysis and dynamic instrumentation for nesC memory safety. 2016. Submitted for publication.
- [9] R. V. Nehme, H. Lim, and E. Bertino. FENCE: continuous access control enforcement in dynamic data stream environments. In *Third ACM Conference on Data and Application Security and Privacy, CODASPY'13, San Antonio, TX, USA, February 18-20, 2013*, pages 243–254, 2013.
- [10] K. Rawlinson. Hp study reveals 70 percent of internet of things devices vulnerable to attack. <http://www8.hp.com/us/en/hp-news/press-release.html?id=1744676#.VpfsZ8ArJcw>, July 2014.
- [11] B. Shebaro, O. Oluwatimi, D. Midi, and E. Bertino. Identidroid: Android can finally wear its anonymous suit. *Transactions on Data Privacy*, 7(1):27–50, 2014.
- [12] A. Singla, A. Mudgerikar, I. Papapanagiotou, and A. A. Yavuz. Haa: Hardware-accelerated authentication for internet of things in mission critical vehicular networks. In *IEEE Military Communications Conference*, 2015.
- [13] S. Sultana, D. Midi, and E. Bertino. Kinesis: a security incident response and prevention system for wireless sensor networks. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems, SenSys '14, Memphis, Tennessee, USA, November 3-6, 2014*, pages 148–162, 2014.
- [14] J. Won, S. Seo, and E. Bertino. A secure communication protocol for drones and smart objects. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 249–260, 2015.
- [15] J. Won, S. Seo, and E. Bertino. White-box attack-resistant dynamic block cipher for vehicular networks. 2016. Submitted for publication.
- [16] A. Wright. Hacking cars. *Commun. ACM*, 54(11):18–19, 2011.

# Data Processing in Modern Hardware

Gustavo Alonso  
Systems Group  
Department of Computer Science  
ETH Zurich  
alonso@inf.ethz.ch

## ABSTRACT

Data processing is changing in radical ways from how it has developed in the last four to five decades.

On the one hand, data science and big data have brought an unprecedented growth and variety in data sizes, demanding workloads, data types, and applications. From studying social networks on graph data to genomics over string matching algorithms; from low latency key value stores used to retrieve user profiles to large scale data appliances focusing on data warehousing; from real time stream data processing to database engines on cloud platforms, the types, scope, and requirements on data management engines has grown enormously.

On the other hand, hardware is no longer a source of performance as it has been in the last decades. Instead, it has become a complex, fast evolving, highly specialized, and heterogeneous platform that requires considerable tuning and effort to use optimally. Today, hardware is not becoming necessarily faster per se but provides instead a wide range of options for accelerating and tuning applications through new features. Unlike what happened in the past, applications in general and database engines in particular, have to work much harder to extract performance improvements from new hardware as the exploitation of these new features is not automatic and often requires a redesign of the system. In addition, many of the opportunities offered by modern hardware are still without adequate support from high level tools

such as compilers or debuggers, placing quite a burden on system designers.

In this talk I will discuss the issues in data processing that arise as a result of modern hardware: the need to deal with parallelism and distribution, the increasing importance of networking, the proliferation of accelerators, and the raise of heterogeneity in the machine. These issues are both a threat and a challenge, demanding a radical redesign of many aspects of data processing and database engines. Using examples from recent work ranging from query scheduling to hardware accelerators, I will present several exciting and radically new directions that are opening up for database research as a result of the advances being made in hardware. An important theme in the talk is the call for database designers and researchers to become proactive and identify the hardware features and characteristics that are needed to better support data processing.

## ACKNOWLEDGMENTS

I would like to thank all the members of the Systems Group for many informal discussions on the evolution of hardware and data processing over the years that have been instrumental to shape the vision outlined in this keynote.



# Finding Users of Interest in Micro-blogging Systems

Camelia Constantin  
Univ. Pierre et Marie Curie  
2 Place Jussieu  
F75005 Paris, France  
camelia.constantin@lip6.fr

Ryadh Dahimene  
CNAM  
2 rue Conté  
F75141 Paris, France  
ryadh.dahimene@cnam.fr

Quentin Grossetti  
CNAM  
2 rue Conté  
F75141 Paris, France  
quentin.grossetti@cnam.fr

Cédric du Mouza  
CNAM  
2 rue Conté  
F75141 Paris, France  
dumouza@cnam.fr

## ABSTRACT

Micro-blogging systems have become a prime source of information. However, due to their unprecedented success, they have to face an exponentially increasing amount of user-generated content. As a consequence finding users who publish quality content that matches precise interest is a real challenge for the average user. This paper presents a new recommendation score which takes advantage both of the social graph topology and of the existing contextual information to recommend users to follow according to user interest. Then we introduce a landmark-based algorithm which allows to scale. The experimental results and the user studies that we conducted confirm the relevance of this recommendation score against concurrent approaches as well as the scalability of the landmark-based algorithm.

## 1. INTRODUCTION

Micro-blogs have become a major trend over the Web 2.0 as well as an important communication vector. Twitter, the main micro-blogging service, has grown in a spectacular manner to reach more than 570 million users in April, 2014 in less than seven years of existence. Currently around 1 million new accounts are added to Twitter every week, while there were only 1,000 in 2008. 500 million tweets are sent every day and on average a Twitter user follows 108 accounts<sup>1</sup>. Facebook is another example with 1.26 billion users who publish on average 36 posts a month. A Facebook user follows on average 130 “friends” which results in 1,500 pieces of information a user is exposed on average when he logs in<sup>1</sup>. Other similar systems like Google+, Instagram, Youtube, Sina Weibo, Identi.ca or Plurk, to quote the largest, also exhibit dramatic growth.

<sup>1</sup><http://expandedramblings.com>

This fast and unprecedented success has introduced several challenges for service providers as well as for their users. While the former have to face a tremendous flow of user generated content, the latter struggle to find relevant data that matches their interests: they usually have to spend a long time to read all the content received, trying to filter out relevant information. Two (complementary) strategies have emerged to help the user to find relevant data that matches his interest in the huge flow of user generated content: posts filtering like in [13, 14, 8] and posts/account searches and/or recommendation like in [7, 4, 5]. Social network systems usually offer the ability to search for posts or accounts that match a set of keywords. This could be a “local” search to filter out the posts received, or a “global” search to query the whole set of existing posts/accounts. For the latter search, there exist two options: some pre-computed posts sets that correspond to the hot topics at query time, or customized searches where the query result is built according to the keywords specified by the user. However, the broad match semantics generally adopted by the searching tools is very limited. Even a ranking score based on the number of keywords is not sufficient to retrieve all posts of interest. Combined with the lack of semantics and the number of posts a day, the large number of searches performed every day also raises scalability issue. For instance in 2012, more searches were performed each month (24 billion) on Twitter than on Yahoo (9.4 billion) or on Bing (4.1 billion).

In this paper we consider the problem of discovering quality content publishers by providing efficient, topological and contextual user recommendations on the top of a micro-blog social graph. Micro-blogging systems are characterized by the existence of a large directed social graph where each user (accounts) can freely decide to connect to any other user for receiving all his posts. In this paper we make the assumption that a link between a user  $u$  and a user  $v$  expresses an interest of  $u$  for one or several topics from the content published by  $v$ . We consequently choose to model the underlying social network graph as a *labeled social graph*, where labels correspond to the topics of interest of the users. Our objective is to propose a recommendation score that captures both the topological proximity and connectivity of a publisher along with his authority regarding a given topic and the interest of the intermediary users between the one to be recommended and the publisher.

The size of the underlying social graph raises challenging issues especially when we consider operations that involve a graph exploration. In order to speed up the recommendation process we propose a fast approximate computation based on landmarks, *i.e.*, we select a set of nodes in the social graph, called *landmarks*, which will play the role of hubs and store data about their neighborhood. This set of landmarks is selected using different strategies we compare experimentally in Section 5.

**Contributions.** In this paper we propose a recommendation system that produces personalized user recommendations. Our main contributions are:

- 1) considering the idea that measures based on the graph topology are good indicators for the user similarity, we propose a topological score which integrates semantic information on users and their relationships;
- 2) furthermore, we introduce a landmark-based approach to improve recommendation computation time and to achieve a 2-3 order of magnitude gain compare to the exact computation;
- 3) an experimental validation of our approach, including a comparative study with other approaches (TwitterRank [26] and Katz [16]).

Observe that we illustrate our proposal in the context of micro-blogging systems, but our model is general and may be used for any social networks where users publish content and receive posts from the accounts they follow.

The paper is organized as follows: after the introduction in Section 1, we present the related work in Section 2. Section 3 describes our model and our recommendation scores along with their composition property. Then we propose our fast recommendation computation based on a landmark strategy in Section 4. Our experimental validation is presented in Section 5 while Section 6 concludes the paper and introduces future work.

## 2. RELATED WORK

Recommendation systems for social networks were recently proposed like [16, 4, 26, 11, 7, 21, 10, 3, 28, 24]. The work in [16] presents a comparison of different topological-based recommendation methods adapted in the context of link-prediction. In [2], authors propose to combine two ranking scores estimated with a fair bets approach on the user invitation graph and on the profile browsing graph. The work in [24] presents a user recommendation system which exploits node similarity scores estimated as a combination of local and global scores. Local score is based on the number of neighbors of the query node and of the recommendation node while global score involved the shortest path between them. Thus the recommendation scores for these approaches are only based on the topology and unlike our proposal do not consider neither content nor authority of the users. On the other side, the work in [20] finds users with high topical authority scores in micro-blogging systems. Unlike our method, those scores are not personalized, the system computes global authority scores, which in our case are used as parameters of the recommendation scores computed for some user.

Other approaches, like [4, 11], consider collaborative filtering. The work in [4] introduces a collaborative tweet ranking based on preference information of the users, authority of the publisher and the quality of the content. Recommendations are produced at a tweet level. Hannon et al. [11] evaluate a set of profiling strategies to provide user recommendations based on content, *e.g.* the tweets of the user or the tweets of his followers, or collaborative filtering. The methods proposed by [21] and [7] also provide tweet recommendations. Pennacchiotti et al. [21] analyze the tweets content as well as the content of the user’s direct neighbors while Diaz-Aviles et al. [7] use the user past interaction to compute rankings in real-time. All these works consider content but unlike our proposal they do not consider paths longer than direct follower/followee links.

Few papers combine content and topology of the social graph. Wend et al. [26] present an extension of the PageRank algorithm named TwitterRank which captures both the link structure and the topical similarity between users. However this similarity is based on topics provided by LDA and their distance-based similarity computation between users does not capture the semantic similarity between topics. We also propose an authority score for an account which estimates the local and global influence of this account for a given topic. Our scoring function also provides higher weight for short paths to favor ”local” recommendations. In [10], authors describe the production recommender system implemented in *Twitter*. It relies on an adaptation of the SALSA algorithm [15] which provides user recommendation in a centralized environment based on a bipartite graph: the user’s circle of trust, computed with random walks from the user considering content properties, and the accounts followed by the users from the circle of trust (the authorities). Our approach is different since it captures the users interest through the labeled social graph, which allows to compute scores based on semantics, on authority and on topology.

To scale and to accelerate our recommendations computations, we pre-compute scores for a subset of nodes named landmarks. Landmark-based approach is a well-known *divide-and-conquer* strategy for the shortest paths problem that have been shown to scale up to very large graphs [25, 9, 22, 23]. The idea is to select a set  $L$  of nodes as landmarks which store the distance to other nodes. The distance  $d(u, v)$  between two nodes  $u$  and  $v$  is then estimated by computing the minimum  $d(u, l) + d(l, v)$ , where  $l \in L$ . Das Sarma et al. [23] chose to pre-compute the time-consuming shortest-path operations for each node in structures called ”sketches” and to use them to provide shortest-path estimations at query-time which enables scaling for large web graphs. Gubichev et al. [9] extend the sketch-based algorithm proposed by [23] to retrieve shortest paths and improve the overall accuracy. Tretyakov et al. [25] use shortest-path trees to achieve an efficient and accurate estimation which support updates. They also introduce a landmark selection strategy that attempts to maximize the shortest-paths coverage. In [22] authors also investigated the impact of landmark selection on the accuracy of distance estimations. They proved that optimizing the landmark coverage is a NP-hard problem and showed experimentally that clever landmark selection strategies yield better results. Similar to these approaches, we employ landmarks for computation scaling and use some of the existing landmark selection strategies in the context of user recommendation.

### 3. MODEL

We introduce in this section the underlying social graph model and our recommendation scores. Table 1 lists the different notations used throughout the paper.

$N, E$	resp. set of nodes and edges
$\Gamma^u, \Gamma^u(t)$	followers for $u$ (resp. total or on topic $t$ )
$\mathcal{T}$	topics vocabulary
$label_n, label_e$	labeling function for nodes and for edges, resp.
$topo_\beta(u, v)$	topological score of $v$ for $u$ with decay factor $\beta$
$\sigma(u, v, t)$	recommendation score of $v$ for $u$ on topic $t$
$\widetilde{\sigma}_S(u, v, t)$	approximate recommendation score considering paths going through a node $n \in S$
$\omega_p(t)$	topical component of the path score for path $p$
$\overline{\omega}_p(t)$	path score for path $p$
$auth(u, t)$	node authority score for $u$ on topic $t$
$\varepsilon_e(t)$	edge relevance of edge $e$ on topic $t$
$\alpha, \beta$	decay factor for an edge and path resp.
$\lambda, \mathcal{L}$	a landmark, the set of landmarks
$P_{u,v}$	set of all paths between $u$ and $v$
$P_{u,\lambda,v}$	set of all paths between $u$ and $v$ through $\lambda$
$\Upsilon_k(\lambda)$	the $k$ -vicinity of $\lambda$
$R_{u,v}$	recommendation vector of $v$ for $u$ for all topics

Table 1: List of notations

#### 3.1 Labeled social graph

We model the *Twitter* social network as a directed labeled graph  $G=(N, E, \mathcal{T}, label_N, label_E)$  where  $N$  is set of vertices such that each vertex  $u \in N$  is a user (account).  $E \subseteq N \times N$  is a set of edges where an edge  $e = (u, v) \in E$  exists if  $u$  follows  $v$ , *i.e.*,  $u$  receives the publications of  $v$ . The labeling function  $label_N : N \rightarrow 2^{\mathcal{T}}$  maps each user to the set of topics that characterize his posts, chosen in a topic vocabulary  $\mathcal{T}$ . The topics associated by the labeling function  $label_E : E \rightarrow 2^{\mathcal{T}}$  to an edge  $e = (u, v)$  describe the interest of the user  $u$  for the posts of  $v$ . In this paper topics are extracted from tweets by using OpenCalais<sup>1</sup> combined with a trained Support Vector Multi-Label Model using Mulan<sup>2</sup> (see Section 5). For a user  $u$ , we define  $\Gamma^u(t)$  the set of nodes following  $u$  on topic  $t$  and by  $\Gamma^u$  the set of all his followers. An example of such graph is depicted in Figure 1. For users  $B$  and  $C$  we display their topics of interest along with an excerpt of their tweets.

#### 3.2 Recommendation

For a user  $u$  and a query composed of several topics  $Q = \{t_1, \dots, t_n\}$ , our model recommends users  $v$  based on the following criteria which consider both graph topology and content semantics:

- (i) *user proximity*:  $u$  trusts his friends, the friends of his friends, etc., but this confidence decreases with distance ;
- (ii) *the number of paths* from  $u$  to  $v$ : user  $v$  is likely to be more important for  $u$  if there are many other relevant users (*i.e.*, linked to  $u$ ) who recommend  $v$ ;
- (iii) *topical path relevance* of the connections between  $u$  and  $v$  with respect to  $Q$ .

<sup>1</sup><http://www.opencalais.com/>

<sup>2</sup><http://mulan.sourceforge.net/>

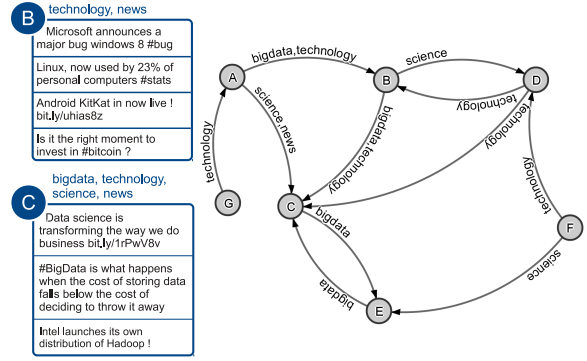


Figure 1: A labeled social graph

Our recommendation score combines the topical relevance of paths with a topological measure which considers *all existing paths* between two nodes  $u$  and  $v$  of the graph. More precisely, the recommendation score  $\sigma(u, v, t)$  of the user  $v$  for user  $u$  on topic  $t$  on paths  $p = u \rightsquigarrow v$  is the sum of all path scores  $\overline{\omega}_p(t)$  and is expressed as follows:

DEFINITION 1 (RECOMMENDATION SCORE).

$$\sigma(u, v, t) = \sum_{p \in P_{u,v}} \overline{\omega}_p(t) = \sum_{p \in P_{u,v}} \beta^{|p|} \omega_p(t) \quad (1)$$

where  $P_{u,v}$  denotes the set of all paths between  $u$  and  $v$ ,  $\overline{\omega}_p(t)$  is the total path score of a path of length  $|p|$  and  $\omega_p(t)$  the topical relevance  $\omega_p(t)$ . The decay factor  $\beta \in [0, 1]$  gives more importance to shorter paths.

The final recommendation score for the query  $Q$  is computed as a weighted linear combination (some are proposed in [1]) where user scores for each individual topic  $t_i \in Q$  are weighted by the relevance of  $t_i$  for the posts of  $u$  which is computed by the topic extraction method (see Section 5).

Note that we can deduce from Equation 1 a score which considers only the topology by ignoring the topical relevance of paths (*i.e.* setting  $\omega_p(t)$  to 1). This score is higher if there exist many short paths between  $u$  and  $v$  and is denoted as :

$$topo_\beta(u, v) = \sum_{p \in P_{u,v}} \beta^{|p|} \quad (2)$$

It corresponds to the Katz score [16] that has been successfully employed for link prediction. We will use it as a baseline for comparison with our method in Section 5.

The topical relevance  $\omega_p(t)$  of a path  $p = u \rightsquigarrow v$  for a topic  $t$  in Equation 1 considers both the relevance of nodes (*user authority*) and the topical relevance of edges (*edge relevance*) on the path  $p$  *w.r.t* the topics of the query  $Q$ . We define in the following these concepts.

**Edge relevance:**

Each edge on a path  $p$  contributes to the score of  $p$  with a semantical score which depends on its topics. Distant edges contribute less to the recommendation score than edges close to  $u$ . More precisely, the relevance of an edge  $e$  at distance  $d$  from  $u$  on path  $p$  for a topic  $t$  is defined as :

$$\varepsilon_e(t) = \alpha^d \times \max_{t' \in label_E(e)} (sim(t', t)) \quad (3)$$

where  $label_E(e)$  is the labeling function that returns the topics associated to the edge  $e$ . The decay factor  $\alpha \in [0, 1]$  decreases the influence of an *edge* according to its distance from  $u$ . The function  $sim : \mathcal{T}^2 \rightarrow \mathbb{R}$  computes the semantic similarity between two topics  $t$  and  $t'$ . We use in the present paper the Wu and Palmer similarity measure [27] on top of the WORDNET<sup>3</sup> database (we have a small number of topics for labeling our dataset without synonymy issues), but other semantic distance measures, such as RESNIK or DISCO<sup>4</sup> could also be used. The choice of the best similarity function is beyond the scope of the current paper. When an *edge* is labeled with several topics, we only keep the maximum similarity to  $t$  among all its topics to avoid high scores for edges labeled with many topics that have small similarity to  $t$ .

### User authority:

We define a per node topical authority function  $auth(u, t)$  of  $u$  on a topic  $t$  which depends on the number of users who follow  $u$  on  $t$ . The authority score is decomposed into two scores: (i) the *local authority score* that gives a higher score to users that are specialized on topic  $t$  than to users  $u$  who publish on a broad range of topics and (ii) the *global popularity score* that gives higher scores to users that are more followed on  $t$ . Combination of local and global scores has also been used to compute authorities for Web pages [12] or micro-blogging [10]. The authority score  $auth(u, t)$  of a user  $u$  on a topic  $t$  is consequently defined as follows :

$$auth(u, t) = \underbrace{\frac{|\Gamma^u(t)|}{|\Gamma^u|}}_{local} \times \underbrace{\frac{\log(1 + \Gamma^u(t))}{\log(1 + \max_{v \in N}(\Gamma^v(t)))}}_{global}$$

where  $|\Gamma^u(t)|$  is the number of followers of  $u$  on  $t$ , and  $|\Gamma^u|$  is its total number of followers. We used the logarithm function to smooth the difference between popular accounts and accounts with very few followers. The local authority is 1 when  $u$  is followed exclusively on  $t$  and the global popularity is 1 when  $u$  is the most followed user on  $t$ . If no other user follows  $u$  on  $t$  both scores are 0. The authority scores for a given topic  $t$  are high for users that are mainly followed on topic  $t$  and that have a significant number of followers. The combination of both local and global scores leads to similar authority scores for very specialized accounts with few followers and for very popular but generalist accounts. Observe for scores update that  $|\Gamma^u|$  and  $|\Gamma^u(t)|$  can be computed on local information of each user, without graph exploration. Oppositely the computation of  $\max_{v \in N}(\Gamma^v(t))$  may be costly since it requires to query the complete graph. However, the log strongly limits the impact of a variation in the popularity of an account with millions of followers, and we can assume this value is stored (and re-computed periodically).

**EXAMPLE 1 (LOCAL AND GLOBAL AUTHORITY).** Consider the example graph in Figure 1, with a sample of tweets for the users  $B$  and  $C$  along with their topics. User  $B$  is more relevant for technology than  $C$ . Indeed  $B$  and  $C$  have the same global popularity with two followers on this topic for both accounts. However the local authority of  $B$  on technology is

<sup>3</sup><http://wordnet.princeton.edu/>

<sup>4</sup>[http://www.linguatools.de/disco/disco\\_en.html](http://www.linguatools.de/disco/disco_en.html)

higher than the one of  $C$  since 2 out of the three topics on which  $B$  is followed are technology, whereas for  $C$  only 2 out of the 6 topics on which it is followed are technology. For the topic bigdata, the local authority of  $B$  and  $C$  is the same (1 out of 3 for  $B$  and 2 out of 6 for  $C$ ) but  $C$  is more followed on bigdata (2 users who follow him) than  $B$  (1 user). Therefore, the total authority of  $C$  on bigdata is higher.

### Topical path relevance:

Finally, we consider that the path relevance of  $p$  is high when both the relevance of the nodes and the one of the edges of  $p$  are high:

$$\omega_p(t) = \sum_{e \in p} \varepsilon_e(t) \times auth(end(e), t) \quad (4)$$

where  $end(e)$  returns the end node of the edge  $e$ . The recommendation score of  $v$  for the user  $u$  on topic  $t$  is then obtained by replacing  $\omega_p(t)$  in equation (1) by its formula given by equation (4). The resulting user recommendation score thus captures the topology (proximity and connectivity) of the graph along with the followers interests (expressed as labeled edges) and the authority score regarding the topic of interest for each user on the path.

**EXAMPLE 2 (TOPICAL PATH RELEVANCE).** In Fig. 1, we want to recommend to  $A$  users on the topic technology (we suppose a search within a range  $k = 2$ ). Users  $D$  and  $E$  can be reached with respectively the paths  $p_1 = A \rightarrow B \rightarrow D$  and path  $p_2 = A \rightarrow C \rightarrow E$ , each of length 2. The relevance of the edge  $A \xrightarrow{bigdata, technology} B$  is higher than the one of  $C \xrightarrow{bigdata} E$ , since the first one is at distance 1 from  $A$ , whereas the second is at distance 2. Moreover, the authority of node  $B$  on technology (computed as  $(local) \times (global) = \frac{2}{3} \times \frac{\log(1+2)}{\log(1+2)}$ ) is higher than the authority of  $C$  on technology ( $\frac{2}{6} \times \frac{\log(1+2)}{\log(1+2)}$ ). Overall, the semantic relevance of the edges on  $p_1$  for technology is higher than the one of edges on  $p_2$  and  $D$  obtains a higher recommendation score than  $E$ .

## 3.3 Score analysis

We will show in the following the iterative formula for score computation and the score composition property that is used in Section 4 for landmark-based computation.

### Iterative score computation.

Recommendation scores  $\sigma(u, v, t)$  (Equation 1) are computed by using the Power Iteration algorithm [19] (see Algorithm 1 in Section 4). It starts by initializing  $\sigma(u, u, t) = 1$  and  $\sigma(u, v, t) = 0$  ( $\forall u \neq v$ ). At each step  $i$ , a new score  $\sigma(u, v, t)^{(i)}$  (that considers all paths from  $u$  to  $v$  with length  $\leq i$ ) is computed by using the scores  $\sigma(u, v, t)^{(i-1)}$  of the neighbors  $w \in \Gamma^v$  computed at step  $(i-1)$ . The computation is performed until convergence. The iterative formula for score computation is the following :

**PROPOSITION 1 (ITERATIVE COMPUTATION).**

$$\sigma^{(i)}(u, v, t) = \sum_{w \in \Gamma^v} (\beta \cdot \sigma^{(i-1)}(u, w, t) + topo_{\alpha\beta}^{(i-1)}(u, w) \cdot \bar{\omega}_{w \rightarrow v}(t)) \quad (5)$$

where  $topo_{\alpha\beta}^{i-1}(u, w)$  is the topological score (see Equation 2) with a decaying factor of  $\alpha \cdot \beta$ . The score  $\bar{\omega}_{w \rightarrow v}(t) = \beta \cdot \alpha$ .

$\max_{t' \in \text{label}_E(w \rightarrow v)}(\text{sim}(t', t)) \cdot \text{auth}(v, t)$  is the score of a path that contains only the edge  $w \rightarrow v$  with topic  $t$ .

PROOF. Suppose a path  $p$  of length  $k \leq i$  from  $u$  to  $v$ . This path can be decomposed into a path  $p_1$  of length  $k-1$  from  $u$  to the neighbor  $w$  of  $v$  and an edge  $e$  from  $w$  to  $v$  of length 1. By using Equations (3) and (4), the score  $\bar{w}_p(t)$  is computed as:  $\bar{w}_p(t) = \beta \cdot \bar{w}_{p_1}(t) + \beta^{|\mathcal{P}_1|} \cdot \alpha^{|\mathcal{P}_1|} \cdot \bar{w}_e(t) = \beta \cdot \bar{w}_{p_1}(t) + \beta^{k-1} \cdot \alpha^{k-1} \cdot (\beta \cdot \alpha \cdot \max_{t' \in \text{label}_E(w \rightarrow v)}(\text{sim}(t', t)) \cdot \text{auth}(v, t))$ . The score  $\bar{w}_{p_1}(t)$  corresponds to a path that finishes at  $w$ .

We can re-organize the paths in Equation 1 by grouping those that pass through the same neighbor  $w$  of  $v$ :  $\sigma(u, v, t) = \sum_{p \in P_{u,v}} \bar{w}_p(t) = \sum_{w \in \Gamma^{v^-}(t)} (\sum_{p \in P_{u,v}, w \in p} \bar{w}_p(t))$ . By replacing  $\bar{w}_p(t)$  into this equation we obtain the iterative score formula.  $\square$

### Score composition.

From the iterative score computation we can deduce for each path  $p$  from a node  $u$  to a node  $v$  its total path score  $\bar{w}_p(t)$  on topic  $t$  from the score of its sub-paths already computed using the following property :

PROPOSITION 2 (RECOMMENDATION SCORE COMPOSITION). Assume a path  $p = p_1.p_2$ , with  $\bar{w}_{p_1}(t)$  and  $\bar{w}_{p_2}(t)$  the total path scores of respectively  $p_1$  and  $p_2$  for a topic  $t$ . The total path score of  $p$  can be computed as:

$$\bar{w}_p(t) = \beta^{|\mathcal{P}_2|} \cdot \bar{w}_{p_1}(t) + \beta^{|\mathcal{P}_1|} \cdot \alpha^{|\mathcal{P}_1|} \cdot \bar{w}_{p_2}(t)$$

PROOF. By induction using the recursive formula, we prove the proposition for paths with length  $k \geq 1$ .  $\square$

### Iterative score computation convergence.

In order to show the convergence of the iterative computation of recommendation scores  $\sigma(u, v, t)$  of users  $v$  for user  $u$  on topic  $t$  (Equation (5)), we express this computation in matrix form as :

$$R_t^{(k+1)} = (\beta A)R_t^{(k)} + (\beta \alpha)S_t T_{\alpha\beta}^{(k)} \quad (6)$$

where  $R_t^{(k)}$  is the recommendation vector for topic  $t$  computed at step  $k$  ( $R_t^{(k)}[v]$  is the recommendation score  $\sigma(u, v, t)$  computed at step  $(k)$ ). Matrix  $A$  is the adjacency matrix of the graph ( $A[v][u] = 1$  if  $u$  follows  $v$ ). Matrix  $S_t$  is the similarity-authority matrix ( $S_t[v][u] = \text{sim}(\max_{t' \in \text{label}_E(u \rightarrow v)}(t', t)) \times \text{auth}(v, t)$ ). Vector  $T_{\alpha\beta}^{(k)}$  is the topological vector at step  $k$  ( $T_{\alpha\beta}^{(k)}[v]$  is the topological score  $\text{topo}_{\alpha\beta}^{(k)}(u, v)$ ). It can be expressed as follows :

$$T_{\alpha\beta}^{(k+1)} = \alpha\beta A T_{\alpha\beta}^{(k)} + I$$

where  $I[u] = 1$  and  $I[v] = 0$  for all  $u \neq v$ . We deduce that the computation convergence is achieved under the following condition :

PROPOSITION 3 (SCORES COMPUTATION CONVERGENCE). If  $\beta < 1/\sigma_{\max}(A)$ , where  $\sigma_{\max}(A)$  is the highest eigen value for  $A$ , then the iterative scores computation of our recommendation scores converges.

PROOF. Based on the recursive formula which defines the topical vector for a given node  $n$ , the topical scores matrix defined by the series expansion

$$T_{\alpha\beta} = \sum_{i=1}^{\infty} \alpha\beta^i A^i = (I - \alpha\beta A)^{-1} - I$$

converges when  $I - \alpha\beta A$  is positive definite, so  $\alpha\beta < 1/\sigma_{\max}(A)$ . Consider a step  $k'$  when convergence is reached for  $T_{\alpha\beta}$ . Then for any  $k > k'$ , we have the recursive computation  $R^{(k+1)} = (\beta A)R^{(k)} + C$  with  $C = (\alpha\beta)S T_{\alpha\beta}^{(\infty)}$  constant. The convergence for  $R^{(k+1)}$  is reached when  $R^{(\infty)} = (\beta A)R^{(\infty)} + C$  thus when  $R^{(\infty)} = (I - \beta A)^{-1} \times C$ . This can be achieved if  $\beta < 1/\sigma_{\max}(A)$ . Since  $\beta > \alpha\beta$ , this later condition is sufficient to ensure convergence.  $\square$

## 4. LANDMARK-BASED COMPUTATION

The recommendation score computation presented in the previous section assumes to explore *all* paths from a user  $u$  to the nodes to be recommended. Computing recommendation scores by graph exploration at  $k$  hops for a graph with  $n$  nodes supposes to consider  $\text{out}_{avg}^k$  paths for the average case ( $\text{out}_{avg}$  denotes the average out degree) and of  $N^k$  paths in the worst case for a complete graph. This might be prohibitive in the context of social graphs with millions of nodes and edges. We rely here on a landmark-based approach to propose fast approximate recommendations.

The computation is performed in two steps: (i) in the preprocessing step we precompute for a sample of nodes in the graph, named landmarks, top- $n$  recommendation scores ( $n$  being a parameter of the system) for every topic  $t \in \mathcal{T}$  and (ii) at query time we compute approximate recommendations by exploring the graph until a given depth (also a parameter of the system) and collect precomputed recommendations from landmarks encountered during this exploration.

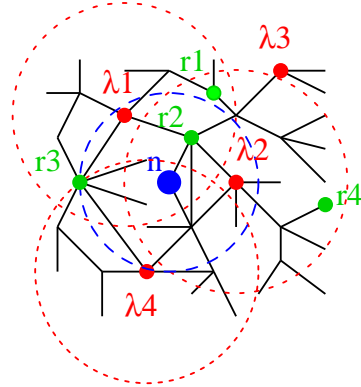


Figure 2: Landmark-based recommendation

EXAMPLE 3. Figure 2 illustrates this approach, where  $n$  is the query node and  $\lambda_1, \lambda_2, \lambda_3$  and  $\lambda_4$  are landmarks. When performing the graph exploration represented by the blue dashed-line from the node  $n$ , the landmarks  $\lambda_1, \lambda_2$  and  $\lambda_4$  are encountered. Node  $r_2$  is encountered during exploration and its score for  $n$  is computed at the same time as the scores for  $\lambda_1, \lambda_2$  and  $\lambda_4$ .  $r_1$  is not encountered during the exploration from  $n$ , but it is encountered by the explorations starting from landmarks  $\lambda_1$  and  $\lambda_2$ . Its recommendation score for  $n$  is estimated by aggregating the scores of  $\lambda_1$  and  $\lambda_2$  for  $n$  computed at query time with the scores of  $r_1$  for  $\lambda_1$  and  $\lambda_2$  which were precomputed.

## 4.1 Preprocessing

For the preprocessing step we consider a subset  $\mathcal{L} \subset N$  of nodes, so-called the landmarks, with  $|\mathcal{L}| \ll |N|$ . Instead of a random sampling, several strategies may be considered to determine  $\mathcal{L}$ . For instance, landmark-based approaches for computing shortest paths within a large graph rely mainly on centrality properties (betweenness or closeness centrality) to determine the sampling. The publisher-follower characteristics of our graph also allow other topology-based sampling, like a selection of the nodes with the most important number of followers (most popular accounts) or the ones that follow the highest number of accounts (most active readers). While the choice of the landmarks may impact the global performances of our approach we do not investigate further the sampling strategies in the current paper. Nonetheless some of these sampling techniques are experimentally compared in Section 5.

---

### Algorithm 1: LANDMARK\_RECMM( $\lambda, max_k, T, \beta, n$ )

---

**Require:** landmark ( $\lambda$ ), maximum exploration ( $max_k$ ), set of topics ( $T$ ), topological decay factor ( $\beta$ ), number of results to return ( $n$ )  
**Ensure:** a set of recommendation list  $R_\lambda$ , a topological vector  $topo_\beta(\lambda)$

```

1:  $\Upsilon_0 := \lambda, k := 0$ 
2: while  $k < max_k$  and  $converged = false$  do
3:    $\Upsilon_{k+1} := \emptyset$ 
4:   for all  $u \in \Upsilon_k$  do
5:      $\Upsilon_{k+1} := \Upsilon_{k+1} \cup \Gamma^u$ 
6:     for all  $v \in \Gamma^u$  do
7:       for all  $t \in T$  do
8:          $\sigma^{(k+1)}(\lambda, v, t) +=$ 
            $\beta \times \sigma^{(k)}(\lambda, u, t) + topo_{\alpha\beta}^k(\lambda, u) \times \bar{w}_{u \rightarrow v}$ 
9:       end for
10:       $topo_\beta^{(k+1)}(\lambda, v) += \beta \times topo_\beta^k(\lambda, u)$ 
11:    end for
12:     $R_t[u] += \sigma^{(k)}(\lambda, u, t)$ 
13:     $topo_\beta(\lambda, u) += topo_\beta^k(\lambda, u)$ 
14:  end for
15:  if  $(\sum_{u \in \Upsilon_k} \sigma^k(\lambda, u, t)) / |R_t| < tol, \forall t \in T$  then
16:     $converged := true$ 
17:  end if
18:   $k := k + 1$ 
19: end while
20: return for all  $t \in T$   $top-n((R_t), topo_\beta(\lambda))$ 

```

---

Algorithm 1 performs the recommendation computation (we remove the initialization of recommendation scores to simplify the presentation) and is used both in the preprocessing and in the approximate score computation step. It takes as parameters the starting node  $\lambda$  of the graph exploration, the maximum exploration depth  $max_k$ , the set of topics on which the recommendations are computed  $T$ , the path decay factor  $\beta$  and the number  $n$  of final results to be kept.

The set of reached nodes at depth  $k$  from  $\lambda$  is called the  $k$ -vicinity of  $\lambda$ , denoted  $\Upsilon_k(\lambda)$ .  $\Upsilon_\infty(\lambda)$  denotes the set of reachable nodes from  $\lambda$ . For each topic  $t \in T$  the algorithm computes a recommendation vector  $R_t$  with  $R_t[u] = \sigma(\lambda, u, t)$  (see Equation 6), where  $u \in \Upsilon_\infty(\lambda)$  is a reachable node from  $\lambda$ . The algorithm also computes the topological scores  $topo_\beta(\lambda, u)$  with decay factor  $\beta$  for all  $u \in \Upsilon_\infty$  (Equation 2), used to estimate the final recommendation scores at query time (see below). Iteration in line 4 allows to ex-

plore the  $k$ -vicinity of  $\lambda$ . For each iteration we add in the  $k$ -vicinity nodes that could be reached with an additional hop (l. 5). For each node  $v$  reached at this step (l. 6), we compute (or update if the node has been already encountered) the recommendation score for each term of the topic vocabulary (l. 7-8), and the node's topological score (l. 10). The score for  $u$  on paths of length  $k$  is added to the sorted topical (l.12) and topological (l.13) lists of  $\lambda$ . Finally, only the top- $n$  recommendations for each vector  $R_t$  and only the top- $n$  topological scores  $topo_\beta(\lambda)$  are stored.

In the preprocessing step, for each landmark  $\lambda \in \mathcal{L}$  we compute the recommendation scores on all topics for all nodes encountered during the iterative computation. So Algorithm 1 runs until the convergence is reached ( $max_k$  is set to a large value) with the parameter  $T$  set to  $\mathcal{T}$ .

## 4.2 Fast approximate recommendation

We now present the algorithm for fast approximate recommendations based on the pre-computations performed for each landmark in the preprocessing step. We assume that we want to recommend to an account  $u$  other accounts for a topic  $t$ .

We first perform a graph exploration starting from  $u$ , similar to the one described by the Algorithm 1, for a given maximal depth  $k$ ,  $max_k$ , set to a small value (e.g. 2 or 3). The graph exploration finds landmarks in the  $k$ -vicinity of  $u$  and computes path scores on topic  $t$  for paths from  $u$  to each encountered landmark. These scores are further combined with the scores stored by the landmarks in order to compute the approximate recommendation scores.

More precisely, we denote  $\Lambda \subseteq \mathcal{L}$  the set of landmarks encountered by the graph exploration. For each  $\lambda \in \Lambda$  the top- $n$  recommended accounts  $v$  along with their recommendation scores  $\sigma(\lambda, v, t)$  and their topological score  $topo_\beta(\lambda, v)$  are already computed in the preprocessing step. The approximate recommendation of a node  $v$  for a user  $u$  is an aggregation of the scores of  $v$  computed by all the landmarks  $\lambda \in \Lambda$  :

**DEFINITION 2** (APPROXIMATE RECOMMENDATION SCORE).  
*The approximate recommendation score of a node  $v$  for a node  $u$  on the topic  $t$  with respect to the set of landmarks  $\Lambda$  is defined as :*

$$\tilde{\sigma}_\Lambda(u, v, t) = \sum_{\lambda \in \Lambda} \tilde{\sigma}_\lambda(u, v, t)$$

where the score  $\tilde{\sigma}_\lambda(u, v, t)$  denotes the score of  $v$  that takes into consideration the set of paths  $P_{u, \lambda, v}$  from  $u$  to  $v$  that pass through the landmark  $\lambda$ .

The score  $\tilde{\sigma}_\lambda(u, v, t)$  is computed by the composition of the scores  $\sigma(u, \lambda, t)$  and  $topo_{\beta\alpha}(u, \lambda)$  obtained during the exploration phase with the scores  $\sigma(\lambda, v, t)$  and  $topo_\beta(\lambda, v)$  that are stored in the sorted lists of  $\lambda$ .

**PROPOSITION 4** (APPROXIMATE SCORE COMPUTATION).  
*The recommendation score of  $v$  for  $u$  with respect to the landmark  $\lambda$  can be computed as follows :*

$$\tilde{\sigma}_\lambda(u, v, t) = \sigma(u, \lambda, t) \times topo_\beta(\lambda, v) + topo_{\beta\alpha}(u, \lambda) \times \sigma(\lambda, v, t)$$

**PROOF.** Any path  $p$  from  $P_{u, \lambda, v}$  could be decomposed into  $p_1$  and  $p_2$ , with  $p_1 \in P_{u, \lambda}$  and  $p_2 \in P_{\lambda, v}$ . Obviously any path  $p = p_1.p_2$  with  $p_1 \in P_{u, \lambda}$  and  $p_2 \in P_{\lambda, v}$  is a path from  $P_{u, \lambda, v}$ .

Consequently, based on Proposition 2) we have:

$$\begin{aligned}
\tilde{\sigma}_\lambda(u, v, t) &= \sum_{p \in P_{u, \lambda, v}} \bar{\omega}_p(t) \\
&= \sum_{p_1 \in P_{u, \lambda}} \sum_{p_2 \in P_{\lambda, v}} \beta^{|p_2|} \bar{\omega}_{p_1}(t) + \beta^{|p_1|} \alpha^{|p_1|} \bar{\omega}_{p_2}(t) \\
&= \sum_{p_1 \in P_{u, \lambda}} \bar{\omega}_{p_1}(t) \cdot \sum_{p_2 \in P_{\lambda, v}} \beta^{|p_2|} + \sum_{p_1 \in P_{u, \lambda}} (\beta \cdot \alpha)^{|p_1|} \cdot \sum_{p_2 \in P_{\lambda, v}} \bar{\omega}_{p_2}(t) \\
&= \sigma(u, \lambda, t) \cdot \text{topo}_\beta(\lambda, v) + \text{topo}_{\beta \cdot \alpha}(u, \lambda) \cdot \sigma(\lambda, v, t)
\end{aligned}$$

□

Note that our approach estimates a lower-bound of the recommendation scores while landmark-based approaches traditionally proposed for shortest paths computation provide score upper-bounds, based on the triangular inequality. Indeed in our setting the approximate scores do not consider all the paths from  $u$  to  $v$ , but only the subset  $P_{u, \lambda, v}$  that pass through  $\lambda$ . However experiments show this approximation allows to retrieve a set of recommendations close to the one retrieved by an exact computation.

---

**Algorithm 2:** APPROX\_RECMM( $u, k, t, \beta, \alpha$ )

---

**Require:** a node  $u$ , a max. depth  $k$ , a topic  $t$ , the decay factor for path  $\beta$  and for edge  $\alpha$ .

**Ensure:** an ordered list of recommendations  $\tilde{R}_t$  for  $u$

```

1: ( $R_t, \text{topo}_{\beta \cdot \alpha}(u)$ ) ← LANDMARK_RECMM( $u, k, t, \beta, \alpha$ )
2: for all  $v \in R_t$  do
3:   if  $v \in \mathcal{L}$  then
4:     for all  $w$  recommended by  $v$  do
5:        $\tilde{R}_t[w] +=$ 
          $\sigma(u, v, t) \cdot \text{topo}_\beta(v, w) + \text{topo}_{\beta \cdot \alpha}(u, v) \cdot \sigma(v, w, t)$ 
6:     end for
7:   end if
8: end for
9: return  $\tilde{R}_t$ 

```

---

We perform our approximate recommendation for a node  $u$  and a topic  $t$  by using Algorithm 2. It first calls the LANDMARK\_RECMM algorithm to compute recommendation scores from  $u$  to all nodes within a distance  $k$  along with their topological score (l. 1). Observe that unlike the preprocessing step, the exploration depth has a small value  $k$  (2 in our experiments) so that the algorithm will not be run until the convergence. The recommendations are computed only for a single topic  $t$ . Note also that the decay factor is here  $\beta \cdot \alpha$ . For each encountered landmark (l. 2-3) we combine its recommendation for the topic  $t$  with the recommendation score computed from  $u$  to the landmark according to Proposition 4 (l. 5).

## 5. EXPERIMENTS

In this section we present the experiments that we have conducted on a real *Twitter* and *DBLP* datasets to validate our structures and algorithms.

### 5.1 The datasets

The *Twitter* dataset we use in our experiments contains approximately 2.2 million users (with their 2.3 billion associated tweets acquired in 2015 from February to April) linked by more than 125 million edges (*i.e* following relationships).

Table 2 describes the main topological properties on the generated dataset. For our *Twitter* dataset these properties are

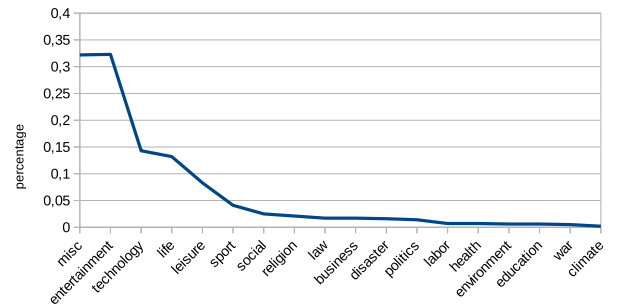
Property	Twitter	DBLP
Total number of nodes	2,182,867	525,567
Total number of edges	125,451,980	20,526,843
Avg. out-degree	57.8	47.3
Avg. in-degree	69.4	53.6
max in-degree	348,595	9,897
max out-degree	185,401	5,052

**Table 2: Datasets topological properties**

very close to the ones of the real *Twitter* network observed in [18].

### Topic extraction.

As already mentioned, in order to generate the topics of the edges we first used the OpenCalais document categorization to tag a subset of the users (nodes) in our graph with topics extracted from their published tweets. This strategy allowed to tag 10 percents of our nodes using a list of 18 standard topics for Web sites/documents proposed by OpenCalais. The user categorization was completed by using a trained Support Vector Multi-Label Model using Mulan, with a precision of 0.90, that associated to each user in the graph his publisher profile (topics on which he publishes). Each follower is characterized by a follower profile containing topics with high frequency among the topics of their followed publishers. Finally the labels of each edge are the topics in the intersection between the corresponding follower and publisher profiles. The resulting graph is a fully labeled social graph with 2.2M nodes and 125M edges. We refer to this dataset as *Twitter*. The edge labels obtained with our generation method show a biased distribution similar to the one observed for Web sites in Yahoo! Directory [17] (see Figure 3).



**Figure 3: Distribution of edges per topic**

For the *DBLP* dataset, we merged different versions of the ArnetMiner *DBLP* datasets<sup>5</sup>. The resulting dataset contains 2,291,100 papers, 1,274,860 authors and 4,191,643 citations. From this dataset we build a graph of author citations by creating a directed edge between author  $u$  and author  $v$  if some paper of  $u$  cites a paper of  $v$ . This results in a final dataset with 525,567 authors and 20,526,843 citations

<sup>5</sup><http://aminer.org/billboard/citation>

between them. Observe that we only kept cited authors. Then we used the Singapore Classification<sup>6</sup> to manually label some of the major conferences. The other conferences are labeled based on the number of authors they have in common with already labeled conferences (topics of two conferences are close if there are many authors that publish in both of them). Paper topics are deduced from the conference topics by assuming that a paper published in a conference is about the main topic of this conference. Author profiles are built from the topics of their published papers. The resulting dataset is summarized in Table 2.

## 5.2 Implementation

We implemented our solution in Java (JVM version 1.7). The experiments were run on a 64-bit server with a 10 cores Intel Xeon processor at 2.20GHz and 128GB of memory. The server is running Linux with a 3.11.10 kernel.

The topic similarities given by the Wu and Palmer similarity scores are pre-computed and stored in memory as a triangular similarity matrix. We considered here only the 18 common topics for Web documents, which results in a 2.5 KB file, but observe that for 10,000 topics the similarity matrix will require around 750MB so can still easily fit in memory. A similar approach was chosen for the similarity matrix of the DBLP dataset. We stored the landmark recommendations as inverted lists: for each landmark, we have a set of accounts recommended along with their recommendation score for each topic from  $\mathcal{T}$ . Landmarks were chosen according to one of the selection strategies presented in Table 4. We compare the quality of our recommendations with two related algorithms chosen as baseline: the standard Katz score [16], which considers only the topology (all paths between two accounts along with their length, given by the topological score in Equation 2), and TwitterRank [26] which captures both the link structure and the topical similarity between users. In the following we denote our score as TR.

The values of parameter  $\beta$  and  $\alpha$  are set to 0.0005 and respectively to 0.85, similarly to the values used for the Katz and the TwitterRank algorithms in [16] and [26].

## 5.3 Quality of the recommendation

We consider a test set of  $T$  edges of the graph together with their corresponding topics representing the ground truth. As observed in [16], to maintain the topological properties of the graph during the evaluation process, the target node of an edge of the test set must have at least  $k_{in}$  in-degree and the source node at least  $k_{out}$  out-degree ( $k_{in} = 3$  and  $k_{out} = 3$  in our experiments). All edges from  $T$  are then removed from the graph. For each edge  $e = u \rightarrow v$  in  $T$  we randomly select 1000 accounts in the graph. We compute recommendation scores for the 1001 accounts (the 1000 accounts and  $v$ ) with respect to  $u$  on the topics of  $e$  and we form a ranked list (similar to [6]) for each topic. For each list, if  $v$  belongs to the  $top-n$  accounts of the ranked list we have a hit, otherwise a miss. The overall recall and precision are defined similarly to [6] with  $\#hits/T$  and  $\#hits/N.T$  respectively. For our experiments we set the test size  $T = 100$  and we average values over 100 trials.

Figure 4 illustrates the accuracy of the different recommendation strategies for the Twitter dataset. We see that

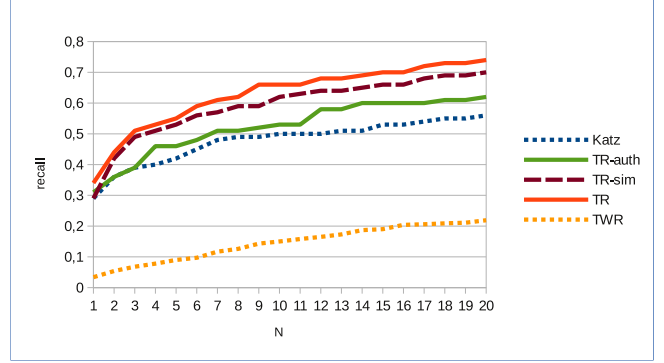


Figure 4: Recall at  $N$  (Twitter)

TwitterRank is outperformed by other algorithms. Indeed only for 4% of the recommendations the account corresponding to the removed edge is found in the top-1 for TwitterRank, while Katz provides as first recommendation the correct account in 29% of the tests and TR in 34%. So TR provides a 8.5 and 1.2 gain with TwitterRank and Katz respectively for the top-1. For the top-10 the improvement remains significant: 3.8 and 1.3 with TwitterRank and Katz respectively.

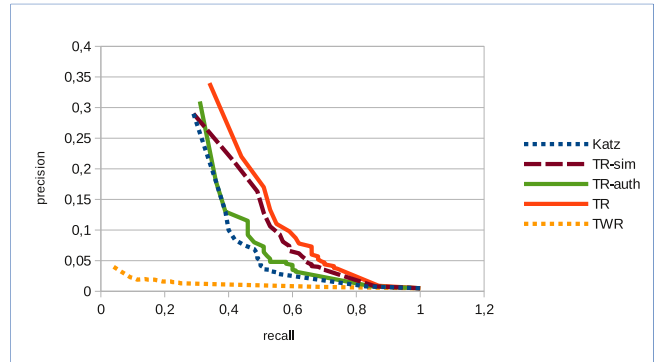


Figure 5: Precision vs recall (Twitter)

Figure 5 confirms that TR outperforms other approaches: for a similar recall value greater than 0.4, the precision of TR is at least twice the one of Katz and one order of magnitude higher than the one of TwitterRank.

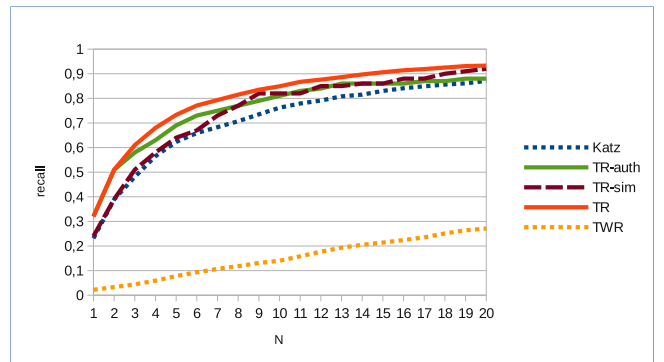


Figure 6: Recall at  $N$  (DBLP)

We observe in Figures 6 and 7 that the DBLP dataset exhibits similar results. The recall however exhibits a faster

<sup>6</sup><http://www.ntu.edu.sg/home/assourav/crank.htm>



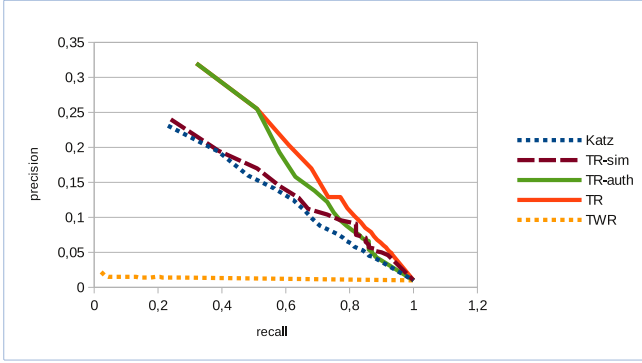


Figure 7: Precision vs recall (DBLP)

increase for TR due to the self-citations phenomenon: authors from a given paper often cite one or several of their previous papers on the topic. These papers may share some citations with the paper corresponding to the edge removed for the selected author. This also explains the faster recall increase for Katz strategy. TwitterRank whose recommendations are essentially based on the popularity (in-degree) of an account reached in the graph does not capture this phenomenon and provides slightly worse results than with the Twitter dataset.

Figure 4 also illustrates the benefit when taking into account both the edge similarity and the authority. Adding the edge similarity to Katz, which takes into account only the topology (number and length of the paths between two nodes), provides a better precision and recall (+11% for the precision and  $N = 20$ , see TR-auth). When we consider our approach without edge-similarity scores, but with topic authorities of the nodes, we improve both recall and precision (+25% compare to Katz precision, see TR-sim). Finally our approach which integrates the topology, the edge similarity and the topic authorities, outperforms these approaches (+32%, +19% and +6% with resp. Katz, TR-auth and TR-sim).

However there exists a large discrepancy for accuracy when considering two dimensions of analysis: the edge removal strategy and the popularity of the topic used for the recommendation. For the top-10, Figure 8 shows that for Twitter we have a very low accuracy, *i.e.* a recall of 0.15, 0.03 and 0.18 for respectively Katz, TwitterRank and TR, when trying to retrieve an account which belongs to the top-10% less followed accounts (TW min). Conversely very popular accounts (top-10% most followed accounts) are most of time retrieved in the top-10 recommendations with a recall between 0.9 and 0.95 for all strategies. This can be explained by their path-based approach which aggregates scores on incoming paths, so accounts with numerous incoming paths got a high score. Observe that for popular accounts, TwitterRank provides the best results. Indeed most of large accounts are labeled with several topics. While TwitterRank score relies on the account popularity and on the presence or not of a label for an account (whatever the number of labels it has), TR score considers for its authority score the number of incoming edges labeled with a given topic. But the more labels an account has, the lower authority score for a given topic it may have. Oppositely an account with a low in-degree rarely has several labels. Our approach that also considers semantic similarity between topics on edges is

then particularly efficient. With the DBLP dataset, authors who belong to the 10% less cited are more likely retrieved than with the Twitter dataset for Katz and TR due to the higher density of the graph. However TwitterRank based on the popularity fails to retrieve these authors. Even for the 10% most popular authors, TwitterRank does not achieve the good results obtained for the Twitter datasets, due to a different distribution of the in-degree. While the 10% most followed accounts in Twitter include few extremely popular accounts and some moderately popular, the 10% most followed authors in DBLP consist in a more uniform dataset regarding the in-degree.

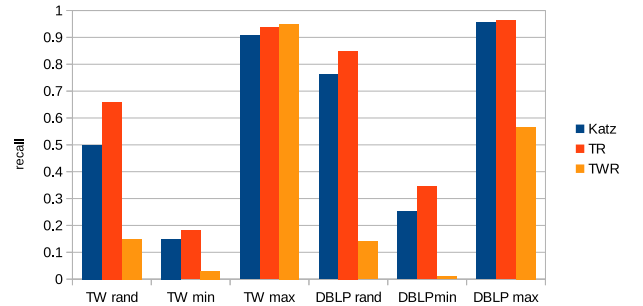


Figure 8: Recall w.r.t. popularity

Since the distribution of edge topics is very biased we also study the impact of the popularity of the topic on the recommendations. Results are depicted in Figure 9 for topics **social**, **leisure** and **technology**. Two main conclusions are underlined with this experiment. First, the less popular an account is, the better accuracy for our recommendations we get. So for an infrequent topic like **social** we get a recall-at-10 for TR, Katz and TwitterRank of respectively 0.959, 0.751 and 0.253. Oppositely for the popular topic **technology** we get respectively 0.462, 0.424 and 0.09. Indeed for a popular topic many accounts may be found in a close and connected neighborhood of the account we want to recommend, possibly with a higher score than for the account formerly linked by the removed edge. Second we observe that TR which considers the semantic similarity between topics always outperforms other strategies.

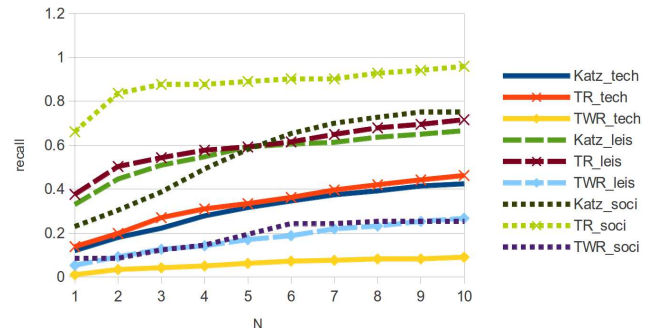


Figure 9: Recall w.r.t. topic popularity

However this experiment does not highlight the quality of the recommendations performed by each algorithm but only the ability to retrieve a removed link between two followers

(link prediction). To estimate the quality of the recommendations we rely on a user validation.

### User Validation Task.

In order to evaluate the relevance of the generated recommendations, we conducted a user validation task on 54 IT users (undergraduate, postgraduate and PhD students, and academics) from which 46% are regular *Twitter* users. We set up an on-line blind test where we ask users to rate the relevance of a set of recommendations for a given topic on a scale from 1 (low relevance) to 5 (high relevance). A recommendation set consists in the top-3 recommendations given by Katz, TR and TwitterRank, so 9 recommended accounts for topics **Technology**, **Social** and **Leisure**. On the interface the recommendation list is shuffled, and for each recommendation we display a sample of 5 randomly chosen tweets from the corresponding account.

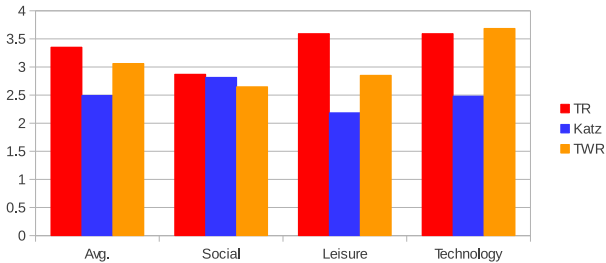


Figure 10: Relevance scores (user validation Twitter)

Figure 10 presents the results of our user validation. We observed that the user during the validation usually mark with the average 2 or 3 value all accounts when he was doubtful about the relevance or not of an account what happened usually when tweets were neutral, unclear, or when they required some knowledges about a given topic (*e.g.* few European people know who is Tom Brady so can not assert this tweet is about the **leisure** topic). So scores greater than value 3 are significant since they means users really observe the relevance of an account.

From this experiment we conclude that on average TR and TwitterRank provide more relevant recommendations according to the topic searched. However according to the popularity of the topics (see Figure 3) we have very different results. The **social** topic gave more homogeneous results with a score between 2.7 for TwitterRank, 2.8 for Katz and 2.9 for TR. The reason for this result is that posts published by these accounts are generally difficult to classify since they mix social and health, or social and politics for instance. Oppositely topics like **leisure** or **technology** are less ambiguous. For these topics, we see that TR and TwitterRank outperform Katz, which was expected since these two approaches consider the content published for their recommendation scores, unlike Katz. While TwitterRank generally recommends accounts with a large number of followers, TR can also recommend smaller account but more-specialized, which results in a better relevance score for topic with a medium popularity like **leisure**, when TwitterRank is slightly better for the most popular topic **technology**.

We also conduct a user validation for the DBLP dataset. We build a list with the top-3 recommendations returned

by each method for researchers from our lab. Observe they belong to different areas (IR, DB, OR, network, software engineering, etc). To illustrate how the different methods may help to discover relevant authors we limit to 100 the number of citations of the authors returned by each algorithm (so we avoid to propose very popular and obvious authors). We propose to each researcher the randomized list with the 9 authors retrieved based on his DBLP entry. He marks each proposal between 1 and 5 according to the relevance of the proposal (*i.e.* the proposed author could have been cited regarding the past publications done by the researcher). We collect 47 answers and results are presented in Table 3.

	Katz	TR	TWR
average mark	2.38	2.47	1.51
# 4 and 5-mark	46	47	11
best answer (%)	0.38	0.50	0.12

Table 3: User validation (DBLP)

The first row shows that both Katz and TR outperform TwitterRank on this dataset. The second row shows that around a third of the recommendations proposed by Katz and TR are considered particularly relevant by our panel (4 or 5-mark) while only 8% are so-considered with TwitterRank. A first rationale is that papers we cite, including papers from co-authors, often cite the same relevant articles within a topically-closed research community. The importance of the semantics on edge is less than with Twitter since researchers, whatever the number of articles they published, cite/are cited by mainly researchers from their community. This explains the close score for Katz and TR. The important role of the popularity in TwitterRank explains its poor results in this context since it proposed popular authors even when there exists a small number of paths between the query author and them. Last row confirms the quality of our recommendations since TR presents for 50% of the tests the best top-3 recommendations, when Katz and TwitterRank achieve respectively 38% and 12%.

## 5.4 Approximate computations

We perform a set of experiments to illustrate the benefits of the landmark-based approximate computations. Since results may be highly related to the choice for the landmarks selection, as underlined in [22], we decided to implement and compare recommendations based on 11 different landmark selection strategies presented in Table 4.

*Size and building time of the landmark index.* A first experiment highlights the important time discrepancy for the landmark selection algorithms (see Table 5). Obviously random selections of the landmarks like RANDOM, BTW-FOL and BTW-PUB are the fastest strategies (around 2ms per landmark), while strategies based on the centrality property are 5 orders of magnitude slower (around 17h) due to  $O(N^2 \cdot \log N + NE)$  centrality complexity (with Johnson’s algorithm). Table 5 also illustrates that the recommendation computation for a given landmark is almost independent of the landmarks selection strategy (between 12 and 15 mns), which means that convergence is achieved in a similar number of steps after exploring a similar number of paths.

*Comparison of the landmark selection strategies for recommendations.* We evaluate our approximate approach

Algorithm	Description
RANDOM FOLLOW	Draw landmarks with a uniform distribution Draw landmarks with a probability depending on their # of followers
PUBLISH	Draw landmarks with a probability depending on their # of publishers
IN-DEG	Landmarks are nodes with highest in-degree
BTW-FOL	Draw landmarks among nodes with # of followers in [min_follow,max_follow]
OUT-DEG	Landmarks are nodes with highest out-degree
BTW-PUB	Draw landmarks among nodes with # of publishers in [min_publis,max_publish]
CENTRAL	Select landmarks that are reachable at a given distance from most of chosen seed nodes
OUT-CEN	Select the landmarks based on the number of different output seeds that they cover
COMBINE	Weighted combination between the CENTRAL and OUT-CEN
COMBINE2	Weighted combination between the BTW-FOL and BTW-PUB

Table 4: Landmarks selection algorithms proposed

Strategy	landmarks	
	select. (ms)	comput. (s)
RANDOM	2.4	756.7
FOLLOW	3,712.8	877.3
PUBLISH	3,614.7	868.6
IN-DEG	459.6	854.3
BTW-FOL	2.4	735.1
OUT-DEG	1,815.7	918.6
BTW-PUB	1.7	822.7
CENTRAL	61,060.2	807.8
OUT-CEN	66,862.3	816.5
COMBINE	130,461.8	818.2
COMBINE2	2.45	805.6

Table 5: Determining landmark w.r.t. strategies

presented in Section 4. We perform a BFS at depth 2 from a query node and combine scores with the ones of the landmarks encountered. (see Algorithm 2). Then we compare the recommendations retrieved with the ones provided by the exact computation with convergence. Average results for 100 landmarks are reported in Table 6.

Strategy	#lnd	time in s (gain)	L10	L100	L1000
RANDOM	2.9	0.93 (338)	0.130	0.124	0.125
FOLLOW	17.5	0.83 (379)	0.377	0.140	0.096
PUBLISH	11.7	0.58 (539)	0.349	0.136	0.100
IN-DEG	58.9	0.84 (373)	0.523	0.149	0.066
BTW-FOL	3.5	0.55 (577)	0.061	0.059	0.058
OUT-DEG	6.2	0.81 (388)	0.518	0.147	0.064
BTW-PUB	2.9	0.54 (585)	0.129	0.127	0.123
CENTRAL	5.3	0.76 (414)	0.134	0.123	0.125
OUT-CEN	4.4	0.74 (425)	0.172	0.131	0.121
COMBINE	4.2	0.71 (443)	0.180	0.125	0.118
COMBINE2	3.7	0.54 (584)	0.129	0.126	0.124

Table 6: Comparison of the landmark selection strategies

First we observe that the number of landmarks encountered during the BFS at distance 2 differs from one strategy to another and ranges from 2.9 on average for the RANDOM strategy to 58.9 for IN-DEG. Centrality approaches lead to less landmarks encountered since they select landmarks among nodes which connect connected subgraphs and a two-hop BFS is more unlikely to visit several connected subgraphs. We notice that the processing time does not depend

on the number of landmarks found, what seems counterintuitive since more landmarks means more computations (score combinations) to perform. The rationale is that we perform pruning when we encounter a landmark during the BFS, to avoid considering twice paths from the BFS which pass through a landmark. Since the recommendation computation is dominated by the BFS exploration and computation, this pruning largely reduces the whole processing time. A second important result is that our approximate computation allows to get a 2-3 order of magnitude gain compared to the exact computation. Finally observe that a strategy which allows to find more landmarks is more tolerant to a landmark departure or to a landmark with outdated recommendation values.

Finally to validate the quality of the approximate computation we report the average Kendall Tau distance between the approximate computation and the exact computation obtained at convergence when a landmark stores respectively the top-10, top-100 or top-1000 recommendations for all topics (see last 3 columns of Table 6). Keeping 1000 recommendations for the landmarks at the pre-processing allows to reach a Kendall Tau distance between 0.06 and 0.13 for the top-100 recommended accounts for a node at query time. Keeping a top-10 at landmarks leads to a higher Kendall Tau distance since a landmark may update at most 10 scores from the top-10 built at distance 2 from the BFS. Consequently an account which is ranked at the 11th place for two landmarks is not kept as a recommendation whereas its aggregate score may be higher than accounts kept as recommendations. Remark that even when storing the top-1000 for each topic, the landmarks recommendations can easily fit in memory since they require 1.4MB storage each.

## 6. CONCLUSION

We present the TR recommendation score which combines topology and semantic information regarding the user interest. To face prohibitive computations with very large graphs, we propose a landmark-based approach which requires a pre-computation step for a small set of identified nodes and achieves a 2-3 order of magnitude gain compare to the exact computation. The experiments and user validation show that TR outperforms other algorithms. As future work we intend to study updating strategies since many following links have a short lifespan. This graph dynamicity may impact the scores stored by the landmarks. Moreover we made the choice to handle the recommendation task in a centralized manner motivated by the current social media architectures like *Twitter Who-to-Follow* service hosted on a single server. However, with the continuous increase of the social graph sizes, distribution strategies must be considered in the future. Regarding our approach, distribution implies to split the graph by taking into account connectivity, but also to perform landmark selections and distributions that allow a node to evaluate the recommendation scores “locally” minimizing network transfer costs.

## 7. REFERENCES

- [1] J. A. Aslam and M. Montague. Models for metasearch. In *SIGIR*, pages 276–284, 2001.
- [2] S. Budalakoti and R. Bekkerman. Bimodal Invitation-Navigation Fair Bets Model for Authority Identification in a Social Network. In *WWW*, pages 709–718, 2012.
- [3] V. Chaoji, S. Ranu, R. Rastogi, and R. Bhatt. Recommendations to Boost Content Spread in Social Networks. In *WWW*, pages 529–538, 2012.

- [4] K. Chen, T. Chen, G. Zheng, O. Jin, E. Yao, and Y. Yu. Collaborative personalized tweet recommendation. In *SIGIR*, pages 661–670, 2012.
- [5] A. Chin, B. Xu, and H. Wang. Who Should I Add as a 'Friend'? : a Study of Friend Recommendations Using Proximity and Homophily. In *MSM*, page 7, 2013.
- [6] P. Cremonesi, Y. Koren, and R. Turrin. Performance of Recommender Algorithms on Top-n Recommendation Tasks. In *RECSYS*, pages 39–46, 2010.
- [7] E. Diaz-Aviles, L. Drumond, Z. Gantner, L. Schmidt-Thieme, and W. Nejdl. What is happening right now ... that interests me?: online topic discovery and recommendation in twitter. In *CIKM*, pages 1592–1596, 2012.
- [8] S. G. Esparza, M. P. O'Mahony, and B. Smyth. Towards the Profiling of Twitter Users for Topic-Based Filtering. In *SGAI*, pages 273–286, 2012.
- [9] A. Gubichev, S. J. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. In *CIKM*, pages 499–508, 2010.
- [10] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. WTF: the Who to Follow Service at Twitter. In *WWW*, pages 505–514, 2013.
- [11] J. Hannon, M. Bennett, and B. Smyth. Recommending twitter users to follow using content and collaborative filtering approaches. In *RECSYS*, pages 199–206, 2010.
- [12] G. Jeh and J. Widom. Scaling Personalized Web Search. In *WWW*, pages 271–279, 2003.
- [13] P. Kapanipathi, F. Orlandi, A. P. Sheth, and A. Passant. Personalized Filtering of the Twitter Stream. In *SPIM*, pages 6–13, 2011.
- [14] K. Koroleva and A. B. Röhler. Reducing Information Overload: Design and Evaluation of Filtering & Ranking Algorithms for Social Networking Sites. In *ECIS*, page 12, 2012.
- [15] R. Lempel and S. Moran. Salsa: The stochastic approach for link-structure analysis. *ACM Transactions on Information Systems*, 19(2):131–160, 2001.
- [16] D. Liben-Nowell and J. Kleinberg. The Link Prediction Problem for Social Networks. In *CIKM*, pages 556–559, 2003.
- [17] T.-Y. Liu, Y. Yang, H. Wan, H.-J. Zeng, Z. Chen, and W.-Y. Ma. Support Vector Machines Classification with a Very Large-Scale Taxonomy. *SIGKDD Explorations*, 7(1):36–43, 2005.
- [18] S. A. Myers, A. Sharma, P. Gupta, and J. Lin. Information network or social network?: The structure of the twitter follow graph. In *WWW Companion Volume*, pages 493–498, 2014.
- [19] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, November 1999.
- [20] A. Pal and S. Counts. Identifying Topical Authorities in Microblogs. In *WSDM*, pages 45–54, 2011.
- [21] M. Pennacchiotti, F. Silvestri, H. Vahabi, and R. Venturini. Making your interests follow you on twitter. In *CIKM*, pages 165–174, 2012.
- [22] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, pages 867–876, 2009.
- [23] A. D. Sarma, S. Gollapudi, M. Najork, and R. Panigrahy. A sketch-based distance oracle for web-scale graphs. In *WSDM*, pages 401–410, 2010.
- [24] P. Symeonidis and E. Tiakas. Transitive Node Similarity: Predicting and Recommending Links in Signed Social Networks. *WWWJ*, 17(4):743–776, 2014.
- [25] K. Tretyakov, A. Armas-Cervantes, L. García-Bañuelos, J. Vilo, and M. Dumas. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *CIKM*, pages 1785–1794, 2011.
- [26] J. Weng, E.-P. Lim, J. Jiang, and Q. He. TwitterRank: Finding Topic-sensitive Influential Twitterers. In *WSDM*, pages 261–270, 2010.
- [27] Z. Wu and M. Palmer. Verbs Semantics and Lexical Selection. In *ACL*, pages 133–138, 1994.
- [28] X. Yang, H. Steck, Y. Guo, and Y. Liu. On top-k Recommendation Using Social Networks. In *RECSYS*, pages 67–74, 2012.

# Slowing the Firehose: Multi-Dimensional Diversity on Social Post Streams

Shiwen Cheng, Marek Chrobak, Vagelis Hristidis  
Department of Computer Science & Engineering  
University of California, Riverside, California, USA  
{schen064, marek, vagelis}@cs.ucr.edu

## ABSTRACT

Web 2.0 users conveniently consume content through subscribing to content generators such as Twitter users or news agencies. However, given the number of subscriptions and the rate of the subscription streams, users suffer from the information overload problem. To address this issue, we propose a novel and flexible diversification paradigm to prune redundant posts from a collection of streams. A key novelty of our diversification model is that it holistically incorporates three important dimensions of social posts, namely content, time and author. We show how different applications, such as microblogging, news or bibliographic services, require different settings for these three dimensions. Further, each dimension poses unique performance challenges towards scaling the diversification model for many users and many high-throughput streams. We show that hash-based content distance measures and graph-based author distance measures are both effective and efficient for social posts. We propose scalable real-time stream processing algorithms leveraging efficient indexes that input a social post stream and output a diversified version of the stream, diversified across all three dimensions. Next, we show how these techniques can be extended to serve multiple users by appropriately reusing indexing and computation where possible. Through extensive experiments on real Twitter data, we show that our diversification model is effective and our solutions are scalable. We show that different algorithms perform best for different application settings.

## 1. INTRODUCTION

Tremendous amounts of online social data are generated every day. For instance, Twitter has reported over 280 million monthly active users in its microblogging service and 500 million Tweets posted per day<sup>1</sup>. One common way to consume social data is through implicit or explicit subscription. For example, almost all news agencies offer RSS feeds for people to subscribe. Google Scholar continuously recommends new publications to its users based on a user's profile and publication history. In a microblogging system like Twitter, one can subscribe to other users' posts by following them.

<sup>1</sup><https://about.twitter.com/company>

All posts matching a user's subscriptions are typically displayed in a convenient central place, such as the user's timeline in Twitter or Facebook. These timelines are updated in real time. A key challenge is that a user could be easily overwhelmed by the number of posts in the timeline, especially if the user is subscribed to many post producers. Further, a user's timeline often contains lots of posts that carry no new information with respect to other similar posts. This data overload issue also happens in other applications with smaller data throughput such as news and research papers. For instance, it has been shown that a primary care physician should read hundreds of medical publications per day to keep up with the medical literature [2].

To alleviate the data overload problem, in this paper we propose a novel way to efficiently and effectively diversify social post streams by pruning redundant posts. By social post streams we mean a broad class of content generated by services where each post, in addition to its textual content, has a unique author and a unique timestamp, and where authors are associated through various social relationships. For instance, in Google Scholar authors are connected by relations such as co-authorship or overlapping research interests. In microblogging sites users are connected by follower/followee relations.

Given a stream consisting of all the posts from a user's subscriptions, our goal is to output in real-time a subset of the stream in which (i) all posts are dissimilar to each other and (ii) any post in the whole stream will be either included or *covered* by a post in the sub-stream. A post covers another post if the two posts are similar in all three similarity dimensions: (a) content, (b) time and (c) author.

Two posts have similar *content* if their text components are similar. Intuitively, all other dimensions being equal, users want to avoid seeing two posts with very similar content. Similarly, the *timestamp* distance of two posts is important in social post diversification. Two posts that have similar content but are far away in terms of post time, may both be of interest to the user. Note that time is widely used for diversifying search results in microblogging systems [10, 14, 4].

The *author* similarity is a more subtle dimension that to the best of our knowledge has not been used before for computing diversity in social media. For example, CNN and Fox News, which both have official Twitter accounts, are dissimilar to each other because they generally have different political views. We compute the distance between two authors through their social connections. In particular, we compare the sets of friends (or followers in the case of Twitter) of the two authors, which has been shown to be a good author similarity measure in social networks [21, 9].

**Challenges:** To summarize, in our model two posts are redundant with respect to each other if they are similar in all of the three

dimensions. It is challenging to apply the proposed diversification model in a large scale social service with high posts throughput. First, we must efficiently compare the content of a new post to the content of all previous posts (within a time window). For this, we apply Hash-based techniques to measure the content similarity between social posts. Hash-based techniques have been applied before to Web documents [11], but not to social posts, which are generally shorter and may heavily rely on abbreviations or URLs.

Second, handling the author dimension is challenging. A naive approach is to check if the author of each new post is similar to the author of each existing post (within a time window). However, we show that depending on the setting (similarity thresholds across the three dimensions), a different indexing data structure is more efficient to achieve real-time posts processing.

Third, the three diversity dimensions offer an opportunity to use the results of the one dimension to prune the work needed for the other dimension. For instance, if a reader knows that posts  $P_1$  and  $P_2$  have high content similarity, then she doesn't need to check if their authors or time are similar.

Fourth, if we move from one user to many users, where each user has a collection of subscriptions, the challenge is how to reuse the computation performed for diversifying one user's stream to diversify streams of other users. We show that we can reuse computation across users only if their shared subscriptions meet a strict condition.

**Previous work on diversity:** There has been much work on diversifying results for documents [15, 1, 3], social posts [10, 14, 4] and database records [5, 6]. However, none of these works can be applied to our setting where: (i) data is streaming and an instant decision must be made on whether a post should be pushed to the user, and (ii) a multi-dimensional diversity model is adopted. In contrast, most previous works focus on the search setting, where a user submits a query and the set of results must be diversified based on content, including work on social posts [10, 14].

The problem studied in this paper is also fundamentally different from previous work on stream summarization [20, 18, 16, 23], because: (i) we do not aim to generate an aggregation of documents, but instead select a subset of posts, and (ii) we define strict coverage constraints to guarantee that not even one uncovered posts is missed.

**Contributions:** In this paper, we make following contributions:

- We propose a new paradigm to define diversity on social posts, by incorporating three important dimensions – content, time and author – and we define corresponding optimization problems (Section 2).
- We study how content similarity can be efficiently applied to social posts, which are generally short and contain abbreviations (Section 3).
- We propose efficient data structures and algorithms to solve the social posts stream diversification problem (Section 4).
- We show how the single-user algorithm can be extended to handle many users, by reusing computation across users (Section 5).
- We perform a comprehensive experimental evaluation, where we focus on microblogging data, which poses the most serious scalability challenges. We show how different algorithms perform better for different diversity needs (Section 6).

Section 7 reviews related work. We conclude in Section 8.

## 2. FRAMEWORK AND PROBLEM DEFINITION

Let  $\mathbf{P}$  represent a stream (ordered set) of social posts. Each post  $P_i$  in  $\mathbf{P}$  has an author  $author(P_i)$ , textual content  $text(P_i)$  and a timestamp  $time(P_i)$  (also referred as  $t_i$ ). We define the distance measures across the three diversity dimensions as follows.

- **Content Distance.** We represent the content distance between two posts  $P_i$  and  $P_j$  as  $dist_c(P_i, P_j)$ . Cosine similarity is a possible way to define the distance, but for efficiency purposes we employ the hash-based simhash measure as explained in Section 3, where we show that simhash is effective for social posts.
- **Time Distance.** The time distance between two posts  $P_i$  and  $P_j$  is denoted as  $dist_t(P_i, P_j) = |t_i - t_j|$ .
- **Author Distance.** We denote the author distance between  $P_i$  and  $P_j$  as  $dist_a(P_i, P_j)$ . For social data, we define the similarity between two authors as the cosine similarity between their friends' vectors, which has been successfully used in previous work to measure the user similarity in Twitter [21, 9]. The author distance is  $(1 - similarity)$ . For other domains other distance measures may be more appropriate.

Next, we define the coverage semantics between posts.

**Definition 1.** (*Post Coverage*) Given a content diversity threshold  $\lambda_c$ , a time diversity threshold  $\lambda_t$  and an author diversity threshold  $\lambda_a$ , two social posts  $P_i$  and  $P_j$  cover each other if:

- $dist_c(P_i, P_j) \leq \lambda_c$  and
- $dist_t(P_i, P_j) \leq \lambda_t$  and
- $dist_a(P_i, P_j) \leq \lambda_a$ .

Note that the coverage semantics between two posts is symmetric. The three thresholds may vary according to the characteristics of a social system as we discuss below. The primary focus of this paper is to study the efficient processing of a posts stream and not to set these threshold values.

We next define the Social Post Stream Diversification (SPSD) problem.

**Problem 1 [Social Post Stream Diversification (SPSD)]** Given a social post stream  $\mathbf{P}$ , and diversity thresholds  $\lambda_c$ ,  $\lambda_t$  and  $\lambda_a$ , compute a sub-stream of posts  $Z \subseteq \mathbf{P}$  that covers  $\mathbf{P}$ , that is,  $\forall P_i \in \mathbf{P} \exists P_j \in Z$ , such that  $P_j$  covers  $P_i$ .

Note we have to compute  $Z$  in *real-time*, i.e., immediately decide whether a post  $P_i$  should be included in  $Z$  at its arrival. That is, we cannot first view the whole stream and then decide which posts should be included in the substream.

In SPSD, there is a single user who consumes the stream and many authors who generate the posts of the stream (a user may also be an author and vice versa). That is, a solution to SPSD should be deployed for each user, for example, as part of the Twitter app of a user. On the other hand, a social network service would rather have a central diversification engine that diversifies the posts for each of its users, so that no client side post processing is required. We refer to this version of SPSD as Multiple-Users SPSD (M-SPSD). Another difference between SPSD and M-SPSD is that in SPSD we can easily support user customized diversity thresholds. Figure 1 shows how SPSD and M-SPSD differ in terms of the setting and deployment.

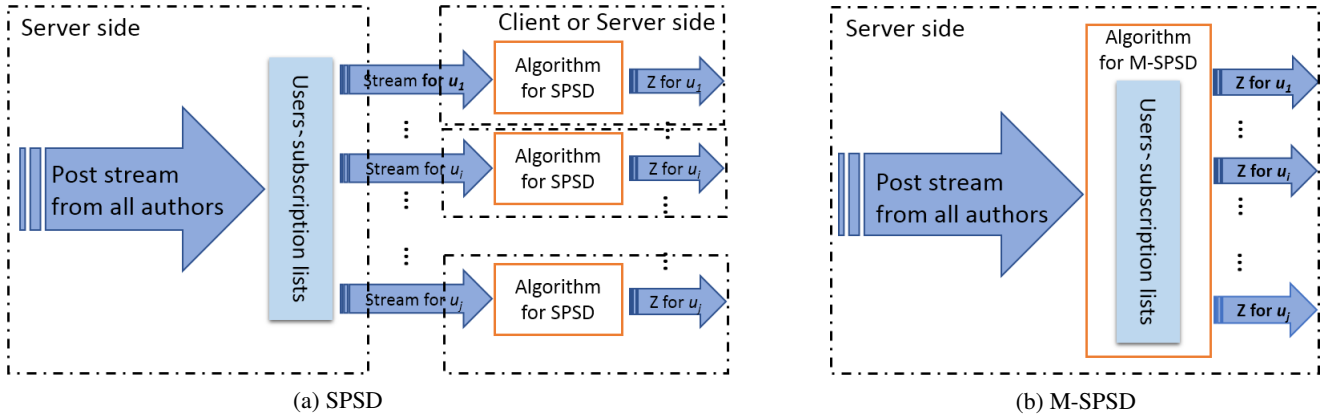


Figure 1: Settings of SPDP and M-SPDP.

**Problem 2 [Multiple-Users Social Post Stream Diversification (M-SPSD)]** Given a social post stream  $P$ , diversity thresholds  $\lambda_c$ ,  $\lambda_t$  and  $\lambda_a$ , and a set of users where each user is subscribed to a subset of the authors, compute a diversified sub-stream for each user.

### 3. CONTENT DISTANCE ESTIMATION FOR MICROBLOGGING POSTS

Among the three diversity dimensions, the content distance is the most expensive to compute, because it must be computed for each new post. This is especially true given our real-time decision semantics described above. In contrast, the author similarity between each pair of authors may be precomputed (e.g., once every week), as it changes slowly over time. For that reason, we cannot afford to use traditional content similarity measures such as cosine similarity. Instead, we turn to hash-based distance measures. In this section we present the details of the employed content distance technique along with an analysis of its effectiveness for microblogging data.

We define the content distance between two posts  $P_i$  and  $P_j$  as the Hamming distance of their SimHash [17] fingerprints. Previous work has applied SimHash on web documents [11] and showed that it is efficient and effective. We represent the SimHash of  $text(P_i)$  as  $S_i$ , which is a 64-bit fingerprint. The Hamming distance of two SimHash fingerprints is the number of different bits between them. According to the experimental analysis in [19], the cosine distance between two texts positively correlates to the Hamming distance of their corresponding SimHash fingerprints.

#### Distribution of SimHash distances in Twitter

First, we study the distribution of SimHash distances on Twitter data. We collected a dataset of 200 thousand tweets from the Twitter Streaming API, which returns a stream of randomly selected substream of Twitter ([12] showed that the stream is not exactly random but this is not too important for our problem). The distribution of the Hamming distances for these tweets is depicted in Figure 2, which shows a perfect normal distribution with mean value 32, as expected, and with most of the distances between 24 to 40.

#### User Study

To further evaluate the effectiveness of SimHash for social posts, we conducted a user study to learn the relationship between the SimHash distance between two posts and the perceived dissimilarity between the posts. A second goal of the study is to learn what is a good SimHash distance threshold (e.g., a threshold of 3 bits was

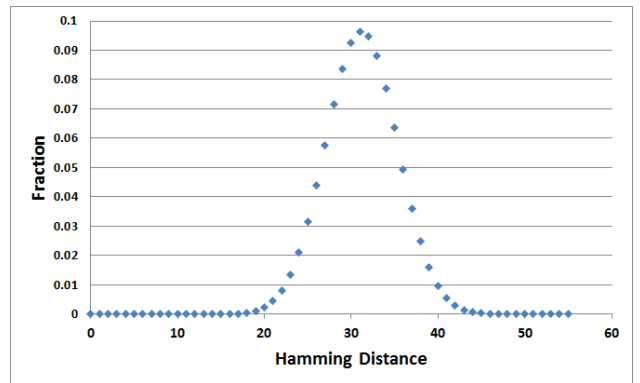


Figure 2: Hamming distance distribution

chosen to define redundant Web pages [11]) and if any preprocessing of the tweet text (e.g., expand shortened URLs) may improve the effectiveness of SimHash.

*Setup and Methods:* In particular, we collected a dataset of 2000 pairs of tweets randomly selected from the 200,000 tweets returned by the Twitter Streaming API, with SimHash distances between 3 and 22 – 100 tweets from each distance value. We chose 3 to 22 because this is the range where we expect to find posts that are very similar (redundant with respect to each other). This range choice is supported by our results below. We recruited 12 undergraduate and graduate students.

We evenly divided these 2000 pairs into 4 groups and distributed them to the 12 students for labeling. The author and timestamp of the posts are hidden. Some examples of these pairs are shown in Table 1. Each group of tweets is labeled by 3 students. The students were asked to mark whether the two tweets in a pair are redundant with respect to each other.

To help the users more accurately label the similarity between two posts, we showed the expanded URL (instead of the shortened one shown in Table 1). We used a majority vote, that is, if at least 2 out of the 3 students labelled a pair as redundant, we labelled the pair as near-duplicates.

*Results:* Out of the 2000 pairs, the users marked 949 pairs as redundant. Figure 3 shows the precision and recall achieved by various SimHash distance values. For each Hamming distance  $h$ , the precision is defined as the fraction of pairs with Hamming distance no more than  $h$  that are true near-duplicates. Recall is the

Table 1: Example tweet pairs and their Hamming distances

Tweet pair	Hamming distance
Over 300 people missing after South Korean ferry sinks. (Reuters) Story: <a href="http://t.co/9w2JrurhKm">http://t.co/9w2JrurhKm</a>	3
Over 300 people missing after South Korean ferry sinks. (Reuters) Story: <a href="http://t.co/E1vKp9JJfe">http://t.co/E1vKp9JJfe</a> "In order to succeed, your desire for success should be greater than your fear of failure" Bill Cosby	8
In order to succeed, your desire for success should be greater than your fear of failure. #quote #success - Bill Cosby	
Alibaba's growth accelerates, U.S. IPO filing expected next week <a href="http://t.co/mUcmLJ4cpc">http://t.co/mUcmLJ4cpc</a> #Technology #Reuters	13
Alibaba's growth accelerates, U.S. IPO filing expected next week: SAN FRANCISCO (Reuters) - Alibaba Group Hold... <a href="http://t.co/aLAV8w4gWF">http://t.co/aLAV8w4gWF</a>	

fraction of the total number of near-duplicate pairs that are detected with Hamming distance at most  $h$ . This graph shows that SimHash distance is an effective measure to identify similar posts.

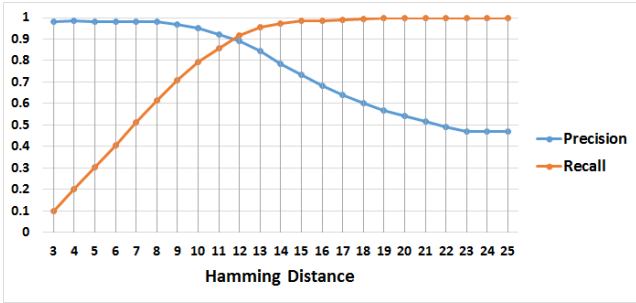


Figure 3: Precision and Recall for Hamming distance. SimHash fingerprints are generated from raw texts of tweets

Next, we study if various text preprocessing methods may improve the precision or recall of SimHash distance measure for microblogs. We first normalize the text by (a) changing all text to lowercase, (b) removing extra white spaces between words, and (c) removing non-alphanumeric characters (such as \*, -, +, /, etc.). Figure 4 plots the precision and recall after we apply the normalization. We see that this graph achieves higher precision and recall values than the original analysis in Figure 3. We also see that the two lines cross for  $distance = 18$ , which achieves precision = 0.96 and recall = 0.95. Hence, we use  $\lambda_c = 18$  as the default content distance threshold in the experiments in Section 6.

We also tried other methods of text preprocessing such as expanding shortened URLs (URLs in tweets are shortened by Twitter), varying the weights of user mentions and hashtags (by creating artificial copies), and expanding abbreviations. However, these methods had no significant impact to the precision and recall.

For completeness, we compared the effectiveness of SimHash to that of cosine similarity (which is much slower as discussed above) in terms of detecting posts with near-duplicate content (redundant). We tried different cosine threshold values and found that the precision and recall lines across at cosine similarity 0.7, where all posts with cosine similarity above 0.7 are marked as redundant. This achieves precision and recall of 0.96 and 0.95 respectively, which is the same as what we achieved using SimHash above. This means that, for detecting near-duplicate in our dataset, SimHash achieves effectiveness similar to cosine similarity. Hence, given the time performance advantage of SimHash, it is the best choice for our problem.

The high threshold value of  $\lambda_c = 18$  for SimHash precludes the

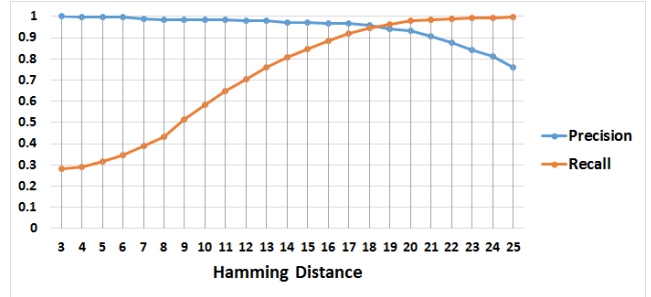


Figure 4: Precision and Recall for Hamming distance. SimHash fingerprints are generated from normalized texts of tweets

use of the efficient SimHash index proposed in [11] which relies on building several copies of the SimHash values table for several permutations of the bits, since the number of these copies is exponential in  $\lambda_c$  (which was only 3 in [11]). Hence, as we discuss in Section 4, other indexing and searching techniques are required.

#### 4. ALGORITHMS FOR SPSPD

In this section, we describe our algorithmic solutions for the SPSPD problem. As explained earlier in Section 3, due to the high Hamming distance threshold we are unable to use existing SimHash indexing techniques, and we must rely on comparing the SimHash value of each new post with those of all the previous ones, leading inevitably to linear time complexity per post in the worst case. We reduce the number of these comparisons by leveraging the other two dimensions, time and author. We first discuss how we handle time diversity, which is simpler, and then we present various approaches for handling author diversity.

**Handling Time Diversity.** According to the diversity model, at the arrival of a post  $P_i$  it can only be covered by the previous posts within a  $\lambda_t$  time distance. Thus, it is sufficient to store only the posts from previous  $\lambda_t$  time in memory for checking the coverage of a new post. One possible implementation is that we could store the posts in a circular array. We track two post indices for the oldest post within a  $\lambda_t$  distance to current time ( $a$ ) and the most recent post ( $b$ ). At the arrival of each post  $P_i$ , we compare it to the posts from most recent post to the oldest (i.e., from index  $b$  to  $a$ ). If we encounter a post  $P_j$  with  $t_i - t_j > \lambda_t$ , we update  $a$  to be index of the post right after  $P_j$ . And we insert a non-redundant post to the array with index  $(b + 1)$  and update  $b = b + 1$ .

Now that we have discussed how to handle time diversity, we focus on the author diversity among the posts in the last  $\lambda_t$  time units. The author similarity relations between all authors form an



author similarity graph  $G$ , which as we discussed above may be periodically precomputed. There is an edge between two authors in  $G$  if their distance is below the threshold  $\lambda_a$ . For each user  $u_i$  who subscribes to a set  $A$  of authors, we define  $G_i$  as the subgraph of  $G$  that contains all the  $A$  authors and the edges among them. In this section, we assume there is only one user (hence, one  $G_i$ ) and in Section 5 we assume multiple users (and  $G_i$ 's).

## 4.1 UniBin

Our first method to solve SPSPD, which we refer as *UniBin*, works as follows: At the arrival of each post  $P_i$  in  $\mathbf{P}$ , we sequentially (from the most recent post to the older ones) compare  $P_i$  to each post in the past  $\lambda_t$  time range in the diversified sub-stream  $Z$ . For each post  $P_j$ , we check whether  $P_j$  meets both: (1) Hamming distance between  $S_i$  and  $S_j$  (SimHash fingerprints of  $P_i$  and  $P_j$ , respectively)  $\leq \lambda_c$ , and (2)  $dist_a(P_i, P_j) \leq \lambda_a$ , which can be achieved by checking whether  $author(P_i)$  and  $author(P_j)$  are the same or neighbors in  $G$ . If no post from the past  $\lambda_t$  time range meets the above two conditions (i.e.,  $P_i$  is not covered by  $Z$ ), then we add  $P_i$  to  $Z$ . Otherwise we do not include  $P_i$  in  $Z$ .

We denote this method as **UniBin** indicating that the posts from all authors are stored in a single *post bin* (e.g., a circular array as described earlier). We illustrate UniBin with an example. In Figure 5a, each node represents an author. Two authors are connected by an edge if they are similar to each other (i.e., the author distance  $\leq \lambda_a$ ). Figure 5b shows the posts from these authors with post distance information in terms of all three diversity dimensions.

We show the update of a post bin for UniBin in Figure 6a. When  $P_1$  arrives, there is no posts in the bin yet. Thus  $P_1$  is not covered hence is added to the bin.  $P_2$  is also added as it is not covered by  $P_1$  (the Hamming distance between  $S_1$  and  $S_2$ ,  $dist_c(P_1, P_2)$ , is larger than the threshold  $\lambda_c$ ). For  $P_3$ , the algorithm first compares it to  $P_2$  which does not cover  $P_3$  (because  $dist_c(P_2, P_3) > \lambda_c$ ). However, it is covered by  $P_1$  because in all three diversity dimensions they are within the distance thresholds (or above similarity threshold). Thus,  $P_3$  is not added. So forth,  $P_4$  is not covered by either  $P_1$  and  $P_2$  and is included in the bin. However, we note that  $P_4$  and  $P_3$  cover each other. Finally,  $P_5$  is covered by  $P_4$ .

## 4.2 NeighborBin

UniBin has to compare a new post (both its author and content SimHash) to all posts in the last  $\lambda_t$  time units. This aggregated time may be considerable given the high frequency of posts, even if the author similarity graph  $G_i$  and the post bin are maintained in memory.

To improve this, we partition the posts by their authors such that for a new post  $P_i$  we only check its coverage by comparing with the posts from  $author(P_i)$  or from  $author(P_i)$ 's similar authors. Specifically, we create a post bin for each author and when a new post  $P_i$  comes, the algorithm sequentially checks posts in the bin identified by  $author(P_i)$  but not other posts. However, we must note that posts from the authors that are neighbors of  $author(P_i)$  in  $G_i$  can potentially cover  $P_i$ . Hence, the post bin of an author also includes the posts of similar authors (neighbors in  $G_i$ ). Thus, we add  $P_i$  to all bins of  $author(P_i)$ 's neighbors in addition to the bin of  $author(P_i)$ , if  $P_i$  is detected as a non-redundant post. We denote this method as **NeighborBin**.

Figure 6b depicts the execution of NeighborBin for the data shown in Figure 5.  $P_1$  is added not only to the bin of its author a1, but also to the bins of a2 and a3, because they are neighbors of a1, as shown in Figure 5a. To check the coverage of  $P_2$ , only the post bin of a2 is accessed where  $P_1$  does not cover  $P_2$ . After that,  $P_2$  is also added to the post bins of a1, a2 and a3. NeighborBin checks the

coverage of  $P_3$  by iterating posts in the bin of a3 where  $P_1$  covers  $P_3$ . When  $P_4$  comes, a4's post bin is blank and thus  $P_4$  is added to the post bins of a3 and a4 without incurring any post comparisons. Finally,  $P_5$  is detected as redundant by checking the bin of a3 ( $author(P_5) = a3$ ) where  $P_4$  covers  $P_5$ .

## 4.3 CliqueBin

In NeighborBin, we index the posts by author aiming to reduce the pairwise post comparisons. But the tradeoff is memory consumption: we have multiple copies of a post in different authors' post bins.

To reduce the overhead on memory consumption incurred by NeighborBin, we identify groups (cliques) of authors that are similar to each other and assign a single bin to them, such that a post generated by any of these authors is only stored in that bin. Specifically we find a *clique edge cover* of  $G_i$ , that is a collection of cliques whose union contains all edges of  $G_i$ . We maintain a post bin per clique (e.g., a map from clique ID to a list of posts). Only the posts from authors in a same clique as  $author(P_i)$  can possibly cover post  $P_i$ . Thus, at the arrival of post  $P_i$ , we check whether it is covered by sequentially comparing it to the posts from only the cliques that contain  $author(P_i)$ . Thus a post  $P_i$  in  $Z$  is stored once for every clique that contains  $author(P_i)$  – instead of once for each neighbor of  $author(P_i)$  in NeighborBin. Note that this approach guarantees that the coverage requirement for posts is satisfied: when a new post  $P_i$  authored by  $a_j$  appears, and  $P_i$  is not similar to earlier posts of  $a_j$  or its neighbors then  $P_i$  will be added to the cliques involving  $a_j$ , because  $a_j$ 's edges are covered by the cliques.

Considering the space consumption, our objective should be to minimize the sum of the sizes of cliques, i.e., the average number of cliques per author is minimized and thus number of copies per post is reduced. This is an NP-hard problem, and hence we have decided to use a simple greedy heuristic. It starts by picking an edge in  $G_i$  to form an initial clique. Then it extends the clique by adding nodes that are neighbors to all the nodes in the clique. When there is no such node, the clique is saved and the algorithm picks another edge not yet included in any found cliques and repeats the above process. We stop when all edges are covered.

Upon a new post  $P_i$ , we use a hashmap (Author2Cliques) to get all the cliques that contains  $author(P_i)$ , and then we check the posts in the corresponding bins. Recall that NeighborBin and UniBin load the author similarity graph  $G_i$  in memory. We can make the same assumption that Author2Cliques is loaded in memory for applying CliqueBin. Similar to the computation of author similarity graph, we assume the clique partition of  $G_i$  and the Author2Cliques mapping are computed offline. We denote this algorithm as **CliqueBin**.

The update of a post bin by CliqueBin is depicted in Figure 6c. Cliques C0 and C1 together cover all the edges in the graph. We can see that  $P_1$  is only stored once in C0's bin (because a1 is in C0) instead of saving 3 copies in NeighborBin as Figure 6b. The same applies to  $P_2$ . Since a3 is in both C0 and C1, during the processing of  $P_3$  CliqueBin may check both bins of C0 and C1.  $P_4$  will only be compared with the bin of C1 because a4 belongs to only C1. Again, CliqueBin checks the coverage of  $P_5$  by iterating both bins of C0 and C1. This example illustrates how CliqueBin can reduce space requirements compared to NeighborBin.

We note that in some cases CliqueBin may have to do a larger number of pairwise post comparisons than NeighborBin. Suppose that after  $P_5$  in the above example author a3 posts  $P_6$  and then author a4 posts  $P_7$ . If  $P_6$  and  $P_7$  are not redundant to any other posts, then  $P_6$  should be added to all four post bins in NeighborBin, and

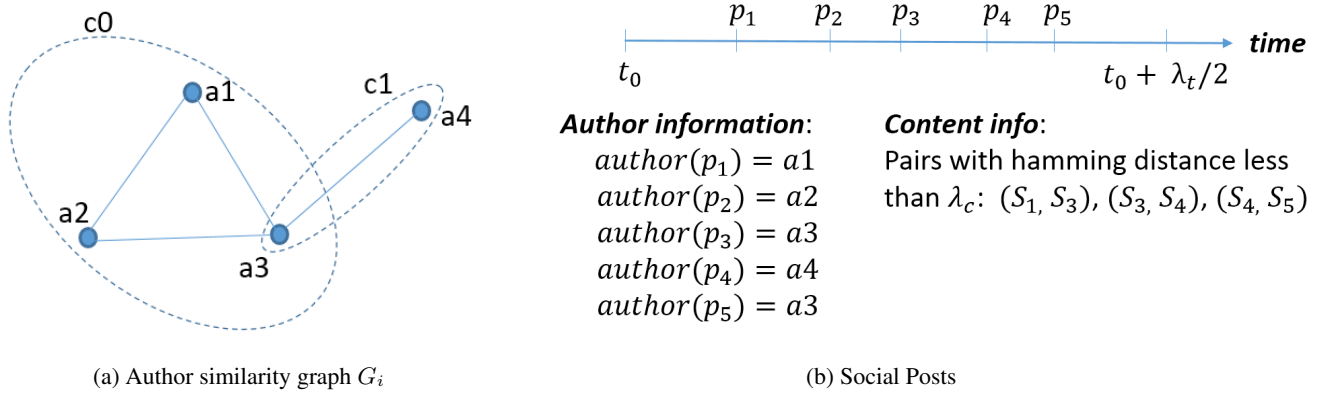


Figure 5: Example of author similarity graph and posts

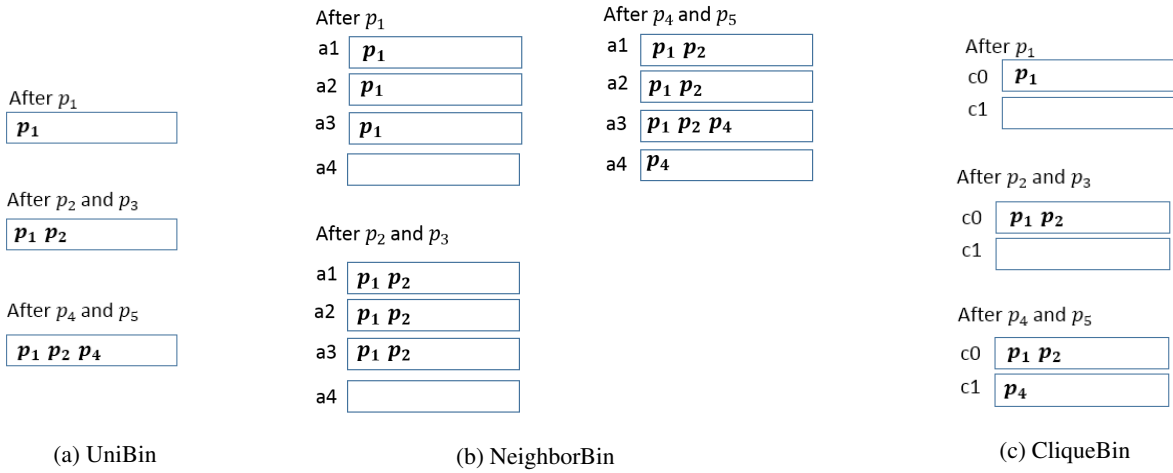


Figure 6: Running example for the three algorithms for SPSD.

to both post bins in CliqueBin. For  $P_7$ , NeighborBin only accesses the bin of  $a_4$  and thus only needs to do two comparisons (with  $P_4$  and  $P_6$ ). In contrast, CliqueBin has to do 5 comparisons: with  $P_1, P_2, P_4$  and twice with  $P_6$  (once in post bin of each clique). We study this experimentally in Section 6.

#### 4.4 Performance Analysis

In this section we show an estimate of the time and space complexity of our algorithms, attempting to capture their performance on realistic data, rather than the worst-case performance. Rigorous derivation of such estimates is challenging, because the behavior of these algorithms heavily depends on the specifics of the data sets, including the topology of the social network. Instead, we provide informal derivations based on several reasonable assumptions about the data set and the graph's topology.

Suppose there are  $m$  subscribed authors, and the total number of posts from these  $m$  authors in a  $\lambda_t$  time range is  $n$ . We assume a ratio of  $r$  ( $\leq 1$ ) posts left after diversification, that is,  $r \cdot n$  non-redundant posts per  $\lambda_t$  time. We also assume that the each author generates the same number of  $\frac{n}{m}$  posts with  $\frac{r \cdot n}{m}$  left after diversification. Further we assume in the author similarity graph, each author has  $d$  neighbors and is in  $c$  ( $\leq d$ ) cliques. We denote  $s$  as the average number of authors in a clique.

Note that cliques may have overlaps. If we define  $q$  as the num-

ber of edges in  $G$  over the total number of edges in  $c$  cliques from  $G$ , we have  $\frac{m \cdot c}{s} = \frac{m \cdot d}{s \cdot (s-1) \cdot q}$ , where both sides compute the number of distinct cliques. Thus we can expect  $c \cdot (s-1) \cdot q = d$  with  $0 \leq q \leq 1$ .

Recall that UniBin puts posts from all authors in  $Z$  into a single post bin. Thus, the total bin size is  $r \cdot n$  in UniBin. Each new post is sequentially compared to each post in the bin and thus the number of post comparisons per new post is  $r \cdot n$ . Each non-redundant post incurs one insertion into the bin.

NeighborBin maintains a set of per-author bins with each bin storing posts from an author and her similar authors. Roughly, each per-author bin stores  $\frac{d+1}{m} \cdot r \cdot n$  posts. Thus the total number of post copies stored in memory is  $(d+1) \cdot r \cdot n$ . At the arrival of a new post  $P_i$ , the number of post comparisons made by NeighborBin is  $\frac{d+1}{m} \cdot r \cdot n$  (compare  $P_i$  to all posts in  $author(P_i)$ 's post bin). Each non-redundant post incurs a total of  $(d+1)$  insertions into the bins.

In CliqueBin, for each non-redundant post  $P_i$  we store its  $c$  copies: one copy in the bin of each clique containing  $author(P_i)$ . Thus, the total size of the clique bins is  $c \cdot r \cdot n$ . CliqueBin compares each new post  $P_i$  to posts in the bins of  $c$  cliques that contain  $author(P_i)$ , which leads to a total of  $\frac{s \cdot c}{m} \cdot r \cdot n$  comparisons. Each non-redundant post incurs a total of  $c$  insertions into the bins.

Table 2 summarizes the performance analysis. We can see that all these results contain the same component  $r \cdot n$ . Obviously, all

Table 2: Performance estimation of the algorithms for SPSD

	UniBin	NeighborBin	CliqueBin
RAM	$r \cdot n$	$(d + 1) \cdot r \cdot n$	$c \cdot r \cdot n$
Comparisons in $\lambda_t$	$r \cdot n^2$	$\frac{d+1}{m} \cdot r \cdot n^2$	$\frac{s \cdot c}{m} \cdot r \cdot n^2$
Insertions in $\lambda_t$	$r \cdot n$	$(d + 1) \cdot r \cdot n$	$c \cdot r \cdot n$

three diversity thresholds effects the ratio of non-redundant post  $r$ . The value of  $n$  is affected by several factors, such as the frequency of the post stream  $\mathbf{P}$  and the setting of time diversity threshold  $\lambda_t$ .

An important factor that affects the performance of the algorithms, especially NeighborBin and CliqueBin, is the topology of the author similarity graph  $G$ . In the above estimates, we use parameters  $d, c, s$  and  $m$  to capture the topology properties. We note that the values of the ratios of  $d, c, s$  to  $m$  are functions of the author diversity threshold  $\lambda_a$ . Given a set of subscribed authors (i.e., with  $m$  fixed), the larger  $\lambda_a$  the denser  $G$  is (in terms of the number of edges). Thus, the number of neighbors per author ( $d$ ) increases with  $\lambda_a$ , which means the performance of NeighborBin will drop if all other settings remain unchanged. We also argue that  $c$  and  $c \cdot s$  increase with the graph’s density, and hence we expect CliqueBin to perform better for smaller  $\lambda_a$ s. In Section 6, we confirm this through experiments on real data set.

In Section 6 we will summarize the use cases for each algorithm based on this theoretical analysis combined with our experimental results.

## 4.5 Summary

We summarize the characteristics of the three algorithms in Table 3. In terms of data structure, UniBin and NeighborBin need the author similarity graph, while CliqueBin needs the mapping of each author to the set of cliques containing the author. As we mentioned, we assume that all these data structures are maintained in memory.

We can see that UniBin requires the least RAM. NeighborBin reduces the post comparisons compared to UniBin, but has high RAM consumption because it maintains multiple copies of a post. CliqueBin outperforms NeighborBin in terms of RAM consumption, by reducing the number of copies per post (and thus insertions per post), but it incurs more post comparisons. Since CliqueBin still maintains multiple copies of a post, it requires more insertions and higher RAM consumption than UniBin. Also, since CliqueBin does not compare posts from non-similar authors, we expect the number of comparisons in CliqueBin to be lower than in UniBin.

## 5. ALGORITHMS FOR MULTIPLE-USERS SPSD (M-SPSD)

In this section, we extend our ideas to solving M-SPSD. When we move from applying the diversity model for one user to multiple users, the crucial question is whether it is possible to reuse the computation performed for diversifying one user’s stream to diversify the other users’ streams.

A simple way to solve M-SPSD is to process the post stream for each user individually. That is, we can apply the algorithm for SPSD on each user’s post stream separately. We denote the corresponding algorithms for M-SPSD as **M\_UniBin**, **M\_NeighborBin** and **M\_CliqueBin** respectively, to distinguish them from the algorithms for SPSD. In this section, we present variations of these algorithms to optimize the diversification process by reusing computations for multiple users who share subscriptions.

If two users do not share any common subscriptions, then their

post streams are disjoint and thus the computation of diversifying one’s stream cannot be reused for diversifying the other users’ post streams. Hence we only consider the cases for optimization when users share the same subset of subscriptions.

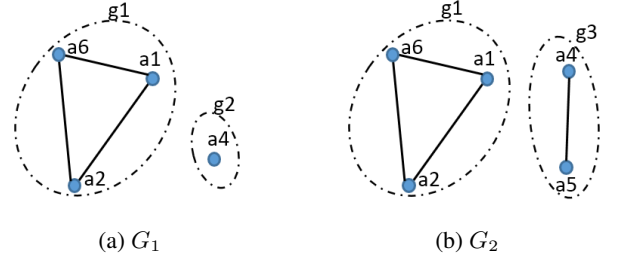


Figure 7: Author similarity graphs of two users  $u_1$  and  $u_2$ .

However, we notice several limitations to reusing the diversification computation across multiple users, even if they share some subscriptions. We use examples to illustrate this. Figure 7 shows two users,  $u_1$  and  $u_2$ , sharing a set of subscriptions  $\{a_1, a_2, a_4, a_6\}$ .

We notice that after diversification  $u_1$  may see a different subset of the posts from  $a_4$  as  $u_2$ .  $u_2$  subscribes to  $a_5$  which is a similar author to  $a_4$ . Thus, it is possible that some posts from  $a_4$  are shown to  $u_1$  but not to  $u_2$  if they are covered by  $a_5$ ’s posts.

However, the same diversified set of posts from  $\{a_1, a_2, a_6\}$  will be shown to  $u_1$  and  $u_2$ . The three authors form a *connected component* (denoted as  $g_1$  in Figure 7) in both  $G_1$  and  $G_2$ . That is, in both  $G_1$  and  $G_2$  there are no other authors similar to any author in  $\{a_1, a_2, a_6\}$ . Hence, posts from other subscribed authors can not cover the posts from  $\{a_1, a_2, a_6\}$ . Thus, the diversification processes on the posts from  $\{a_1, a_2, a_6\}$  are exactly the same for  $u_1$  and  $u_2$ . This means that we can reuse the data structures and computation across  $u_1$  and  $u_2$  for diversifying the post stream from  $\{a_1, a_2, a_6\}$ .

Based on these observations, we can optimize the diversification process for multiple users if they subscribe to a same set of authors that form a connected component. We can then consider a post stream (a subset of  $\mathbf{P}$ ) of each connected component separately, apply the diversification algorithm on it, and then merge the diversified post streams together.

For this, we first process the author similarity graph  $G_i$  of each user  $u_i$  to compute all connected components of all  $G_i$ s. (Since different  $G_i$ s may overlap, some nodes may appear in several components.) For each distinct connected component  $g_i$ , we run one of the proposed algorithms for SPSD on the post stream by the authors in  $g_i$ . User  $u_i$ ’s post stream consists of the union of the diversified post streams from all connected components in  $G_i$ .

For example, as shown in Figure 8b, we can apply the UniBin algorithm for three distinct connected components ( $g_1, g_2$  and  $g_3$ ), that is, we maintain a single post bin for each of the three components. Then the posts shown to  $u_1$  is the union of the two diversified post streams from  $g_1$  and  $g_2$ . We refer this algorithm as  $S\_UniBin$ . For comparison, we show the example for  $M\_UniBin$  in Figure 8a.  $M\_UniBin$  maintains a post bin for each user separately. To extend NeighborBin, we maintain a per-author post bin for each author in a distinct connected component  $g_i$ . To extend CliqueBin, we do the clique partition for each  $g_i$ , then maintain a per-clique post bin as described earlier.

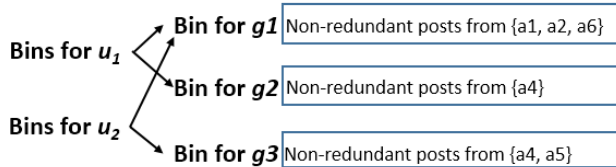
We denote the three algorithms with the above optimization as **S\_UniBin**, **S\_NeighborBin** and **S\_CliqueBin** respectively.

Table 3: Differences between the three algorithms for SPSD

		UniBin	NeighborBin	CliqueBin
Data Structures		(1) Author similarity graph (2) A single post bin storing posts from all authors.	(1) Author similarity graph (2) A post bin per author storing posts from the author and her neighbors.	(1) Author clique mapping (2) A post bin per clique storing posts from all the authors in the clique.
Properties	RAM	Low	High	Moderate
	Comparisons	High	Low	Moderate
	Insertions	Low	High	Moderate



(a) M\_UniBin



(b) S\_UniBin

Figure 8: Example of M\_UniBin and S\_UniBin.

## 6. EXPERIMENTAL EVALUATION

### 6.1 Data Set and Experimental Settings

We conducted our experiments on Twitter data. The authors in [22] published a Twitter social graph dataset consisting of more than 660,000 Twitter authors (accounts). Computing the author similarity graph for the whole data set would be prohibitive, as it requires comparing all pairs of authors. Instead, we used a subgraph of 20,150 authors obtained by randomly picking an initial author, and adding authors that are reachable through Breadth First Search on the follower-followee graph.

We computed all pairwise author similarity for these 20,150 Twitter authors. The author similarity distribution is depicted in Figure 9, where the x-axis shows the author similarity value and y-axis shows the fraction of author pairs with similarity values larger than the value indicated by x-axis. It shows that 2.3% author pairs are with similarity  $\geq 0.2$  and 0.6% pairs are with similarity  $\geq 0.3$ .

Further, we crawled the tweets of these twitter authors using Twitter REST API<sup>2</sup> for one day. The tweets data set contains 233,311 tweets, which means these Twitter authors post slightly over 10 tweets per author per day. After we removed some short tweets that have less than two words or only contain meaningless tokens, there are 213,175 tweets left.

We implemented all algorithms in Java. We ran our experiments on machines with Quad Core Intel(R) Xeon(R) E3-1230 v2@3.30GHz CPU and 16GB RAM.

### 6.2 Performance of the algorithms for SPSD

In this section, we evaluate the performance of the three algorithms for SPSD. We assume that a user follows all the Twitter authors in our dataset, and we run the algorithms on the user’s post stream which consists of 213,175 posts in one day.

First, we study the effect of the three diversity dimensions: time, content and author. Figure 10 shows the number of tweets left after diversification under different settings by removing diversity dimensions and varying diversity thresholds. Incorporating all three diversity dimensions with reasonable diversity thresholds, the di-

<sup>2</sup><https://dev.twitter.com/overview/documentation>

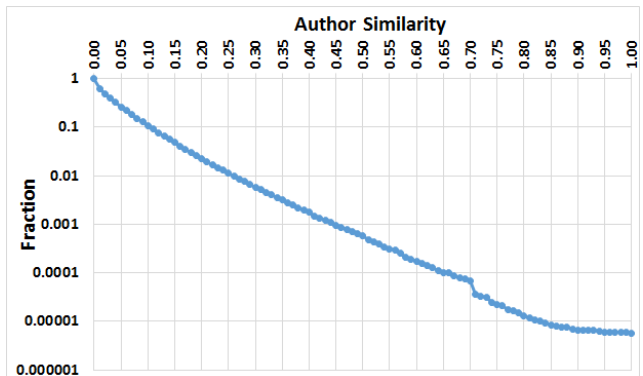


Figure 9: Author similarity distribution in our data set

versification model prunes about 10% redundant posts. We notice that incorporating only some of these dimensions will largely change the size of diversified stream. It means that all three dimensions play an important role in diversifying tweet data.

#### 6.2.1 Performance of the algorithms under different diversity settings

The analysis in Section 4.4 indicates that the performance of the three algorithms for SPSD is effected by several factors such as the diversity thresholds and the post stream throughput. These diversity settings could change the relative performance between the three algorithms. In this section, we study the performance of each algorithm under different settings and we experimentally show that each algorithm outperforms the other two in certain settings. Supported by former analysis and experimental results, we will summarize use cases for each algorithm.

**Varying time diversity threshold  $\lambda_t$ .** In Figure 11, we present the performance of UniBin, NeighborBin and CliqueBin under different time diversity thresholds ( $\lambda_t$ ). In this experiment, we set  $\lambda_c = 18$  (according to the results in Figure 4) and  $\lambda_a = 0.7$  (i.e., we consider two authors are similar if the cosine similarity between

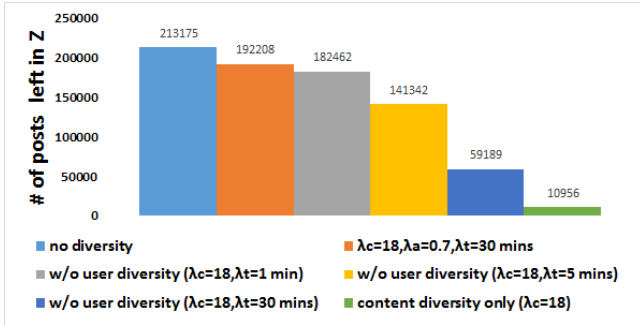


Figure 10: Number of tweets left after applying diversification in our data set

their followee vector is  $\geq 0.3$  and thus distance is  $\leq 0.7$ ). The running time shows the execution time for an algorithm to ingest the 213,175 posts.

In Figure 11a we can see that the running time of all three algorithms decreases with smaller  $\lambda_t$ s. The reason is that with a smaller  $\lambda_t$ , the algorithms perform fewer pairwise post comparisons (depicted in Figure 11c). NeighborBin and CliqueBin outperform UniBin in terms of running time. We also notice that CliqueBin is more efficient than NeighborBin when  $\lambda_t$  is small (e.g.,  $\leq 10$  minutes). This gives us evidence for the summarization of use cases in Table 4 for NeighborBin and CliqueBin.

Smaller  $\lambda_t$  also reduces the RAM consumption because the algorithms store shorter history of  $Z$  in post bins. As expected, NeighborBin requires more memory than UniBin and CliqueBin.

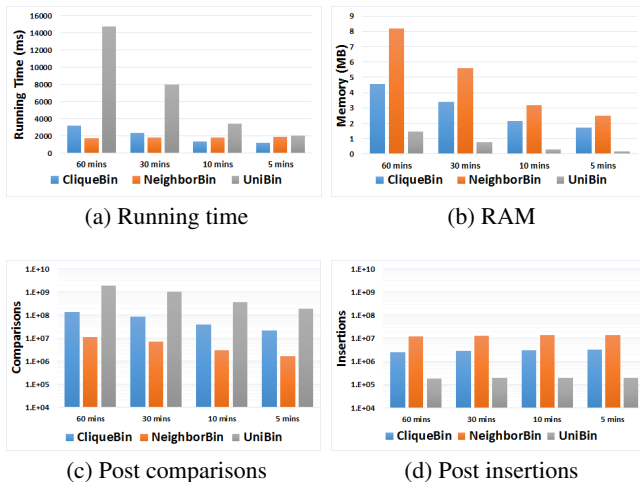


Figure 11: Performance of the three algorithms under different time diversity thresholds  $\lambda_t$ .

**Varying content diversity threshold  $\lambda_c$ .** We also study the performance of the three algorithms by varying  $\lambda_c$ . For this, we set  $\lambda_t = 30$  mins and  $\lambda_a = 0.7$  and we vary the  $\lambda_c$  from 9 to 18. Figure 12 depicts the results. It shows that, for all the three algorithms, the change of content diversity threshold only slightly affects the performance. The reason is that SimHash can effectively detect tweets with near-duplicate content for  $\lambda_c \geq 9$  as we can see in Figure 4. With  $\lambda_c$  changing from 9 to 18, the precision is already stable. The recall is lower with smaller  $\lambda_c$ , which means more posts will be detected as non-redundant. But this increase in number of

non-redundant posts is slight, and thus the increase in the number of comparisons and insertions does not affect the overall efficiency significantly.

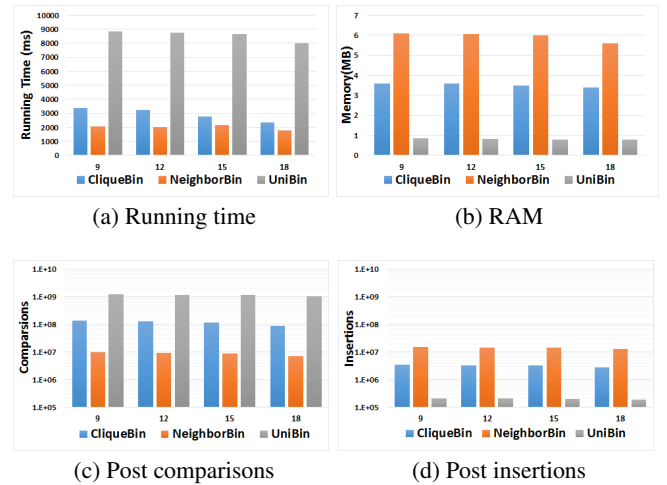


Figure 12: Performance of the three algorithms under different content diversity thresholds  $\lambda_c$ .

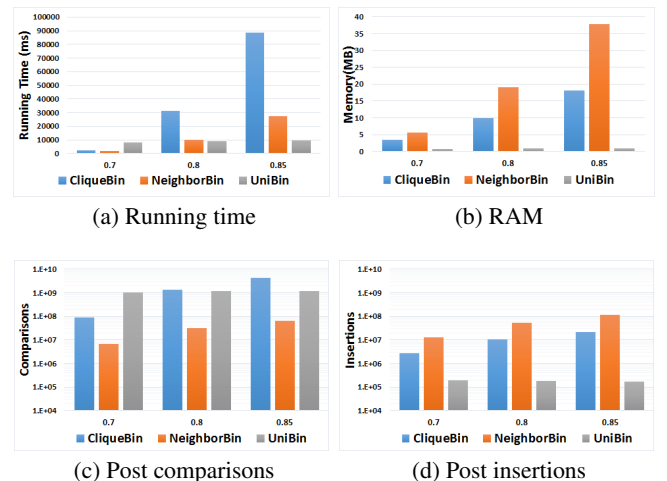


Figure 13: Performance of the three algorithms under different author diversity thresholds  $\lambda_a$ .

**Varying author diversity threshold  $\lambda_a$ .** Further, we study the performance by varying  $\lambda_a$ . The results are presented in Figure 13 where we set  $\lambda_t = 30$  mins and  $\lambda_c = 18$ .

We observe that the author diversity threshold  $\lambda_a$  significantly affects the overall performance of NeighborBin and CliqueBin but not UniBin. When  $\lambda_a$  increases, the author similarity graph gets denser and thus the number of neighbors per author and the number of cliques per author both increase. For instance, when  $\lambda_a = 0.7$  the number of neighbors per author ( $d$ ) is 113.7, the number of cliques per author ( $c$ ) is 29 and the average size of a clique ( $s$ ) is 20 in our data set. They change to 437.3, 106 and 38 correspondingly with  $\lambda_a = 0.8$ . Hence, the number of copies per post in NeighborBin and CliqueBin increases. This explains that in Figure 13 the memory consumption by NeighborBin and CliqueBin

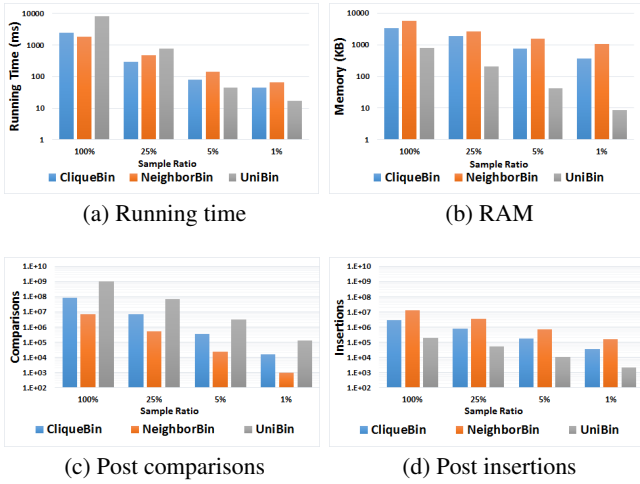


Figure 14: Performance of the three algorithms under different post rates.

increases sharply with larger  $\lambda_a$ s. However, the number of non-duplicate posts does not vary much with different  $\lambda_a$ s in our data set; thus the performance of UniBin is stable.

We note that when  $\lambda_a$  is large the performance of NeighborBin and CliqueBin (in terms of both memory consumption and running time) is significantly worse than UniBin. Hence, we expect UniBin is the best choice among these three algorithms in use cases where  $\lambda_a$  is large, as we summarize in Table 4.

**Varying post stream throughputs.** We also study the performance of the algorithms under different post stream throughputs. We test this in two ways: (i) varying subscriptions’ post rate, and (ii) varying the number of subscriptions. For both, we keep  $\lambda_t = 30 \text{ mins}$ ,  $\lambda_a = 0.7$  and  $\lambda_c = 18$ .

*Varying post generation rate.* For this, we randomly sample the posts from the 21,050 authors and solve SPSP on the sampled post stream. We conduct experiments for the sample ratio 25%, 5% and 1% and present the results in Figure 14. The results show that when the throughput is low (the same ratio is low) UniBin outperforms the other two algorithms. We can also see that CliqueBin performs better than NeighborBin with a moderate or small post generation rate.

*Varying the number of subscribed authors.* The results shown above are for the case of one user subscribing (following) all Twitter authors in our dataset. In this experiment, we randomly sample Twitter authors in our dataset with different sample sizes. We assume that a user subscribes to all authors in one sample and we run the algorithms on the user’s post stream. The results in Figure 15 show that UniBin slightly outperforms the other two when the number of subscriptions is small.

To summarize, UniBin delivers better performance than NeighborBin and CliqueBin when the stream throughput is low. This is consistent with our analysis in Section 4.4 – see also Table 4.

## 6.2.2 Discussion

Through extensive experiments, we observe that each algorithm outperforms the other two in certain cases. In Table 4 we summarize the best choice of algorithm in different use cases based on our analysis and experimental study.

UniBin is the most memory efficient among the three algorithms. Thus in applications with limited RAM UniBin should be consid-

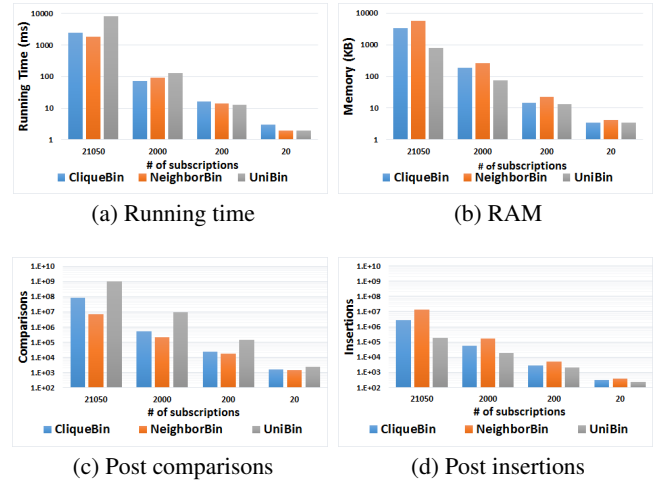


Figure 15: Performance of the three algorithms varying the number of subscribed authors.

ered. Further, when the stream throughput is low (we tested it with small number of subscriptions and low post generation rate), UniBin performs better than the other two. According to the analysis in Table 2, we expect that the number of comparisons increases super-linearly with  $n$  (the number of posts in a  $\lambda_t$  time range), however the number of insertions increases sub-linearly with  $n$ . With a lower stream throughput (smaller  $n$ ) the overhead of insertions in NeighborBin and CliqueBin is a large contribution to the total running time. When  $n$  is small enough, the overhead on insertions becomes larger than the saving on comparisons for NeighborBin and CliqueBin compared with UniBin. The similar reasoning can be applied to explain why UniBin is the best choice when  $\lambda_t$  is very small. To clarify, in Figure 11 we did not include the results by setting  $\lambda_t = 1 \text{ min}$  where UniBin performs best among the three algorithms. We argued that with a larger  $\lambda_a$  both  $d$  (number of neighbors per author) and  $c$  (number of cliques per author) increase and thus NeighborBin and CliqueBin both have higher number of comparisons and insertions. Thus we can see UniBin is preferable when  $\lambda_a$  is set large. One example use case for UniBin is News RSS Feed reader, where the author similarity graph is dense. Generally, news agents form clusters (e.g., by their political views) such that in each cluster the news agents are similar to each other from a user’s perspective. Another use case could be Google Scholar where the post (scientific publication) throughput is low.

In other cases, CliqueBin or NeighborBin will be the better choice. They both perform well in cases with a high or moderate stream throughput, which is very common for online social networks. The tie breaker between them is the time diversity threshold  $\lambda_t$ , as we analyzed  $\lambda_t$  determines the tradeoffs between costs of comparisons and insertions. CliqueBin is a better choice if  $\lambda_t$  is set moderately. For example, in Twitter information is time sensitive and thus people may be interested in reading posts with related content but with time distance larger than, say, minutes. For applications where the value of  $\lambda_t$  could be in hours or even days, NeighborBin can be applied. For example, Twitch<sup>3</sup> is a platform on which people can watch and share video game shows. Users may not be interested in watching the video record of the same match that posted at different time. Even in Twitter some users may prefer to customize the  $\lambda_t$  to a larger value, in order to reduce the post volume if they

<sup>3</sup><http://www.twitch.tv/>

Table 4: Use cases of the three algorithms for SPSD

Conditions	Algorithm choice	Example use case
Very small $\lambda_t$ OR low stream throughput OR large $\lambda_a$ (dense $G$ ) OR RAM is a critical limitation	UniBin	News RSS Feed, Google Scholar
Large $\lambda_t$ AND small $\lambda_a$ (sparse $G$ ) AND high stream throughput	NeighborBin	Twitch
Moderate $\lambda_t$ AND small $\lambda_a$ (sparse $G$ ) AND high stream throughput	CliqueBin	Twitter

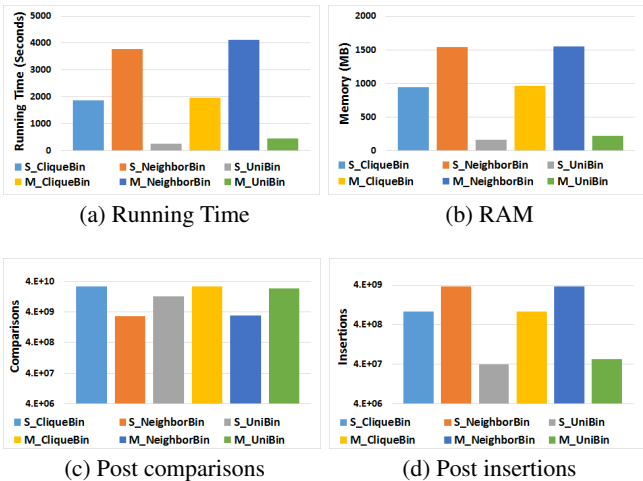


Figure 16: Performance of the algorithms for M-SPSD.

follow a large number of authors.

### 6.3 Performance of the algorithms for M-SPSD

We consider now the scenario where each Twitter author is also a user. Each user subscribes to (follows) a set of authors which we can get from the original follower-followee social graph. Then we run the algorithms solving M-SPSD for these 21,050 users in our data set. For the experiments in this section, we set  $\lambda_t = 30 mins$ ,  $\lambda_a = 0.7$  and  $\lambda_c = 18$ .

The average number of subscriptions in our sampled user data set is 443.6 and the median is 187. Since we only crawled the posts and computed the author similarity graph for the set of 21,050 authors, we ignored the subscriptions that are not in this set. Then the average number of subscriptions per user drops to 130 and the median is 20. We should note that this reduces the probability of different users sharing common subscriptions.

Figure 16 presents the performance of the algorithms. It shows that the proposed optimization (reusing computation and data structure across multiple users described in Section 5) improves time efficiency as well as memory consumption. Specifically, S\_UniBin uses 43% less running time and 27% less memory than M\_UniBin. In the S\_UniBin method, posts are stored separately by connected components. This reduces the number of comparisons significantly over M\_UniBin. We also observe that S\_NeighborBin reduces the running time of M\_NeighborBin by 8% while S\_CliqueBin improves M\_NeighborBin by 4% in running time.

S\_UniBin achieves superior performance. We also notice that

S\_NeighborBin requires fewer post comparisons than S\_UniBin but many more insertions. We think that S\_UniBin outperforms S\_NeighborBin and S\_CliqueBin also because its post access pattern is sequential while in the other two are not (each post bin is a map).

## 7. RELATED WORK

**Time Aware Diversity.** The authors of [7] solve the problem of maintaining the  $k$  most diverse results in a sliding window over a stream. MaxMin semantics is used. They maintain a data structure called the cover tree and show how to incrementally add new and remove expired results from this tree. The cover tree cannot be used for our diversity semantics because it cannot handle simultaneous similarity in three dimensions: time, content and author.

**Diversification on Microblogging Posts.** The work of [4] studies the problem of diversifying posts in microblogging systems. In their problem setting, users subscribe several queries (topics). However, in practice users are more often subscribing to authors, which is the setting of the problem we studied in this paper. In [4] they apply strict coverage semantics similar to ours, but limited only to time and content diversity. Unlike in our model, in [4] the content diversity is guided by the inputted queries where no inter-post content similarity is considered. They also studied the stream variation of their problem in which they allow a lag upon a new post to decide whether it should be outputted. In our problem, the diversity model is required to make the decision immediately at the arrival of a post.

**Document Stream Summarization.** The authors of [20] work on the problem of summarizing a Twitter stream. They model the summarization problem as a facility location problem. Give a budget of  $k$ , they aim to select  $k$  tweets that maximize the similarity to the whole tweets set. They incorporate the time factor to measure the document similarity of two posts. But unlike in our problem, instead of using a hard (boolean) threshold, they consider an exponential decay to the content similarity based on their timestamp difference. In the work of [13], the authors apply clustering techniques for Twitter stream summarization. Tweets are clustered according to content similarity. For each cluster, they build a word graph or phrase graph and pick frequent sentences (“paths” in the graph) to construct a summary. The sentences in the summary may not be in any original tweet. The authors of [18] propose a one-pass online clustering algorithm to cluster tweets, and then they generate online summaries by selecting  $k$  tweets (one from each cluster) that have high LexRank [8] score. In [16], the authors apply topic modeling for personalized time-aware tweet summarization. However, all these work do not consider author similarity to measure the similarity between tweets.

**Detecting Duplicate Tweets.** In [21], the authors propose to use machine learning methods to detect near-duplicates in tweets. For

this, they construct a rich set of syntactic, semantic and contextual features. They aim to distinguish different levels of near-duplicates, e.g. exact copy, strong near-duplicate, or weak near-duplicate.

## 8. CONCLUSION

In this paper, we studied the novel problem of diversifying social post streams by incorporating diversity in three dimensions: content, time and author. We illustrated the challenges of solving the problem and proposed various algorithms to efficiently handle these challenges. We showed the tradeoffs between our proposed algorithms and argued the use cases for them. We also studied the problem of applying the proposed diversification model for multiple users in a social system. Extensive experiments proved the effectiveness of our model and efficiency of proposed algorithms.

## 9. ACKNOWLEDGEMENT

Marek Chrobak is supported by National Science Foundation (NSF) grants CCF-1217314 and CCF-1536026. Vagelis Hristidis is supported by NSF grants IIS-1216007 and IIS-1447826.

## 10. REFERENCES

- [1] R. Agrawal, S. Gollapudi, A. Halverson, and S. Jeong. Diversifying search results. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pages 5–14, 2009.
- [2] B. S. Alper, J. A. Hand, S. G. Elliott, S. Kinkade, M. J. Hauan, D. K. Onion, and B. M. Sklar. How much effort is needed to keep up with the literature relevant for primary care? *Journal of the Medical Library association*, 92(4):429, 2004.
- [3] G. Capannini, F. M. Nardini, R. Perego, and F. Silvestri. Efficient diversification of web search results. *Proceedings of the VLDB Endowment*, 4(7):451–459, 2011.
- [4] S. Cheng, A. Arvanitis, M. Chrobak, and V. Hristidis. Multi-query diversification in microblogging posts. In *17th International Conference on Extending Database Technology*, pages 133–144, 2014.
- [5] E. Demidova, P. Fankhauser, X. Zhou, and W. Nejdl. Divq: diversification for keyword search over structured databases. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 331–338. ACM, 2010.
- [6] M. Drosou and E. Pitoura. Disc diversity: Result diversification based on dissimilarity and coverage. *Proceedings of the VLDB Endowment*, 6(1):13–24, Nov 2012.
- [7] M. Drosou and E. Pitoura. Dynamic diversification of continuous data. In *15th International Conference on Extending Database Technology*, pages 216–227. ACM, 2012.
- [8] G. Erkan and D. R. Radev. Lexrank: Graph-based lexical centrality as salience in text summarization. *Journal of Artificial Intelligence Research*, 22(1):457–479, 2004.
- [9] A. Goel, A. Sharma, D. Wang, and Z. Yin. Discovering similar users on twitter. In *11th Workshop on Mining and Learning with Graphs*, 2013.
- [10] M. Koniaris, G. Giannopoulos, T. Sellis, and Y. Vasilieou. Diversifying microblog posts. In *Web Information Systems Engineering–WISE 2014*, pages 189–198. Springer, 2014.
- [11] G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web*, pages 141–150. ACM, 2007.
- [12] F. Morstatter, J. Pfeffer, H. Liu, and K. M. Carley. Is the sample good enough? comparing data from twitter’s streaming api with twitter’s firehose. In *Proceedings of the 7th International AAI Conference on Web and Social Media*, 2013.
- [13] A. Olariu. Clustering to improve microblog stream summarization. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 220–226, 2012.
- [14] M. G. Ozsoy, K. D. Onal, and I. S. Altıngövdü. Result diversification for tweet search. In *Web Information Systems Engineering–WISE 2014*, pages 78–89. Springer, 2014.
- [15] D. Rafiei, K. Bharat, and A. Shukla. Diversifying web search results. In *Proceedings of the 19th international conference on World Wide Web*, pages 781–790, 2010.
- [16] Z. Ren, S. Liang, E. Meij, and M. de Rijke. Personalized time-aware tweets summarization. In *Proceedings of the 36th international ACM SIGIR conference on research and development in information retrieval*, pages 513–522, 2013.
- [17] C. Sadowski and G. Levin. Simhash: Hash-based similarity detection. Technical report, Technical report, Google, 2007.
- [18] L. Shou, Z. Wang, K. Chen, and G. Chen. Sumblr: continuous summarization of evolving tweet streams. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 533–542. ACM, 2013.
- [19] S. Sood and D. Loguinov. Probabilistic near-duplicate detection using simhash. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1117–1126. ACM, 2011.
- [20] H. Takamura, H. Yokono, and M. Okumura. Summarizing a document stream. In *Advances in Information Retrieval*, pages 177–188. Springer, 2011.
- [21] K. Tao, F. Abel, C. Hauff, G.-J. Houben, and U. Gadiraju. Groundhog day: near-duplicate detection on twitter. In *Proceedings of the 22nd international conference on World Wide Web*, pages 1273–1284, 2013.
- [22] X. Wang, H. Liu, P. Zhang, and B. Li. Identifying information spreaders in twitter follower networks. Technical Report TR-12-001, School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ 85287, USA, 2012.
- [23] X. Yang, A. Ghoting, Y. Ruan, and S. Parthasarathy. A framework for summarizing and analyzing twitter feeds. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 370–378. ACM, 2012.



# Social, Structured and Semantic Search

Raphaël Bonaque  
INRIA & U. Paris-Sud, France  
raphael.bonaque@inria.fr

François Goasdoué  
U. Rennes 1 & INRIA, France  
fg@irisa.fr

Bogdan Cautis  
U. Paris-Sud & INRIA, France  
bogdan.cautis@lri.fr

Ioana Manolescu  
INRIA & U. Paris-Sud, France  
ioana.manolescu@inria.fr

## ABSTRACT

Social content such as blogs, tweets, news etc. is a rich source of interconnected information. We identify a set of requirements for the meaningful exploitation of such rich content, and present a new data model, called  $S3$ , which is the first to satisfy them.  $S3$  captures *social* relationships between users, and between users and content, but also the *structure* present in rich social content, as well as its *semantics*. We provide the first top- $k$  keyword search algorithm taking into account the social, structured, and semantic dimensions and formally establish its termination and correctness. Experiments on real social networks demonstrate the efficiency and qualitative advantage of our algorithm through the joint exploitation of the social, structured, and semantic dimensions of  $S3$ .

## 1. INTRODUCTION

The World Wide Web (or Web, in short) was designed for users to interact with each other by means of pages interconnected with hyperlinks. Thus, the Web is the earliest inception of an *online* social network (whereas “real-life” social networks have a much longer history in social sciences). However, the technologies and tools enabling large-scale online social exchange have only become available recently. A popular model of such exchanges features: *social network users*, who may be connected to one another, *data items*, and the possibility for users to *tag* data items, i.e., to attach to an item an annotation expressing the user’s view or classification of the item. Variants of this “user-item-tag” (UIT) model can be found e.g., in [18, 21, 30]. In such contexts, a user, called *seeker*, may ask a query, typically as a set of keywords. The problem then is to find the best query answers, taking into account both the relevance of items to the query, and the social proximity between the seeker and the items, based also on tags. Today’s major social networks e.g., Facebook [7], all implement some UIT variant. We identify a set of basic requirements which UIT meets:

**R0.** UIT models *explicit social connections* between users, e.g.,  $u_1$  is a friend of  $u_0$  in Figure 1, to which we refer throughout this paper unless stated otherwise. It also captures *user endorsement (tags)* of data items, as UIT search algorithms *exploit both the user endorsement and the social connections* to return items most likely to interest the seeker, given his social and tagging behavior.

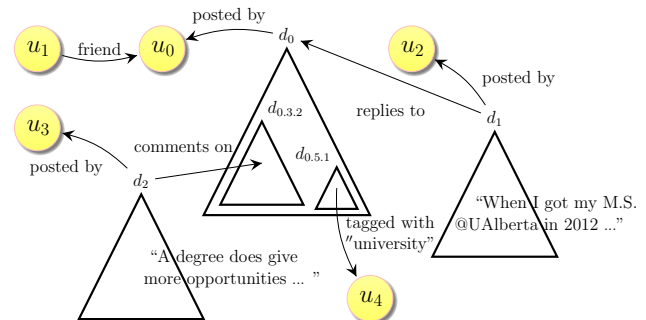


Figure 1: Motivating example.

To fully exploit the content shared in social settings, we argue that the model used for such data (and, accordingly, the query model) must also satisfy the requirements below:

**R1.** The current wealth of publishing modes (through social networks, blogs, interlinked Web pages etc.) allows many different relations between items. For example, document  $d_1$  *replies to* document  $d_0$  (think for instance of opposite-viewpoint articles in a heated debate), while document  $d_2$  *comments on* the paragraph of  $d_0$  identified by the URI  $d_{0.3.2}$ . The model must capture **relations between items**, in particular since **they may lead to implicit relations between users**, according to their manipulations of items. For instance, the fact that  $u_2$  posted  $d_1$  as a reply to  $d_0$ , posted by  $u_0$ , entails that  $u_2$  at least read  $d_0$ , and thus some form of exchange has taken place between  $u_0$  and  $u_2$ ; if one looked for *explicit* social connections only, we would wrongly believe that  $u_0$  and  $u_2$  have no relation to each other.

**R2.** Items shared in social media often have a rich structured content. For instance, the article  $d_0$  comprises many sections, and paragraphs, such as the one identified by the URI  $d_{0.3.2}$ . **Document structure must be reflected in the model** in order to return *useful* document fragments as query results, instead of a very large document or a very small snippet of a few words (e.g., exactly the search keywords). Document structure also helps discern when users have *really* interacted through content. For instance,  $u_3$  has interacted with  $u_0$ , since  $u_3$  comments on the fragment  $d_{0.3.2}$  of  $u_0$ ’s article  $d_0$ . In contrast, when user  $u_4$  tags with “university” the fragment  $d_{0.5.1}$  of  $d_0$ , disjoint from  $d_{0.3.2}$ ,  $u_4$  may not even have read the same text as  $u_3$ , thus the two likely did not interact.

**R3.** **Item and tag semantics** must be modelled. Social Web data encapsulates users’ knowledge on a multitude of topics; ontologies, either general such as DBpedia or Google’s Knowledge Base, or application-specific, can be leveraged to *give query answers which cannot be found without relying on semantics*. For instance, assume  $u_1$  looks for information about *university graduates*: document  $d_1$  states that  $u_2$  holds a M.S. degree. Assume a knowledge base specifies that *a M.S. is a degree* and that *someone*

$U$ URIs	$L$ literals	$\mathcal{K}$ keywords	$Ext(k)$ extension of $k$
$\Omega$ users	$D$ documents	$T$ tags	I graph instance

**Table 1: Main data model notations.**

having a degree is a graduate. The ability to return as result the snippet of  $d_1$  most relevant to the query is directly conditioned by the ability to exploit the ontology (and the content-based interconnections along the path:  $u_1$  friend of  $u_0$ ,  $u_0$  posted  $d_0$ ,  $d_1$  replied to  $d_0$ ).

**R4.** In many contexts, tagging may apply to tags themselves, e.g., in annotated corpora, where an annotation (tag) obtained from an analysis can further be annotated with provenance details (when and how the annotation was made) or analyzed in its turn. Information from higher-level annotations is obviously still related to the original document. The model should allow expressing **higher-level tags**, to exploit their information for query answering.

**R5.** The data model and queries should have **well-defined semantics**, to precisely characterize computed results, ensure correctness of the implementation, and allow for optimization.

**R6.** The model should be **generic** (not tied to a particular social network model), **extensible** (it should allow easy extension or customization, as social networks and applications have diverse and rapidly evolving needs), and **interoperable**, i.e., it should be possible to get richer / more complete answers by integrating different sources of social connections, facts, semantics, or documents. This ensures in particular independence from any proprietary social network viewpoint, usefulness in a variety of settings, and a desirable form of “monotonicity”: the more content is added to the network, the more its information value increases.

This work makes the following contributions.

1. We present  $\mathcal{S}3$ , a novel *data model* for structured, semantic-rich content exchanged in social applications; it is the first model to meet the requirements **R0** to **R6** above.
2. We revisit *top-k social search for keyword queries*, to retrieve the most relevant *document fragments* w.r.t. the social, structural, and semantical aspects captured by  $\mathcal{S}3$ . We identify a set of *desirable properties of the score function* used to rank results, provide a *novel query evaluation algorithm* called  $\mathcal{S}3_k$  and *formally establish its termination and correctness*; the algorithm intelligently exploits the score properties to stop *as early as possible*, to return answers fast, with little evaluation effort.  $\mathcal{S}3_k$  is the first to formally guarantee a specific result in a structured, social, and semantic setting.
3. We implemented  $\mathcal{S}3_k$  based on a concrete score function (extending traditional ones from XML keyword search) and experimented with *three real social datasets*. We demonstrate the *feasibility* of our algorithm, and its *qualitative advantage* over existing approaches: it finds relevant results that would be missed by ignoring any dimension of the graph.

An  $\mathcal{S}3$  instance can be exploited in many other ways: through structured XML and/or RDF queries as in [9], searching for users, or focusing on annotations as in [4]; one could also apply graph mining etc. In this paper, we first describe the data model, and then revisit the top-k document search problem, since it is the most widely used (and studied) in social settings.

In the sequel, Section 2 presents the  $\mathcal{S}3$  data model, while Section 3 introduces a notion of generic score and instantiates it through a concrete score. Section 4 describes  $\mathcal{S}3_k$ , we present experiments in Section 5, then discuss related works in Section 6 and conclude.

## 2. DATA MODEL

We now describe our model integrating social, structured, and semantic-rich content into a *single weighted RDF graph*, and based on a small set of  *$\mathcal{S}3$ -specific RDF classes and properties*. We

Constructor	Triple	Relational notation
Class assertion	$s \text{ type } o$	$o(s)$
Property assertion	$s \text{ p } o$	$p(s, o)$
Constructor	Triple	Relational notation
Subclass constraint	$s \prec_{sc} o$	$s \subseteq o$
Subproperty constraint	$s \prec_{sp} o$	$s \subseteq o$
Domain typing constraint	$s \leftrightarrow_d o$	$\Pi_{\text{domain}(s)} \subseteq o$
Range typing constraint	$s \leftrightarrow_r o$	$\Pi_{\text{range}(s)} \subseteq o$

**Figure 2: RDF (top) and RDFS (bottom) statements.**

present weighted RDF graphs in Section 2.1, and show how they model social networks in Section 2.2. We add to our model structured documents in Section 2.3, and tags and user-document interactions in Section 2.4; Section 2.5 introduces our notion of social paths. Table 1 recaps the main notations of our data model.

**URIs and literals** We assume given a set  $U$  of Uniform Resource Identifiers (URIs, in short), as defined by the standard [28], and a set of literals (constants) denoted  $L$ , disjoint from  $U$ .

**Keywords** We denote by  $\mathcal{K}$  the set of all possible *keywords*: it contains all the URIs, plus the stemmed version of all literals. For instance, stemming replaces “graduation” with “graduate”.

### 2.1 RDF

An *RDF graph* (or *graph*, in short) is a set of *triples* of the form  $s \text{ p } o$ , stating that the *subject*  $s$  has the *property*  $p$  and the value of that property is the *object*  $o$ . In relational notation (Figure 2),  $s \text{ p } o$  corresponds to the tuple  $(s, o)$  in the binary relation  $p$ , e.g.,  $u_1$  hasFriend  $u_0$  corresponds to hasFriend( $u_1, u_0$ ). We consider every triple is *well-formed* [27]: its subject belongs to  $U$ , its property belongs to  $U$ , and its object belongs to  $\mathcal{K}$ .

**Notations** We use  $s, p, o$  to denote a subject, property, and respectively, object in a triple. Strings between quotes as in “string” denote literals.

**RDF types and schema** The property type built in the RDF standard is used to specify to which *classes* a resource belongs. This can be seen as a form of resource typing.

A valuable feature of RDF is RDF Schema (RDFS), which allows enhancing the resource descriptions provided by RDF graphs. An RDF Schema declares *semantic constraints* between the classes and the properties used in these graphs, through the use of four RDF built-in properties. These constraints can model:

- subclass relationships, which we denote by  $\prec_{sc}$ ; for instance, any *M.S.Degree* is also a *Degree*;
- subproperty relationships, denoted  $\prec_{sp}$ ; for instance, *workingWith* someone also means being *acquaintedWith* him;
- typing of the first attribute (or domain) of a property, denoted  $\leftrightarrow_d$ , e.g., the domain of *hasDegreeFrom* is a *Graduate*;
- typing of the second attribute (or range) of a property, denoted  $\leftrightarrow_r$ , e.g., the range of *hasDegreeFrom* is an *University*.

Figure 2 shows the constraints we use, and how to express them. In this figure, domain and range denote respectively the first and second attributes of a property. The figure also shows the relational notation for these constraints, which in RDF are interpreted under the open-world assumption [1], i.e., as *deductive constraints*. For instance, if a graph includes the triples hasFriend  $\leftrightarrow_d$  Person and  $u_1$  hasFriend  $u_0$ , then the triple  $u_1$  type Person holds in this graph even if it is not explicitly present. This *implicit* triple is due to the  $\leftrightarrow_d$  constraint in Figure 2.

**Saturation RDF entailment** is the RDF reasoning mechanism that allows making explicit all the implicit triples that hold in an RDF graph  $G$ . It amounts to repeatedly applying a set of normative immediate *entailment* rules (denoted  $\vdash_{\text{RDF}}^i$ ) on  $G$ : given some triples

explicitly present in  $G$ , a rule adds some triples that directly follow from them. For instance, continuing the previous example,

$$u_1 \text{ hasFriend } u_0, \text{ hasFriend } \hookrightarrow_r \text{ Person } \vdash_{\text{RDF}}^i \\ u_0 \text{ type Person}$$

Applying immediate entailment  $\vdash_{\text{RDF}}^i$  repeatedly until no new triple can be derived is known to lead to a unique, finite fixpoint graph, known as the *saturation* (a.k.a. closure) of  $G$ . RDF entailment is part of the RDF standard itself: the answers to a query on  $G$  must take into account all triples in its saturation, since *the semantics of an RDF graph is its saturation* [27].

In the following, we assume, without loss of generality, that all RDF graphs are saturated; many saturation algorithms are known, including incremental [10] or massively parallel ones [26].

**Weighted RDF graph** Relationships between documents, document fragments, comments, users, keywords etc. naturally form a graph. We encode each edge from this graph by a *weighted RDF triple* of the form  $(s, p, o, w)$ , where  $(s, p, o)$  is a regular RDF triple, and  $w \in [0, 1]$  is termed the *weight* of the triple. Any triple whose weight is not specified is assumed to be of weight 1.

We define the saturation of a weighted RDF graph as the saturation derived *only from its triples whose weight is 1*. Any entailment rule of the form  $a, b \vdash_{\text{RDF}}^i c$  applies only if the weight of  $a$  and  $b$  is 1; in this case, the entailed triple  $c$  also has the weight 1. We restrict inference in this fashion to distinguish triples which certainly hold (such as: “a M.S. is a degree”, “ $u_1$  is a friend of  $u_0$ ”) from others whose weight is computed, and carries a more quantitative meaning, such as “the similarity between  $d_0$  and  $d_1$  is 0.5”<sup>1</sup>.

**Graph instance I and S3 namespace** We use  $I$  to designate the weighted RDF instance we work with. The RDF Schema statements in  $I$  allow a semantic interpretation of keywords, as follows:

**DEFINITION 2.1 (KEYWORD EXTENSION).** *Given an S3 instance  $I$  and a keyword  $k \in \mathcal{K}$ , the extension of  $k$ , denoted  $\text{Ext}(k)$ , is defined as follows:*

- $k \in \text{Ext}(k)$
- for any triple of the form  $b \text{ type } k, b \prec_{\text{sc}} k$  or  $b \prec_{\text{sp}} k$  in  $I$ , we have  $b \in \text{Ext}(k)$ .

For example, given the keyword *degree*, and assuming that  $\text{M.S.} \prec_{\text{sc}} \text{degree}$  holds in  $I$ , we have  $\text{M.S.} \in \text{Ext}(\text{degree})$ . The extension of  $k$  does not generalize it, in particular it does not introduce any loss of precision: whenever  $k'$  is in the extension of  $k$ , the RDF schema in  $I$  ensures that  $k'$  is an *instance*, or a *specialization* (particular case) of  $k$ . This is in coherence with the principles behind the RDF schema language<sup>2</sup>.

For our modeling purposes, we define below a small set of RDF classes and properties used in  $I$ ; these are shown prefixed with the S3 namespace. The next sections show how  $I$  is populated with triples derived from the users, documents and their interactions.

## 2.2 Social network

We consider a set of social network users  $\Omega \subset U$ , i.e., each user is identified by a URI. We introduce the special RDF class S3:user, and for each user  $u \in \Omega$ , we add:  $u \text{ type S3:user} \in I$ .

<sup>1</sup>One could generalize this to support inference over triples of any weight, leading to e.g., “ $u_1$  is of type Person with a weight of 0.5”, in the style of probabilistic databases.

<sup>2</sup>One could also allow a keyword  $k' \in \text{Ext}(k)$  which is only close to (but not a specialization of)  $k$ , e.g., “student” in  $\text{Ext}(\text{“graduate”})$ , at the cost of a loss of precision in query results. We do not pursue this alternative here, as we chose to follow standard RDF semantics.

To model the relationships between users, such as “friend”, “co-worker” etc., we introduce the special property S3:social, and model any concrete relationship between two users by a triple whose property specializes S3:social. Alternatively, one may see S3:social as the *generalization of all social network relationships*.

Weights are used to encode the strength  $w$  of each relationship going from a user  $u_1$  to a user  $u_2$ :  $u_1 \text{ S3:social } u_2 \ w \in I$ . As customary in social network data models, the higher the weight, the closer we consider the two users to be.

**Extensibility** Depending on the application, it may be desirable to consider that two users satisfying some condition are involved in a social interaction. For instance, if two people have worked the same year for a company of less than 10 employees (such information may be in the RDF part of our instance), they must have *worked together*, which could be a social relationship. This is easily achieved with a query that retrieves all such user pairs (in SPARQL or in a more elaborate language [9] if the condition also carries over the documents), and builds a  $u \text{ workedWith } u'$  triple for each such pair of users. Then it suffices to add these triples to the instance, together with the triple:  $\text{workedWith } \prec_{\text{sp}} \text{ S3:social}$ .

## 2.3 Documents and fragments

We consider that content is created under the form of structured, tree-shaped *documents*, e.g., XML, JSON, etc. A document is an unranked, ordered tree of *nodes*. Let  $N$  be a set of node names (for instance, the set of allowed XML element and attribute names, or the set of node names allowed in JSON). Any node has a *URI*. We denote by  $D \subset U$  the set of all node URIs. Further, each node has a *name* from  $N$ , and a *content*, which we view as a *set of keywords* from  $\mathcal{K}$ : we consider each text appearing in a document has been broken into words, stop words have been removed, and the remaining words have been stemmed to obtain our version of the node’s text content. For example, in Figure 1, the text of  $d_1$  might become {“M.S.”, “UAlberta”, “2012”}.

We term any subtree rooted at a node in document  $d$  a *fragment* of  $d$ , implicitly defined by the URI of its root node. The set of fragments (nodes) of a document  $d$  is denoted  $\text{Frag}(d)$ . We may use  $f$  to refer interchangeably to a fragment or its URI. If  $f$  is a fragment of  $d$ , we say  $d$  is an *ancestor* of  $f$ .

To simplify, we use *document and fragment interchangeably*; both are identified by the URI of their unique root node.

**Document-derived triples** We capture the *structural relationships* between documents, fragments and keywords through a set of RDF statements using S3-specific properties. We introduce the RDF class S3:doc corresponding to the documents, and we translate:

- each  $d \in D$  into the I triple  $d \text{ type S3:doc}$ ;
- each document  $d \in D$  and fragment rooted in a node  $n$  of  $d$  into  $n \text{ S3:partOf } d$ ;
- each node  $n$  and keyword  $k$  appearing in the content of  $n$  into  $n \text{ S3:contains } k$ ;
- each node  $n$  whose name is  $m$ , into  $n \text{ S3:nodeName } m$ .

**EXAMPLE 2.1.** *Based on the sample document shown in Figure 1, the following triples are part of I:*

$$d_{0.3.2} \text{ S3:partOf } d_{0.3} \quad d_1 \text{ S3:contains "M.S."} \\ d_{0.3} \text{ S3:partOf } d_0 \quad d_1 \text{ S3:nodeName text}$$

The following constraints, part of  $I$ , model the natural relationships between the S3:doc class and the properties introduced above:

$$\text{S3:partOf } \leftrightarrow_d \text{ S3:doc} \quad \text{S3:partOf } \hookrightarrow_r \text{ S3:doc} \\ \text{S3:contains } \leftrightarrow_d \text{ S3:doc} \quad \text{S3:nodeName } \leftrightarrow_d \text{ S3:doc}$$

which read: the relationship S3:partOf connects pairs of fragments (or documents); S3:contains describes the content of a fragment; and S3:nodeName associates names to fragments.

**Fragment position** We will need to assess how closely related a given fragment is to one of its ancestor fragments. For that, we use a function  $pos(d, f)$  which returns the *position* of fragment  $f$  within document  $d$ . Concretely,  $pos$  can be implemented for instance by assigning Dewey-style IDs to document nodes, as in [19, 22]. Then,  $pos(d, f)$  returns the list of integers  $(i_1, \dots, i_n)$  such that the path starting from  $d$ 's root, then moving to its  $i_1$ -th child, then to this node's  $i_2$ -th child etc. ends in the root of the fragment  $f$ . For instance, in Figure 1,  $pos(d_{0.3.2}, d_0)$  may be  $(3, 2)$ .

## 2.4 Relations between structure, semantics, users

We now show how dedicated S3 classes and properties are used to encode all kinds of connections between users, content, and semantics in a single S3 instance.

**Tags** A typical user action in a social setting is to *tag* a data item, reflecting the user's opinion that the item is related to some concept or keyword used in the tag. We introduce the special class S3:relatedTo to *account for the multiple ways in which a user may consider that a fragment is related to a keyword*. We denote by  $T$  the set of all tags.

For example, in Figure 1,  $u_4$  tags  $d_{0.5.1}$  with the keyword "university", leading to the triples:

a type S3:relatedTo      a S3:hasSubject  $d_{0.5.1}$   
a S3:hasKeyword "university"      a S3:hasAuthor  $u_4$

In this example,  $a$  is a *tag* (or annotation) resource, encapsulating the various tag properties: its content, who made it, and on what. The tag subject (the value of its S3:hasSubject property) is either a document or another tag. The latter allows to express *higher-level annotations*, when an annotation (tag) can itself be tagged.

A tag may lack a keyword, i.e., it may have no S3:hasKeyword property. Such no-keyword tags model *endorsement* (support), such as like on Facebook, retweet on Twitter, or +1 on Google+.

Tagging may differ significantly from one social setting to another. For instance, star-based rating of restaurants is a form of tagging, topic-based annotation of text by expert human users is another, and similarly a natural language processing (NLP) tool may tag a text snippet as being about some entity. Just like the S3:social property can be specialized to model arbitrary social connections between users, subclasses of S3:relatedTo can be used to model different kinds of tags. For instance, assuming  $a_2$  is a tag produced by a NLP software, this leads to the I triples:

$a_2$  type NLP:recognize  
NLP:recognize  $\prec_{sc}$  S3:relatedTo

**User actions on documents** Users *post* (or author, or publish) content, modeled by the dedicated property S3:postedBy. Some of this content may be *comments* on (or replies / answers to) other fragments; this is encoded via the property S3:commentsOn.

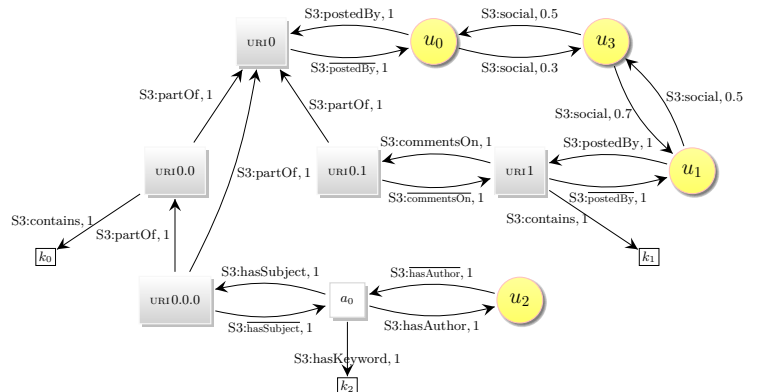
**EXAMPLE 2.2.** In Figure 1,  $d_2$  is posted by  $u_3$ , as a comment on  $d_{0.3.2}$ , leading to the following I triples:

$d_2$  S3:postedBy  $u_3$        $d_2$  S3:commentsOn  $d_{0.3.2}$

As before, we view any concrete relation between documents e.g., *answers to*, *retweets*, *comments on*, *is an old version of* etc. as a specialization (sub-property) of S3:commentsOn; the corresponding connections lead to implicit S3:commentsOn triples, as explained in Section 2.1. Similarly, forms of authorship connecting users to their content are modeled by specializing S3:postedBy.

Class	Semantics
S3:user	the users (the set of its instances is $\Omega$ )
S3:doc	the documents (the set of its instances is $D$ )
S3:relatedTo	generalization of item "tagging" with keywords (the set of all instances of this class is $T$ : the set of tags)
Property	Semantics
S3:postedBy	connects users to the documents they posted
S3:commentsOn	connects a comment with the document it is about
S3:partOf	connects a fragment to its parent nodes
S3:contains	connects a document with the keyword(s) it contains
S3:nodeName	asserts the name of the root node of document
S3:hasSubject	specifies the subject (document or tag) of a tag
S3:hasKeyword	specifies the keyword of a tag
S3:hasAuthor	specifies the poster of a tag
S3:social	generalization of social relationships in the network

**Table 2: Classes and properties in the S3 namespace.**



**Figure 3: Sample S3 instance I.**

This allows integrating (querying together) many social networks over partially overlapping sets of URIs, users and keywords.

**Inverse properties** As syntactic sugar, to simplify the traversal of connections between users and documents, we introduce a set of *inverse properties*, denoted respectively  $\overline{\text{S3:postedBy}}$ ,  $\overline{\text{S3:commentsOn}}$ ,  $\overline{\text{S3:hasSubject}}$  and  $\overline{\text{S3:hasAuthor}}$ , with the straightforward semantics:  $s \overline{p} o \in I$  iff  $o p s \in I$  where  $\overline{p}$  is the inverse property of  $p$ . For instance,  $u_0$  S3:friend  $u_1$  in Figure 1.

Table 2 summarises the above S3 classes and properties, while Figure 3 illustrates an I instance.

## 2.5 Social paths

We define here social paths on I, established either through explicit social links or through user interactions. We call **network edges** those I edges encapsulating quantitative information on the links between user, documents and tags, i.e., *edges whose properties are in the namespace S3 other than S3:partOf*, and whose subjects and objects are either users, documents, or tags. For instance, in Figure 3,  $u_1$  S3:social  $u_3$  0.5 and  $\text{URI0}$  S3:postedBy  $u_1$  are network edges;  $\text{URI0.0}$  S3:contains  $k_0$  and  $\text{URI0.1}$  S3:partOf  $\text{URI0}$  are not. The intuition behind the exclusion of S3:partOf is that *structural relations between fragments, or between fragments and keywords, merely describe data content and not an interaction*. However, if two users comment on the same fragment, or one comments on a fragment of a document posted by the other (e.g.,  $u_2$  and  $u_0$  in Figure 1), this is form of social interaction.

When two users interact with unrelated fragments of a same document, such as  $u_3$  and  $u_4$  on disjoint subtrees of  $d_0$  in Figure 1, this does not establish a social link between  $u_3$  and  $u_4$ , since they may not even have read the same text<sup>3</sup>. We introduce:

<sup>3</sup>To make such interactions count as social paths would only re-

**DEFINITION 2.2 (DOCUMENT VERTICAL NEIGHBORHOOD).** Two documents are vertical neighbors if one of them is a fragment of the other. The function  $\text{neigh}: U \rightarrow 2^U$  returns the set of vertical neighbors of an URI.

In Figure 3,  $\text{URI0}$  and  $\text{URI0.0.0}$  are vertical neighbors, so are  $\text{URI0}$  and  $\text{URI0.1}$ , but  $\text{URI0.0.0}$  and  $\text{URI0.1}$  are not. In the sequel, due to the strong connections between nodes in the same vertical neighborhood, we consider (when describing and exploiting social paths) that a path entering through any of them can exit through any other; a vertical neighborhood acts like a single node only and exactly from the perspective of a social path<sup>4</sup>. We can now define social paths:

**DEFINITION 2.3 (SOCIAL PATH).** A social path (or simply a path) in  $I$  is a chain of network edges such that the end of each edge and the beginning of the next one are either the same node, or vertical neighbors.

We may also designate a path simply by the list of nodes it traverses, when the edges taken along the path are clear. In Figure 3,  $u_2 \xrightarrow{u_2 \text{ S3:hasAuthor } a_0 \ 1} a_0 \xrightarrow{a_0 \text{ S3:hasSubject } \text{URI0.0.0} \ 1} \text{URI0.0.0} \dashrightarrow \text{URI0} \xrightarrow{\text{URI0} \text{ S3:postedBy } u_0 \ 1} u_0$  is an example of such a path (the dashed line:  $\text{URI0.0.0} \dashrightarrow \text{URI0}$ , is not an edge in the path but a connection between vertical neighbors,  $\text{URI0.0.0}$  been the end of an edge and  $\text{URI0}$  the beginning of the next edge). Also, in this Figure, there is no social path going from  $u_2$  to  $u_1$  avoiding  $u_0$ , because it is not possible to move from  $\text{URI0.1}$  to  $\text{URI0.0.0}$  through a vertical neighborhood.

**Social path notations** The set of all social paths from a node  $x$  (or one of its vertical neighbours) to a node  $y$  (or one of its vertical neighbors) is denoted  $x \rightsquigarrow y$ . The length of a path  $p$  is denoted  $|p|$ . The restriction of  $x \rightsquigarrow y$  to paths of length exactly  $n$  is denoted  $x \rightsquigarrow_n y$ , while  $x \rightsquigarrow_{\leq n} y$  holds the paths of at most  $n$  edges.

**Path normalization** To harmonize the weight of each edge in a path depending on its importance, we introduce path normalization, which modifies the weights of a path's edge as follows. Let  $n$  be the ending point of a social edge in a path, and  $e$  be the next edge in this path. The normalized weight of  $e$  for this path, denoted  $e.n_w$ , is defined as:

$$e.n_w = e.w / \sum_{e' \in \text{out}(\text{neigh}(n))} e'.w$$

where  $e.w$  is the weight of  $e$ , and  $\text{out}(\text{neigh}(n))$  the set of network edges outgoing from any vertical neighbor of  $n$ . This normalizes the weight of  $e$  w.r.t. the weight of edges outgoing from any vertical neighbor of  $n$ . Observe that  $e.n_w$  depends on  $n$ , however  $e$  does not necessarily start in  $n$ , but in any of its vertical neighbors. Therefore,  $e.n_w$  indeed depends on the path (which determines the vertical neighbor  $n$  of  $e$ 's entry point).

In the following, we assume all social paths are normalized.

**EXAMPLE 2.3.** In Figure 3, consider the path:

$$p = u_0 \xrightarrow{u_0 \text{ S3:postedBy } \text{URI0} \ 1} \text{URI0} \dashrightarrow \text{URI0.0.0} \xrightarrow{\text{URI0.0.0} \text{ S3:hasSubject } a_0 \ 1} a_0$$

Its first edge is normalized by the edges leaving  $u_0$ : one leading to  $\text{URI0}$  (weight 1) and the other leading to  $u_3$  (weight 0.3). Thus, its normalised weight is  $1/(1 + 0/3) = 0.77$ .

Its second edge exits  $\text{URI0.0.0}$  after a vertical neighborhood traversal  $\text{URI0} \dashrightarrow \text{URI0.0.0}$ . It is normalized by the edges leaving  $\text{neigh}(\text{URI0})$ , i.e., all the edges leaving a fragment of  $\text{URI0}$ . Its normalised weight is  $1/(1 + 1 + 1 + 1) = 0.25$ .

quire simple changes to the path normalization introduced below.  
<sup>4</sup>In other contexts, e.g., to determine their relevance w.r.t. a query, vertical neighbors are considered separately.

$S_3$  meets the requirements from Section 1, as follows. Genericity, extensibility and interoperability (**R6**) are guaranteed by the reliance on the Web standards RDF (Section 2.1) and XML/JSON (Section 2.3). These enable specializing the  $S_3$  classes and properties, e.g., through application-dependent queries (see Extensibility in Section 2.2). Our document model (Section 2.3) meets requirement **R2**; the usage of RDF (Section 2.1) ensures **R3**, while the relationships introduced in Section 2.4 satisfy **R1** as well as **R4** (higher-level tags). For what concerns **R5** (formal semantics), the data model has been described above; we consider queries next.

### 3. QUERYING AN $S_3$ INSTANCE

Users can search  $S_3$  instances through keyword queries; the answer consists of the  $k$  top-score fragments, according to a joint structural, social, and semantic score. Section 3.1, defines queries and their answers. After some preliminaries, we introduce a generic score, which can be instantiated in many ways, and a set of feasibility conditions on the score, which suffice to ensure the termination and correctness of our query answering algorithm (Section 3.3). We present our concrete score function in Section 3.4.

#### 3.1 Queries

$S_3$  instances are queried as follows:

**DEFINITION 3.1 (QUERY).** A query is a pair  $(u, \phi)$  where  $u$  is a user and  $\phi$  is a set of keywords.

We call  $u$  the seeker. We define the top- $k$  answers to a query as the  $k$  documents or fragments thereof with the highest scores, further satisfying the following constraint: the presence of a document or fragment at a given rank precludes the inclusion of its vertical neighbors at lower ranks in the results<sup>5</sup>. As customary, top- $k$  answers are ranked using a score function  $s(q, d)$  that returns for a document  $d$  and query  $q$  a value in  $\mathbb{R}$ , based on the graph  $I$ .

**DEFINITION 3.2 (QUERY ANSWER).** A top- $k$  answer to the query  $q$  using the score  $s$ , denoted  $T_{k,s}(q)$ , is recursively defined as a top- $k-1$  answer, plus a document with the best score among those which are neither fragments nor ancestors of the documents in the top- $k-1$  answer.

Observe that a query answer may not be unique. This happens as soon as several documents have equal scores for the query, and this score happens to be among the  $k$  highest.

#### 3.2 Connecting query keywords and documents

Answering queries over  $I$  requires finding best-scoring documents, based on the direct and indirect connections between documents, the seeker, and search keywords. The connection can be direct, for instance, when the document contains the keyword, or indirect, when a document is connected by a chain of relationships to a search keyword  $k$ , or to some keyword from  $k$ 's extension.

We denote the set of direct and indirect connections between a document  $d$  and a keyword  $k$  by  $\text{con}(d, k)$ . It is a set of three-tuples  $(\text{type}, \text{frag}, \text{src})$  such that:

- $\text{type} \in \{\text{S3:contains}, \text{S3:relatedTo}, \text{S3:commentsOn}\}$  is the **type** of the connection,
- $f \in \text{Frag}(d)$  is the **fragment** of  $d$  (possibly  $d$  itself) due to which  $d$  is involved in this connection,
- $\text{src} \in \Omega \cup D$  (users or documents) is the **source** (origin) of this connection (see below).

<sup>5</sup>This assumption is standard in XML keyword search, e.g., [6].

Below we describe the possible situations which create connections. Let  $d, d'$  be documents or tags, and  $f, f'$  be fragments of  $d$  and  $d'$ , respectively<sup>6</sup>. Further, let  $k, k'$  be keywords such that  $k' \in \text{Ext}(k)$ , and  $\text{src} \in \Omega \cup D$  be a user or a document.

**Documents connected to the keywords of their fragments** If the fragment  $f$  contains a keyword  $k$ , then:

$$(S3:\text{contains}, f, d) \in \text{con}(d, k)$$

which reads: “ $d$  is connected to  $k$  through a  $S3:\text{contains}$  relationship due to  $f$ ”. This connection holds even if  $f$  contains not  $k$  itself, but some  $k' \in \text{Ext}(k)$ . For example, in Figure 1, if the keyword “university” appears in the fragment whose URI is  $d_{2.7.5}$ , then  $\text{con}(d_2, \text{“university”})$  includes  $(S3:\text{contains}, d_{2.7.5}, d_2)$ . Observe that a given  $k'$  and  $f$  may lead to many connections, if  $k'$  specializes several keywords and/or if  $f$  has many ancestors.

**Connections due to tags** For every tag  $a$  of the form

$$\begin{array}{ll} \text{a type } S3:\text{relatedTo} & \text{a } S3:\text{hasSubject } f \\ \text{a } S3:\text{hasAuthor } \text{src} & \text{a } S3:\text{hasKeyword } k' \end{array}$$

$\text{con}(d, k)$  includes  $(S3:\text{relatedTo}, f, \text{src})$ . In other words, whenever a fragment  $f$  of  $d$  is tagged by a source  $\text{src}$  with a specialization of the keyword  $k$ , this leads to a  $S3:\text{relatedTo}$  connection between  $d$  and  $k$  due to  $f$ , whose source is the tag author  $\text{src}$ . For instance, the tag  $a$  of  $u_4$  in Figure 1 creates the connection  $(S3:\text{relatedTo}, d_{0.5.1}, u_4)$  between  $d_0$  and “university”.

More generally, if a tag  $a$  on fragment  $f$  has any type of connection (not just  $S3:\text{hasKeyword}$ ) to a keyword  $k$  due to source  $\text{src}$ , this leads to a connection  $(S3:\text{relatedTo}, f, \text{src})$  between  $d$  and  $k$ . The intuition is that the tag adds its connections to the tagged fragment and, transitively, to its ancestors. (As the next section shows, the importance given to such connections decreases as the distance between  $d$  and  $f$  increases.)

If the tag  $a$  on  $f$  is a simple endorsement (it has no keyword), the tag inherits  $d$ 's connections, as follows. Assume  $d$  has a connection of type  $\text{type}$  to a keyword  $k$ : then,  $a$  also has a  $\text{type}$  connection to  $k$ , whose source is  $\text{src}$ , the tag author. The intuition is that when  $\text{src}$  endorses (`likes`, `+1s`) a fragment,  $\text{src}$  agrees with its content, and thus connects the tag, to the keywords related to that fragment and its ancestors. For example, if a user  $u_5$  endorsed  $d_0$  in Figure 1 through a no-keyword tag  $a_5$ , the latter tag is related to “university” through:  $(S3:\text{relatedTo}, d_{0.5.1}, u_5)$ .

**Connections due to comments** When a comment on  $f$  is connected to a keyword, this also connects any ancestor  $d$  of  $f$  to that keyword; the connection source carries over, while the type of  $d$ 's connection is  $S3:\text{commentsOn}$ . For instance, in Figure 1, since  $d_2$  is connected to “university” through  $(S3:\text{contains}, d_{2.7.5}, d_2)$  and since  $d_2$  is a comment on  $d_{0.3.2}$ , it follows that  $d_0$  is also related to “university” through  $(S3:\text{commentsOn}, d_{0.3.2}, d_2)$ .

### 3.3 Generic score model

We introduce a set of proximity notions, based on which we state the conditions to be met by a score function, for our query evaluation algorithm to compute a top-k query answer.

**Path proximity** We consider a measure of proximity *along one path*, denoted  $\overrightarrow{\text{prox}}$ , between 0 and 1 for any path, such that:

- $\overrightarrow{\text{prox}}(()) = 1$ , i.e., the proximity is maximal on an empty path (in other words, from a node to itself),
- for any two paths  $p_1$  and  $p_2$ , such that the start point of  $p_2$  is in the vertical neighborhood of the end point of  $p_1$ :

$$\overrightarrow{\text{prox}}(p_1 || p_2) \leq \min(\overrightarrow{\text{prox}}(p_1), \overrightarrow{\text{prox}}(p_2)),$$

<sup>6</sup>We here slightly extend notations, since tags do not have fragments: if  $d$  is a tag, we consider that its only fragment is  $d$ .

where  $||$  denotes path concatenation. This follows the intuition that proximity along a concatenation of two paths is at most the one along each of these two components paths: proximity can only decrease as the path gets longer.

**Social proximity** associates to two vertices connected by at least one social path, a comprehensive measure over *all the paths* between them. We introduce such a global proximity notion, because different paths traverse different nodes, users, documents and relationships, all of which may impact the relation between the two vertices. Considering all the paths gives a *qualitative* advantage to our algorithm, since it enlarges its knowledge to the types and strength of all connections between two nodes.

**DEFINITION 3.3 (SOCIAL PROXIMITY).** *The social proximity measure  $\text{prox} : (\Omega \cup D \cup T)^2 \rightarrow [0, 1]$ , is an aggregation along all possible paths between two users, documents or tags, as follows:*

$$\text{prox}(a, b) = \oplus_{\text{path}} (\{(\overrightarrow{\text{prox}}(p), |p|), p \in a \rightsquigarrow b\}),$$

where  $| \cdot |$  is the number of vertices in a path and  $\oplus_{\text{path}}$  is a function aggregating a set of values from  $[0, 1] \times \mathbb{N}$  into a single scalar value.

Observe that the set of all paths between two nodes may be infinite, if the graph has cycles; this is often the case in social graphs. For instance, in Figure 3, a cycle can be closed between  $(u_0, \text{URI0}, u_0)$ . Thus, in theory, the score is computed over a potentially infinite set of paths. However, in practice, our algorithm works with *bounded social proximity* values, relying only on paths of a bounded length:

$$\text{prox}^{\leq n}(a, b) = \oplus_{\text{path}} (\{(\overrightarrow{\text{prox}}(p), |p|), p \in a \rightsquigarrow_{\leq n} b\})$$

Based on the proximity measure, and the connections between keywords and documents introduced in Section 3.2, we define:

**DEFINITION 3.4 (GENERIC SCORE).** *Given a document  $d$  and a query  $q = (u, \phi)$ , the score of  $d$  for  $q$  is:*

$$\text{score}(d, (u, \phi)) = \oplus_{\text{gen}} (\{(k, \text{type}, \text{pos}(d, f), \text{prox}(u, \text{src})) \mid k \in \phi, (\text{type}, f, \text{src}) \in \text{con}(d, k)\})$$

where  $\oplus_{\text{gen}}$  is a function aggregating a set of (keyword, relationship type, importance of fragment  $f$  in  $d$ , social proximity) tuples into a value from  $[0, 1]$ .

Importantly, the above score *reflects the semantics, structure, and social content of the S3 instance*, as follows.

First,  $\oplus_{\text{gen}}$  aggregates over the keywords in  $\phi$ . Recall that tuples from  $\text{con}(d, k)$  account not only for  $k$  but also for keywords  $k' \in \text{Ext}(k)$ . This is how *semantics* is injected into the score.

Second, the score of  $d$  takes into account the relationships between fragments  $f$  of  $d$ , and keywords  $k$ , or  $k' \in \text{Ext}(k)$ , by using the sequence  $\text{pos}(d, f)$  (Section 2.3) as an indication of the structural importance of the fragment within the document. If the sequence is short, the fragment is likely a large part of the document. Document *structure* is therefore taken into account here both *directly* through  $\text{pos}$ , and *indirectly*, since the  $\text{con}$  tuples also propagate relationships from fragments to their ancestors (Section 3.2).

Third, the score takes into account the *social* component of the graph through  $\text{prox}$ : this accounts for the relationships between the seeker  $u$ , and the various parties (users, documents and tags), denoted  $\text{src}$ , due to which  $f$  may be relevant for  $k$ .

**Feasibility properties** For our query answering algorithm to converge, the generic score model must have some properties which we describe below.

**1. Relationship with path proximity** This refers to the relationship between path proximity and score. First, the score should only

increase if one adds *more paths* between a seeker and a data item. Second, the contribution of the paths of length  $n \in \mathbb{N}$  to the social proximity can be expressed using the contributions of shorter “pre-fixes” of these paths, as follows. We denote by  $ppSet^n(a, b)$  the set of the path proximity values for all paths from  $a$  to  $b$  of length  $n$ :

$$ppSet^n(a, b) = \{\overrightarrow{prox}(p) \mid p \in a \rightsquigarrow_n b\}$$

Then, the first property is that there exists a function  $U_{prox}$  with values in  $[0, 1]$ , taking as input (i) the bounded social proximity for path of length at most  $n - 1$ , (ii) the proximity along paths of length  $n$ , and (iii) the length  $n$ , and such that:

$$prox^{\leq n}(a, b) = prox^{\leq n-1}(a, b) + U_{prox}(prox^{\leq n-1}(a, b), ppSet^n(a, b), n)$$

**2. Long paths attenuation** The influence of social paths should decrease as they get longer; intuitively, the farther away two items are, the weaker their connection and thus their influence on the score. More precisely, there exists a bound  $B_{prox}^>n$  tending to 0 as  $n$  grows, and such that:

$$B_{prox}^>n \geq prox - prox^{\leq n}$$

**3. Score soundness** The score of a document should be positively correlated with the social proximity from the seeker to the document fragments that are relevant for the query.

Denoting  $score_{[g]}$  the score where the proximity function  $prox$  is replaced by a continuous function  $g$  having the same domain  $(\Omega \cup D \cup T)^2$ ,  $g \mapsto score_{[g]}$  must be monotonically increasing and continuous for the uniform norm.

#### 4. Score convergence

This property bounds the score of a document and shows how it relates to the social proximity. It requires the existence of a function  $B_{score}$  which takes a query  $q = (u, \phi)$  and a number  $B \geq 0$ , known to be an upper bound on the social proximity between the seeker and any source: for any  $d$ , query keyword  $k$ , and  $(type, f, src) \in con(d, k)$ , we know that  $prox(u, src) \leq B$ .  $B_{score}$  must be positive, and satisfy, for any  $q$ :

- for any document  $d$ ,  $score(d, q) \leq B_{score}(q, B)$ ;
- $\lim_{B \rightarrow 0} (B_{score}(q, B)) = 0$  (tends to 0 like  $B$ ).

We describe a concrete *feasible score*, i.e., having the above properties, in the next section.

### 3.4 Concrete score

We start by instantiating  $\overrightarrow{prox}$ ,  $prox$  and  $score$ .

**Social proximity** Given a path  $p$ , we define  $\overrightarrow{prox}(p)$  as the product of the normalized weights (recall Section 2.5) found along the edges of  $p$ . We define our concrete social proximity function  $prox(a, b)$  as a weighted sum over all paths from  $a$  to  $b$ :

$$prox(a, b) = C_\gamma \times \sum_{p \in a \rightsquigarrow b} \frac{\overrightarrow{prox}(p)}{\gamma^{|p|}}$$

where  $\gamma > 1$  is a scalar coefficient, and  $C_\gamma = \frac{\gamma-1}{\gamma}$  is introduced to ensure that  $prox \leq 1$ . Recall that by Definition 3.3,  $prox$  requires a  $\oplus_{path}$  aggregation over the (social proximity, length) pairs of the paths between the two nodes. Hence, this concrete social proximity corresponds to choosing:

$$\oplus_{path}(S) = C_\gamma \times \sum_{(sp, len) \in S} \frac{sp}{\gamma^{len}}$$

where  $(sp, len)$  is a (social proximity, length) pair from its input.

**EXAMPLE 3.1. Social proximity** Let us consider in Figure 3 the social proximity from  $u_0$  to  $URI0$ , using the  $\overrightarrow{prox}$  and  $\oplus_{path}$  previously introduced. An edge connects  $u_0$  directly to  $URI0$ , leading to the normalized path  $p$ :

$$p = u_0 \xrightarrow{u_0 \text{ S3:postedBy } URI0 \frac{1}{1+0.3}} URI0$$

which accounts for a partial social proximity:

$$prox^{\leq 1}(u_0, URI0) = \frac{\overrightarrow{prox}(p)}{\gamma^{|p|}} = \frac{1/(1+0.3)}{\gamma^1}$$

This social proximity generalizes Katz distance [17]; other common distances may be used, e.g., SimRank [14].

**Score function** We define a simple concrete  $S3$  score function which, for a document  $d$ , is the product of the scores of each query keyword in  $d$ . The score of a keyword is summed over all the connections between the keyword and the document. The weight for a given connection and keyword only depends on the social distance between the seeker and the sources of the keyword, and the structural distance between the fragment involved in this relation and  $d$ , namely the length of  $pos(d, f)$ . Both distances decrease exponentially as the path length grows. Formally:

**DEFINITION 3.5 (S3<sub>k</sub> SCORE).** Given a query  $(u, \phi)$ , the  $S3_k$  score of a document  $d$  for the query is defined as:

$$score(d, (u, \phi)) = \prod_{k \in \phi} \left( \sum_{(type, f, src) \in con(d, k)} \eta^{|pos(d, f)|} \times prox(u, src) \right)$$

for some damping factor  $\eta < 1$ .

Recall from Definition 3.4 that an aggregation function  $\oplus_{gen}$  combines the contributions of (keyword, relationship type, importance, social proximity) tuples in the score. The above definition corresponds to the following  $\oplus_{gen}$  aggregator:

$$\oplus_{gen}(S) = \prod_{k \in \phi} \left( \sum_{\substack{rel, prox \\ \exists type, (k, type, rel, prox) \in S}} \eta^{|rel|} \times prox \right)$$

Note that if we ignore the social aspects and restrict ourselves to top- $k$  search on documents (which amounts to  $prox = 1$ ),  $\oplus_{gen}$  gives the best score to the lowest common ancestor (LCA) of the nodes containing the query keywords. Thus, our score extends typical XML IR works, e.g., [6] (see also Section 6).

Obviously, there are many possible ways to define  $\oplus_{gen}$  and  $\oplus_{path}$ , depending on the application. In particular, different types of connections may not be accounted for equally; our algorithm only requires a *feasible score* (with the feasibility properties).

**THEOREM 3.1 (SCORE FEASIBILITY).** The  $S3_k$  score function (Definition 3.5) has the feasibility properties (Section 3.3).

The proof appears in our technical report [3].

## 4. QUERY ANSWERING ALGORITHM

In this section, we describe our *Top-k* algorithm called  $S3_k$ , which computes the answer to a query over an  $S3$  instance using our  $S3_k$  score, and formally state its correctness.

### 4.1 Algorithm

The main idea, outlined in Algorithm 1, is the following. The instance is explored starting from the seeker and going to other vertices (users, documents, or resources) at increasing distance. At the  $n$ -th iteration, the  $I$  vertices explored are those connected to the seeker by at least a path of length at most  $n$ . We term *exploration border* the set of graph nodes reachable by the seeker through a path of length exactly  $n$ . Clearly, the border changes as  $n$  grows.

During the exploration, documents are collected in a set of *candidate* answers. For each candidate  $c$ , we maintain a score interval: its *currently known lowest possible score*, denoted  $c.lower$ , and its

**Algorithm 1:**  $S3_k$  – Top- $k$  algorithm.

---

**Input** : a query  $q = (u, \phi)$   
**Output**: the best  $k$  answers to  $q$  over an  $S3$  instance  $I, T_{k,s}(q)$

```

1  $candidates \leftarrow []$  // initially empty list
2  $discarded \leftarrow \emptyset$ 
3  $borderPath \leftarrow []$ 

4  $allProx \leftarrow \delta_u$  //  $\delta_u[v] = \begin{cases} 1 & \text{if } v = u \\ 0 & \text{otherwise} \end{cases}$ 
5  $threshold \leftarrow \infty$  // Best possible score of a document not yet explored, updated in ComputeCandidatesBounds
6  $n \leftarrow 0$ 
7 while not StopCondition( $candidates$ ) do
8    $n \leftarrow n + 1$ 
9   ExploreStep()
10  ComputeCandidatesBounds()
11  CleanCandidatesList()
12 return  $candidates[0, k - 1]$ 

```

---

$q = (u, \phi)$	Query: seeker $u$ and keyword set $\phi$
$k$	Result size
$n$	Number of iterations of the main loop of the algorithm
$candidates$	Set of documents and/or fragments which are candidate query answers at a given moment
$discarded$	Set of documents and/or fragments which have been ruled out of the query answer
$borderPath[v]$	Paths from $u$ to $v$ explored at the last iteration ( $a \rightsquigarrow_n v$ )
$allProx[v]$	Bounded social proximity ( $prox^{\leq n}$ ) between the seeker $u$ and a node $v$ , taking into account all the paths from $u$ to $v$ known so far
$connect[c]$	Connections between the seeker and the candidate $c$ : $connect[c] = \{(k, type, pos(d, f), src)   k \in \phi, (type, f, src) \in con(c, k)\}$
$threshold$	Upper bound on the score of the documents not visited yet

**Table 3: Main variables used in our algorithms.****Algorithm 2:** Algorithm StopCondition

---

**Input** :  $candidates$  set  
**Output**: true if  $candidates[0, k - 1]$  is  $T_{k,s}(q)$ , false otherwise

```

1 if  $\exists d, d' \in candidates[0, \dots, k - 1], d \in neigh(d')$  then
2   return false
3  $min\_topk\_lower \leftarrow \infty$ 
4 foreach  $c \in candidates[0, \dots, k - 1]$  do
5    $min\_topk\_lower \leftarrow \min(min\_topk\_lower, c.lower)$ 
6  $max\_non\_topk\_upper \leftarrow candidates[k].upper$ 
7 return  $\max(max\_non\_topk\_upper, threshold) \leq min\_topk\_lower$  // Boolean result

```

---

highest possible score, denoted  $c.upper$ . These scores are updated as new paths between the seeker and the candidates are found. Candidates are kept sorted by their highest possible score; the  $k$  first are the answer to the query when the algorithm stops, i.e., when no candidate document outside the current first  $k$  can have an upper bound above the minimum lower bound within the top  $k$  ranks.

Further, the search algorithm relies on three tables:

- $borderPath$  is a table storing, for a node  $v$  in  $I$ , the set of paths of length  $n$  between  $u$  (the seeker) and  $v$ , where  $n$  is the current distance from  $u$  that the algorithm has traversed.
- $allProx$  is a table storing, for a node  $v$  in  $I$ , the proximity between  $u$  and  $v$  taking into account all the paths known so far from  $u$  to  $v$ . Initially, its value is 0 for any  $v \neq u$ .
- $connect$  is a table storing for a candidate  $c$  the set of connections (Section 3.2) discovered so far between the seeker and  $c$ .

These tables are updated during the search. While they are defined on all the  $I$  nodes, we only compute them gradually, for the nodes on the exploration border.

**Termination condition** Of course, search should not explore the whole graph, but instead stop as early as possible, while returning

**Algorithm 3:** Algorithm ExploreStep

---

**Update:**  $borderPath$  and  $allProx$

```

1 if  $n = 1$  then
2    $borderPath \leftarrow out(\{u\})$ 
3 else
4   foreach  $v \in I$  do
5      $newBorderPath[v] \leftarrow \emptyset$ 
6   foreach  $p \in borderPath$  do
7     foreach network edge  $e$  in  $out(neigh(p.end))$  do
8        $m \leftarrow e.target$ 
9       if  $m$  is a document or a tag then
10        GetDocuments( $m$ )
11         $newBorderPath[m].add(p|e)$ 
12    $borderPath \leftarrow newBorderPath$ 
13 foreach  $v \in I$  do
14    $newAllProx[v] \leftarrow allProx[v] + U_{prox}(allProx[v],$ 
15      $\{prox^{\leq n}(p), p \in borderPath[v]\}, n)$ 
16  $allProx \leftarrow newAllProx$ 

```

---

the correct result. To this aim, we maintain during the search an upper bound on the score of score of all documents unexplored so far, named  $threshold$ . Observe that we do not need to return the exact score of our results, and indeed we may never narrow down the (lower bound, upper bound) intervals to single numbers; we just need to make sure that no document unexplored so far is in among the top  $k$ . Algorithm 2 outlines the procedure to decide whether the search is complete: when (i) the candidate set does not contain documents such that one is a fragment of another, and (ii) no document can have a better score than the current top  $k$ .

**Any-time termination** Alternatively, the algorithm can be stopped at any time (e.g., after exhausting a time budget) by making it return the  $k$  best candidates based on their current upper bound score.

**Graph exploration** Algorithm 3 describes one search step (iteration), which visits nodes at a social distance  $n$  from the seeker. For the ones that are documents or tags, the GetDocuments algorithm (see hereafter) looks for related documents that can also be candidate answers (these are added to  $candidates$ );  $discarded$  keeps track of related documents with scores too low for them to be candidates. The  $allProx$  table is also updated using the  $U_{prox}$  function, whose existence follows from the first score feasibility property (Section 3.3), to reflect the knowledge acquired from the new exploration border ( $borderPath$ ). Observe that Algorithm 3 computes  $prox^{\leq n}(u, src)$  iteratively using the first feasibility property; at iteration  $n$ ,  $allProx[src] = prox^{\leq n}(u, src)$ .

**Computing candidate bounds** The ComputeCandidateBounds algorithm (shown in [3]) maintains during the search the lower and upper bounds of the  $candidates$ , as well as  $threshold$ . A candidate's lower bound is computed as its score where its social proximity to the user<sup>7</sup> is approximated by its bounded version, based only on the paths explored so far:

$$\oplus_{gen}(\{(kw, type, pos(d, f), allProx[src]) | kw \in \phi, (type, f, src) \in con(d, kw)\})$$

This is a lower bound because, during exploration, a candidate can only get closer to the seeker (as more paths are discovered).

A candidate's upper bound is computed as its score, where the social proximity to the user is replaced by the sum between the bounded proximity and the function  $B_{prox}^{>n}(u, src)$ , whose existence follows from the long path attenuation property (Section 3.3).

<sup>7</sup>The actual (exact) social proximity requires a complete traversal of the graph; our algorithms work with approximations thereof.



The latter is guaranteed to offset the difference between the bounded and actual social proximity:

$$\oplus_{gen}(\{(kw, type, pos(d, f), allProx[src] + B_{prox}^{>n}(u, src)) \mid kw \in \phi, (type, f, src) \in con(d, kw)\})$$

The above bounds rely on  $con(d, k)$ , the set of all connections between a candidate  $d$  and a query keyword  $k$  (Section 3.2); clearly, the set is not completely known when the search starts. Rather, connections accumulate gradually in the *connect* table (Algorithm `GetDocuments`), whose tuples are used as approximate (partial)  $con(d, k)$  information in `ComputeCandidateBounds`.

Finally, `ComputeCandidateBounds` updates the relevance threshold using the known bounds on *score* and *prox*. The new bound estimates the best possible score of the unexplored documents.

**Cleaning the candidate set** Algorithm `CleanCandidateList` removes from *candidates* documents that cannot be in the answer, i.e., those for which  $k$  candidates with better scores are sure to exist, as well as those having a candidate neighbor with a better score. The algorithm is delegated to [3].

**Getting candidate documents** Given a candidate document or tag  $x$ , Algorithm `GetDocuments` checks whether some documents unexplored so far, reachable from  $x$  through a chain of `S3:partOf`, `S3:commentsOn`, `S3:commentsOn`, `S3:hasSubject`, or `S3:hasSubject` edges, are candidate answers. If yes, they are added to *candidates* and the information necessary to estimate their score is recorded in *connect*. The algorithm is detailed in [3].

## 4.2 Correctness of the algorithm

The theorems below state the correctness of our algorithm for any score function with the feasibility properties identified in Section 3.3. The proofs are quite involved, and they are delegated to [3]. The core of the proofs is showing how the score feasibility properties entail a set of useful properties, in particular related to early termination (convergence).

**THEOREM 4.1 (STOP CORRECTNESS).** *When a stop condition is met, the first  $k$  elements in candidates are a query answer.*

We say the tie of two equal-score documents  $d, d'$  is *breakable* if examining a set of paths of bounded length suffices to decide their scores are equal. (In terms of our score feasibility properties, this amounts to  $B_{prox}^{>n} = 0$  for some  $n$ .) Our generic score function (Definition 3.5) does not guarantee all ties are breakable. However, any finite-precision number representation eventually brings the lower and upper bounds on  $d$  and  $d'$ 's scores too close to be distinguished, de facto breaking ties.

**THEOREM 4.2 (CORRECTNESS WITH BREAKABLE TIES).** *If there exists a query answer of size  $k$  and all ties are breakable then Algorithm 1 returns a query answer of size  $k$ .*

**THEOREM 4.3 (ANYTIME CORRECTNESS).** *Using anytime termination, Algorithm 1 eventually returns a query answer.*

In our experiments (Section 5), the threshold-based termination condition was always met, thus we never needed to wait for convergence of the lower and upper bound scores.

## 5. IMPLEMENTATION & EXPERIMENTS

We describe experiments creating and querying S3 instances. We present the datasets in Section 5.1, while Section 5.2 outlines our implementation and some optimizations we brought to the search algorithm. We report query processing times in Section 5.3, study the quality of our returned results in Section 5.4, then we conclude.

I<sub>1</sub> (Twitter)

Users	492,244
S3:social edges	17 544 347
Documents	467,710
Fragments (non-root)	1,273,800
Tags	609,476
Keywords	28,126,940
Tweets	999,370
Retweets	85%
Reply to users' status	6.9%
String-keyword associations extracted from DBpedia	3,301,425
S3:social edges per user having any (average)	317
Nodes (without keywords)	2 972 560
Edges (without keywords)	24 554 029

I<sub>2</sub> (Vodkaster)

Users	5,328
S3:social edges (vdk:follow)	94,155
Documents (movie comments)	330,520
Fragments (non-root)	529,432
Keywords	3,838,662
Movies	20,022

I<sub>3</sub> (Yelp)

Users	366,715
S3:social edges (yelp:friend)	3,868,771
Documents (reviews)	2,064,371
Keywords	59,614,201
Businesses	61,184

Figure 4: Statistics on our instances.

## 5.1 Datasets, queries, and systems

**Datasets** We built three datasets, I<sub>1</sub>, I<sub>2</sub>, and I<sub>3</sub>, based respectively on content from Twitter, Vodkaster and Yelp.

The instance I<sub>1</sub> was constructed starting from tweets obtained through the public streaming Twitter API. Over a one-day interval (from May 2nd 2014 16h44 GMT to May 3rd 2014 12h44 GMT), we gathered roughly one million tweets. From every tweet that is not a retweet, we created a document having three nodes (*i*) a *text* node: from the *text* field of the tweet, we extracted named entities and words (using the Twitter NLP tools library [20]) and matched them against a general-purpose ontology we created from DBpedia (see below); (*ii*) a *date* node, and (*iii*) a *geo* node: if the tweet included a human readable location, we inserted it in this node. The RDF graph of our instance was built from DBpedia datasets, namely: *Mapping-based Types*, *Mapping-based Properties*, *Persondata* and *Lexicalizations Dataset*. These were chosen as they were the most likely to contain concepts (names, entities etc.) occurring in tweets. Tweet text was *semantically enriched* (connected to the RDF graph) as follows: within the *text* fields, we replaced each word  $w$  for which a triple of the form  $u \text{ foaf:name } w$  holds in the DBpedia knowledge base, by the respective URI  $u$ .

When a tweet  $t'$  authored by user  $u$  is a *retweet* of another tweet  $t$ , for each hashtag  $h$  introduced by  $t'$ , we added to I<sub>1</sub> the triples: a type `S3:relatedTo`, a `S3:hasSubject`  $t$ , a `S3:hasKeyword`  $h$  and a `S3:hasAuthor`  $u$ . If a tweet  $t''$  was a *reply* to another tweet  $t$ , we considered  $t''$  a comment on  $t$ . Whenever  $t$  was present in our dataset<sup>8</sup>, we added the corresponding `S3:commentsOn` triple in I<sub>1</sub>. The set of users  $\Omega_{I_1}$  corresponds to the set of user IDs having posted tweets, and we created links between users as follows. We assigned to every pair of users  $(a, b)$  a value  $u_{\sim}(a, b) = t \cdot js_1(a, b) + (1 - t) \cdot js_2(a, b)$ , where  $js_1, js_2$  give the Jaccard similarities of the sets of keywords appearing in each user's posts, respectively, in each user's comments. Whenever this similarity was above a threshold, we created an edge of weight  $u_{\sim}$  between the two users. Through experiments on this dataset, we set the threshold to 0.1.

The instance I<sub>2</sub> uses data from Vodkaster, a French social network dedicated to movies. The data comprises *follower* relations between the users and a list of comments on the movies, in French, along with their author. Whenever user  $u$  follows user  $v$  we in-

<sup>8</sup>The corpus may contain a re-tweet of a tweet we do not capture; this is unavoidable unless one has access to the full Twitter history.

cluded  $u$  `vdk:follow`  $v$  in  $I_2$ , where `vdk:follow` is a custom sub-property of  $S3$ :`social`. We translate the first comment on each film into a document; each additional comment was then considered a comment on the first. The text of each comment was stemmed, then each stemmed sentence was made a fragment of the comment.

The instance  $I_3$  is based on Yelp [29], a crowd-sourced reviews website about businesses. This dataset contains a list of textual reviews of businesses, and the friend list of each user. As for  $I_2$ , we considered that the first review of a business is commented on by the subsequent reviews of the same business. For each user  $u$  friend with user  $v$ , we added `u yelp:friend v` to  $I_3$ , where `yelp:friend` is a  $S3$ :`social` subproperty modeling social Yelp connections. Reviews were also semantically enriched using DBpedia.

Table 4 shows the main features of the three quite different data instances.  $I_1$  is by far the largest.  $I_2$  was not matched with a knowledge base since its content is in French;  $I_2$  and  $I_3$  have no tags.

**Queries** For each instance we created workloads of 100 queries, based on three independent parameters:

- $f$ , the keyword frequency: either *rare*, denoted ‘-’ (among the 25% least frequent in the document set), or *common*, denoted ‘+’ (among the 25% most frequent)
- $l$ , the number of keywords in the query: 1 or 5
- $k$ , the expected number of results: 5 or 10

This lead to a total of 8 workloads, identified by  $qset_{f,l,k}$ , for each dataset. To further analyze the impact of varying  $k$ , we added 10 more workloads for  $I_1$ , where  $f \in \{+, -\}$ ,  $l = 1$ , and  $k \in [1, 5, 10, 50]$  (used in Figure 7). We stress here that injecting semantics in our workload queries, by means of keyword extensions (Definition 2.1), increased their size on average by 50%.

**Systems** Our algorithms were fully implemented in Python 2.7; the code has about 6K lines. We stored some data tables in PostgreSQL 9.3, while others were built in memory, as we explain shortly. All our experiments were performed on a 4 cores Intel Xeon E3-1230 V2 @3.30GHz with 16Go of RAM, running Debian 8.1.

No existing system directly compares with ours, as we are the first to consider fine-granularity content search with semantics in a social network. To get at least a rough performance comparison, we used the Java-based code provided by the authors of the top- $k$  social search system described in [18], based on the UIT (user, item tag) model, and referred to as **TopkS** from now on. The data model of TopkS is rather basic, since its documents (*items*) have no internal structure nor semantics and tags have no semantic connection between them. Further, (*user, user, weight*) tuples reflect weighted links between users. TopkS computes a social score and a content-based score for each item; the overall item score is then  $\alpha \times$  social score  $+(1 - \alpha) \times$  content score, where  $\alpha$  is a parameter of TopkS.

We adapted our instances into TopkS’s simpler data model. From  $I_1$ , we created  $I'_1$  as follows: (i) the relations between users were kept with their weight; (ii) every tweet was merged with all its retweets and replies into a single item, and (iii) every keyword  $k$  in the content of a tweet that is represented by item  $i$  posted by user  $u$  led to introducing the (user, item, tag) triple  $(u, i, k)$ . To obtain  $I'_2$  and  $I'_3$ , each movie or business becomes an item, each word extracted from a review leads to a (user, item, tag) tuple.

## 5.2 Implementation and optimizations

We briefly discuss our implementation, focusing on optimizations w.r.t. the conceptual description in Section 4.

The first optimization concerns the computation of *prox*, required for the score (Definition 3.5). While the score involves connections between documents and keywords found on any path, in practice  $S3_k$  explores paths (and nodes) increasingly far from

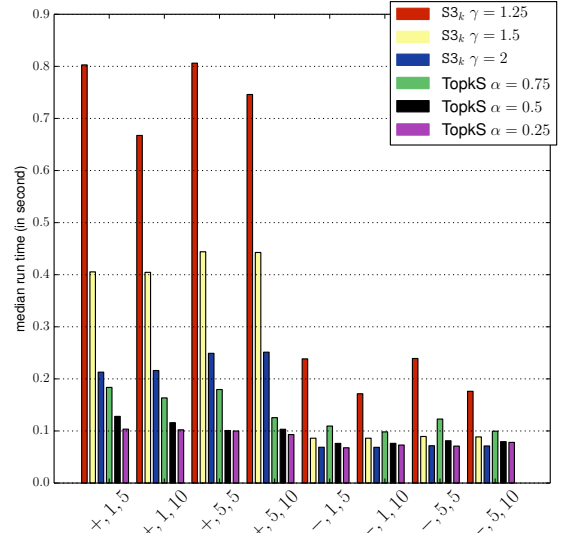


Figure 5: Query answering times on  $I_1$  (Twitter).

the seeker, and stores such paths in *borderPath*, which may grow very large and hurt performance. To avoid storing *borderPath*, we compute for each explored vertex  $v$  the weighted sum over all paths of length  $n$  from the seeker to this vertex:

$$borderProx(v, n) = \sum_{p \in u \rightarrow v, |p|=n} \frac{prox(p)}{\gamma^n}$$

and compute *prox* directly based on this value.

Furthermore, Algorithm `GetDocuments` considers documents reachable from  $x$  through edges labeled  $S3$ :`partOf`,  $S3$ :`commentsOn`,  $S3$ :`commentsOn`,  $S3$ :`hasSubject` or  $S3$ :`hasSubject`. Reachability by such edges defines a *partition* of the documents into *connected components*. Further, by construction of *con* tuples (Section 3.2), connections carry over from one fragment to another, across such edges. Thus, a fragment matches the query keywords iff its component matches it, leading to an efficient pruning procedure: we compute and store the partitions, and test that each keyword (or extension thereof) is present in every component (instead of fragment). Partition maintenance is easy when documents and tags are added, and more expensive for deletions, but luckily these are rarer.

The query answering algorithm creates in RAM the *allProx* table and two sparse matrices, computed only once: *distance*, encoding the graph of network edges in  $I$  (accounting for the vertical neighborhood), and *component*, storing the component of each fragment or tag. Thus, Algorithm 3, which computes *allProx* and finds new components to explore, relies on efficient matrix and vector operations. For instance, the new distance vector *borderProx* w.r.t. the seeker at step  $n + 1$  is obtained by multiplying the distance matrix with the previous distance vector from step  $n$ . The documents and the RDF graph, on the other hand are not stored in RAM, and are queried using a PostgreSQL database.

The search algorithm can be *parallelized* in two ways. First, within an iteration, we discover new documents in different components in parallel. Second, when *borderProx* is available in the current iteration, we can start computing the next *borderProx* using the distance matrix. More precisely, Algorithm 3 (`ExploreStep`) can be seen as consisting of two main blocks: (i) computing the new *borderProx* using the (fixed) distance matrix and the previous *borderProx* (lines 1-12 except line 10); (ii) computing *allProx* using the new *borderProx* and the previous *allProx* (lines 13-16) plus the call to `GetDocuments` (line 10). The latter algorithm consists of two parts: (iii) identifying the newly discovered components, respectively (iv) testing the documents they contain. We used 8 concurrent threads, each running a task of one of the forms (i)-(iv), above, and synchronized them with a custom

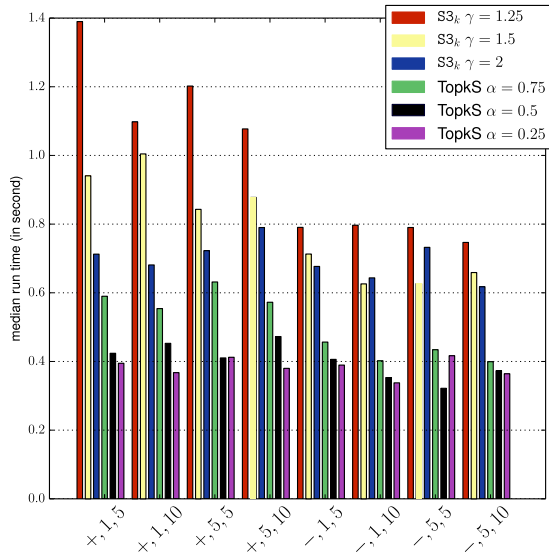


Figure 6: Query answering times on  $I_3$  (Yelp).

scheduler. This reduced the query answering time on average by a factor of 2.

### 5.3 Query answering times

Figures 5 and 6 show the running times of  $S3_k$  on the  $I_1$  and  $I_3$  instances; the results on the smaller instance  $I_2$  are similar [3]. We used different values of the  $\gamma$  social proximity damping factor (Section 3.4) and the  $\alpha$  parameter of TopkS. For each workload, we plot the average time (over its 100 queries). *All runs terminated by reaching the threshold-based stop condition* (Algorithm 2).

A first thing to notice is that while all running times are comparable, TopkS runs consistently faster. This is mostly due to the different proximity functions: our *prox*, computed from all possible paths, has a much broader scope than TopkS, which explores and uses only one (shortest) path. In turn, as we show later, we return a significantly *different* set of results, due to *prox*'s broader scope and to considering document structure and semantics.

Decreasing the  $\gamma$  in  $S3_k$  reduces the running time. This is expected, as  $\gamma$  gives more weight to nodes far from the seeker, whose exploration is costly. Similarly, *increasing*  $\alpha$  in TopkS forces to look further in the graph, and affects negatively its performance.

The influence of  $k$  is more subtle. When the number of candidates is low and the exploration of the graph is not too costly, higher  $k$  values allow to include most candidates among the  $k$  highest-scoring ones. This reduces the exploration needed to refine their bounds enough to clarify their relative ranking. In contrast, if the number of candidates is important and the exploration costly, a small  $k$  value significantly simplifies the work. This can be seen in Figure 7 where, with frequent keywords, increasing  $k$  does not affect the 3 fastest quartiles but significantly slows down the slowest quartile, since the algorithm has to look further in the graph.

The same figure also shows that rare-keyword workloads (whose labels start by  $-$ ) are faster to evaluate than the frequent-keyword ones (workload labels starting with  $+$ ). This is because finding rare keywords tends to require exploring longer paths. Social damping at the end of such paths is high, allowing to decide that possible matches found even farther from the seeker will not make it into the top- $k$ . In contrast, matches for frequent keywords are found soon, while it is still possible that nearby exploration may significantly change their relative scores. In this case, more search and computations are needed before the top- $k$  elements are identified.

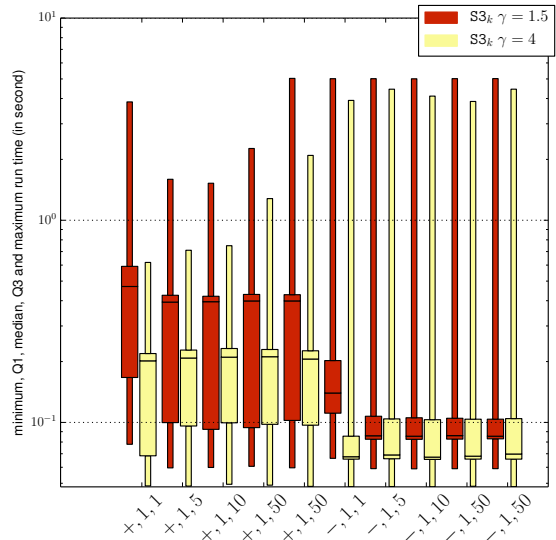


Figure 7: Query answering times on  $I_1$  when varying  $k$ .

Measure \ Instance	$I_1$	$I_2$	$I_3$
Graph reachability	12%	23%	41%
Semantic reachability	83%	100%	78%
$L_1$	8%	10%	4%
Intersection size	13.7%	18.4%	5.6%

Figure 8: Relations between  $S3_k$  and TopkS answers.

### 5.4 Qualitative comparison

We compare now the answers of our  $S3_k$  algorithm and those of TopkS from a *qualitative* angle.  $S3_k$  follows links between documents to access further content, while TopkS does not; we term *graph reachability* the fraction of candidates reached by our algorithm which are not reachable by the TopkS search. Further, while  $S3_k$  takes into account semantics by means of semantic extension (Definition 2.1), TopkS only relies on the query keywords. We call *semantic reachability* the ratio between the number of candidates examined by an algorithm *without* expanding the query, and the number of candidates examined *with* query expansion. Observe that some  $S3_k$  candidates may be ignored by TopkS due to the latter's lack of support for *both* semantics and connections between documents. Finally, we report two measures of distance between the results of the two algorithms. The first is the *intersection size* i.e., the fraction of  $S3_k$  results that TopkS also returned. The second,  $L_1$ , is based on Spearman's well-known *foot rule* distance between lists, defined as:

$$L_1(\tau_1, \tau_2) = 2(k - |\tau_1 \cap \tau_2|)(k+1) + \sum_{i \in \tau_1 \cap \tau_2} |\tau_1(i) - \tau_2(i)| - \sum_{\substack{\tau \in \{\tau_1, \tau_2\} \\ i \in \tau \setminus (\tau_1 \cap \tau_2)}} \tau(i)$$

where  $\tau_j(i)$  is the rank of item  $i$  in the list  $\tau_j$ .

The averages of these 4 measures over the 8 workloads on each instance appear in Figure 8. The ratios are low, and show that different candidates translate in different answers (the low  $L_1$  stands witness for this). Few  $S3_k$  results can be attained by an algorithm such as TopkS, which ignores semantics and relies only on the shortest path between the seeker and a given candidate.

### 5.5 Experiment conclusion

Our experiments have demonstrated first the ability of the  $S3$  data model to *capture very different social applications*, and to query them meaningfully, accounting for their structure and enriching them with semantics. Second, we have shown that  $S3_k$  query answering can be quite efficient, even though considering all paths between the seeker and a candidate answer slows it down

w.r.t. simpler algorithms, which rely on a shortest-path model. We have experimentally verified the expected impact of the social damping factor  $\gamma$  and of the result size  $k$  on running time. Third, and most importantly, we have shown that taking into account in the relevance model the social, structured, and semantic aspects of the instance bring a *qualitative gain*, enabling meaningful results that would not have been reachable otherwise.

## 6. RELATED WORK

Prior work on *keyword search in databases* spreads over different research directions:

**Top-k search in a social environment** uses UIT models [18, 21, 30] we outlined in Section 1. Top-k query results are found based on a score function accounting for the presence of each keyword in the tags of a candidate item, and a simple social distance based on the length of the social edge paths; query answering algorithms are inspired from the general top-k framework of [8]. As documents are considered atomic, and relations between them are ignored, requirements **R1**, **R2** and **R4** are not met. Further, the lack of semantics also prevents **R5**. Recent developments tend to focus on performance and scalability, or the integration of more attributes such as locality or temporality [7, 16], without meeting the abovementioned requirements. Location and time can be added to generic scores but this is outside of the scope of this paper.

**Semi-structured document retrieval based on keywords** relies mostly on the *Least Common Ancestors* approach, by which a set of XML nodes containing the requested keywords are resolved into one result item, their common ancestor node [6, 23]. This field pioneered by [11], encompassed by our model, generalizes LCA constraints but lacks both social and semantics, and thus meets only **R2**. Other recent developments in this area, including more flexible and comprehensive reasoning patterns, have been presented in [2] but have the same limitations. IR-style search in relational databases [12, 13] considers key-foreign key relationships between items, but ignores text structure, semantics, and social aspects.

**Semantic search on full-text documents**, either via RDF [15, 25] or a semantic similarity measure [24], allows to query interconnected, semantic rich unstructured textual documents or entities, thus meeting **R1**, **R5** and **R6**. Efforts to consider XML structure in such semantics-rich models [9] also enable **R2**.

**Personalized IR in a social context** adapts the answers to a user's query, taking into account her interests and those of her direct and indirect social connections [5]; this meets **R1** but not **R2** nor **R3**.

All the aforementioned models can be seen as partial views over the  $S_3$  model we devised, and they could easily be transcribed into it modulo some minor variations; for instance, Facebook's Graph-Search [7] is a restricted form of SPARQL query one could ask over an  $S_3$  instance. Slight adaptations may be needed for social contexts tolerating *similarity* between keywords that goes beyond the strict specialization relation (in RDF sense) we consider. We have hinted in Section 2 how this could be included.

## 7. CONCLUSION

We devised the  $S_3$  data model for structured, semantic-rich content exchanged in social applications. We also provided the  $S_{3k}$  top-k keyword search algorithm, which takes into account the social, structural and semantical aspects of  $S_3$ . Finally, we demonstrated the practical interest of our approach through experiments on three real social networks.

Next, we plan to extend  $S_{3k}$  to a massively parallel in-memory computing model to make it scale further. We also consider gen-

erating user-centric knowledge bases to be used in  $S_{3k}$ , to further adapt results to the user's semantic perspective.

## References

- [1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [2] C. Aksoy, A. Dimitriou, and D. Theodoratos. Reasoning with Patterns to effectively answer XML keyword queries. *The VLDB Journal*, 2015.
- [3] R. Bonaque, B. Cautis, F. Goasdoué, and I. Manolescu. Social, structured and semantic search, extended version. <https://hal.inria.fr/hal-01218116>, 2015.
- [4] P. Buneman, E. V. Kostylev, and S. Vansummeren. Annotations are relative. In *ICDT*. ACM, 2013.
- [5] D. Carmel, N. Zwerdling, I. Guy, S. Ofek-Koifman, N. Har'El, I. Ronen, E. Uziel, S. Yogev, and S. Chernov. Personalized social search based on the user's social network. In *CIKM*, 2009.
- [6] L. J. Chen and Y. Papakonstantinou. Supporting top-k keyword search in XML databases. In *ICDE*, 2010.
- [7] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, et al. Unicorn: A system for searching the social graph. *PVLDB*, 2013.
- [8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4), 2003.
- [9] F. Goasdoué, K. Karanasos, Y. Katsis, J. Leblay, I. Manolescu, and S. Zampetakis. Growing triples on trees: an XML-RDF hybrid model for annotated documents. *VLDB Journal*, 2013.
- [10] F. Goasdoué, I. Manolescu, and A. Roatis. Efficient query answering against dynamic RDF databases. In *EDBT*, 2013.
- [11] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRank: Ranked keyword search over XML documents. In *SIGMOD*, 2003.
- [12] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, 2003.
- [13] V. Hristidis, H. Hwang, and Y. Papakonstantinou. Authority-based keyword search in databases. *ACM TODS*, 33(1), 2008.
- [14] G. Jeh and J. Widom. Simrank: a measure of structural-context similarity. In *SIGKDD*, 2002.
- [15] W. Le, F. Li, A. Kementsietsidis, and S. Duan. Scalable keyword search on large RDF data. *IEEE TKDE*, 26(11), 2014.
- [16] Y. Li, Z. Bao, G. Li, and K.-L. Tan. Real time personalized search on social networks. In *ICDE*, 2015.
- [17] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *JASIST*, 2007.
- [18] S. Maniu and B. Cautis. Network-aware search in social tagging applications: instance optimality versus efficiency. In *CIKM*, 2013.
- [19] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. Ordpats: Insert-friendly XML node labels. In *SIGMOD*, 2004.
- [20] A. Ritter, S. Clark, Mausam, and O. Etzioni. Named entity recognition in tweets: An experimental study. In *EMNLP*, 2011.
- [21] R. Schenkel, T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, and G. Weikum. Efficient top-k querying over social-tagging networks. In *SIGIR*, 2008.
- [22] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, 2002.
- [23] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum. TopX: efficient and versatile top-k query processing for semistructured data. *The VLDB Journal*, 2008.
- [24] M. Theobald, R. Schenkel, and G. Weikum. Efficient and self-tuning incremental query expansion for top-k query processing. In *SIGIR*, 2005.
- [25] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In *ICDE*, 2009.
- [26] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. E. Bal. WebPIE: A web-scale parallel inference engine using MapReduce. *J. Web Sem.*, 2012.
- [27] Resource Description Framework. <http://www.w3.org/RDF>.
- [28] Uniform Resource Identifier. <http://tools.ietf.org/html/rfc3986>.
- [29] Yelp Dataset Challenge. [http://www.yelp.com/dataset\\_challenge](http://www.yelp.com/dataset_challenge).
- [30] S. A. Yahia, M. Benedikt, L. V. Lakshmanan, and J. Stoyanovich. Efficient network aware search in collaborative tagging sites. *PVLDB*, 2008.

# Indexing Query Graphs to Speedup Graph Query Processing

Jing Wang  
School of Computing Science  
University of Glasgow, UK  
j.wang.3@research.gla.ac.uk

Nikos Ntarmos  
School of Computing Science  
University of Glasgow, UK  
nikos.ntarmos@glasgow.ac.uk

Peter Triantafillou  
School of Computing Science  
University of Glasgow, UK  
peter.triantafillou@glasgow.ac.uk

## ABSTRACT

Subgraph/supergraph queries although central to graph analytics, are costly as they entail the NP-Complete problem of subgraph isomorphism. We present a fresh solution, the novel principle of which is to acquire and utilize knowledge from the results of previously executed queries. Our approach, *i*GQ, encompasses two component subindexes to identify if a new query is a subgraph/supergraph of previously executed queries and stores related key information. *i*GQ comes with novel query processing and index space management algorithms, including graph replacement policies. The end result is a system that leads to significant reduction in the number of required subgraph isomorphism tests and speedups in query processing time. *i*GQ can be incorporated into any sub/supergraph query processing method and help improve performance. In fact, it is the only contribution that can speedup significantly both subgraph and supergraph query processing. We establish the principles of *i*GQ and formally prove its correctness. We have implemented *i*GQ and have incorporated it within three popular recent state of the art index-based graph query processing solutions. We evaluated its performance using real-world and synthetic graph datasets with different characteristics, and a number of query workloads, showcasing its benefits.

## Categories and Subject Descriptors

H.2.4 [Systems]: Query processing

## General Terms

Design, Performance

## Keywords

Graph query processing, indexing, query result caching

## 1. INTRODUCTION

Graph structured data are prevalent in many modern big data applications, ranging from chemical, bioinformatics,

and other scientific datasets to social networking and social-based applications (such as recommendation systems). In biology, for example, there is a great need to model “structured interaction networks”. These abound when studying species, proteins, drugs, genes, and molecular and chemical compounds, etc. In these graphs, nodes can model species, genes, etc. and edges reflect relationships between them. Molecular compounds, consisting of atoms and their bonds, are naturally modeled as graphs. Ditto for social networks, where nodes refer to people and edges to their relationships.

Developing systems and algorithms that can store, manage, and provide analysis over large numbers of (potentially large) graphs is a formidable challenge. Already, there exist several very large graph datasets. For instance, the PubChem[34] chemical compound dataset contains more than 35 million graphs and ChEBI[11] (Chemical Entities of Biological Interest) dataset contains more than half a million graphs. Further applications extend to software development and debugging[27] and to similarity searching in medical datasets[32]. As a result, a large number of graph data management systems, optimised for handling graph data, have emerged (e.g., Neo4J[4], InifiteGraph[20]). This is in addition to graph management systems designed by big data companies for their own purposes (e.g., Twitter’s FlockDB[38], Google’s Pregel[28]) and the list is continuously expanding. Hence, the demand for high performance data analytics in graph data systems is steadily increasing.

Central to graph analytics, is the need to locate patterns in dataset graphs. Informally, given a query graph, the system is called to identify which of the stored graphs in its dataset contain it (subgraph matching), or are contained in it (supergraph matching). This is a very costly operation as it entails the NP-Complete[14] problem of subgraph isomorphism and even its most popular solutions [9, 25, 39] are computationally very expensive. This problem is exacerbated when dealing with datasets storing large numbers of graphs, as the number of required subgraph isomorphism tests grows. Furthermore, performance deteriorates significantly with increasing graph sizes.

The key driver of our work is the realization that in many applications, it is natural to expect that queries submitted in the past share subgraph or supergraph relationships with queries of the future. As one example, consider chemical graph datasets, where queries use the graph representation of chemical entities. Such queries are naturally hierarchical: At the base, we see chemical elements. Then, there are graphs depicting chemical compounds (consisting of chemical elements), while there are also techniques to

create chemical compound clusters out of similar chemical compounds[34]. Similarly, in protein datasets there is also a hierarchy of queries for aminoacids, proteins, protein mixtures, proteins of uni-cell bacteria, all the way to those of multi-cell organisms. Finally, typical tools for social network analysis (SNA – e.g., Pajek[10]) provide the ability to produce graphs by filtering nodes and/or edges from other graphs. Using such graphs as queries in exploratory interactive SNA induces again the previous characteristic. For instance, consider the (query) graphs for analyzing friendship networks: such networks within the USA are subgraphs of friendship networks within North America, which in turn are subgraphs of the complete friendship network graph. The conclusion is that, in many applications, any query can itself be a subgraph or supergraph of a previously issued query. Up to now, this natural subgraph/supergraph relationship among queries has not been exploited.

## 2. PERSPECTIVES AND RELATED WORK

The problem of subgraph/supergraph query processing has been extensively studied. A prominent paradigm in the literature is the *filter then verify* paradigm. Essentially, this is an index-based class of methods. During indexing, the dataset graphs are reduced to their features (a *feature* being any substructure of a graph, be it path, tree, cycle, or arbitrary subgraph), which are inserted into an index structure (e.g., tree, trie, hash table, etc.). Given a query graph  $g$ ,  $g$  is also decomposed into its features, following the same process as for dataset graphs. Then the index is searched for  $g$ 's features; for subgraph queries, the set of graphs that contain all of said features are returned, whereas for supergraph queries the returned set consists of graphs all of whose features are contained in  $g$ 's features. This set is called the *candidate set* and producing it constitutes the filtering stage of query processing.

All known algorithms guarantee that there will be no *false negatives*; that is, for subgraph (resp. supergraph) queries, all graphs in the dataset that can possibly contain (resp. are contained in) the query graph will be included in the candidate set. However, *false positives* are possible – not all graphs in the candidate set contain (resp. are contained in) the query graph. And herein lies the primary source of problems, since a subgraph subgraph isomorphism test must be performed against each graph in the candidate set, during the verification stage of query processing. The major focus of related work then is how to reduce the number of false positives, i.e., the number of unnecessary subgraph isomorphism tests.

Approaches in the literature can be classified along two dimensions: whether they employ (frequent) mining techniques or an exhaustive enumeration for the production of features, and based on the type of features of the dataset graphs they index (e.g., paths, trees, subgraphs). Note that exhaustive enumeration can yield huge indices and may take a prohibitively long time to do so. For this reason, all exhaustive enumeration approaches limit the size of features to a typically fairly small number of edges (i.e., 10 or less).

Mining-based approaches, both for supergraph queries ([5, 51, 46, 6, 52]) and subgraph queries (e.g., [41, 7, 52]) utilize techniques to mine for frequent (or *discriminating*, in [6]) (sub)graphs among the dataset graphs that are then indexed. Other mining-based approaches, like Tree+ $\Delta$ [49] and TreePi[45] mine for and index frequent trees. Last, Lin-

dex[43] and LWinindex[44] utilize the frequent mining algorithms of previous approaches, and are thus able to index and query several feature types. Typically such approaches tend to mine for more complex structures, which presents a trade-off between the complexity and time required for the indexing process vis-a-vis the potential for higher pruning power during query processing. However, numerous related performance studies [21, 12, 15, 17, 22] have shown that feature-mining approaches tend to be comparatively worse performers.

On the other hand, SING[12], GraphGrep[16] and GraphGrepSX[3] perform exhaustive enumeration, listing all paths of dataset graphs up to a certain path length. Similarly, CT-Index[22] indexes trees and cycles, whereas Grapes[15] indexes paths along with location information.

A different approach, which does not index features as above, is presented in gCode[53]. For each graph  $G$  in the graph dataset, gCode computes a signature per vertex of  $G$  (essentially reflecting the vertex's neighbourhood) and then computes a signature for  $G$  itself. The latter is a tree structure combining the signatures of all its vertices.

With respect to the verification stage, approaches also differ on how this is performed. In some works, verification is performed by applying any (exact) subgraph isomorphism algorithm of choice (see [25] for a detailed insightful comparative evaluation) after the filtering stage. Indeed, this can be the default choice for all approaches and there is a large variety of subgraph isomorphism algorithms available. Most such algorithms are influenced by Ullman's early work [39]. Arguably, the algorithm that is now the most widely used is the VF2[9] algorithm. Last, several approaches store and utilize location information in their index to achieve further filtering ([45, 12, 15]).

Recent performance studies [17, 21] have shown that CT-Index[22] and Grapes[15] are high performing approaches. CT-Index[22] is based on deriving canonical forms for the (tree, cycle) features of a graph  $G$ , to the fact that for trees and cycles finding string-based canonical forms can be done in linear time (unlike general graphs). These string representations of a graph's features are then hashed into a bitmap structure per graph  $G$ . Checking whether a query graph  $g$  can possibly be a subgraph of a graph  $G$ , can be done with simple bitwise operators between the bitmap of  $g$  and that of  $G$  (as supergraphs must contain all features of a subgraph). Last, its verification stage is then based on VF2.

Grapes[15] is designed to exploit parallelism available in multi-core machines. It exhaustively enumerates all paths (up to a maximum length), which are then inserted into a trie with their location information. This operation is performed in parallel by several threads, each of which works on a portion of the graph, producing its own trie, and subsequently all tries are merged together to form the path index of a graph. Grapes then computes (typically) small connected components of graphs in the candidate set, on which the verification (subgraph isomorphism test) is performed.

An insightful discussion and comparative performance evaluation of several indexing techniques for subgraph query processing (published prior to 2010) can be found in [17]. Furthermore, in [21] we presented a systematic performance and scalability study of several older as well as current state-of-the-art index-based approaches for subgraph query processing. We are not aware of similar in-depth studies of solutions to supergraph query processing; however, [44] pro-

vides a concise overview of related approaches.

On a related note, recent work also deals with graph querying against historical graphs, identifying subgraphs enduring graph mutations over time [35], which can be viewed as a variation whereby graph snapshots in time can be viewed as different graphs. Also, the research community has recently started looking into subgraph queries against a single, very large graph consisting of possibly billions of nodes [36]. To accelerate the query processing, SPath [48] proposes a path-at-a-time fashion, which proves to be more efficient than traditional vertex-at-a-time methods, whereas [36] makes use of a memory cloud and [33, 1, 24] exploit MapReduce. In this subgraph querying problem for the single large graph setting, the goal is to expedite the subgraph isomorphism itself, whereas in the setting with many dataset graphs, the target of subgraph querying problem is to minimize the number of isomorphism tests that need to be performed. Our work focuses on the latter setting and leaves for future work the application of our ideas to the former setting.

There has also been considerable work on approximate graph pattern matching. Relevant techniques (e.g., [22, 18, 37, 40, 42, 47, 50, 33, 13]) perform subgraph matching with support for wildcards and/or approximate matches. These solutions are not directly related to our work, as we expedite exact index-based subgraph/supergraph query processing.

Caching of the results of path/tree queries has been explored in XML databases [26, 2, 29]. The problem we focus on is considerably different, as the queries we deal with are in the form of graphs (not just paths/trees), thus entailing the NP-Complete problem of subgraph isomorphism. Furthermore, in our setting queries retrieve stored graphs that contain the query graph (subgraph queries) or are contained in it (supergraph queries), and we exploit both supergraph and subgraph relationships among queries themselves, as opposed to only subsumption (i.e., supergraph) relationships. Moreover, our graph replacement policy also takes into account the subgraph isomorphism costs, as opposed to just the size or popularity of cached queries.

Last, [23] presents a cache for targeted historical queries against a large social graph. In this case, each query is centered around a uniquely identified node in the social graph, and the objective is to avoid maintaining and/or reconstructing complete snapshots of the social graph, but to instead use a set of static “views” – i.e., snapshots of neighborhoods of nodes – to rewrite incoming queries. [23] does not deal with subgraph/supergraph query processing; rather, the nature of the queries means that containment can be decided by simply measuring the distance of the central query node to the center of each view, while also taking into account the diameter of these two graphs. Furthermore, the authors do not provide a cache replacement strategy, but rather an algorithm to compute the optimal cache contents given a set of queries. *iGQ* could well be used to both generalise and expedite query processing in [23].

## 2.1 The *iGQ* Perspective

In this work we offer a new perspective and a strategy for improving subgraph/supergraph query processing performance and scalability. Our approach rests on the following three observations: First, in related works there exists an implicit assumption that graph queries will be similarly structured to the dataset graphs. In general this is not guaranteed to hold (e.g., in exploratory analytics), and when

query graphs have no match in the dataset graphs, query processing cannot benefit at all from indexes that are solely constructed on dataset graphs. Second, even when query graphs have matches against dataset graphs, the system performs expensive computations during query processing and simply throws away all (painstakingly and laboriously) derived knowledge (i.e., previous querying result). Third, the success of known approaches depends on and exploits the fact that dataset graphs share features (e.g., when mining for frequent features) and/or that dataset graph features contain or are contained in other graph features (e.g., when using tries to index dataset graph features). However, they completely fail to investigate and exploit such similarities between query graphs.

As mentioned, it is natural in many applications for new queries to bear subgraph/supergraph relationships with previously issued queries. Our efforts in this work centre on exploiting this characteristic to further improve the performance of query processing. Therefore, instead of “mining” only the stored graphs and creating relevant indexes on them, we also “mine” *query graphs* and accumulate the knowledge produced by the system when running queries, creating a *query index* in addition to the *dataset index*. Our insights identify which is the relevant accumulated knowledge and how to exploit it during query processing in order to further reduce the number of subgraph isomorphism tests. *iGQ* can accommodate any proposed index for sub or supergraph query processing and help expedite both query types.

## 2.2 Contributions

The contributions of this work are that we:

- Provide a new perspective to the problem of subgraph/supergraph query processing, with insights as to how the work performed by the system when executing queries can be appropriately managed to improve the performance of future queries.
- Detail the *iGQ* approach, based on a query index structure and associated query processing algorithms, which can reduce the number of isomorphism tests performed during query processing.
- Present the *iGQ* framework, showing how to incorporate *iGQ* within existing approaches, and the two *iGQ* components: a subgraph query index and a supergraph query index. The subgraph index of *iGQ* can be based on any existing subgraph index (over query graphs, not dataset graphs). The supergraph index on the other hand is a new index to swiftly determine supergraph status between new and previous queries.
- Address the issue of index space management, providing mechanisms for index updates and a graph replacement policy, deciding contents of query index.
- Implement *iGQ*, incorporate it within three popular approaches for graph query processing, and provide experimental results using real-world datasets and a number of query workloads, showcasing *iGQ*’s benefits against competitive state of the art methods.

## 3. PROBLEM FORMULATION

We consider undirected labeled graphs. For simplicity, we assume that only vertices have labels; all our results straightforwardly generalize to graphs with edge labels.

DEFINITION 1. A labeled graph  $G = (V, E, l)$  consists of

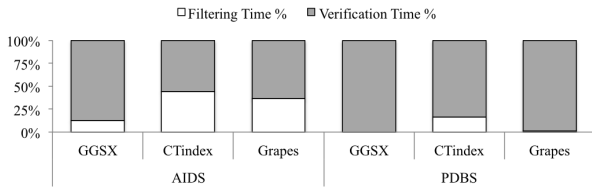


Figure 1: Dominance of the Verification Time on the Overall Query Processing Time of Three Subgraph Querying Algorithms on Two Different Real-world Graph Datasets

a set of vertices  $V(G)$  and edges  $E(G) = \{(u, v), u \in V, v \in V\}$ , and a function  $l : V \rightarrow U$ , where  $U$  is the label set, defining the domain of labels of vertices.

A sequence of vertices  $(v_0, \dots, v_n)$  s.t.  $\exists (v_i, v_{i+1}) \in E$ , constitutes a path of length  $n$ . A simple path is a path where no vertices are repeated. A cycle is a path of length  $n > 1$ , where  $v_0 = v_n$ . A simple cycle is a cycle with no repeated vertices (other than  $v_0$  and  $v_n$ ). A connected graph is one where there exists a path between any pair of its vertices.

**DEFINITION 2.** A graph  $G_i = (V_i, E_i, l_i)$  is subgraph-isomorphic to a graph  $G_j = (V_j, E_j, l_j)$ , (by abuse of notation) denoted by  $G_i \subseteq G_j$ , when there exists an injection  $\phi : V_i \rightarrow V_j$ , such that  $\forall (u, v) \in E_i, u, v \in V_i, \Rightarrow (\phi(u), \phi(v)) \in E_j$  and  $\forall u \in V_i, l_i(u) = l_j(\phi(u))$ .

Informally, there is a subgraph isomorphism  $G_i \subseteq G_j$  if  $G_j$  contains a subgraph that is isomorphic to  $G_i$ . In this case, we say that  $G_i$  is a subgraph of (or contained in)  $G_j$ , or inversely that  $G_j$  is a supergraph of (contains)  $G_i$  (denoted by  $G_j \supseteq G_i$ ).

**DEFINITION 3.** The subgraph querying problem entails a set  $D = \{G_1, \dots, G_n\}$  containing  $n$  graphs, and a query graph  $g$ , and determines all graphs  $G_i \in D$  such that  $g \subseteq G_i$ .

**DEFINITION 4.** The supergraph querying problem entails a set  $D = \{G_1, \dots, G_n\}$  containing  $n$  graphs, and a query graph  $g$ , and determines all graphs  $G_i \in D$  such that  $g \supseteq G_i$ .

The  $i$ GQ index,  $\mathbb{I}$ , will be called to index the features of query graphs; then we shall say that query graph  $g$  is indexed by  $i$ GQ and (by abuse of notation) denote it by  $g \in \mathbb{I}$ . We denote with  $\mathbb{I}_{sub}(g)$  all query graphs currently contained in  $\mathbb{I}$  that are supergraphs of  $g$  (answers to  $g$ , if  $g$  was a subgraph query); i.e.,  $\mathbb{I}_{sub}(g) = \{G \in \mathbb{I} \wedge g \subseteq G\}$ . Similarly, we denote with  $\mathbb{I}_{super}(g)$  all query graphs currently contained in  $\mathbb{I}$  that are subgraphs of  $g$  (answers to  $g$ , if  $g$  was a supergraph query); i.e.,  $\mathbb{I}_{super}(g) = \{G \in \mathbb{I} \wedge g \supseteq G\}$ .

## 4. iGQ PRINCIPLES

We firstly discuss our findings from experiments we ran regarding the major performance obstacles we need to overcome if we are to bring about further query processing time reductions. Subsequently, we present the  $i$ GQ framework, followed by an explanation of how the components of  $i$ GQ are utilized for further performance improvements, and related formal proofs of correctness. As mentioned, so far related work has not considered benefiting from the execution of previous queries. Thus, despite devoting a lot of resources to such queries, the results derived cannot be put to good use to improve performance of future subgraph queries.

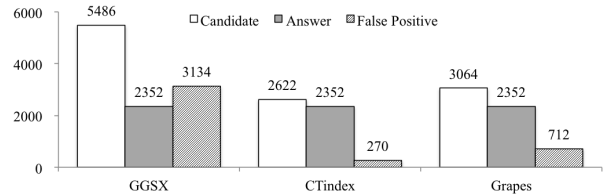


Figure 2: Average Number of Candidates, Answer Set Size, and False Positives in the AIDS Dataset

## 4.1 Insights

We report on the fundamentals of the performance of three state of the art approaches, GraphGrepSX[3] (GGSX), Grapes[15], and CT-Index[22], over three real datasets and one synthetic dataset with different characteristics. These characteristics will be presented in detail in the experimental evaluation section. Briefly, AIDS represents a graph DB consisting of 40,000 very small, sparse graphs, while PDBS is a graph dataset containing 600 large graphs. Please note that the way the queries were generated is standard among related work [15, 22].

### Subgraph Query Performance: Where Does Time Go?

There are two key components of the overall query processing time: filtering time (to process the index and produce the candidate set) and verification time (to perform the verification of all candidate graphs). Fig. 1 shows what percentage of the total query processing time is attributed to each component.

The dominance of the verification step is clear. This holds across the three different approaches that employ different indexing methods and utilize different strategies for cutting down the cost of subgraph isomorphism. Recall that subgraph isomorphism performance is highly sensitive to the size of both the input graph and the stored graph. Hence, we would expect that for smaller stored graphs (as in the AIDS dataset) the verification step would be much faster. Notably, however, even when graphs are very small, the verification step is the biggest performance inhibitor and as graphs become larger (e.g., PDBS) the verification step becomes increasingly responsible for nearly the total query processing time. Of course, given the NP-Completeness of subgraph isomorphism, one would expect that verification would dominate, especially for large graphs. But the fact that even with very small graphs this holds is noteworthy.

### Filtering Power: Is It Good Enough?

The second fundamental point pertains to how one can reduce the verification cost. Related works highlight that their approaches prove to be very powerful in terms of filtering out the vast majority of DB graphs. In Figures 2 and 3 we show our results with respect to the average size of candidate sets and of the answer set, as well as the average number of false positives for the AIDS and PDBS datasets.

First, note that different algorithms behave differently in different datasets (e.g., Grapes significantly outperforms CT-Index in PDBS while the reverse holds for AIDS). Second, note that despite the powerful filtering of an approach, when the DB contains a large number of graphs (see Figure 2) in absolute numbers, there is a very large number of unnecessary subgraph isomorphism tests (i.e., false positives)



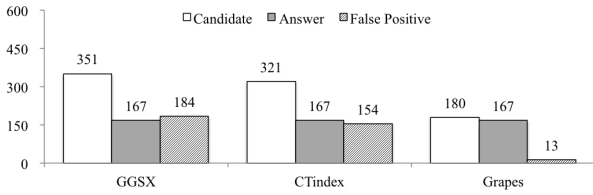


Figure 3: Average Number of Candidates, Answer Set Size, and False Positives in the PDBS Dataset

that is required. The above two combined imply that even the best algorithm will suffer from a large number of unnecessary subgraph isomorphism tests under some datasets.

Turning our attention to Figure 3 we see that for DBs with medium to small number of graphs, the high filtering power can indeed result in requiring only a relatively small number of subgraph isomorphism tests. However, considerable percentages of false positives can appear in the candidate sets of even top-performing algorithms; e.g., CT-Index, which exhibited the best filtering in the AIDS dataset, has an almost 50% false positive ratio in the PDBS dataset. Furthermore, not all subgraph isomorphism tests for the graphs in the candidate set are equally costly. As the cost of subgraph isomorphism testing depends on the size of the graph, the larger graphs in the candidate set contribute a much greater proportion of the total cost of the verification step. Note that, naturally, false positive graphs tend to be the largest graphs in the DB, since these have a higher probability to contain all features of query graphs.

Note that we placed emphasis on the number of unnecessary subgraph isomorphism tests (i.e., the false positives), as we can improve filtering further by reducing this number. However, this is not the only source of possible improvements. As we shall show later, *iGQ* can improve on the number of subgraph isomorphism tests even beyond this, by exploiting knowledge gathered during query execution.

The insights that can be drawn are as follows:

- Despite the fact that state of the art techniques (based on indexing features of DB graphs) can enjoy high filtering capacity, there is still large room for improvement, as even the best approaches may perform large numbers of unnecessary subgraph isomorphism tests.
- Improving further the filtering power of approaches can significantly improve query processing time, as this will reduce the number of subgraph isomorphism tests, which dominates the overall querying time.
- Even approaches that are purported to enjoy great filtering powers, can behave much more poorly under different datasets.
- Unnecessary subgraph isomorphism tests are not solely caused by false positives; even graphs in the candidate set that are true positives can be unnecessarily tested if the system fails to exploit this knowledge (accrued by previous query executions).

## 4.2 The *iGQ* Framework

*iGQ* aims to augment the functionality and benefits offered by any one of the subgraph and/or supergraph indexing methods in the literature. Let us call the chosen method  $\mathbb{M}$ . The *iGQ* framework consists of method  $\mathbb{M}$  and the two components of  $\mathbb{I}$ ,  $\mathbb{I}_{sub}$  and  $\mathbb{I}_{super}$ . For the sake of simplicity, we shall first describe the operation of *iGQ* when  $\mathbb{M}$  is a

method for subgraph query processing (denoted  $\mathbb{M}_{sub}$ ). Initially, method  $\mathbb{M}_{sub}$  builds its graph dataset index as per usual. The *iGQ* index,  $\mathbb{I}$ , starts off empty; it is then populated as queries arrive and are executed by  $\mathbb{M}_{sub}$ .

Upon the arrival of a query  $g$ , the query processing process is parallelized. One thread uses method  $\mathbb{M}_{sub}$ 's algorithms and indexing structure to breakdown the query graph into its features, and uses its index to produce a candidate set of graphs,  $CS(g)$ , as usual. Additionally,  $\mathbb{I}$  will obtain as many of the intermediate and final results from method  $\mathbb{M}$ 's execution as possible; e.g., it will obtain the features of the query graph, to be compared to those stored in  $\mathbb{I}$  (from previously-executed queries). At this point, two separate threads will be created: one will check whether the query graph is a subgraph of previous query graphs and the other will check whether it is a supergraph of previous query graphs. These cases yield different opportunities for optimization and are discussed separately below.

In the following we proceed to describe the function of each component of the *iGQ* framework and how it is all brought together. For the formal proofs of correctness that follow, for simplicity, we make the following assumptions.

**Assumptions.** The *iGQ* index components,  $\mathbb{I}_{sub}$  and  $\mathbb{I}_{super}$  work correctly. That is:

$$G \in \mathbb{I}_{sub}(g) \Rightarrow g \subseteq G \quad (1)$$

and

$$G \in \mathbb{I}_{super}(g) \Rightarrow g \supseteq G \quad (2)$$

We will prove that these assumptions hold in sections 6.1 and 6.2.

### 4.2.1 The Subgraph Case: $\mathbb{I}_{sub}$

This case occurs when a new query  $g$  is a subgraph of a previous query  $G$ . When  $G$  was executed by the system, the  $\mathbb{I}_{sub}$  component of *iGQ* indexed  $G$ 's features. Additionally, *iGQ* stored the results computed by  $\mathbb{M}_{sub}$  for  $G$ .

Fig. 4 depicts an example for the subgraph case of *iGQ*. A new query  $g$  is "sent" to method  $\mathbb{M}_{sub}$ 's graph index, producing a candidate set,  $CS(g)$ , which in this case contains the four graphs  $\{g_1, g_2, g_3, g_4\}$ . Similarly,  $g$  is "sent" to the *iGQ* subgraph component,  $\mathbb{I}_{sub}$ , from where it is determined that there exists a previous query  $G$ , such that  $g \subseteq G$ . *iGQ* then retrieves the answer set,  $Answer(G)$  (previously produced by method  $\mathbb{M}_{sub}$  and indexed by  $\mathbb{I}_{sub}$ ); in this case,  $Answer(G) = \{g_1, g_2\}$ . The reasoning then proceeds as follows. Consider graph  $g_1 \in CS(g)$ . Since from  $\mathbb{I}_{sub}$  it has been concluded that  $g \subseteq G$  and from the answer set of  $G$  we know that  $G \subseteq g_1$ , it necessarily follows that  $g \subseteq g_1$ . Similarly, we conclude that  $g \subseteq g_2$ . Hence, there is no point in testing  $g$  for subgraph isomorphism against  $g_1$  or  $g_2$ , as the answer is already known. Therefore, one can safely subtract graphs  $g_1, g_2$  from  $\mathbb{M}_{sub}$ 's candidate set, and test only the remaining graphs (reducing the number of subgraph isomorphism tests in this example by 50%). After the verification stage,  $g_1, g_2$  are added to the final answer set.

In the general case,  $g$  may be a subgraph of multiple previous query graphs  $G_i$  in  $\mathbb{I}_{sub}$ . Following the above reasoning, we can safely remove from  $CS(g)$  all graphs appearing in the answer sets of all query graphs  $G_i$ , as they are bound to be supergraphs of  $g$ ; that is, the set of graphs submitted by

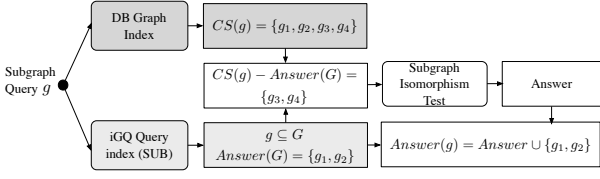


Figure 4: iGQ Processing of a Subgraph Query in the Subgraph Case

iGQ for subgraph isomorphism testing is given by:

$$CS_{sub}(g) = CS(g) \setminus \bigcup_{G_i \in \mathbb{I}_{sub}(g)} Answer(G_i) \quad (3)$$

Finally, if  $Answer_{sub}(g)$  is the subset of graphs in  $CS_{sub}(g)$  verified to be containing  $g$  through subgraph isomorphism testing, the final answer set for query  $g$  will be:

$$Answer(g) = Answer_{sub}(g) \cup \bigcup_{G_i \in \mathbb{I}_{sub}(g)} Answer(G_i) \quad (4)$$

LEMMA 1. *The iGQ answer in the subgraph case does not contain false positives.*

PROOF. Assume that a false positive was produced by iGQ; particularly, consider the first ever false positive produced by  $\mathbb{I}_{sub}$ , i.e., for some query  $g$ ,  $\exists G_{FP}$  such that  $g \not\subseteq G_{FP}$  and  $G_{FP} \in Answer(g)$ . Note that  $G_{FP}$  cannot be in  $Answer_{sub}(g)$ , as the latter contains only those graphs from  $CS_{sub}(g)$  that have been verified to be supergraphs of  $g$  after passing the subgraph isomorphism test, and hence  $g \not\subseteq G_{FP} \Rightarrow G_{FP} \notin Answer_{sub}(g)$ . Therefore, by formula (4),  $G_{FP} \in Answer(g) \Rightarrow \exists G$  such that  $G \in \mathbb{I}_{sub}(g)$  and  $G_{FP} \in Answer(G)$ . But (by formula (1))  $G \in \mathbb{I}_{sub}(g) \Rightarrow g \subseteq G$ , and  $G_{FP} \in Answer(G) \Rightarrow G \subseteq G_{FP}$ . Thus  $g \subseteq G_{FP}$  (a contradiction).  $\square$

LEMMA 2. *iGQ in the subgraph case does not introduce false negatives.*

PROOF. Assume that a false negative was produced by iGQ; particularly, consider the first ever false negative produced by  $\mathbb{I}_{sub}$ , i.e., for some query  $g$ ,  $\exists G_{FN}$  such that  $g \subseteq G_{FN}$  and  $G_{FN} \notin Answer(g)$ . As method  $\mathbb{M}_{sub}$  is assumed to be correct, it cannot produce any false negatives when processing query  $g$ , hence  $g \subseteq G_{FN} \Rightarrow G_{FN} \in CS(g)$ . Then, the only possibility for error is that  $G_{FN}$  was removed using formula (3); i.e.,  $G_{FN} \notin CS_{sub}(g)$ . That implies that  $\exists G$  such that  $G \in \mathbb{I}_{sub}(g)$  and  $G_{FN} \in Answer(G)$ . But then, by formula (4),  $G_{FN}$  will be added to  $Answer_{sub}(g)$  and thus  $G_{FN} \in Answer(g)$  (a contradiction).  $\square$

THEOREM 1. *The iGQ answer in the subgraph case of query processing is correct.*

PROOF. There are only two possibilities for error; iGQ can produce false negatives or false positives. The theorem then follows straightforwardly from Lemmas 1 and 2.  $\square$

#### 4.2.2 The Supergraph Case: $\mathbb{I}_{super}$

This case occurs when a new query  $g$  is a supergraph of a previous query  $G$ . Fig. 5 depicts an example for the supergraph case of iGQ. Again, the subgraph query processing method  $\mathbb{M}_{sub}$  produces a candidate set,  $CS(g)$  that, say, contains four graphs  $\{g_1, g_2, g_3, g_4\}$ . Running  $g$  through  $\mathbb{I}_{super}$ ,

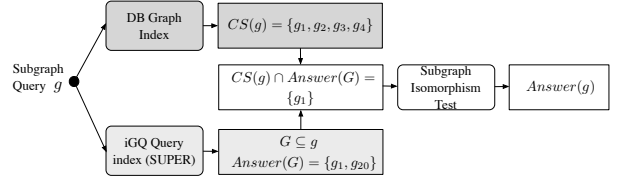


Figure 5: iGQ Processing of a Subgraph Query in the Supergraph Case

it is determined that there exists a previous query graph  $G$  such that  $G \subseteq g$ . Also  $\mathbb{I}_{super}$  supplies the stored answer set for  $G$ ,  $Answer(G) = \{g_1, g_2\}$ .

The reasoning then proceeds as follows. Consider graph  $g_2 \in CS(g)$ . We know from  $\mathbb{I}_{super}$  that  $g_2 \notin Answer(G)$ . Now, if  $g \subseteq g_2$  were to indeed be true, since  $G \subseteq g$ , then it must also hold that  $G \subseteq g_2$ ; that is,  $Answer(G)$  would have to contain  $g_2$  as well, which is a contradiction. Therefore, it is safe to conclude that  $g \not\subseteq g_2$  and thus  $g_2$  can be safely removed from  $CS(g)$ . Similarly, we can also safely remove graphs  $g_3, g_4$  from  $CS(g)$ , reducing in this case the number of required subgraph isomorphism tests by 75%. Thus, only  $g_1$  needs to be isomorphism-tested in this example.

In the general case,  $g$  may be a supergraph of multiple previous query graphs  $G_i$  in  $\mathbb{I}_{super}$ . By the above reasoning, only those graphs appearing in the answer sets of all queries  $G_i$  may actually be supergraphs of  $g$ ; thus the set of graphs submitted by iGQ for subgraph isomorphism testing is:

$$CS_{super}(g) = CS(g) \cap \bigcap_{G_i \in \mathbb{I}_{super}(g)} Answer(G_i) \quad (5)$$

The final answer produced for query  $g$  by iGQ,  $Answer(g)$ , will be the subset of graphs in  $CS_{super}(g)$  that have been verified by the subgraph isomorphism test.

LEMMA 3. *The iGQ answer in the supergraph case does not contain false positives.*

PROOF. This trivially follows by construction as all graphs in  $Answer(g)$  have passed through subgraph isomorphism testing at the final stage of processing.  $\square$

LEMMA 4. *The iGQ answer in the supergraph case does not introduce false negatives.*

PROOF. Assume false negatives are possible and consider the first ever false negative produced by  $\mathbb{I}_{super}$ ; i.e., for some query  $g$ ,  $\exists G_{FN}$  such that  $g \subseteq G_{FN}$  and  $G_{FN} \notin Answer(g)$ . Method  $\mathbb{M}_{sub}$  does not produce in its candidate set any false negatives (as will be formally proven shortly), hence  $G_{FN} \in CS(g)$ . Then, the only possibility for error is for iGQ to have removed graph  $G_{FN}$  from  $CS_{super}(g)$  with formula (5). This implies that  $\exists G$  such that  $G \in \mathbb{I}_{super}(g)$  and  $G_{FN} \notin Answer(G)$ . But since  $G \in \mathbb{I}_{super}(g)$ , by equation (2),  $G \subseteq g$ , and then  $g \subseteq G_{FN} \Rightarrow G \subseteq G_{FN} \Rightarrow G_{FN} \in Answer(G)$  (a contradiction).  $\square$

THEOREM 2. *The iGQ answer in the supergraph case of query processing is correct.*

PROOF. There are only two possibilities for error; iGQ can produce false negatives or false positives. The theorem then follows straightforwardly from Lemmas 3 and 4.  $\square$

### 4.3 iGQ and Optimal Performance

There are two special cases that warrant further emphasis, since they introduce the greatest possible benefits.

First, note that *iGQ* can easily recognize the case where a new query,  $g$ , is exactly the same as a previous query contained in  $\mathbb{I}$ . Specifically, this holds when  $\exists G \in \mathbb{I}$  such that  $g \subseteq G$  or  $g \supseteq G$ , and  $g$  and  $G$  have the same number of nodes and edges. When this holds, since  $\mathbb{I}$  stores the result for  $G$ , we can return directly and completely avoid the subgraph isomorphism testing as the actual result for  $g$  is known! As the subgraph isomorphism test dominates the query execution time, this is expected to be a large performance improvement.

Second, consider the supergraph part of *iGQ*. If  $\exists G \in \mathbb{I}_{super}(g)$  such that  $G \subseteq g$  and  $Answer(G) = \emptyset$ , then we can completely omit the verification stage again: If there were a dataset graph  $G'$  such that  $g \subseteq G'$ , since  $G \subseteq g$  we would conclude that  $G \subseteq G'$ , which necessarily implies that  $G' \in Answer(G)$ , which contradicts the fact that  $Answer(G) = \emptyset$ . Thus, no such graph  $G'$  can exist and it is safe to stop query processing at this stage.

### 4.4 iGQ and Supergraph Query Processing

As mentioned earlier, *iGQ* can expedite both subgraph and supergraph query processing. In the latter case, the components of *iGQ* ( $\mathbb{I}_{sub}$ ,  $\mathbb{I}_{super}$ ) remain unchanged, but the handling of the return answer sets is the exact inverse of what happens for subgraph queries. Briefly, given a supergraph query processing method  $\mathbb{M}_{super}$  and a supergraph query  $g$ , the union of the answer sets of graphs in  $\mathbb{I}_{super}(g)$  are removed from  $CS_{super}(g)$  and added to  $Answer_{super}(g)$  to produce the final answer, and the graphs not appearing in the intersection of the answer sets of graphs in  $\mathbb{I}_{sub}(g)$  are completely subtracted from  $CS_{sub}(g)$ . Also, the first optimal case mentioned above still holds, but the second optimal case is inversed with the processing terminating when  $\exists G \in \mathbb{I}_{sub}(g)$  such that  $Answer(G) = \emptyset$ . The intuition behind this design and the proof of correctness of *iGQ* for supergraph query processing, follow the same reasoning as above and are omitted for space reasons. The elegance afforded by the double use of *iGQ* is unique.

## 5. iGQ INDEX SPACE MANAGEMENT

As queries arrive continuously and the space to store  $\mathbb{I}$  is finite, *iGQ* requires methods for (i) efficiently handling this space and (ii) ensuring that it is best utilized, keeping those query graphs that increase its performance impact.

### 5.1 iGQ Graph Replacement Policy

Our replacement policy differs fundamentally from standard replacement policies: Unlike traditional cache replacement, whereby replacing a page or a file block saves one IO, different graphs in  $\mathbb{I}$  bring about different benefits, as is shown below. We identify three key principles.

**Increase the use of *iGQ* index.**  $\mathbb{I}$  should contain popular graphs; this is typical of all replacement algorithms. We define the popularity of a graph  $g$  as  $P(g) = \frac{H(g)}{M(g)}$ , where  $H(g)$  is the number of times a graph  $g \in \mathbb{I}$  has been found to be a subgraph or supergraph of query graphs (*hit*), and  $M(g)$  is the total number of all queries processed since  $g$  was added to the *iGQ* index. In essence, this models the fraction of queries affected over time by  $g$  being in  $\mathbb{I}$ .

**Reduce the number of subgraph isomorphism tests.** Ideal graphs for  $\mathbb{I}$  are graphs that bring about the greatest possible reductions in the number of executed subgraph isomorphism tests. Let  $R(g)$  be the total number of graphs removed from the candidate sets of incoming queries because of  $g$  being in  $\mathbb{I}$ . Then this component is computed as  $\frac{R(g)}{H(g)}$  – the per-*hit* average number of subgraph isomorphism tests alleviated by  $g$ .

**Reduce the cost of each subgraph isomorphism test.** A graph  $g \in \mathbb{I}$  is more desirable if it helps avoid subgraph isomorphism tests on the biggest graphs from  $\mathbb{M}$ 's *CS*. This is so since we also wish to remove from *CS* graphs with expensive subgraph isomorphism tests. We denote by  $C(g)$  the total cost of the subgraph isomorphism tests alleviated as a result of  $g$  being in  $\mathbb{I}$ . In order to estimate this value, we extend the asymptotic complexity analysis of [8] to the case of subgraph isomorphism. Specifically, given graphs with  $L$  labels, graph  $g'$  with  $n$  nodes, and graph  $G_i$  with  $N_i \geq n$  nodes, the cost  $c(g', G_i)$  of subgraph isomorphism of  $g'$  against  $G_i$  is given by:

$$c(g', G_i) = \frac{N_i \times N_i!}{L^{n+1} \times (N_i - n)!}$$

$C(g)$  is then computed as the sum over all  $c(g', G_i)$ , for all  $g'$  whose *CS*( $g'$ ) was reduced by removing  $G_i$  as a result of  $g$  being in  $\mathbb{I}$ , and  $\frac{C(g)}{R(g)}$  gives the average cost reduction per alleviated test.

Ideal graphs for  $\mathbb{I}$  are those that could help future queries as much as possible. To quantify such a contribution, we introduce the notion of graph *utility*,  $U(g)$ , defined as:

$$U(g) = \frac{H(g)}{M(g)} \times \frac{R(g)}{H(g)} \times \frac{C(g)}{R(g)} = \frac{C(g)}{M(g)}$$

That is, the utility of a graph  $g$  in *iGQ* is equal to the probability of  $g$  being used for an incoming query (i.e., being *hit*), times the average savings in number of subgraph isomorphism tests per such hit, times the average cost for a single subgraph isomorphism test. The replacement policy is then based on this, with the graph with the smallest  $U(g)$  being evicted.

### 5.2 iGQ Index Maintenance Policy

For all graphs in  $\mathbb{I}$  we maintain the metadata mentioned above (i.e.,  $C(g), M(g)$ ). Additionally, we store the actual query graphs that are indexed by  $\mathbb{I}$  in a separate store coined  $\mathbb{I}_{graphs}$ . To facilitate index updates without interfering with query processing performance, we employ the concepts of *query window size*,  $W$ , and *cache size*,  $\mathcal{C}$ , with  $W \leq \mathcal{C}$ . As new graph queries arrive, they are processed as outlined above, update the metadata for graphs in  $\mathbb{I}$ , and are inserted into a temporary storage  $\mathbb{I}_{temp}$ . When  $W$  new queries have been processed, we consult the metadata to locate the  $W$  graphs in  $\mathbb{I}$  with the lowest utility values. The graph data for those graphs is removed from  $\mathbb{I}_{graphs}$  and replaced by the graphs in  $\mathbb{I}_{temp}$ . The latter is then emptied, and a “shadow” index,  $\mathbb{I}_{shadow}$ , is built over graphs in  $\mathbb{I}_{graphs}$ . Incoming queries keep being served by  $\mathbb{I}$  and updating its metadata. When the shadow indexing is over,  $\mathbb{I}_{shadow}$  replaces  $\mathbb{I}$  (with a pointer swap). Finally, metadata for graphs removed from  $\mathbb{I}$  is also removed from the metadata store ( $C(g), M(g)$ ).

## 6. iGQ ALGORITHMS AND STRUCTURES

The proofs of correctness provided by the previous section, assume that  $\mathbb{I}_{sub}$  and  $\mathbb{I}_{super}$  provide correct results (recall formulas (1) and (2)). We shall now discuss the associated mechanisms and prove that they hold.

---

**Algorithm 1** The Supergraph Index in iGQ

---

```
1: Input: Set  $\mathbb{Q}$  of (previous) queries  $g_1, g_2, \dots, g_n$ 
2: Output: Supergraph index of previous queries  $\mathbb{I}_{super}$ 
3:
4: Initialize  $\mathbb{I}_{super}$  to an empty TRIE
5: for all  $g_i \in \mathbb{Q}$  do
6:   Extract all features of  $g_i$  and insert them in set  $F(g_i)$ 
7:    $NF[g_i] = |F(g_i)|$ 
8:   for all features  $f \in F(g_i)$  do
9:      $o =$  number of occurrences of  $f$  in  $g_i$ 
10:     $\mathbb{I}_{super}.insert(f, \{g_i, o\})$ 
11:   end for
12: end for
13: return  $\mathbb{I}_{super}$ 
```

---

### 6.1 Finding Supergraphs in $\mathbb{I}_{sub}$

This case represents a microcosm of our original problem, where instead of indexing and querying dataset graphs, we index and query previous query graphs. Hence, any approach from the related works can be adapted for this purpose. Actually, as iGQ can complement any existing approach,  $\mathbb{M}_{sub}$ , we can utilize  $\mathbb{M}_{sub}$ 's method for subgraph query processing for the subgraph case of iGQ, or any other method appropriate for iGQ's special characteristics (i.e., relatively small set of small graphs). Note that the assumed correct method  $\mathbb{M}_{sub}$  precludes false negatives and subgraph isomorphism testing of all candidates precludes false positives. Hence, formula (1)'s assumption is trivially satisfied.

### 6.2 Finding Subgraphs in $\mathbb{I}_{super}$

The problem of supergraph query processing has also received some attention (e.g., in [5, 44, 46, 6, 51]). In principle, any of these algorithms can be utilized for the task at hand within iGQ. However, we choose to propose a new approach, which is efficient yet simple and avoids the complexities and overheads involved in the above general approaches. The point is that we want a method for supergraph query processing that can easily fit within the framework of iGQ and perform both subgraph and supergraph query indexing and processing. Algorithm 1 shows how  $\mathbb{I}_{super}$  is created. Briefly,  $\mathbb{I}_{super}$  is a trie, storing features of queries. For each feature  $f$  it stores a pair  $\{g_i, o\}$  for each graph  $g_i$  in which  $f$  appears, where  $o$  is its number of occurrences in  $g_i$ . For each  $g_i$  it also stores the number of distinct features ( $NF[g_i]$ ) it contains.

Algorithm 2 illustrates how  $\mathbb{I}_{super}$  identifies candidates  $CS$  that are potential subgraphs of query  $g$ . The idea is to find those graphs that contain *only* features included in the query graph  $g$  (lines 19–22; the check for  $count(g_i)$  on line 20, ensures that all individual features of  $g_i$  are contained in  $g$ ), and where for each such graph  $g_i$  a feature  $f$  occurs at most as many times as  $f$  occurs in  $g$  (line 12). Last, the graphs in  $CS$  are isomorphically tested to verify that  $g_i \subseteq g$ .

It is straightforward to see that no false negatives can exist in  $CS$ . Assume there is a false negative  $g_i$  such that  $g_i \subseteq g$  and  $g_i \notin CS$ . Since  $g_i \subseteq g$ , any feature  $f$  in  $g_i$  appears no more times than  $f$  appears in  $g$ , thus  $g_i$  would be added to  $\mathbb{G}$  on every execution of line 12. As  $g_i \subseteq g$ , all of  $g_i$ 's features must appear in  $g$ . Thus,  $g_i$  would pass the if-clause at line 20 and be added to  $CS$  (contradiction). Moreover, subgraph isomorphism testing of all members of  $CS$  precludes false positives. Hence, formula (2)'s assumption holds.

---

**Algorithm 2** Supergraph Query Processing in iGQ

---

```
1: Input: Query graph  $g$  and  $\mathbb{I}_{super}$ 
2: Output: Candidate set  $CS$  of potential subgraphs of  $g$ 
3:
4: Initialize multiset  $\mathbb{G} = \emptyset$ 
5: Extract all features of query graph  $g$ ,  $F(g)$ 
6: for all features  $f \in F(g)$  do
7:    $O[f, g] =$  number of occurrences of  $f$  in  $g$ 
8: end for
9: for all features  $f \in F(g)$  do
10:  if  $f \in \mathbb{I}_{super}$  then
11:    for all  $\{g_i, o\} \in \mathbb{I}_{super}.get(f)$  do
12:      if  $o \leq O[f, g]$  then
13:         $\mathbb{G}.insert(g_i)$ 
14:      end if
15:    end for
16:  end if
17: end for
18: for all graphs  $g_i \in \mathbb{G}$  do
19:    $count(g_i) =$  number of occurrences of  $g_i$  in  $\mathbb{G}$ 
20:   if  $count(g_i) == NF[g_i]$  then
21:      $CS.add(g_i)$ 
22:   end if
23: end for
24: return  $CS$ 
```

---

### 6.3 iGQ System Operation

Fig. 6 depicts the complete iGQ system operation when used to expedite a subgraph query processing method  $\mathbb{M}_{sub}$ . Please keep in mind, though, that iGQ can be integrated with any subgraph and/or supergraph querying method. Given a new subgraph query  $g$ :

1. The query is sent to three separate processing threads in parallel and also stored in the query *window*.
2. In the first thread,  $\mathbb{M}_{sub}$  uses its *Dataset Graph Index* to filter the dataset graphs and produce the candidate set  $CS(g)$ , as usual.
3. The remaining two threads perform filtering along the subgraph (section 4.2.1) and supergraph path (section 4.2.2). Their results are combined to prune  $CS(g)$ , based on formulae (3) and (5).
4. The resulting candidate set,  $CS_{igq}(g)$ , undergoes subgraph isomorphism testing to produce  $Ans_{igq}(g)$ .
5. Since this is for a subgraph query, the graphs pruned during processing along the subgraph path in step 3 are added to  $Ans_{igq}(g)$  to produce the final answer set,  $Answer(g)$  (see formula (4)).
6. Metadata maintained throughout the processing of  $g$ , including  $Answer(g)$  and its subgraphs/supergraphs detected during step 3, are added to the metadata store,  $Stat(iGQ\_Graph)$ .
7. If the query window is full, the system uses the above metadata to select appropriate cached graphs to evict. Said graphs are replaced by the graphs in the window.
8. Finally, the iGQ index is updated to reflect the new contents of the cache (section 5.2 details the maintenance of the iGQ index).

## 7. PERFORMANCE EVALUATION

We have implemented the iGQ algorithms and report on experiments evaluating its performance on the savings of

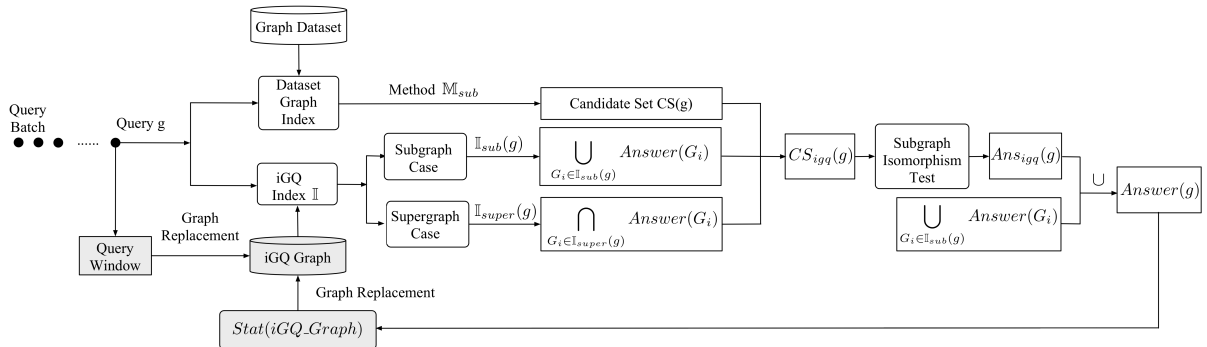


Figure 6: Operation of *iGQ* on Top of a Subgraph Query Processing Method  $M_{sub}$

subgraph query processing time (supergraph query processing time is omitted for space reason) and on the number of subgraph isomorphism tests.

## 7.1 Experimental Setup

Experiments were run on a Dell R920 system (4 Intel Xeon(R) CPUs (15 cores each), 512GB RAM, 1TB disk), and on a cluster of four Dell R720’s (each with 2 Intel Xeon(R) CPUs (8 cores each), 64GB RAM, 1TB disk).

**Algorithms.** In addition to our implementation of *iGQ* query processing, we also secured access to implementations of three recent high performing subgraph query processing methods, GraphGrepSX[3] (GGSX), Grapes[15], and CT-Index[22]. In addition to their competitive performance, these three methods represent interesting design decisions. GGSX indexes paths (up to a certain maximum length, equal to 4 in these experiments) and uses the VF2 subgraph isomorphism algorithm for its verification stage. Grapes, like GGSX, also indexes paths (of up to length of 4), but utilizes location information in the filtering stage to expedite the verification stage, essentially focusing only on connected components of the dataset graphs that may contain the query graph. CT-Index indexes trees (of maximum size 6), and cycles (of maximum size 8) in hash-based bitmap structures (4096-bit wide), and uses a modified VF2 for its verification stage.

The implementations for Grapes and GGSX were obtained from the corresponding project web sites[15, 3]. For Grapes, we present two alternatives, Grapes and Grapes(6), which use 1 and 6 threads respectively. For fairness, we altered the code of Grapes so to stop query processing when the first match was found, instead of looking for all matches of a query within each stored graph. For CT-Index we obtained the JAR file from one of the authors, which we then reverse-engineered to derive its code in Java. Subsequently, we integrated the *iGQ* algorithms of Section 4.2.1 within Grapes, CT-Index, and GGSX, yielding three different versions of *iGQ*, denoted as *iGQ\_Grapes*, *iGQ\_CT-Index*, and *iGQ\_GGSX*. In this way, (i) we validate our claim that *iGQ* can be incorporated into existing approaches, and (ii) we show that it can introduce significant performance gains during subgraph query processing of any of these approaches.

**Datasets.** We have employed three real-world datasets and one synthetic dataset with different characteristics, outlined in Table 1. AIDS is the Antiviral Screen Dataset of the National Cancer Institute, containing topological structures of molecules [30]. PDBS[19] is a dataset of graphs representing DNA, RNA and proteins. As AIDS and PDBS contain typical but relatively sparse graphs, we have per-

dataset	unique vertex labels	graphs in dataset	average node degree	num. nodes per graph			num. edges per graph		
				avg	std.dev	max	avg	std.dev	max
AIDS	62	40,000	2.09	45	22	245	47	23	250
PDBS	10	600	2.13	2,939	3,217	16,431	3,064	3,264	16,781
PPI	46	20	9.23	4,943	2,717	10,186	26,667	26,361	89,674
Synthetic	20	1,000	19.52	892	417	7,135	7,991	5	8,007

Table 1: Characteristics of Datasets

formed further experiments on dense datasets, including the PPI dataset and a synthetic dataset. PPI[15] models large and dense protein interaction networks and consists of 20 graphs. We also used the generator provided by [7] to create a much larger number (1,000) of much denser graphs.

**Query Workloads.** Unfortunately, despite the availability of graph datasets, the community does not enjoy well established benchmarks and/or real-world query logs for these datasets. So all related works synthesize queries derived from components of the dataset graphs. We follow this established principle for generating our workloads, whereby queries are generated from the original dataset graphs as follows. There are 3 key probability distributions to consider here. The first governs how a graph is selected from the dataset graphs. The second governs how a node is selected within this graph. Given these, we produce 4 query workloads: *uni-uni*, *uni-zipf*, *zipf-uni*, and *zipf-zipf*, with, e.g., *zipf-uni* denoting that dataset graphs have a popularity (probability of being selected) following a Zipf distribution, while nodes within the selected graph have a popularity drawn from a uniform distribution. The probability density function of the Zipf distribution is given by:  $p(x) = \frac{x^{-\alpha}}{\zeta(\alpha)}$ , where  $\zeta$  is the Riemann Zeta function[31]. The default value for  $\alpha$  was 1.4 – we have also used  $\alpha = 1.1$  representing a much smaller skew and  $\alpha = 2.0$  representing a stronger skew (as a reference point, web page popularities follow a Zipf with  $\alpha = 2.4$  [31]). The third governs the size of each graph query: query sizes are uniformly at random selected from 4, 8, 12, 16, 20 edges. Once a graph and a node within this graph have been selected, we then perform a BFS traversal of the latter’s neighborhood, with unvisited edges of each traversed node included in the generated graph, until the desired query size is reached.

For AIDS and PDBS, we ran 3,000 queries for each experiment. The first  $W$  of these queries were used to warm-up the index. We then used the remaining queries to measure the times and candidate set sizes with and without *iGQ* for each algorithm. By default we use a cache size  $C = 500$  and a batch window (and warm-up set) size  $W = 100$  queries – we have also used  $C = 1000, W = 200$  and  $C = 1500, W = 300$  with a 5,000-query workload to test cache size impact. We

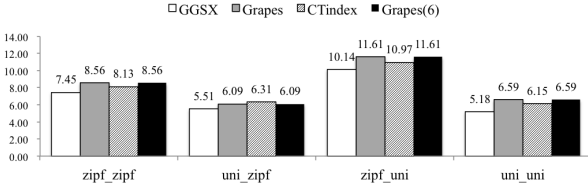


Figure 7: Speedup in Number of Subgraph Isomorphism Tests for AIDS

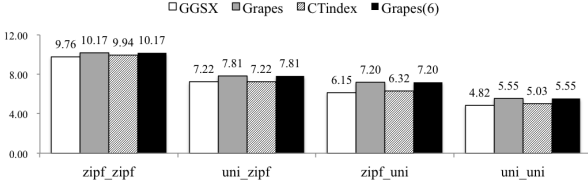


Figure 8: Speedup in Number of Subgraph Isomorphism Tests for PDBS

further tested *iGQ* against PPI and the synthetic dataset, in order to examine its performance under larger and denser graphs. In these cases, queries take 1-2 orders of magnitude more time to execute, hence for practical reasons we reduced the query workload to 500 queries. The batch window (and warm-up set) size were set to  $W = 20$  queries, with cache sizes of  $C = 100, 200, 300$  and Zipf skew  $\alpha = 1.4, 2.0, 2.4$ .

We report the speedup (reduction) achieved by *iGQ*, defined as the ratio of the average performance of the traditional method  $M$  over the average performance of  $iGQ_M$ , for the number of subgraph isomorphism tests and the query processing time.

## 7.2 Filtering Power Speedup

We first examine the *filtering power*, reflecting the speedup in the number of subgraph isomorphism tests performed. This metric facilitates a qualitative analysis of performance, independent of implementation and system details. Fig. 7 and 8 depict results for the AIDS and PDBS datasets respectively, across all four query workloads. The reduction in the number of subgraph isomorphism tests is evident (speedups of  $5\times$  to  $11\times$ ). Fig. 9 shows how Zipf skew  $\alpha$  affects this metric for the PDBS dataset, using one of the fastest methods (Grapes(6)). Results for the AIDS dataset and the other algorithms are similar and omitted for space reasons. As expected, with more skewness come increased benefits by *iGQ*.

Fig. 10 focuses on speedup across queries grouped by size (e.g., Q4 groups queries with 4 edges). As *iGQ* does not maintain separate caches per query size, the various query groups compete for the same space. Thus, some of them may seem to exhibit a lower speedup for larger cache sizes (e.g., the speedup of Q16 drops slightly when going from  $C = 200$  to 300); however, the speedup for the whole workload exhibited a steady rise (2.18, 2.45 and 2.53 for  $C = 100, 200$  and 300 respectively; figure omitted due to space reasons). Last, Fig. 11 depicts the results for the synthetic dataset.

## 7.3 Query Processing Speedup

Fig. 12 and 13 show the query processing time speedup for the AIDS/PDBS datasets. Interestingly, juxtaposing Fig. 12 against Fig. 7 (and Fig. 13 against Fig. 8) we see that reductions in the number of subgraph isomorphism tests do not directly translate into equal gains in query processing

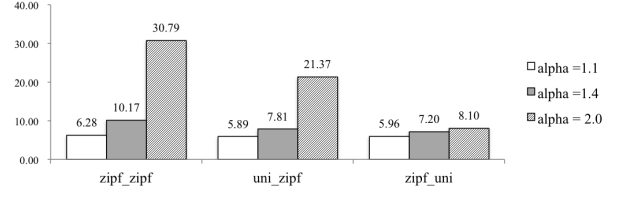


Figure 9: Speedup in Number of Subgraph Isomorphism Tests for PDBS/Grapes(6) vs Zipf Skew  $\alpha$

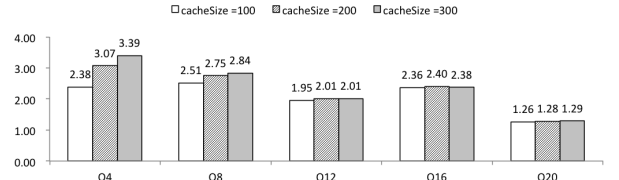


Figure 10: Speedup in Number of Subgraph Isomorphism Tests for PPI/Grapes(6)/zipf - zipf( $\alpha = 1.4$ )/Query Groups

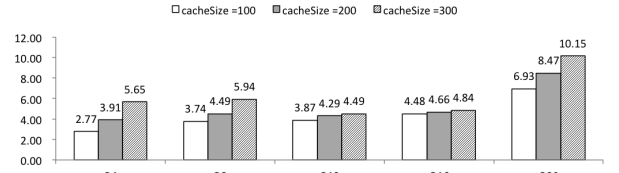


Figure 11: Speedup in Number of Subgraph Isomorphism Tests for Synthetic/Grapes(6)/zipf - zipf( $\alpha = 2.4$ )/Query Groups

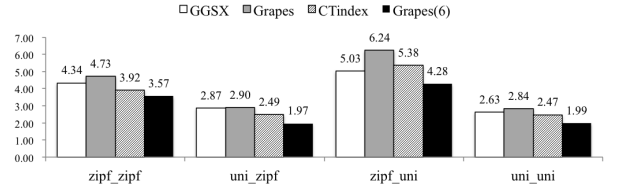


Figure 12: Speedup in Query Processing Time for AIDS

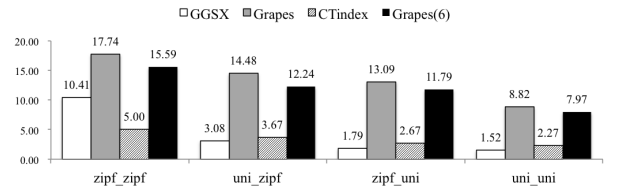


Figure 13: Speedup in Query Processing Time for PDBS

times. This is due to some large graphs in the candidate sets not being pruned away by the current index contents. We would expect this to be ameliorated as cache sizes increase. Indeed Fig. 14 shows this for Grapes(6) as cache size varies from 500 to 1,000 and 1,500 queries. Results for other cases are similar and omitted for space reasons.

Fig. 15 shows the impact of Zipf skew  $\alpha$  on query processing speedup for the Grapes(6) algorithms on the PDBS dataset. Again, with more skewness come greater benefits, up to impressive levels. Interestingly, juxtaposing Fig. 15 against Fig. 9 tells a different story. We see that reductions in the number of subgraph isomorphism tests translate into higher gains in query processing times. This is because of the replacement algorithm that maintains in the index those

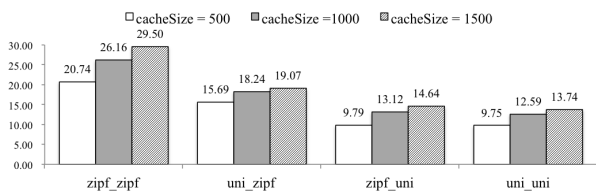


Figure 14: Speedup in Query Processing Time for PDBS/Grapes(6) vs Cache Size

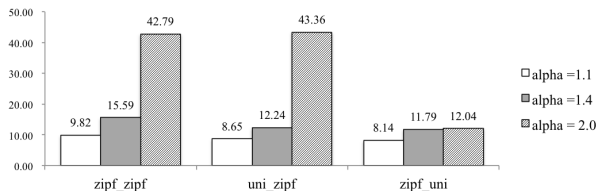


Figure 15: Speedup in Query Processing Time for PDBS/Grapes(6) vs Zipf Skew  $\alpha$

query graphs  $g$  with the higher utility; i.e., which help to avoid expensive subgraph isomorphism tests against graphs in the candidate sets. Fig. 16 and Fig. 17 show the speedup for the query processing time, corresponding to Fig. 10 and Fig. 11, respectively.

Last, Fig. 18 plots the index size for  $iGQ$  for  $C = 500$  graph queries, versus that of the three algorithms we’ve considered so far, for the AIDS dataset. In the default configurations,  $iGQ$  adds a negligible space overhead on top of the base indexes (less than 1%). In addition to the default configurations for said algorithms, Fig. 18 also plots the index sizes for the immediately larger configurations (i.e., for max path length of 5 for Grapes and GGSX, and for trees of size 7, cycles of size 9, and 8192 bits per bitmap for CT-Index). Note that this minimal increase in the feature size results in almost double the space requirements for the base indexes. On the other hand, these larger indexes bring a performance improvement of less than 10% in all cases (figure omitted due to space reasons), which is virtually negligible when compared to the gains provided by  $iGQ$ .

Overall,  $iGQ$  is shown to introduce significant to impressive performance gains, against the state of the art methods in the literature. We have actually conducted a detailed performance evaluation of most related algorithms[21] and selected GGSX, Grapes(1), Grapes(6), and CT-Index as those showing the best performance. Regardless of the method, when incorporating  $iGQ$  with it, large performance gains ensue. These gains are robust and are manifested in all four different query workloads we have presented and, most importantly, with a minimal space overhead.

## 8. CONCLUSIONS

We have presented a novel perspective and solution to the graph querying problem, departing from related work in three ways: First, it constructs query indexes, as opposed to simply relying on dataset graph indexes. Second, it maintains the knowledge the system produced when executing previous queries. Third, it can be used to expedite both subgraph and supergraph queries. We showed how these can help improve the performance of future queries and provided formal proof of correctness. The proposed  $iGQ$  framework consists of (i) a subgraph index, (ii) a supergraph index, (iii) a method for efficiently maintaining the index,

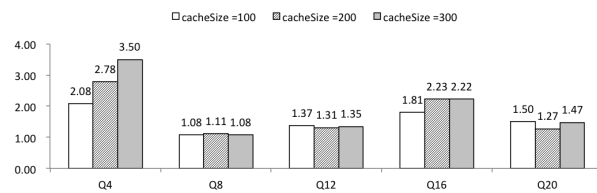


Figure 16: Speedup in Query Processing Time for PPI/Grapes(6)/zipf - zipf( $\alpha = 1.4$ )/Query Groups

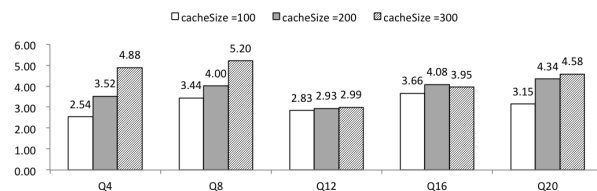


Figure 17: Speedup in Query Processing Time for Synthetic/Grapes(6)/zipf - zipf( $\alpha = 2.4$ )/Query Groups

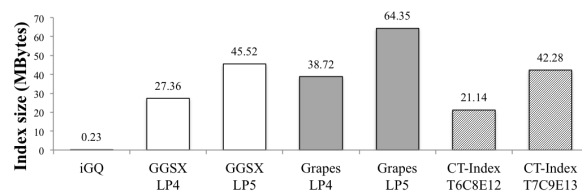


Figure 18: Absolute Index Sizes (in MBytes) for AIDS

including a graph replacement policy, and (iv) any popular method for indexing and processing subgraph or supergraph queries. We incorporated  $iGQ$  within 3 popular methods from related work, showcasing its wide applicability. Last, our performance evaluation on both real-world and synthetic datasets with various query workloads showed  $iGQ$ ’s significant performance gains and negligible space overhead.

## 9. REFERENCES

- [1] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using Map-Reduce. In *Proc. IEEE ICDE*, pages 62–73, 2013.
- [2] A. Balmin, F. Ozcan, S. K. Beyer, J. R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *Proc. VLDB*, pages 60–71, 2004.
- [3] V. Bonnici, et al. Enhancing graph database indexing by suffix tree structure. In *Proc. IAPR PRIB*, 2010. <http://cs.nyu.edu/shasha/papers/graphgrep/>.
- [4] R. V. Bruggen. *Learning Neo4j*. O’Reilly Media, 2013.
- [5] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu. Towards graph containment search and indexing. In *Proc. VLDB*, 2007.
- [6] J. Cheng, Y. Ke, A. W.-C. Fu, and J. X. Yu. Fast graph query processing with a low-cost index. *VLDBJ*, 20(4):521–539, 2010.
- [7] J. Cheng, Y. Ke, W. Ng, and A. Lu. FG-index: towards verification-free query processing on graph databases. In *Proc. ACM SIGMOD*, 2007.
- [8] L. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the VF graph matching algorithm. In *Proc. ICIAP*, 1999.
- [9] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento.

- A (sub) graph isomorphism algorithm for matching large graphs. *IEEE TPAMI*, 26(10):1367–1372, 2004.
- [10] W. de Nooy, A. Mrvar, and V. Batagelj. *Exploratory Social Network Analysis with Pajek*. Cambridge University Press, 2005.
- [11] K. Degtyarenko, J. Hastings, P. de Matos, and M. Ennis. Chebi: An open bioinformatics and cheminformatics resource. *Curr. Protoc. Bioinformatics*, 14(26):1–20, 2009.
- [12] R. Di Natale, A. Ferro, R. Giugno, M. Mongiovì, A. Pulvirenti, and D. Shasha. Sing: Subgraph search in non-homogeneous graphs. *BMC bioinformatics*, 11(1):96, 2010.
- [13] M. Elseidy, E. Abdelhamid, S. Skiadopoulou, and P. Kalnis. GRAMI: Frequent subgraph and pattern mining in a single large graph. *PVLDB*, 7(7), 2014.
- [14] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.
- [15] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. *PLoS One*, 8(10):e76911, 2013. <http://ferrolab.dmi.unict.it/GRAPES/>.
- [16] R. Giugno and D. Shasha. GraphGrep: A fast and universal method for querying graphs. In *Proc. IEEE ICPR*, 2002.
- [17] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. iGraph: A framework for comparisons of disk-based graph indexing techniques. *PVLDB*, 3(1-2):449–459, 2010.
- [18] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *Proc. IEEE ICDE*, 2006.
- [19] Y. He, et al. Structure of decay-accelerating factor bound to echovirus 7: a virus-receptor complex. *PNAS*, 99:10325–10329, 2002.
- [20] InfiniteGraph. <http://www.objectivity.com/infinitegraph>.
- [21] F. Katsarou, N. Ntarmos, and P. Triantafillou. Performance and scalability of indexed subgraph query processing methods. *PVLDB*, 8(12), 2015.
- [22] K. Klein, N. Kriege, and P. Mutzel. CT-index: Fingerprint-based graph indexing combining cycles and trees. In *Proc. IEEE ICDE*, 2011.
- [23] G. Koloniari and E. Pitoura. Partial view selection for evolving social graphs. In *Proc. GRADES*, 2013.
- [24] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in MapReduce. *PVLDB*, 2015.
- [25] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.
- [26] K. Lillis and E. Pitoura. Cooperative XPath caching. *SIGMOD’08*, pages 327–338, 2008.
- [27] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. In *Proc. ESEC/FSE*, 2005.
- [28] G. Malewicz, et al. Pregel: a system for large-scale graph processing. In *Proc. ACM PODC*, 2009.
- [29] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *Proc. VLDB*, pages 469–480, 2005.
- [30] NCI - DTP AIDS antiviral screen dataset. [http://dtp.nci.nih.gov/docs/aids/aids\\_data.html](http://dtp.nci.nih.gov/docs/aids/aids_data.html).
- [31] M. Newman. Power laws, Pareto distributions and Zipf’s law. *Contemporary Physics*, 46:323–351, 2005.
- [32] E. Petras and C. Faloutsos. Similarity searching in medical image databases. *IEEE TKDE*, 9(3), 1997.
- [33] T. Plantenga. Inexact subgraph isomorphism in MapReduce. *J. Parallel Distrib. Comput.*, 73:164–175, 2013.
- [34] PubChem. <https://pubchem.ncbi.nlm.nih.gov/>.
- [35] K. Semertzidis and E. Pitoura. Durable graph pattern queries on historical graphs. In *Proc. IEEE ICDE*, 2016 (to appear).
- [36] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.
- [37] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *Proc. IEEE ICDE*, 2008.
- [38] Twitter FlockDB. <https://github.com/twitter/flockdb>.
- [39] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.
- [40] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *Proc. IEEE ICDE*, 2007.
- [41] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proc. ACM SIGMOD*, 2004.
- [42] X. Yan, F. Zhu, P. S. Yu, and J. Han. Feature-based similarity search in graph structures. *ACM TODS*, 31(4):1418–1453, 2006.
- [43] D. Yuan and P. Mitra. Lindex: a lattice-based index for graph databases. *VLDBJ*, 22(2):229–252, 2013.
- [44] D. Yuan, P. Mitra, and C. L. Giles. Mining and indexing graphs for supergraph search. *PVLDB*, 6(10), 2013.
- [45] S. Zhang, M. Hu, and J. Yang. TreePi: A Novel Graph Indexing Method. In *Proc. IEEE ICDE*, 2007.
- [46] S. Zhang, J. Li, H. Gao, and Z. Zou. A novel approach for efficient supergraph query processing on graph databases. In *Proc. EDBT*, 2009.
- [47] S. Zhang, J. Yang, and W. Jin. Sapper: subgraph indexing and approximate matching in large graphs. *PVLDB*, 3(1-2):1185–1194, 2010.
- [48] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 2010.
- [49] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta >= graph. In *Proc. VLDB*, 2007.
- [50] X. Zhao, et al. Efficient processing of graph similarity queries with edit distance constraints. *VLDBJ*, 22(6):727–752, Dec 2013.
- [51] G. Zhu, X. Lin, W. Zhang, W. Wang, and H. Shang. Prefindex : An efficient supergraph containment search technique. In *Proc. SSDBM*, 2010.
- [52] Y. Zhu, J. Yu, and L. Qin. Leveraging graph dimensions in online graph search. *PVLDB*, 8(1), 2014.
- [53] L. Zou, L. Chen, J. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *Proc. ACM EDBT*, 2008.



# GSCALER: Synthetically Scaling A Given Graph

J.W. Zhang  
National University of Singapore  
jiangwei@nus.edu.sg

Y.C. Tay  
National University of Singapore  
dcstayc@nus.edu.sg

## ABSTRACT

Enterprises and researchers often have datasets that can be represented as graphs (e.g. social networks). The owner of a large graph may want to scale it down to a smaller version, e.g. for application development. On the other hand, the owner of a small graph may want to scale it up to a larger version, e.g. to test system scalability. This paper investigates the *Graph Scaling Problem* (GSP):

*Given a directed graph  $G$  and positive integers  $\tilde{n}$  and  $\tilde{m}$ , generate a similar directed graph  $\tilde{G}$  with  $\tilde{n}$  nodes and  $\tilde{m}$  edges.*

This paper presents a graph scaling algorithm GSCALER for GSP. Analogous to DNA shotgun sequencing, GSCALER, decomposes  $G$  into small pieces, scales them, then uses the scaled pieces to construct  $\tilde{G}$ . This construction is based on the indegree/outdegree correlation of nodes and edges.

Extensive tests with real graphs show that GSCALER is scalable and, for many graph properties, it generates a  $\tilde{G}$  that has greater similarity to  $G$  than other state-of-the-art solutions, like Stochastic Kronecker Graph and *UpSizeR*.

## 1. INTRODUCTION

The emergence of online social networks, like Facebook and Twitter, has attracted considerable research. However, their enormous sizes make any experiment on the entire graph impractical. It is therefore often necessary to obtain a smaller version of the graph for experiments. We call this the **scaling down** problem.

At the other end of the scale, a new social network service provider may have a small graph, but wants to test the scalability of their system. They may therefore want to have a larger (and necessarily) synthetic version of their current empirical graph. We call this the **scaling up** problem.

These two problems arise in other contexts as well, e.g. where the graph represents router topology or web links. They illustrate the *Graph Scaling Problem* (GSP):

*Given a directed graph  $G$  and positive integers  $\tilde{n}$  and  $\tilde{m}$ , generate a similar directed graph  $\tilde{G}$  with  $\tilde{n}$  nodes and  $\tilde{m}$  edges.*

There are many possible ways to define “similarity”, depending on the context, but we believe the definitions must all be in terms of graph properties, like indegree distribution, clustering coefficient, effective diameter, etc.

However, it is impossible for  $G$  and  $\tilde{G}$  to have exactly the same properties; e.g. if  $G$  and  $\tilde{G}$  have the same degree distributions, then the larger graph must necessarily have smaller density. One must therefore select the graph properties that are to be preserved when scaling. GSP facilitates this selection by allowing  $\tilde{n}$  and  $\tilde{m}$  to be specified separately.

Related work in the literature have objectives that are different from GSP. There are many papers on graph sampling, such as gSH, BFS, forest fire and frontier sampling [1, 3, 18, 21, 23, 30, 35]. They can be viewed as examples of scaling down, since they produce a  $\tilde{G}$  that is a subgraph of  $G$ ; this can have data protection issues that do not arise if  $\tilde{G}$  is synthetic. Moreover, graph sampling cannot generate a  $\tilde{G}$  that is larger than  $G$ .

Other related work use generative models that can produce a  $\tilde{G}$  that is smaller or larger than  $G$ . For example, an Erdős-Rényi model generates a graph of any size  $n$  with a specified edge probability  $p$  [9]; Chung and Lu’s model generates graphs with a specified degree distribution [4]; and Stochastic Kronecker Graphs [19,20] are generated from an initiator by applying Kronecker product. However, these do not allow a choice of both  $\tilde{n}$  and  $\tilde{m}$ .

In contrast, we propose GSCALER, a solution to GSP that deviates from previous work by using a technique that is analogous to DNA shotgun sequencing [31]. The latter breaks a long DNA strand into smaller ones that are easier to sequence, then use these smaller sequences to reconstruct the sequence in the original strand.

Similarly, GSCALER (i) breaks the given  $G$  into two sets  $S_{in}$  and  $S_{out}$  of small pieces, (ii) scales them by size to  $\tilde{S}_{in}$  and  $\tilde{S}_{out}$ ; (iii) merges these pieces to give a set  $\tilde{S}_{bi}$  of larger pieces, then (iv) assembles  $\tilde{G}$  from the pieces in  $\tilde{S}_{bi}$ .

This paper makes the following contributions:

1. We present GSCALER, an algorithm for solving GSP.
2. We prove that GSCALER (i) does not generate multiple edges between two nodes, and (ii) has small degree distribution error even when the average degree of  $\tilde{G}$  differs from that of  $G$ .

3. We present experiments that compare GSCALER to 4 other techniques, using 2 real graphs and 7 properties.

We begin by surveying related work in Sec. 2. We describe GSCALER in Sec. 3, and prove that it does not generate multiple edges between any two nodes. Sec. 4 reviews the graph properties, state-of-the-art algorithms and datasets that are used for comparison. Sec. 5 then proves that GSCALER has small error, and presents the experimental comparison to other algorithms. We discuss the choice of  $\tilde{n}$  and  $\tilde{m}$  in Sec. 6, before Sec. 7 concludes with a summary.

## 2. RELATED WORK

The closest work in graph scaling from the literature are graph sampling algorithms and generative models.

For graph sampling, the main approaches are node-based sampling, edge-based sampling and traversal-based sampling which produce a subgraph of the original graph  $G$ .

Node-based sampling selects a set of nodes  $V_{sub}$  from  $G$ , then  $\tilde{G}$  is just the induced graph of this set of nodes  $V_{sub}$ . Authors in [32] pointed out that node-based sampling may not preserve a power law degree distribution because of bias induced by high degree nodes.

Similarly, traditional edge-based sampling selects edges randomly. However, this might result in a sparsely connected graph  $\tilde{G}$  [21]. Some other edge-based sampling variants [1, 16, 21] sample the graph using edge selection/deletion and combination with node selection/deletion.

Most graph sampling techniques focus on traversal-based sampling [14]. Breadth first sampling (BFS) [3, 18, 35] and random walk sampling (RW) [12, 30] are the most basic and well-known algorithms. Similar to BFS, snow ball sampling [13] (SBS) is widely used in sociology studies. Metropolis-Hastings Random Walk (MHRW) [12, 27] is a Markov-Chain Monte Carlo algorithm which samples unbiased subgraph in undirected social graphs. However, MHRW suffers from *sample-rejection problem*. Later, rejection-controlled Metropolis-Hastings (RCMH) [26] is proposed to reduce the sample-rejection ratio.

Frontier sampling is a multi-dimensional random walk which results in better estimators for some graph properties [30]. A probabilistic version of SBS, forest fire ( $FF$ ) [21, 23] captures some important observations in real social networks, e.g. small diameter.

Most traversal-based sampling requires random access to a node's neighbors, which might not be feasible for large graphs (that cannot fit into memory). Hence, streaming graph sampling algorithms are proposed, e.g. induced edge sampling (ES-i) [2]. As mentioned previously, graph sampling algorithms are limited to *scaling down* problem.

For generative models, the Erdős-Rényi model generates a graph of any size  $n$  with a specified edge probability  $p$  [9]. There are variants of random models that generate graphs with specific graph properties [4, 28], e.g. the Chung-Lu model generates graphs with a specified degree distribution.

One group of generative models [5, 6, 11, 17] employ the strategy of *preferential attachment*. They obey a simple rule: a new node  $u$  attaches to the graph at each time step, and adds an edge  $e_{uv}$  to an existing node  $v$  with a probability  $p$  proportional to the degree of the node  $v$ .

Another type of generative models is *recursive matrix* model [7, 19, 20], which recursively multiplies the adjacency matrix. For example, Stochastic Kronecker Graph (SKG) [20]

Notation	Description
$G(V, E)$	original graph
$\tilde{G}(V, E)$	scaled graph
$n/\tilde{n}$	number of nodes in $G/\tilde{G}$
$m/\tilde{m}$	number of edges in $G/\tilde{G}$
$f_{in}/\tilde{f}_{in}$	$G/\tilde{G}$ 's indegree distribution
$f_{out}/\tilde{f}_{out}$	$G/\tilde{G}$ graph's outdegree distribution
$f_{bi}/\tilde{f}_{bi}$	$G/\tilde{G}$ graph's bidegree distribution
$f_{corr}/\tilde{f}_{corr}$	$G/\tilde{G}$ graph's edge correlation distribution
$S_{in}/\tilde{S}_{in}$	set of pieces with incoming edges in $G/\tilde{G}$
$S_{out}/\tilde{S}_{out}$	set of pieces with outgoing edges in $G/\tilde{G}$
$\tilde{S}_{bi}$	set of pieces with incoming and outgoing edges in $\tilde{G}$
$ct_{\Delta}$	count function of the pieces in set $\Delta$ , $\Delta$ can be $S_{in}, \tilde{S}_{in}$ and so on.
$I(\alpha')/O(\alpha')$	total number of available incoming/outgoing edges for nodes with bidegree $\alpha'$
$D(f, \tilde{f})$	KS-D statistics of between $f$ and $\tilde{f}$

Table 1: Notation

recursively multiplies the graph initiator  $K_1$  through Kronecker product, which results in a self-similar graph. SKG captures most social network properties, such as small diameter and power law degree distribution.

Scaling problem appears in other fields as well. For example, *UpSizeR* is a pioneer tool which synthetically scales a relational dataset [33]. *UpSizeR*'s focus is on preserving correlation among tuples from multiple tables. In relational terms, *GSP* requires preservation of correlation among tuples in a single table (for the edges).

For Resource Description Framework (RDF), the AO benchmark [8] is the first tool that scales down an RDF dataset. Later, RBench [29] is proposed to both scale down and up. However, these two benchmarks are evaluated with different metrics (*dataset coherence, relationship specialty, literal diversity*), so it would be unfair to use them for comparison.

## 3. GRAPH SCALER (GSCALER)

Given a graph  $G(V, E)$ ,  $|V|$  and  $|E|$  may need to scale by different factors to maintain similarity for certain properties (e.g. density). Hence, GSCALER allows the user to specify the target  $\tilde{n}$  and  $\tilde{m}$ . As shown in Fig.1, the scaling has the following 4 steps:

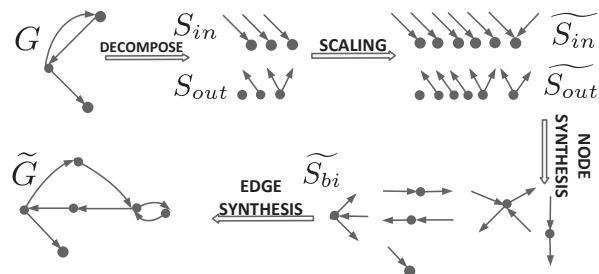


Figure 1: The 4 steps in Gscaler.

GSCALER first decomposes  $G$  into 2 sets  $S_{in}$  and  $S_{out}$ .  $S_{in}$  consists of **pieces**, where a piece is a node with its incoming edges (minus the source nodes). Similarly,  $S_{out}$  consists of

---

**Algorithm 1:** GSCALER( $G, \tilde{n}, \tilde{m}$ )

---

```
1  $S_{in}, S_{out}, f_{bi}, f_{corr} = \text{DECOMPOSE}(G)$ 
2  $\widetilde{S}_{in} = \text{SCALING}(S_{in}, \tilde{n}, \tilde{m})$ 
3  $\widetilde{S}_{out} = \text{SCALING}(S_{out}, \tilde{n}, \tilde{m})$ 
4  $\widetilde{S}_{bi} = \text{NODE\_SYNTHESIS}(\widetilde{S}_{in}, \widetilde{S}_{out}, f_{bi})$ 
5  $\widetilde{G} = \text{EDGE\_SYNTHESIS}(\widetilde{S}_{bi}, f_{corr})$ 
```

---

pieces, where a piece is a node with its outgoing edges (minus the target nodes). Each node in  $G$  generates 2 pieces, one in  $S_{in}$  and one in  $S_{out}$ .

After that, GSCALER scales  $S_{in}$  and  $S_{out}$  to get the scaled sets of pieces  $\widetilde{S}_{in}$  and  $\widetilde{S}_{out}$  with  $\tilde{n}$  nodes and  $\tilde{m}$  edges.

In node synthesis,  $\widetilde{S}_{in}$  and  $\widetilde{S}_{out}$  are used to form a set  $\widetilde{S}_{bi}$  of larger pieces. Each new piece has a node with incoming edges (with no source nodes) and outgoing edges (with no target nodes); it will generate one node in  $\widetilde{G}$ .

The last step (edge synthesis) is to link pieces in  $\widetilde{S}_{bi}$  which finally results in  $\widetilde{G}$ . Edge synthesis is similar to a jigsaw puzzle, where you want to fit the small pieces together. Algo. 1 summaries the workflow for GSCALER.

In the following, we will explain each step in detail, and use a running example to show how  $G$  is scaled to  $\widetilde{G}$  in Fig.1. As presented in Fig.1,  $n = 3$ ,  $\tilde{n} = 6$ ,  $m = 3$ ,  $\tilde{m} = 7$ .

### 3.1 DECOMPOSE

This step is straightforward, and Fig. 1 shows the pieces in  $S_{in}$  and  $S_{out}$  from decomposing  $G$ .

### 3.2 SCALING

To simplify the explanation, we just use  $S_{in}$  for demonstration. Let  $N = \{0, 1, 2, 3, \dots\}$ . We use the *count* function  $ct$  to denote  $k$  pieces in  $\Delta$  has property  $x$ :

$$ct_{\Delta}(x) = k, k \in N \quad (1)$$

For example,  $ct_{S_{in}}(x) = k$  means  $k$  pieces in  $S_{in}$  have indegree  $x$ . For both  $S_{in}$  and  $S_{out}$ , the scaling process takes the following three steps:

- **node scaling.** For indegree  $x$ , let  $A_x = ct_{S_{in}}(x) \times \frac{\tilde{n}}{n}$ . As  $A_x$  might not be an integer, we round-off  $\widetilde{S}_{in}$  as follows:

$$ct_{\widetilde{S}_{in}}(x) = \begin{cases} \lfloor A_x \rfloor & \text{with probability } A_x - \lfloor A_x \rfloor \\ \lceil A_x \rceil & \text{with probability } \lceil A_x \rceil - A_x \end{cases} \quad (2)$$

For example, if  $A_x = 4.8$ , then with 0.8 probability,  $ct_{\widetilde{S}_{in}}(x) = 5$ , and with 0.2 probability,  $ct_{\widetilde{S}_{in}}(x) = 4$ . Hence, Eq.2 can be rewritten as

$$E[ct_{\widetilde{S}_{in}}(x)] = ct_{S_{in}}(x) \times \frac{\tilde{n}}{n} \quad (3)$$

Fig.2 shows the  $\widetilde{S}_{in}$  and  $\widetilde{S}_{out}$  that we get after this scaling.

- **node adjustment.** With randomness in node scaling (Eq.2), we may have  $|\widetilde{S}_{in}| \neq \tilde{n}$ . If so,  $|\tilde{n} - |\widetilde{S}_{in}||$  nodes



**Figure 2:**  $\widetilde{S}_{in}, \widetilde{S}_{out}$  after node scaling for  $G$  in Fig.1.

---

**Algorithm 2:** SCALING( $S_{in}, \tilde{n}, \tilde{m}$ )

---

```
1 node_scaling( $S_{in}, \tilde{n}$ )
2 node_adjustment( $\widetilde{S}_{in}, \tilde{n}, S_{in}$ )
  /*  $h_0/l_0$  is the upper/lower bound where x
  varies. By default,  $h_0/l_0$  should be the
  highest/lowest degree in the graph.  $h_0/l_0$ 
  will be further extended if needed. t is the
  edge difference threshold where the loop
  stops.  $ct(x)$  refers to  $ct_{\widetilde{S}_{in}}(x)$ . */
3 initialize  $h = h_0, l = l_0, t$ 
5 while  $|\sum_x ct(x) \times x - \tilde{m}| > t$  do
6   if  $l >= h$  then
7      $l = l_0; h = h_0;$ 
8   if  $\tilde{m} > \sum_x ct(x) \times x$  then
9     if  $ct(l) > 0$  then
10       $ct(l) --; ct(h) ++;$ 
11       $l ++; h --;$ 
12    else  $l ++;$ 
13  else
14    if  $ct(h) > 0$  then
15       $ct(l) ++; ct(h) --;$ 
16       $l ++; h --;$ 
17    else  $h --;$ 
18 adjust  $|\sum_x ct(x) \times x - \tilde{m}|$  edges
```

---

with random degree are added to or removed from  $\widetilde{S}_{in}$ . For  $G$  in Fig. 1, such adjustment is not needed.

- **edge adjustment.** Next, the number of scaled edges must equal to  $\tilde{m}$ , i.e.  $\sum_x ct_{\widetilde{S}_{in}}(x) \times x = \tilde{m}$ .

If  $\sum_x ct_{\widetilde{S}_{in}}(x) \times x < \tilde{m}$ , we increase the number of high degree nodes, and decrease the number of low degree nodes. If  $\sum_x ct_{\widetilde{S}_{in}}(x) \times x > \tilde{m}$ , we decrease the number of high degree nodes, and increase the number of low degree nodes. The details are shown in Algo. 2.

In our running example,  $\sum_x ct_{\widetilde{S}_{in}}(x) \times x = 6 < 7 = \tilde{m}$ . Hence, we increase the number of high degree nodes (indegree=2), and decrease the number of low degree (indegree=1) nodes. Thus, we have 1 node with indegree=2 and 5 nodes with indegree=1 for  $\widetilde{S}_{in}$ . After the edge adjustment, the correct  $\widetilde{S}_{in}, \widetilde{S}_{out}$  are shown in Fig. 1.

### 3.3 NODE SYNTHESIS

Now we have  $\widetilde{S}_{in}$  and  $\widetilde{S}_{out}$ , and we match 1 piece in  $\widetilde{S}_{in}$  to 1 piece in  $\widetilde{S}_{out}$  and merge them to give a larger piece. This synthesis follows a *bidegree* distribution  $f_{bi} : N^2 \rightarrow [0, 1]$ , where  $f_{bi}(d_1, d_2) = z$  means a fraction  $z$  of nodes have bidegree  $(d_1, d_2)$ . We say a node  $u$  has bidegree  $(d_1, d_2)$  if it has indegree= $d_1$  and outdegree= $d_2$ . For  $G$  in Fig. 1, the corresponding  $f_{bi}$  is listed in Table 2.

GSCALER loops through  $f_{bi}(d_1, d_2)$  to synthesize nodes. However, for a desired bidegree  $(d_1, d_2)$ ,  $\widetilde{S}_{in}$  and  $\widetilde{S}_{out}$  may not have the necessary pieces. Hence, a *neighboring*  $(d_1', d_2')$  will be used. GSCALER uses a greedy heuristic that matches pieces by minimizing the Manhattan distance

$$\|(d_1, d_2) - (d_1', d_2')\|_1 = |d_1 - d_1'| + |d_2 - d_2'|.$$

---

**Algorithm 3: NODE SYNTHESIS** ( $\widetilde{S}_{in}, \widetilde{S}_{out}, f_{bi}$ )

---

```

1 while  $\widetilde{S}_{in}$  and  $\widetilde{S}_{out}$  not empty do
2   for  $f_{bi}(d_1, d_2)$  do
3     while  $[f_{bi}(d_1, d_2) \times \widetilde{m}] > 0$  do
4        $(d_1', d_2') \leftarrow \text{Manhattan}(d_1, d_2)$ 
5        $\widetilde{S}_{bi} \leftarrow (d_1', d_2')$ 
6       update  $\widetilde{S}_{in}, \widetilde{S}_{out}, f_{bi}(d_1, d_2)$ 

```

---

For Table 2, when GSCALER sees  $f_{bi}(1, 0) = \frac{1}{3}$ , it will first generate 1 node with bidegree (1, 0), and generate the other node with bidegree (1, 1). Next, GSCALER sees  $f_{bi}(1, 1) = \frac{1}{3}$ , two nodes both having bidegree (1, 1) are generated. Lastly, GSCALER sees  $f_{bi}(1, 2) = \frac{1}{3}$ , and two nodes with bidegree (1, 2), and (2, 2) are generated.

Algo.3 summarizes the node synthesis. The synthesized pieces for  $\widetilde{S}_{bi}$  in  $\widetilde{G}$  are shown in Fig. 1. Note that each piece in  $\widetilde{S}_{bi}$  maps to a node in  $\widetilde{G}$ .

### 3.4 EDGE SYNTHESIS

Now we are almost done with the graph scaling, we only need to link the edges. This is similar to a jigsaw puzzle, we only need to make sure that each piece links to another correctly. When linking the pieces from  $\widetilde{S}_{bi}$ , we link 1 outgoing edge from a source node  $v_s \in \widetilde{S}_{bi}$  to 1 incoming edge from a target node  $v_t \in \widetilde{S}_{bi}$ . There are numerous ways of joining the nodes. GSCALER synthesizes edges based on the edge correlation function

$$f_{corr} : N^2 \times N^2 \rightarrow [0, 1],$$

where  $f_{corr}(\alpha_s, \alpha_t) = z$  means a fraction  $z$  of the edges have a source node with bidegree  $\alpha_s$  and a target node with bidegree  $\alpha_t$ . The  $f_{corr}$  for  $G$  is listed in Table 3.

Instead of synthesizing edges one by one based on  $f_{corr}$  directly, GSCALER undergoes *Correlation Function Scaling* to scale  $f_{corr}$  to  $\widetilde{f}_{corr}$  for  $\widetilde{G}$ . After a suitable  $\widetilde{f}_{corr}$  is found, GSCALER links edges based on  $\widetilde{f}_{corr}$ .

#### 3.4.1 Correlation Function Scaling

GSCALER loops through  $f_{corr}$  to produce  $\widetilde{f}_{corr}$ . For each  $f_{corr}(\alpha_s, \alpha_t)$ , it does the following *Iterative Correlating*:

##### Manhattan Minimization

GSCALER chooses the *closest*  $(\alpha_s', \alpha_t')$  for  $\widetilde{f}_{corr}$  by minimizing  $\|\alpha_s - \alpha_s'\|_1 + \|\alpha_t - \alpha_t'\|_1$ . For  $f_{corr}((1, 2), (1, 0)) = 1/3$  in Table 3, GSCALER chooses (1, 2) for  $\alpha_s'$  and (1, 0) for  $\alpha_t'$ .

##### Increment Probability Maximization

GSCALER increments  $\widetilde{f}_{corr}(\alpha_s', \alpha_t')$  by  $p$ , where  $p$  needs to be the **largest** number that satisfies the following constraints:

- C1.**  $p \leq f_{corr}(\alpha_s, \alpha_t)$ .
- C2.**  $p \leq \min\{\frac{O(\alpha_s')}{\widetilde{m}}, \frac{I(\alpha_t')}{\widetilde{m}}\}$ , where  $I(\alpha_t')$  is the total number of available *incoming* edges for nodes with bidegree  $\alpha_t'$ , and  $O(\alpha_s')$  is the total number of available *outgoing* edges for nodes with bidegree  $\alpha_s'$ . C2 guarantees incremented number of edges does not exceed the total available number of edges of source nodes and target nodes.
- C3.**  $p \leq \frac{ct_{\widetilde{S}_{bi}}(\alpha_s') \times ct_{\widetilde{S}_{bi}}(\alpha_t')}{\widetilde{m}} - \widetilde{f}_{corr}(\alpha_s', \alpha_t')$ . C3 ensures

$f_{bi}(1, 0) = \frac{1}{3}$
$f_{bi}(1, 1) = \frac{1}{3}$
$f_{bi}(1, 2) = \frac{1}{3}$

Table 2: Bidegree distribution for  $G$  in Fig. 1.

$f_{corr}((1, 2), (1, 0)) = \frac{1}{3}$
$f_{corr}((1, 2), (1, 1)) = \frac{1}{3}$
$f_{corr}((1, 1), (1, 2)) = \frac{1}{3}$

Table 3: Edge Correlation for  $G$  in Fig.1.

the total number of edges from source nodes to target nodes is not more than the maximal number of edges allowed from source nodes to target nodes (*no multiple edges*).

For  $f_{corr}((1, 2), (1, 0)) = 1/3$  in Table 3:

By **C1**,  $p \leq 1/3$ .

By **C2**,  $p \leq \min\{\frac{O((1,2))}{7}, \frac{I((1,0))}{7}\} = \min\{\frac{2}{7}, \frac{1}{7}\} = \frac{1}{7}$ .

By **C3**,  $p \leq \frac{ct_{\widetilde{S}_{bi}}((1,2)) \times ct_{\widetilde{S}_{bi}}((1,0))}{7} - \widetilde{f}_{corr}((1, 2), (1, 0)) = \frac{1 \times 1}{7} - 0$ .

Hence, the incremental value is  $p = \min\{\frac{1}{3}, \frac{1}{7}, \frac{1}{7}\} = \frac{1}{7}$ .

##### Value Update

Next, GSCALER updates the distributions:

- $\widetilde{f}_{corr}(\alpha_s', \alpha_t') \leftarrow \widetilde{f}_{corr}(\alpha_s', \alpha_t') + p$ .
- $O(\alpha_s') \leftarrow O(\alpha_s') - p \times \widetilde{m}$ .
- $I(\alpha_t') \leftarrow I(\alpha_t') - p \times \widetilde{m}$ .

For  $f_{corr}((1, 2), (1, 0)) = 1/3$  in Table 3, GSCALER gets  $\widetilde{f}_{corr}((1, 2), (1, 0)) = 1/7$ ,  $O((1, 2)) = 1$ ,  $I((1, 0)) = 0$ . Table 4 shows the resulting scaled  $\widetilde{f}_{corr}$ .

After iterative correlating, and due to the no multiple edges constraint, it is possible that  $\sum \widetilde{f}_{corr}(\alpha_s', \alpha_t') < 1$ , which we fix by **random swapping**. This swap first randomly permutes the leftover bidegree from  $I$  and  $O$  without violating C3, then takes one element with bidegree  $\gamma_s'$  from  $O$  and one element with bidegree  $\gamma_t'$  from  $I$  to swap with generated  $\widetilde{f}_{corr}(\alpha_s', \alpha_t')$ .

The idea is to break 1 edge from some source node  $v_s$  with bidegree  $\alpha_s'$  to some target node  $v_t$  with bidegree  $\alpha_t'$ , and form 2 new edges: 1 edge pointing from some node with bidegree  $\gamma_s'$  to the other node with bidegree  $\alpha_t'$ , and 1 edge pointing from some node with bidegree  $\alpha_s'$  to some node with bidegree  $\gamma_t'$ . If C3 allows, then update  $\widetilde{f}_{corr}$  as follows:

- $\widetilde{f}_{corr}(\alpha_s', \gamma_t') \leftarrow \widetilde{f}_{corr}(\alpha_s', \gamma_t') + \frac{1}{\widetilde{m}}$ .
- $\widetilde{f}_{corr}(\gamma_s', \alpha_t') \leftarrow \widetilde{f}_{corr}(\gamma_s', \alpha_t') + \frac{1}{\widetilde{m}}$ .
- $\widetilde{f}_{corr}(\alpha_s', \alpha_t') \leftarrow \widetilde{f}_{corr}(\alpha_s', \alpha_t') - \frac{1}{\widetilde{m}}$ .

One successful swap thus increases  $\widetilde{f}_{corr}$  by  $\frac{1}{\widetilde{m}}$ .

In the worst case (this did not happen in our experiments), after random swaps,  $\sum \widetilde{f}_{corr}(\alpha_s', \alpha_t') < 1$  might still hold. We just leave  $\widetilde{f}_{corr}$  as it is, and we will introduce some *dummy nodes* to link these  $(1 - \sum \widetilde{f}_{corr}(\alpha_s', \alpha_t')) \times \widetilde{m}$  edges

$\widetilde{f}_{corr}((1, 2), (1, 0)) = \frac{1}{7}$	$\widetilde{f}_{corr}((1, 2), (1, 1)) = \frac{1}{7}$
$\widetilde{f}_{corr}((1, 1), (1, 2)) = \frac{1}{7}$	$\widetilde{f}_{corr}((2, 2), (1, 1)) = \frac{2}{7}$
$\widetilde{f}_{corr}((1, 1), (2, 2)) = \frac{2}{7}$	

Table 4: Edge Correlation for  $\widetilde{G}$  in Fig. 1.

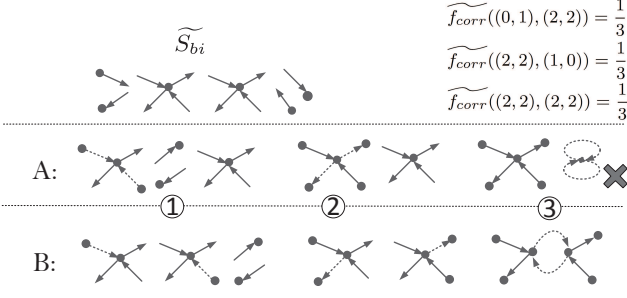


Figure 3: Local linking of source and destination.

later in Sec.3.4.2.

### 3.4.2 Edge Linking

After  $\widetilde{f}_{corr}$  is generated, GSCALER links the  $p \times \widetilde{m}$  edges locally for each  $\widetilde{f}_{corr}(\alpha_s', \alpha_t') = p$ . Note that some **linking** steps are related, since  $\alpha_s'$  might appear in a series of local linking steps, such as

$$\widetilde{f}_{corr}(\alpha_s', \alpha_{t_1}') = p_1, \dots, \widetilde{f}_{corr}(\alpha_s', \alpha_{t_k}') = p_k$$

Hence, one must make sure that after  $\widetilde{f}_{corr}(\alpha_s', \alpha_{t_1}') = p_1$  is done,  $\widetilde{f}_{corr}(\alpha_s', \alpha_{t_2}') = p_2, \dots, \widetilde{f}_{corr}(\alpha_s', \alpha_{t_k}') = p_k$  are still possible to be linked with no multiple edges. To emphasize the importance of this step, we use the following example to demonstrate both good and bad approaches.

In Fig.3,  $\widetilde{S}_{bi}$  and  $\widetilde{f}_{corr}$  are presented at the top.  $\widetilde{S}_{bi}$  has 2 pieces of bidegree (2, 2), 2 pieces of bidegree (1, 0), and 2 pieces of bidegree (0, 1). The dotted edges represent newly linked edges at each step. If an edge has no source or target node attached, then it has not linked any two nodes yet. Example A demonstrates a failed strategy, while example B demonstrates a successful strategy.

For  $\widetilde{f}_{corr}((0, 1), (2, 2)) = \frac{1}{3}$ , it corresponds to local linking step ①, which links two edges from (0, 1) to (2, 2). Both examples link edges successfully. Example A links all 2 edges to 1 target node with bidegree (2, 2), whereas example B links 2 edges to 2 different target nodes with bidegree (2, 2).

For  $\widetilde{f}_{corr}((2, 2), (1, 0)) = \frac{1}{3}$ , it corresponds to local linking step ②, which links 2 edges from (2, 2) to (1, 0). Both examples link edges successfully. Example A links all 2 edges from 1 source node with bidegree (2, 2). Example B links 2 edges from 2 different source nodes with bidegree (2, 2).

For  $\widetilde{f}_{corr}((2, 2), (2, 2)) = \frac{1}{3}$ , it corresponds to local linking step ③, which links edges from (2, 2) to (2, 2). Example A produces multiple edges, which is not allowed, whereas example B successfully produces the graph.

Apart from the multiple edge constraint, the algorithm must be efficient as well. The naive idea of back tracking previous edge linking processes is obviously not practical for large graphs. GSCALER not only generates  $\widetilde{G}$  with no multiple edges, it also links edges in **linear** time (see below). GSCALER first selects the proper *source* nodes  $L_s$  and *target* nodes  $L_t$ , and then link the edges from  $L_s$  to  $L_t$ .

Let  $\mathcal{S}_{\alpha_s'}/\mathcal{T}_{\alpha_t'}$  be a *queue* of the source/target nodes with *bidegree*  $\alpha_s'/\alpha_t'$  from  $\widetilde{S}_{bi}$ . For each  $\widetilde{f}_{corr}(\alpha_s', \alpha_t') = p$ , GSCALER dequeues & enqueues  $p \times \widetilde{m}$  elements from  $\mathcal{S}_{\alpha_s'}$  to  $L_s$ , and dequeues & enqueues  $p \times \widetilde{m}$  elements from  $\mathcal{T}_{\alpha_t'}$  to  $L_t$ . More specifically, whenever one element is dequeued from

$\mathcal{S}_{\alpha_s'}/\mathcal{T}_{\alpha_t'}$ , it will be put into  $L_s/L_t$ , and then been enqueued to  $\mathcal{S}_{\alpha_s'}/\mathcal{T}_{\alpha_t'}$  again. For example, if we dequeues & enqueues 7 elements from  $\mathcal{S}_{(1,1)} = [1, 2, 3, 4, 5]$  to  $L_s$ , then  $L_s = [1, 2, 3, 4, 5, 1, 2]$ , and  $\mathcal{S}_{(1,1)} = [3, 4, 5, 1, 2]$ . In general,

$$L_s = \{u_1^{\lceil \frac{p \times \widetilde{m}}{|\mathcal{S}_{\alpha_s'}|} \rceil}, \dots, u_{r_s}^{\lceil \frac{p \times \widetilde{m}}{|\mathcal{S}_{\alpha_s'}|} \rceil}, u_{r_s+1}^{\lfloor \frac{p \times \widetilde{m}}{|\mathcal{S}_{\alpha_s'}|} \rfloor}, \dots, u_{|\mathcal{S}_{\alpha_s'}|}^{\lfloor \frac{p \times \widetilde{m}}{|\mathcal{S}_{\alpha_s'}|} \rfloor}\}, \forall u_i \in \mathcal{S}_{\alpha_s'}$$

$$L_t = \{v_1^{\lceil \frac{p \times \widetilde{m}}{|\mathcal{T}_{\alpha_t'}|} \rceil}, \dots, v_{r_t}^{\lceil \frac{p \times \widetilde{m}}{|\mathcal{T}_{\alpha_t'}|} \rceil}, v_{r_t+1}^{\lfloor \frac{p \times \widetilde{m}}{|\mathcal{T}_{\alpha_t'}|} \rfloor}, \dots, v_{|\mathcal{T}_{\alpha_t'}|}^{\lfloor \frac{p \times \widetilde{m}}{|\mathcal{T}_{\alpha_t'}|} \rfloor}\}, \forall v_j \in \mathcal{T}_{\alpha_t'}$$

$$r_s = p \times \widetilde{m} \bmod |\mathcal{S}_{\alpha_s'}|, \quad r_t = p \times \widetilde{m} \bmod |\mathcal{T}_{\alpha_t'}|, \quad (4)$$

where  $v_1^k$  means  $v_1$  appears  $k$  times in  $L_t$ , and  $k$  is called the **multiplicity** of  $v_1$ .

After  $L_s$  and  $L_t$  are generated, GSCALER links edges from  $L_s$  to  $L_t$ . Before that, we prove a theorem for **Compound Multiplicity Reduction**; we later use this to show GSCALER does not generate multiple edges between two nodes.

**THEOREM 1.** [Compound Multiplicity Reduction]. Given multisets  $\mathbb{U} = \{u_1^{m_{u_1}}, \dots, u_{\theta_0}^{m_{u_{\theta_0}}}\}$ ,  $\mathbb{V} = \{v_1^{m_{v_1}}, \dots, v_{\theta_1}^{m_{v_{\theta_1}}}\}$ ,

$$\sum_{u_i \in \mathbb{U}} m_{u_i} = |\mathbb{U}| = |\mathbb{V}| = \sum_{v_j \in \mathbb{V}} m_{v_j}, \quad (5)$$

$$\max_{u_i \in \mathbb{U}} m_{u_i} - \min_{u_i \in \mathbb{U}} m_{u_i} \leq 1, \quad (6)$$

$$\text{and } \max_{v_j \in \mathbb{V}} m_{v_j} - \min_{v_j \in \mathbb{V}} m_{v_j} \leq 1, \quad (7)$$

there exists a multiset  $\mathbb{W} = \{w_k | w_k(0) \in \mathbb{U}, w_k(1) \in \mathbb{V}\}$

$$\text{such that } \mathbb{U} = \bigcup_k \{w_k(0)\}, \quad \mathbb{V} = \bigcup_k \{w_k(1)\}, \quad (8)$$

$$\max_{w_k \in \mathbb{W}} m_{w_k} - \min_{w_k \in \mathbb{W}} m_{w_k} \leq 1, \quad (9)$$

$$\forall w_k \in \mathbb{W}, m_{w_k} \leq \lceil \frac{|\mathbb{W}|}{\theta_0 \times \theta_1} \rceil. \quad (10)$$

**PROOF.** Reorder  $\mathbb{U}, \mathbb{V}$  to

$$\mathbb{U} = (u_1, \dots, u_1, u_2, \dots, u_2, \dots, u_{\theta_0}, \dots, u_{\theta_0}),$$

$$\mathbb{V} = (v_1, v_2, \dots, v_{\theta_1}, v_1, v_2, \dots, v_{\theta_1}, v_1, \dots).$$

Construct  $\mathbb{W}$  as follows:  $\forall w_k \in \mathbb{W}$ ,  $w_k = (\mathbb{U}(k), \mathbb{V}(k))$ , where  $\mathbb{U}(k)$  means the  $k$ th element in  $\mathbb{U}$ . Therefore, Eq.8 is satisfied. Further, for any  $u_i$ , let

$$\mathbb{W}_{u_i} = ((u_i, \mathbb{V}(c_i)), (u_i, \mathbb{V}(c_i + 1)), \dots, (u_i, \mathbb{V}(d_i)))$$

be the sequence of all elements in  $\mathbb{W}$  containing  $u_i$  as first coordinate. Consider  $\mathbb{V}_{u_i} = (\mathbb{V}(c_i), \mathbb{V}(c_i + 1), \dots, \mathbb{V}(d_i))$ , a sequence of elements in  $\mathbb{V}$  pairing with  $u_i$ . Note  $\mathbb{V}_{u_i}$  is a *periodic sequence* with *period*  $= \theta_1$ . Thus, the maximum multiplicity in  $\mathbb{V}_{u_i}$  is  $\lceil \frac{|\mathbb{V}_{u_i}|}{\theta_1} \rceil$ , and similarly for  $\mathbb{W}_{u_i}$ . Hence, the maximum multiplicity of  $\mathbb{W}$  is

$$\max_{w_k \in \mathbb{W}} m_{w_k} = \max_{1 \leq i \leq \theta_0} \lceil \frac{|\mathbb{V}_{u_i}|}{\theta_1} \rceil \quad (11)$$

It is trivial that

$$|\mathbb{V}_{u_i}| = m_{u_i}, \forall i, 1 \leq i \leq \theta_0 \quad (12)$$

Moreover, by Eq.6, we will have

$$\forall u_i \in \mathbb{U}, m_{u_i} \leq \lceil \frac{|\mathbb{U}|}{\theta_0} \rceil \quad (13)$$

Hence, by Eq.11, Eq.12, Eq.13, we will have

$$\max_{w_k \in \mathbb{W}} m_{w_k} = \max_{1 \leq i \leq \theta_0} \lceil \frac{|\mathbb{V}_{u_i}|}{\theta_1} \rceil \leq \lceil \frac{\lceil \frac{|\mathbb{U}|}{\theta_0} \rceil}{\theta_1} \rceil = \lceil \frac{|\mathbb{U}|}{\theta_0 \times \theta_1} \rceil \quad (14)$$

Since  $|\mathbb{U}| = |\mathbb{W}|$ , therefore

$$\max_{w_k \in \mathbb{W}} m_{w_k} \leq \lceil \frac{|\mathbb{W}|}{\theta_0 \times \theta_1} \rceil, \quad (15)$$

so Eq.10 holds. Similarly, the minimum multiplicity of  $\mathbb{W}$  is

$$\min_{w_k \in \mathbb{W}} m_{w_k} = \min_{1 \leq i \leq \theta_0} \lfloor \frac{|\mathbb{V}_{u_i}|}{\theta_1} \rfloor \geq \lfloor \frac{|\mathbb{W}|}{\theta_0 \times \theta_1} \rfloor \quad (16)$$

Eq.9 follows from Eq.15 and Eq.16. We call such a  $\mathbb{W}$  a **compound multiset**.  $\square$

For edge linking between  $L_s$  and  $L_t$ , GSCALER links  $u_1 \in L_s$  to  $v_2 \in L_t$  to form one edge  $(u_1, v_2)$  in  $L_e$  (edge set). By Eq.4,  $L_s/L_t$  satisfies  $\mathbb{U}/\mathbb{V}$  in *Theorem 1* respectively, and it is easy to see that  $L_e$  is the compound  $\mathbb{W}$  in *Theorem 1*.

**THEOREM 2.** *Given non-empty  $L_s(\mathbb{U})$  dequeued&enqueued from  $\mathcal{S}_{\alpha_s'}$ , and non-empty  $L_t(\mathbb{V})$  dequeued&enqueued from  $\mathcal{T}_{\alpha_t'}$ , the maximum multiplicity for edge set  $L_e(\mathbb{W})$  as described in *Theorem 1* is 1.*

**PROOF.** If  $|L_s| \leq |\mathcal{S}_{\alpha_s'}|$ , then every element in  $L_s$  is unique. Hence, the maximum multiplicity for elements in  $L_s$  is 1, so  $L_e$  has maximum multiplicity of 1. The case is similar for  $|L_t| \leq |\mathcal{T}_{\alpha_t'}|$ .

If  $|L_s| > |\mathcal{S}_{\alpha_s'}|$  and  $|L_t| > |\mathcal{T}_{\alpha_t'}|$ , then  $L_s$  has  $|\mathcal{S}_{\alpha_s'}|$  distinct elements, and  $L_t$  has  $|\mathcal{T}_{\alpha_t'}|$  distinct elements. By *Theorem 1*, the maximum multiplicity of elements in  $L_e$  is

$$\lceil \frac{|L_e|}{|\mathcal{S}_{\alpha_s'}| \times |\mathcal{T}_{\alpha_t'}|} \rceil \quad (17)$$

Since  $|L_s| = |L_e|$ , and  $L_s$  has  $\widetilde{f}_{corr}(\alpha_s', \alpha_t') \times \tilde{m}$  elements, the maximum multiplicity of elements in  $L_e$  is

$$\lceil \frac{\widetilde{f}_{corr}(\alpha_s', \alpha_t') \times \tilde{m}}{|\mathcal{S}_{\alpha_s'}| \times |\mathcal{T}_{\alpha_t'}|} \rceil$$

Moreover, by C3 in Sec.3.4.1, we know that

$$\widetilde{f}_{corr}(\alpha_s', \alpha_t') \times \tilde{m} \leq |\mathcal{S}_{\alpha_s'}| \times |\mathcal{T}_{\alpha_t'}|, \quad (18)$$

$$\text{so } \lceil \frac{|L_e|}{|\mathcal{S}_{\alpha_s'}| \times |\mathcal{T}_{\alpha_t'}|} \rceil = \lceil \frac{\widetilde{f}_{corr}(\alpha_s', \alpha_t') \times \tilde{m}}{|\mathcal{S}_{\alpha_s'}| \times |\mathcal{T}_{\alpha_t'}|} \rceil \leq 1 \quad (19)$$

$\square$

Hence, by *Theorem 2*, there are no multiple edges from  $L_s$  to  $L_t$ . Therefore, GSCALER successfully produces a graph without multiple edges.

Moreover, the generation runs in **linear** time: For each  $\widetilde{f}_{corr}(\alpha_s', \alpha_t')$ , the generation for  $L_s$  and  $L_t$  is in linear time: After  $L_s, L_t$  are generated, the  $L_e$  is formed by sequentially matching the elements in  $L_s$  and  $L_t$  as described in *Theorem 1*. Hence, the total edge linking time is *linear*. Given the edge correlation  $\widetilde{f}_{corr}$  in Table 4, GSCALER generates  $\tilde{G}$  as presented in Fig. 1.

Finally, as stated at the end of Sec.3.4.1, there might be some small possibility that  $\sum \widetilde{f}_{corr}(\alpha_s', \alpha_t') < 1$  holds. This is the theoretical worst case (which has not happened in our experiments). To resolve this, GSCALER introduces  $\epsilon$  *dummy nodes* into  $\tilde{G}$  for the purpose of maintaining degree distribution similarity. After the dummy nodes are introduced, all the  $1 - \sum \widetilde{f}_{corr}(\alpha_s', \alpha_t')$  edges are linked to

---

**Algorithm 4:** EDGE SYNTHESIS( $\widetilde{S}_{bi}, f_{corr}$ )

---

```

1  $\widetilde{f}_{corr} \leftarrow$  correlation_function_scaling( $\widetilde{S}_{bi}, f_{corr}$ )
2 while  $\widetilde{f}_{corr}(\alpha_s', \alpha_t') = p$  do
3   /* produce the  $L_s$  and  $L_t$  */
4    $L_s, L_t \leftarrow$  local_set( $\alpha_s', \alpha_t', p, \tilde{m}$ )
5   /* link the edges from  $L_s$  to  $L_t$  */
6    $\tilde{G} \leftarrow$  edge_linking( $L_s, L_t$ )
7 add dummy nodes to  $\tilde{G}$  (if necessary)

```

---

these dummy nodes. To avoid multiple edges, we can set  $\epsilon = \max_{\alpha} \{I(\alpha), O(\alpha)\}$ ; this  $\epsilon$  is obviously not the smallest possible. However, such an  $\epsilon$  is already small(negligible) compared to  $|\tilde{G}|$ , and such a scenario is rare. Algo.4 summarizes the edge synthesis.

## 4. EVALUATION

We first review the graph properties that we use for similarity measurement.

### 4.1 Graph Properties

There are numerous graph properties that can be chosen as the similarity measurement criteria, e.g. degree distribution, diameter, k-core distribution, etc. We choose the 7 most common graph properties used in the literature. Let  $N = \{0, 1, 2, 3, \dots\}$  and consider the following local and global graph properties:

1. **Indegree distribution**  $f_{in} : N \rightarrow [0, 1]$   
 $f_{in}(d) = z$  means a portion  $z$  of nodes have indegree  $d$ .
2. **Outdegree distribution**  $f_{out} : N \rightarrow [0, 1]$   
 $f_{out}(d) = z$  means a portion  $z$  of nodes have outdegree  $d$ .
3. **Bidegree distribution**  $f_{bi} : N^2 \rightarrow [0, 1]$   
 Defined in Sec 3.3.
4. **Ratio of largest strongly connected component (SCC)**  
 An SCC of  $G$  is a maximal set of nodes such that for every node pair  $u$  and  $v$ , there is a directed path from  $u$  to  $v$  and another from  $v$  to  $u$ . The ratio is the number of nodes in the SCC divided by  $|V|$ .
5. **Average clustering coefficient (CC)**  
 For node  $v_i$ , let  $N_i$  be the set of its neighbors. The local clustering coefficient [34]  $C_i$  for nodes  $v_i$  is defined by
 
$$C_i = \frac{|\{(v_j, v_k) : v_j, v_k \in N_i, (v_j, v_k) \in E\}|}{|N_i|(|N_i| - 1)}$$
 The average clustering coefficient [15]  $\bar{C} = \frac{\sum_{v_i \in V} C_i}{|V|}$
6. **Average shortest path length (ASPL)**  
 For  $u$  and  $v$  in  $V$ , the *pairwise distance*  $d(u, v)$  is the number of edges in the shortest path from  $u$  to  $v$ ;  $d(u, v) = \infty$  iff there is no path from  $u$  to  $v$  (where  $\infty$  is some number greater than  $|E|$ ). The ASPL is
 
$$\frac{\sum_{d(u,v) < \infty} d(u, v)}{|V| \times (|V| - 1)}.$$

## 7. Effective diameter [21]

The effective diameter is the smallest  $k \in N$  that is greater than 90% of all pairwise distances that satisfy  $d(u, v) < \infty$ .

## 4.2 Measuring Similarity

For a scalar graph property  $\alpha$ , let  $\alpha_G$  denote the  $\alpha$  value for  $G$  and  $\alpha_{\tilde{G}}^{(i)}$  denote the  $\alpha$  value for  $\tilde{G}$  constructed with algorithm  $\mathcal{A}^{(i)}$ . We can compare  $\mathcal{A}^{(1)}$  and  $\mathcal{A}^{(2)}$  by the absolute difference  $|\alpha_{\tilde{G}}^{(i)} - \alpha_G|$  or the relative difference  $|\alpha_{\tilde{G}}^{(i)} - \alpha_G|/\alpha_G$ . These are equivalent since

$$|\alpha_{\tilde{G}}^{(1)} - \alpha_G| < |\alpha_{\tilde{G}}^{(2)} - \alpha_G| \iff \frac{|\alpha_{\tilde{G}}^{(1)} - \alpha_G|}{\alpha_G} < \frac{|\alpha_{\tilde{G}}^{(2)} - \alpha_G|}{\alpha_G}$$

However, an  $\alpha$  like effective diameter is an integer, whereas a property like average clustering coefficient has  $\alpha < 1$ . Some information on  $\alpha_G$  is thus lost if we plot relative differences, so we will plot absolute differences instead.

For a degree distribution  $f : N^d \rightarrow [0, 1]$ , we follow Leskovec and Faloutsos [21] and use a Kolmogorov-Smirnov (KS)  $D$ -statistic to measure the difference between distributions  $f_G$  and  $f_{\tilde{G}}$ . For  $d = 1$ , the statistic is defined as

$$\sup_x |F_G(x) - F_{\tilde{G}}(x)|$$

where  $F_G$  and  $F_{\tilde{G}}$  are the cumulative distribution functions (cdf) for  $f_G$  and  $f_{\tilde{G}}$ . For  $d > 1$ , defining the cdf is not straightforward, since there are  $2^d - 1$  possibilities. In this paper, we adopt Fasano and Franceschini’s computationally efficient variant [10] of the KS  $D$ -statistic.

## 4.3 Algorithms

We compare GSCALER to state-of-art algorithms (for graph sampling, generative models, and database scaling) that have been widely used as baselines for comparison.

- In Random Walk with Escaping (RW), a starting node  $v_0$  is chosen uniformly at random. Each step in the random walk samples an unvisited neighbor uniformly at random, and there is a probability 0.8 (following [21]) that the walk restarts at  $v_0$ . If the walk reaches a dead end, the walk restarts with a new  $v_0$ .
- In Forest Fire (FF) [21], a  $v_0$  is similarly chosen. At each step, first choose a positive integer  $y$  that is geometrically distributed with mean  $p_f(1 - p_f)$ , and let  $x = \lceil y \times c \rceil$ ,  $c \in [0, 1]$ . (We follow Leskovec and Faloutsos and set  $p_f = 0.7$  and  $c = 1$ .) Pick uniformly at random neighbors  $w_1, \dots, w_x$  that are not yet sampled, then recursively repeat these steps at each  $w_i$ .
- In Stochastic Kronecker Graph (SKG) [20], SKG first trains a  $N_1$  by  $N_1$  matrix  $K_1$ , where  $N_1$  is typically set as 2. Then recursively multiply  $K_1$  through *Kronecker* product. Thus,  $d$  multiplication of Kronecker product results in a graph of  $N_1^d$  nodes. In our experiment, we use the  $N_1^d$  closest to  $\tilde{n}$  as the target number  $\tilde{n}$ .
- *UpSizer* was designed to synthetically scale a relational dataset by maintaining the degree distribution [33]. However, *UpSizer* does not allow free choice of  $\tilde{m}$ . For our experiments, we transform the graph to relational tables so *UpSizer* can scale it.

## 4.4 Datasets

We pick 2 real directed graphs from Stanford’s collection of networks [24]. They are large enough that it makes sense for scaling down, but small enough for global properties like diameter to be determined in reasonable time. These two graphs were also used by previous authors [20–22, 25].

- *Epinions* ( $|V| = 75879$ ,  $|E| = 508837$ ) for the website [Epinions.com](http://Epinions.com), where an edge  $(x, y)$  indicates user  $x$  trusts user  $y$ .
- *Slashdot* is a website for technology-related news, where users tag others as friend or foe. The graph contains friend/foe links between the users of Slashdot. The dataset *Slashdot0811* ( $|V| = 77360$ ,  $|E| = 828161$ ) was from November 2008.

Given the space constraint, we choose to compare more properties for 2 graphs, instead of comparing fewer properties for more graphs.

## 5. RESULTS AND DISCUSSION

All experiments are done on a Linux machine with 128GB memory and AMD Opteron 2.3GHz processor. For *SKG*, we use the C++ implementation [24]. For *UpSizer*, we use the authors’ C++ implementation<sup>1</sup>. For *FF*, *RW* and *GSCALER*, we implemented them in Java.

These algorithms use the number of nodes  $n$  to specify sample size or scale factor, so we define  $s = \tilde{n}/n$ . In our experiments, we set  $s = \frac{1}{5}, \frac{1}{7.5}, \frac{1}{10}, \frac{1}{12.5}, \frac{1}{15}, \frac{1}{17.5}, \frac{1}{20}$  for scaling down, and  $s = 2, 3, 4, 5, 6$  for scaling up.

GSCALER allows the user to specify the number of edges  $\tilde{m}$ . However, an  $\tilde{m}$  that is arbitrarily small or arbitrarily large will make it impossible for  $\tilde{G}$  to be similar to  $G$ . To be fair, we choose an  $\tilde{m}$  for GSCALER that yields the best results. We will revisit this issue in Sec.6.

For each  $s$  value, we run each algorithm 10 times (using different seeds) on each dataset. The average of these 10 runs is then plotted as one data point.

### 5.1 Execution Time

For a fair comparison, we exclude the *I/O* time for all algorithms.

Execution time for *SKG* has two parts: training time and running time. *SKG* needs to train the graph initiator matrix  $K_1$ , where  $K_1$  is a 2 by 2 matrix in our case;  $K_1$  can be pre-computed. After  $K_1$  is trained, *SKG* uses  $K_1$  to generate a graph with  $2^k$  nodes.

To get a better understanding of *SKG*’s time complexity, we plot both training time and running time. *SKG-Train* represents the time needed for training  $K_1$ , while *SKG-Run* represents the running time of graph generation given  $K_1$ .

Fig.4 and Fig.5 show the execution time for all algorithms using log scale.

For both datasets, *SKG-Train* is very large for training the graph initiator  $K_1$ . However, after  $K_1$  is trained, graph generation is fast (seconds), so *SKG-Run* is small. That aside, GSCALER has the smallest execution time, which is faster than *SKG-Train* by about 2 orders of magnitude, and faster than *RW*, *FF*, *UpSizer* by about 1 order of magnitude.

<sup>1</sup><http://www.comp.nus.edu.sg/~upsizer/>

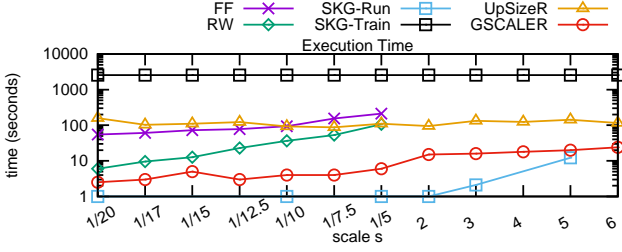


Figure 4: Execution time (log scale) for *Slashdot*. FF and RW do not work for  $s > 1$ .

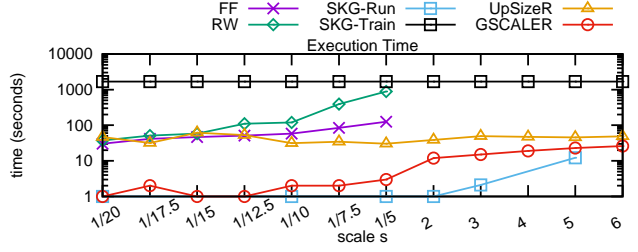


Figure 5: Execution time (log scale) for *Epinions*. FF and RW do not work for  $s > 1$ .

## 5.2 Theoretical Bounds of Degree Distribution

Before looking at the similarity comparison, we first show theoretically that GSCALER performs well on varying  $\tilde{m}$  and  $\tilde{n}$  for indegree/outdegree distribution. To save space, we just consider indegree distribution throughout this section. The proof for the outdegree distribution is similar.

**THEOREM 3.** *Given original graph  $G$ 's indegree distribution is  $f_{in}$ , and GSCALER scales  $G$  to  $\tilde{G}$ , where  $\tilde{m}$  and  $\tilde{n}$  scale by the same ratio  $s$ . Then,  $E[D(f_{in}, \tilde{f}_{in})] = 0$ .*

**PROOF.** As explained in Sec.3.2, GSCALER scales  $S_{in}$  with the following criterion:

$$E[ct_{\tilde{S}_{in}}(x)] = ct_{S_{in}}(x) \times \frac{\tilde{n}}{n}$$

Now, consider the current scaled number of nodes  $|\tilde{V}|$  and number of edges  $|\tilde{E}|$ . After node scaling,

$$\begin{aligned} E[|\tilde{V}|] &= E\left[\sum_x ct_{\tilde{S}_{in}}(x)\right] = \sum_x E[ct_{\tilde{S}_{in}}(x)] = \sum_x ct_{S_{in}}(x) \times s \\ &= n \times s = \tilde{n}. \end{aligned}$$

$$\begin{aligned} E[|\tilde{E}|] &= E\left[\sum_x ct_{\tilde{S}_{in}}(x) \times x\right] = \sum_x E[ct_{\tilde{S}_{in}}(x) \times x] \\ &= \sum_x E[ct_{\tilde{S}_{in}}(x)] \times x = \sum_x ct_{S_{in}}(x) \times s \times x \\ &= \sum_x ct_{S_{in}}(x) \times x \times s = m \times s = \tilde{m}. \end{aligned}$$

Hence, no *edge adjustment* is expected in Sec.3.2. Thus,

$$\forall x, E[\tilde{f}_{in}(x) - f_{in}(x)] = 0.$$

Consequently,  $E[D(f_{in}, \tilde{f}_{in})] = 0$ .  $\square$

However, a GSCALER user may want to have a  $\tilde{G}$  with average degree different from  $G$ , i.e.  $\tilde{m}$  and  $\tilde{n}$  scale by different factors. In this case, GSCALER can still produce a similar degree distribution with small and bounded error.

**THEOREM 4.** *Given an original graph  $G$ 's indegree distribution is  $f_{in}$ , and GSCALER scales  $G$  to  $\tilde{G}$ , where  $\tilde{n} = n \times s$ ,  $\tilde{m} = m \times s \times (1 + r)$  and  $r \neq 0$ . Then,*

$$E[D(f_{in}, \tilde{f}_{in})] \leq \frac{2m|r|}{n \times d^*}$$

where  $d^*$  is approximately the largest degree of  $G$ .

**PROOF.** As shown in the proof of *Theorem 3*, after node scaling,  $E[|\tilde{V}|] = n \times s = \tilde{n}$  and  $E[|\tilde{E}|] = m \times s \neq \tilde{m}$ .

Hence, *edge adjustment* is needed, as stated in Sec.3.2. In total, we are expecting  $m \times s \times (1 + r)$  edges. Hence,  $|m \times rs|$  edges are expected to be added/removed. We refer to  $ct_{\tilde{S}_{in}}(x)$  as  $ct(x)$  if there is no ambiguity.

Consider  $r > 0$ , so  $m \times rs$  edges are to be added. This is done by the edge adjustment operation as stated in Algo.2: (i)  $ct(l) - -$ ,  $ct(h) + +$ , (ii)  $l + +$ ,  $h - -$ . The net effect is to add  $h - l$  edges per adjustment.

Let  $d_m$  be the maximum degree of  $G$ . We assume  $l$  is initiated as 0, while  $h$  is initialized as  $d_m$ .

**Case  $ct(l) > 0$  for all the first  $k$  adjustment:** Then GSCALER will add

$$d_m, d_m - 2, d_m - 4 \dots, d_m - [(k-1) \bmod \lceil \frac{d_m}{2} \rceil] \times 2 \quad (20)$$

edges for the first  $k$  adjustments. Let  $T_k$  be the total number of edges added by first  $k$  adjustments, then

$$T_k \geq \frac{d_m}{2} \times k \quad (21)$$

Since  $m \times rs$  edges are expected to be added, the expected number of adjustments  $k$  satisfies

$$k \leq \frac{m \times rs}{\frac{d_m}{2}} = \frac{2m \times rs}{d_m} \quad (22)$$

Moreover, each edge adjustment changes  $\tilde{f}_{in}$  by

(i) decrementing  $\frac{1}{n}$  for some  $\tilde{f}_{in}(x_i)$ , where  $x_i < \frac{d_m}{2}$

(ii) incrementing  $\frac{1}{n}$  for some  $\tilde{f}_{in}(x_j)$ , where  $x_j > \frac{d_m}{2}$

Thus, the expected total decremental changes made to  $\tilde{f}_{in}$  is  $\frac{1}{n} \times k$ . By Eq.22,

$$\frac{1}{n} \times k \leq \frac{1}{n} \times \frac{2m \times rs}{d_m} = \frac{2m \times rs}{sn \times d_m} = \frac{2mr}{n \times d_m} \quad (23)$$

Since  $D(f_{corr}, \tilde{f}_{corr})$  measures largest difference between the cumulative function of  $f_{corr}$ ,  $\tilde{f}_{corr}$ . Hence, by Eq.23,

$$E[D(f_{corr}, \tilde{f}_{corr})] \leq \frac{2mr}{n \times d^*}, \text{ where } d^* = d_m \quad (24)$$

**Case  $\exists l_j, ct(l_j) = 0$ .** Assume at the  $i$ th adjustment,  $ct(l_0) = 0$  for some  $l_0$ , where  $i$  is the smallest.

Then GSCALER shifts  $l$  to  $l_0 + 1$ , and try to decrease  $ct(l_0 + 1)$ . Assume  $ct(l_0 + 1) > 0$ , then we can do  $ct(l_0 + 1) - -$ .

1.  $l_0 + 1 \neq h$ : By Eq.20, the number of edges added at  $i$ th step is  $d_m - [(i-1) \bmod \lceil \frac{d_m}{2} \rceil] \times 2 - 1$ . Hence,  $T_i = T_{i-1} + d_m - [(i-1) \bmod \lceil \frac{d_m}{2} \rceil] \times 2 - 1$ . By Eq.21,

$$T_i + 1 \geq \frac{d_m}{2} \times i \quad (25)$$



Then,  $T_i \geq \frac{d_m}{2} \times i - 1 = \frac{d_m - 1}{2} \times i + (\frac{i}{2} - 1)$

Since  $T_1 = d_m - 1 \geq \frac{d_m - 1}{2}$ , then  $T_i \geq \frac{d_m - 1}{2} \times i \forall i \in N$

2.  $l_0 + 1 = h$ : Then  $l$  will be reset to 0, and  $h$  will be reset to  $d_m$ , so the number of edges at  $i$ th adjustment is obviously larger than  $d_m - [(i-1) \bmod \lceil \frac{d_m}{2} \rceil] \times 2 - 1$ . Hence, Eq.25 will hold as well.

Therefore, during the first  $k$  adjustments, if we encounter  $w$  different  $l_0, l_1, \dots, l_{w-1}$ , such that  $ct(l_j) = 0, \forall 0 \leq j < w$ , then by mathematical induction, one can still conclude that

$$T_i \geq \frac{d_m - w}{2} \times i.$$

Hence, let  $d^* = d_m - w$  and follow the proof after Eq.21; then  $E[D(\widehat{f}_{corr}, \widetilde{f}_{corr})] \leq \frac{2mr}{n \times d^*}$ , where  $d^*$  is close to  $d_m$  in general.

The proof for  $r < 0$  is similar  $\square$

### 5.3 Experimental Results

All figures in this section (except Figs.7, 9, 11,12) plot the similarity measures (Sec. 4.2) comparing  $G$  and  $\widetilde{G}$ , where

- the horizontal axis is scale factor  $s = \frac{\widetilde{n}}{n}$ ;
- the vertical axis is KS D-statistic for indegree, outdegree and bidegree distributions;
- the vertical axis is absolute error for the other 5 properties (effective diameter, largest SCC ratio, etc.);
- the true value for all datasets' graph properties are provided above each figure;
- we choose  $\widetilde{m}$  to give best results for GSCALER; the other algorithms do not allow a free choice of  $\widetilde{m}$ .

We first look at *scaling down* for all the algorithms. As mentioned in Sec.1, scaling down in GSP has an objective that is different from graph sampling. However, we use graph sampling algorithms for comparison because GSP is a new problem, so there is no other algorithms for comparison except *UpSizeR* and *SKG*.

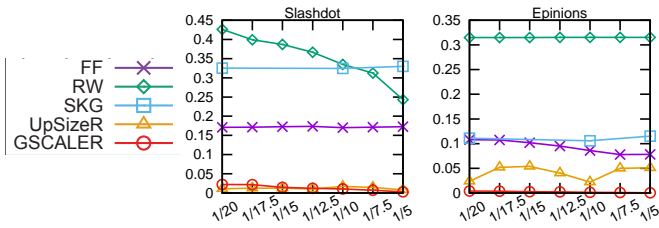


Figure 6: KS-D statistics for Indegree Distribution

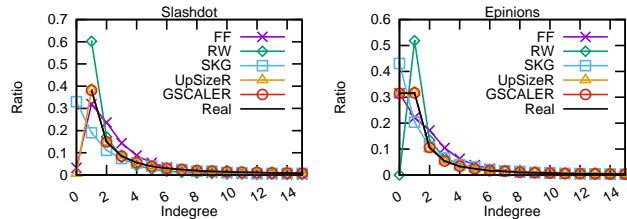


Figure 7: Indegree Distribution Plot

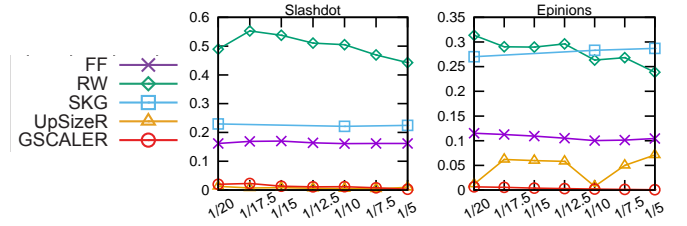


Figure 8: KS-D statistics for Outdegree Distribution

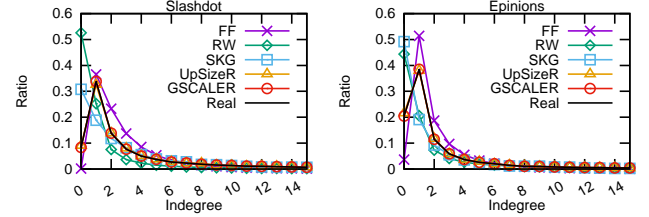


Figure 9: Outdegree Distribution Plot

#### 5.3.1 Indegree Distribution

For the indegree distribution, Fig. 6 shows that both GSCALER and *UpSizeR* perform very well for *Slashdot*, with error  $< 0.01$ . For *Epinions*, *UpSizeR* has an error of 0.05 on average, whereas GSCALER again has a small error  $< 0.01$ .

For more details, Fig. 7 plots the indegree distribution for  $s = 0.2$ , where the x-axis is the indegree, and the y axis is the ratio of nodes having that indegree. We only show the plot up to *indegree* = 15, which covers more than 87% of all nodes.

The plot shows that GSCALER and *UpSizeR* mimics  $G$ 's indegree distribution very well. Although all algorithms produce power law shaped distributions, only *UpSizeR* and GSCALER give a close fit for the empirical distribution.

#### 5.3.2 Outdegree Distribution

For the outdegree distribution, Fig. 8 similarly shows that both GSCALER and *UpSizeR* perform very well for *Slashdot*, with error  $< 0.03$  on average. For *Epinions*, *UpSizeR* has an error of 0.05 on average, whereas GSCALER still has an error  $< 0.01$ .

The outdegree distributions are plotted in Fig. 9. We observe that GSCALER and *UpSizeR* also closely match  $G$ 's outdegree distribution. Again, although all algorithms produce power law shaped distributions, *UpSizeR* and GSCALER give the best fit for the empirical outdegree distribution.

#### 5.3.3 Bidegree Distribution

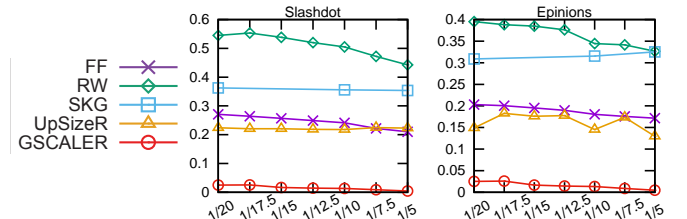


Figure 10: KS-D statistics for Bidegree Distribution

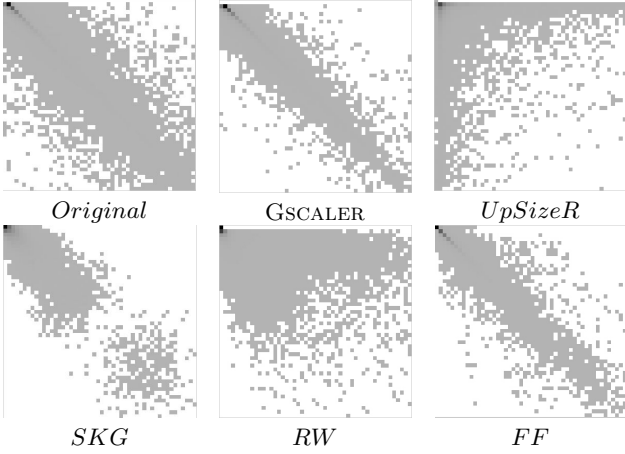


Figure 11: Bidegree distribution for *Slashdot*

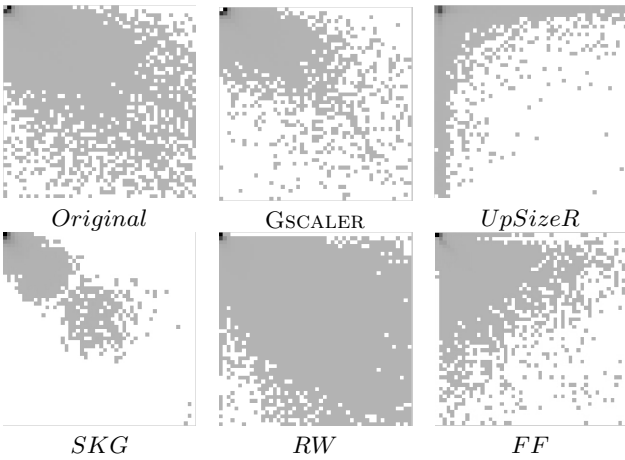


Figure 12: Bidegree distribution for *Epinions*

Even though *UpSizeR* matches the empirical indegree and outdegree distributions, it is not able to capture the bidegree distribution that describes the correlation between indegree and outdegree. Such correlation is especially important for social network graphs. For example, in *Epinions*, the number of people he/she trusts and the number of people who trust him/her are correlated. As shown in Fig.10, only GSCALER captures such correlation very well.

For a detailed look at the bidegree distributions, we give 2-dimensional plots Fig. 11 and Fig. 12. We transform the bidegree distribution to a  $k \times k$  matrix  $B$ . Each cell  $i, j$  represents the portion of nodes with bidegree  $(i, j)$ . In other words,  $B[i, j] = f_{bi}(i, j)$ . When visualizing  $B$ , we use a gray scale intensity plot for cell  $i, j$  to indicate  $B[i, j]$ . The larger  $f_{bi}$  is, the darker the cell  $(i, j)$  is. In our case, we set  $k = 50$  which covers more than 90% of total nodes.

Fig. 11 and Fig. 12 show *indegree* is positively correlated to *outdegree*. For both *Slashdot* and *Epinions*, GSCALER is the best algorithm in capturing this indegree/outdegree correlation. We also observe that *UpSizeR* tends to have indegree and outdegree negatively correlated. *SKG* has very concentrated and similar-shaped plots for both datasets. We suspect this is because of the self-similar matrix operation, Kronecker product. For *FF*, it captures the bidegree correlation for *Slashdot*'s bidegree, but not for *Epinions*.

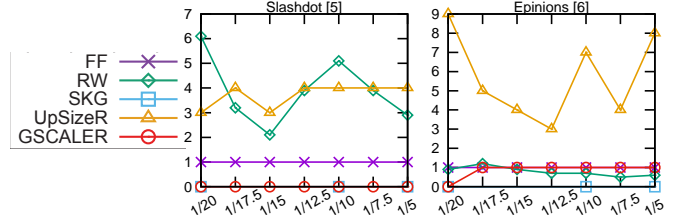


Figure 13: Absolute Error for Effective Diameter

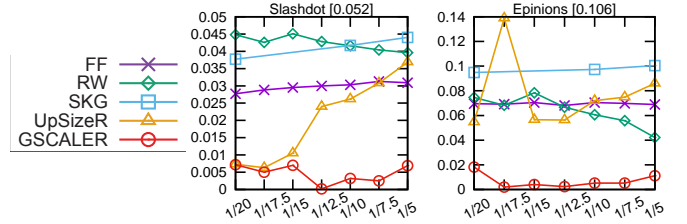


Figure 14: Absolute Error for Average CC

### 5.3.4 Effective Diameter

Fig. 13 shows that, for *Slashdot*, GSCALER produces exactly the real effective diameter; for *Epinions*, it produces an effective diameter with an absolute error no larger than 1. Overall, GSCALER, *FF*, *SKG* are the best algorithms in producing similar effective diameters.

### 5.3.5 Average Clustering Coefficient

For both datasets, GSCALER significantly reduces the error for average clustering coefficient. Fig. 14 shows that, for *Slashdot*, the average error for the other algorithms are between 0.03 and 0.045. This corresponds to a relative error of 60% to 90%. However, GSCALER only has an absolute error 0.005 on average, which corresponds to a relative error  $< 10\%$ . Similarly, for *Epinions*, GSCALER also improves the relative error from 70% to 10% on average.

### 5.3.6 Largest SCC Ratio

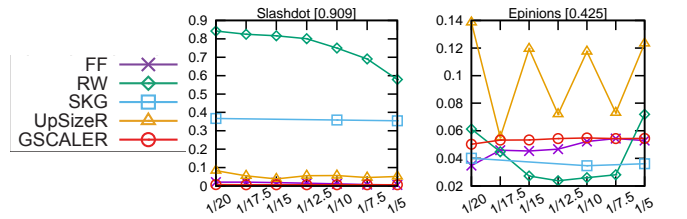


Figure 15: Absolute Error for Largest SCC Ratio

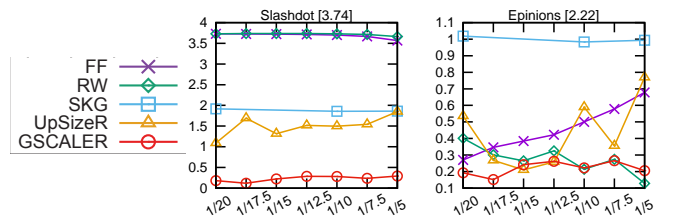


Figure 16: Absolute Error for ASPL

For largest SCC ratio, Fig. 15 shows that *FF*, *UpSizeR*, and GSCALER have the best performance for *Slashdot*. For *Epinions*, *SKG* is the best performing algorithm, whereas GSCALER only loses to *SKG* by an error  $< 0.01$ .

Note that, GSCALER and *FF* are the best performing algorithms which produce best similar largest SCC ratio for both datasets on average.

### 5.3.7 Average Shortest Path Length

Fig. 16 shows that GSCALER is the best algorithm in producing similar *ASPL* for the scaled graph.

For *Slashdot*, GSCALER has relative error  $< 10\%$ , whereas the second best performing algorithm *UpSizeR* has the average relative error  $40\%$

For *Epinions*, GSCALER is the most stable and accurate algorithm, with a consistent absolute error  $< 0.25$ . *RW* performs well for large  $s$ , but does badly for small  $s$ .

### 5.3.8 Scaling Up

To save space, Fig.17 plots performance of both *Slashdot* and *Epinions* for each graph property.

Similar to scaling down, GSCALER improves the accuracy of indegree/outdegree/bidegree distribution significantly. For effective diameter, GSCALER and *SKG* are the best performing algorithms. GSCALER is the best algorithm which produces the most accurate results for average clustering coefficient, largest SCC ratio, and average shortest path length.

### 5.3.9 Summary of Comparisons

For indegree, outdegree and bidegree distributions, GSCALER reduces the error from about 0.1 for the other algorithms to about 0.01. This is expected since GSCALER uses  $f_{bi}$  to construct  $\tilde{G}$ , and agrees with the theoretical bound in Sec. 5.2.

GSCALER only uses  $\mathcal{P}_{sub} = \{f_{bi}, f_{corr}\}$  to construct  $\tilde{G}$ , but measures similarity to  $G$  with a larger set  $\mathcal{P}$  of both local and global properties listed in Sec. 4.1. The results show that enforcing  $\mathcal{P}_{sub}$  suffices to induce similarity for  $\mathcal{P}$ .

## 6. LIMITATION

Unlike the other algorithms, a user can choose both  $\tilde{n}$  and  $\tilde{m}$  for GSCALER. Table 5 illustrates GSCALER’s accuracy for *Slashdot*, using different  $\tilde{n}$  and  $\tilde{m}$ .

However, GSCALER cannot guarantee similarity for arbitrary choices of  $\tilde{n}$  and  $\tilde{m}$  — the error bound in *Theorem 4* becomes loose for large  $|r|$ . For example, if  $n = 1000$  and  $m = 5000$ , but  $\tilde{n} = 2000$  and  $\tilde{m} = 500000$ , one cannot expect to find any  $\tilde{G}$  similar to  $G$ .

There is a Densification Law [23] that says, if  $G_1, G_2, G_3, \dots$ , are snapshots of a growing graph, then

$$E(G_i) \propto V(G_i)^\alpha \quad \text{for some } 1 \leq \alpha \leq 2$$

If  $\tilde{G}$  follows such a law, then  $(\tilde{m}/m) = (\tilde{n}/n)^\alpha$ . If  $\tilde{n} = sn$  and  $\tilde{m} = (1+r)sm$ , then

$$\frac{\tilde{m}}{\tilde{n}^\alpha} = \frac{(1+r)sm}{(sn)^\alpha} = \frac{m}{n^\alpha} \frac{1+r}{s^{(\alpha-1)}}$$

Therefore, for  $\tilde{G}$  to follow the Densification Law, the user must choose  $r > 0$  for  $s > 1$ , and  $r < 0$  for  $s < 1$ . In other words,  $\tilde{m} > m\tilde{n}/n$  for  $\tilde{n}/n > 1$ , and  $\tilde{m} < m\tilde{n}/n$  for  $\tilde{n}/n < 1$ .

Note that modeling the evolution of  $n$  and  $m$  is an interesting problem that is relevant, but orthogonal to GSP.

Slashdot		Effective Diameter	Largest SCC	ASPL	CC
$n = 77360$	$m = 828161$	5	0.909	3.74	0.052
$\tilde{n} = 4421$	$\tilde{m} = 32179$	5	0.916	3.59	0.058
$\tilde{n} = 4421$	$\tilde{m} = 33831$	5	0.916	3.52	0.053
$\tilde{n} = 4421$	$\tilde{m} = 34399$	5	0.916	3.62	0.058
$\tilde{n} = 15472$	$\tilde{m} = 141583$	5	0.916	3.57	0.053
$\tilde{n} = 15472$	$\tilde{m} = 144895$	5	0.916	3.51	0.060
$\tilde{n} = 15472$	$\tilde{m} = 147849$	5	0.916	3.45	0.059
$\tilde{n} = 154720$	$\tilde{m} = 1669903$	5	0.917	3.71	0.055
$\tilde{n} = 154720$	$\tilde{m} = 1671288$	5	0.917	3.71	0.062
$\tilde{n} = 154720$	$\tilde{m} = 1672057$	5	0.917	3.70	0.062
$\tilde{n} = 386800$	$\tilde{m} = 4182213$	5	0.917	3.89	0.058
$\tilde{n} = 386800$	$\tilde{m} = 4186353$	5	0.917	3.89	0.048
$\tilde{n} = 386800$	$\tilde{m} = 4190908$	5	0.917	3.89	0.053

Table 5: Gscaler accuracy for different  $\tilde{n}$  and  $\tilde{m}$ .

## 7. CONCLUSION

We considered the problem of synthetically scaling a given graph. Our solution GSCALER first breaks  $G$  into pieces, scales them, then merges them using the degree and correlation functions from  $G$ .

Different from previous approaches, GSCALER gives user a choice for  $\tilde{n}$  and  $\tilde{m}$ . We proved that GSCALER does not produce multiple edges between two nodes, and has a small distribution error even when the average degree of  $\tilde{G}$  differs from the original graph  $G$ .

Experiments with 2 well-known real datasets show that the  $\tilde{G}$  constructed by GSCALER is more similar to  $G$  for most properties than random walk, forest fire, *UpSizeR* and Stochastic Kronecker Graph.

Our current work aims to extend GSCALER to scale relational databases by representing the tables as graphs.

## 8. REFERENCES

- [1] N. K. Ahmed, N. Duffield, et al. Graph sample and hold: A framework for big-graph analytics. In *Proc. KDD*, pages 1446–1455, 2014.
- [2] N. K. Ahmed, J. Neville, and R. Kompella. Network sampling: From static to streaming graphs. *TKDD*, 8(2):7, 2014.
- [3] Y.-Y. Ahn, S. Han, et al. Analysis of topological characteristics of huge online social networking services. In *Proc. WWW*, pages 835–844, 2007.
- [4] W. Aiello, F. Chung, and L. Lu. A random graph model for power law graphs. *Experimental Mathematics*, 10(1):53–66, 2001.
- [5] R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47, 2002.
- [6] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
- [7] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.
- [8] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *Proc. SIGMOD*, pages 145–156. ACM, 2011.
- [9] P. L. Erdős and A. Rényi. On the evolution of random graphs. In *Publication of the Mathematical Institute of*

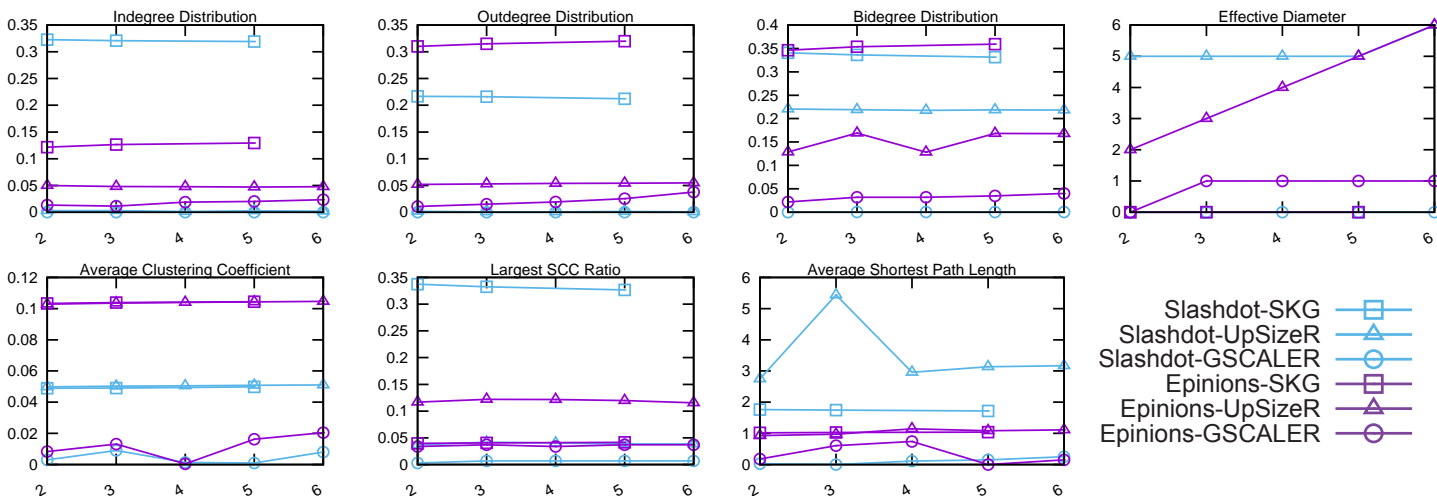


Figure 17: Scaling Up Experiments ( $s$  values on horizontal axes; legend format is dataset-algorithm.)

- the Hungarian Academy of Science*, pages 17–61, 1960.
- [10] G. Fasano and A. Franceschini. A multidimensional version of the Kolmogorov-Smirnov test. *Monthly Notices Royal Astronomical Society* 255(1), 1987.
- [11] A. D. Flaxman, A. M. Frieze, and J. Vera. A geometric preferential attachment model of networks ii. *Internet Mathematics*, 4(1):87–111, 2007.
- [12] M. Gjoka, M. Kurant, et al. Walking in Facebook: A case study of unbiased sampling of OSNs. In *Proc. INFOCOM*, pages 2498–2506, 2010.
- [13] L. A. Goodman. Snowball sampling. *The Annals of Mathematical Statistics*, pages 148–170, 1961.
- [14] P. Hu and W. C. Lau. A survey and taxonomy of graph sampling. *CoRR*, abs/1308.5865, 2013.
- [15] A. Kemper. *Valuation of Network Effects in Software Markets*. Physica, 2010.
- [16] V. Krishnamurthy, M. Faloutsos, et al. Reducing large Internet topologies for faster simulations. In *Proc. NETWORKING*, pages 328–341, 2005.
- [17] R. Kumar, P. Raghavan, et al. Extracting large-scale knowledge bases from the web. In *VLDB*, volume 99, pages 639–650, 1999.
- [18] S. Lee, P. Kim, and H. Jeong. Statistical properties of sampled networks. *Physical Review E* 73(1), 2006.
- [19] J. Leskovec, D. Chakrabarti, et al. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *Proc. PKDD 2005*, pages 133–145. Springer, 2005.
- [20] J. Leskovec, D. Chakrabarti, et al. Kronecker graphs: An approach to modeling networks. *J. Machine Learning Research*, 11:985–1042, 2010.
- [21] J. Leskovec and C. Faloutsos. Sampling from large graphs. In *Proc. KDD*, pages 631–636, 2006.
- [22] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Signed networks in social media. In *Proc. SIGCHI*, pages 1361–1370, 2010.
- [23] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187, 2005.
- [24] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, June 2014.
- [25] J. Leskovec, K. J. Lang, et al. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.
- [26] R.-H. Li, J. Yu, L. Qin, R. Mao, and T. Jin. On random walk based graph sampling. In *ICDE*, pages 927–938, 2015.
- [27] N. Metropolis, A. W. Rosenbluth, et al. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.
- [28] S. Mussmann, J. Moore, J. J. P. III, and J. Neville. Incorporating assortativity and degree dependence into scalable network models. In *Proc. AAAI*, 2015.
- [29] S. Qiao and Z. M. Özsoyoğlu. Rbench: Application-specific rdf benchmarking. In *Proc. SIGMOD*, pages 1825–1838. ACM, 2015.
- [30] B. Ribeiro and D. Towsley. Estimating and sampling graphs with multidimensional random walks. In *Proc. IMC*, pages 390–403, 2010.
- [31] R. Staden. A strategy of dna sequencing employing computer programs. *Nucleic Acids Research*, 6(7):2601–2610, 1979.
- [32] M. Stumpf, C. Wiuf, and R. May. Subnets of scale-free networks are not scale-free: Sampling properties of networks. *PNAS* 102(12), pages 4221–4224, 2005.
- [33] Y. C. Tay, B. T. Dai, D. T. Wang, E. Y. Sun, Y. Lin, and Y. Lin. Upsizer: Synthetically scaling an empirical relational database. *Inf. Syst.*, 38(8):1168–1183, 2013.
- [34] D. J. Watts and S. H. Strogatz. Collective dynamics of ‘small-world’ networks. *Nature*, 393(6684):409–10.
- [35] C. Wilson, B. Boe, et al. User interactions in social networks and their implications. In *Proc. EuroSys*, pages 205–218, 2009.

# Storing and Analyzing Historical Graph Data at Scale

Udayan Khurana  
IBM TJ Watson Research Center  
ukhurana@us.ibm.com

Amol Deshpande  
University of Maryland  
amol@cs.umd.edu

## ABSTRACT

The work on large-scale graph analytics to date has largely focused on the study of static properties of graph snapshots. However, a static view of interactions between entities is often an oversimplification of several complex phenomena like the *spread of epidemics*, *information diffusion*, *formation of online communities*, and so on. Being able to find temporal interaction patterns, visualize the evolution of graph properties, or even simply compare snapshots across time, adds significant value in reasoning over graphs. However, due to the lack of underlying data management support, an analyst today has to manually navigate the added temporal complexity of dealing with large evolving graphs. In this paper, we present a system, called *Historical Graph Store*, that enables users to store large volumes of historical graph data and to express and run complex temporal graph analytical tasks against that data. It consists of two key components: (1) a *Temporal Graph Index* (TGI), that compactly stores large volumes of historical graph evolution data in a partitioned and distributed fashion – TGI also provides support for retrieving snapshots of the graph as of any timepoint in the past or evolution histories of individual nodes or neighborhoods; and (2) a *Temporal Graph Analysis Framework* (TAF), for expressing complex temporal analytical tasks and for executing them in an efficient and scalable manner using Apache Spark. Our experiments demonstrate our system’s efficient storage, retrieval and analytics across a wide variety of queries on large volumes of historical graph data.

## 1. INTRODUCTION

Graphs are useful in capturing behavior involving interactions between entities. Several processes are naturally represented as graphs – social interactions between people, financial transactions, biological interactions among proteins, geospatial proximity of infected livestock, and so on. Many problems based on such graph models can be solved using well-studied algorithms from graph theory or network science. Examples include finding driving routes by computing shortest paths on a network of roads, finding user communities through dense subgraph identification in a social network, and many others. Numerous graph data management systems have been developed over the last decade, including special-

ized graph database systems like Neo4j, Titan, etc., and large-scale graph processing frameworks such as GraphLab [27], Pregel [29], Giraph, GraphX [12], GraphChi [24], etc.

However much of the work to date, especially on cloud-scale graph data management systems, focuses on managing and analyzing a single (typically, current) static snapshot of the data. In the real world, however, interactions are a dynamic affair and any graph that abstracts a real-world process changes over time. For instance, in online social media, the friendship network on Facebook or the “follows” network on Twitter change steadily over time, whereas the “mentions” or the “retweet” networks change much more rapidly. Dynamic cellular networks in biology, evolving citation networks in publications, dynamic financial transactional networks, are few other examples of such data. Lately, we have seen an increasing merit in dynamic modeling and analysis of network data to obtain crucial insights in several domains such as cancer prediction [38], epidemiology [15], organizational sociology [16], molecular biology [9], information spread on social networks [26] amongst others.

In this work, our focus is on providing the ability to analyze and to reason over the entire history of the changes to a graph. There are many different types of analyses of interest. For example, an analyst may wish to study the evolution of well-studied static graph properties such as centrality measures, density, conductance, etc., over time. Another approach is through the search and discovery of temporal patterns, where the events that constitute the pattern are spread out over time. Comparative analysis, such as juxtaposition of a statistic over time, or perhaps, computing aggregates such as *max* or *mean* over time, possibly gives another style of knowledge discovery into temporal graphs. Most of all, a primitive notion of just being able to access past states of the graphs and performing simple static graph analysis, empowers a data scientist with the capacity to perform analysis in arbitrary and unconventional patterns.

Supporting such a diverse set of temporal analytics and querying over large volumes of historical graph data requires addressing several data management challenges. Specifically, there is a want of techniques for storing the historical information in a compact manner, while allowing a user to retrieve graph snapshots as of any time point in the past or the evolution history of a specific node or a specific neighborhood. Further, the data must be stored and queried in a distributed fashion to handle the increasing scale of the data. There is also a need for an expressive, high-level, easy-to-use programming framework that will allow users to specify complex temporal graph analysis tasks, while ensuring those tasks can be executed efficiently in a data-parallel fashion across a cluster.

In this paper, we present a graph data management system, called *Historical Graph Store* (HGS), that provides an ecosystem for managing and analyzing large historical traces of graphs. HGS con-

sists of two key distinct components. First, the *Temporal Graph Index (TGI)*, is an index that compactly stores the entire history of a graph by appropriately partitioning and encoding the differences over time (called *deltas*). These deltas are organized to optimize the retrieval of several temporal graph primitives such as neighborhood versions, node histories, and graph snapshots. TGI is designed to use a distributed key-value store to store the partitioned deltas, and can thus leverage the scalability afforded by those systems (our implementation uses Apache Cassandra<sup>1</sup> key-value store). TGI is a tunable index structure, and we investigate the impact of tuning the different parameters through an extensive empirical evaluation. TGI builds upon our prior work on DeltaGraph [21], where the focus was on retrieving individual snapshots efficiently; TGI extends DeltaGraph to support efficient retrieval of subgraphs instead of only full snapshots, retrieval of histories of nodes or subgraphs over past time intervals, and features a highly scalable design over DeltaGraph.

The second component of HGS is a *Temporal Graph Analysis Framework (TAF)*, which provides an expressive framework to specify a wide range of temporal graph analysis tasks. TAF is based on a novel set of *temporal graph operands* and *operators* that enable parallel execution of the specified tasks at scale in a cluster environment. The execution engine is implemented on Apache Spark [40], a large-scale in-memory cluster computing framework.

**Outline:** The rest of the paper is organized as follows. In Section 2, we survey the related work on graph data stores, temporal indexing, and other topics relevant to the scope of the paper. In Section 3, we provide a sketch of the overall system, including key aspects of the underlying components. We then present TGI and TAF in detail in Sections 4 and 5, respectively. In Section 6, we provide an empirical evaluation, and conclude with a summary and a list of future directions in Section 7.

## 2. RELATED WORK

In the recent years, there has been much work on graph storage and graph processing systems and numerous systems have been designed to address various aspects of graph data management. Some examples include Neo4J, Titan<sup>2</sup>, GBase [19], Pregel [29], Giraph, GraphX [12], GraphLab [27], and Trinity [36]. These systems use a variety of different models for representation, storage, and querying, and there is a lack of standardized or widely accepted models for the same. Most graph querying happens through programmatic access to graphs in languages such as Java, Python or C++. Graph libraries such as Blueprints<sup>3</sup> provide a rich set of implementations for graph theoretic algorithms. SPARQL [33] is a language used to search patterns in linked data. It works on an underlying RDF representation of graphs. T-SPARQL [13] is a temporal extension of SPARQL. He et al. [17], provide a language for finding sub-graph patterns using a graph as a query primitive. Gremlin<sup>4</sup> is a graph traversal language over the property graph data model, and has been adopted by several open-source systems. For large-scale graph analysis, perhaps the most popular framework is the vertex-centric programming framework, adopted by Giraph, GraphLab, GraphX, and several other systems; there have also been several proposals for richer and more expressive programming frameworks in recent years. However, most of these prior systems largely focus on analyzing a single snapshot of the graph data, with very little support for handling dynamic graphs, if any.

<sup>1</sup><https://cassandra.apache.org>

<sup>2</sup><http://thinkaurelius.github.io/titan/>

<sup>3</sup><https://github.com/tinkerpop/blueprints/wiki>

<sup>4</sup><https://github.com/tinkerpop/gremlin>

A few recent papers address the issues of storage and retrieval in dynamic graphs. In our prior work, we proposed DeltaGraph [21], an index data structure that compactly stores the history of all changes in a dynamic graph and provides efficient snapshot reconstruction. G\* [25] stores multiple snapshots compactly by utilizing commonalities. ImmortalGraph [30] is an in-memory system for processing dynamic graphs, with the objectives of shared storage and computation for overlapping snapshots. Ghrab et al. [11] provide a system of network analytics through labeling graph components. Gedik et al. [10], describe a block-oriented and cache-enabled system to exploit spatio-temporal locality for solving temporal neighborhood queries. Koloniari et al. [23] also utilize caching to fetch selective portions of temporal graphs they refer to as partial views. LLAMA [28] uses multiversioned arrays to represent a mutating graph, but their focus is primarily on in-memory representation. There is also recent work on streaming analytics over dynamic graph data [8, 7], but it typically focuses on analyzing only the recent activity in the network (typically over a sliding window).

Temporal graph analytics is an area of growing interest. Evolution of shortest paths in dynamic graphs has been studied by Huo et al. [18], and Ren et al. [34]. Evolution of community structures in graphs has been of interest as well [5, 14]. Change in page rank with evolving graphs [3], and the study of change in centrality of vertices, path lengths of vertex pairs, etc. [32], also lie under the larger umbrella of temporal graph analysis. Ahn et al. [1] provide a taxonomy of analytical tasks over evolving graphs. Barrat et al. [4], provide a good reference for studying several dynamic processes modeled over graphs. Our system significantly reduces the effort involved in building and deploying such analytics over large volumes of graph data.

Temporal data management for relational databases was a topic of active research in the 80s and early 90s. Snapshot index [39] is an I/O optimal solution to the problem of snapshot retrieval for transaction-time databases. Salzberg and Tsotras [35] present a comprehensive survey of temporal data indexing techniques, and discuss two extreme approaches to supporting snapshot retrieval queries, referred to as the *Copy* and *Log* approaches. While the copy approach relies on storing new copies of a snapshot upon every point of change in the database, the log approach relies on storing everything through changes. Their hybrid is often referred to as the *Copy+Log* approach. We omit a detailed discussion of the work on temporal databases, and refer the interested reader to a representative set of references [37, 31, 35]. Other data structures, such as Interval Trees [2] and Segment trees [6] can also be used for storing temporal information. Temporal aggregation in scientific array databases is another related topic of interest, but the challenges there are significantly different. Kaufmann et al. [20] propose an in-memory index in SAP HANA that addresses temporal aggregation, joins, and snapshot construction. The applicability of temporal relational data management techniques to graphs is restricted due to lack of (efficient) support for graph specific retrieval such as fetching neighborhoods, or histories of nodes over time. Our work in this paper focuses on techniques for a wide variety of temporal graph retrieval and analysis on entire graph histories that are primarily stored on disk.

## 3. OVERVIEW

In this section, we introduce key aspects related to HGS. We begin with the data model, followed by the key challenges and concluding with an overview of the system.

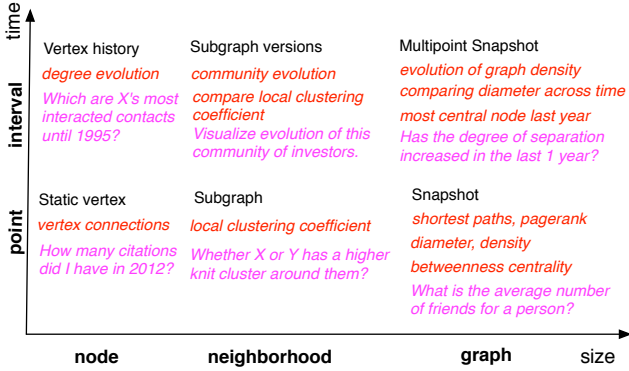


Figure 1: The scope of temporal graph analytics can be represented across two different dimensions - time and entity. The chart lists retrieval tasks (black), graph operations (red), example queries (magenta) at different granularities of time and entity size.

### 3.1 Data Model

Under a discreet notion of time, a time-evolving graph  $G^T = (V^T, E^T)$  may be expressed as a collection of graph *snapshots* over different time points,  $\{G^0 = (V^0, E^0), G^1, \dots, G^t\}$ . The vertex set  $V^i$  for a snapshot consists of a set of vertices (nodes), each of which has a unique identifier (constant over time), and an arbitrary number of key-value attribute pairs. The edge sets  $E^i$  consist of edges that each contain references to two valid nodes in the corresponding vertex set  $V^i$ , information about the direction of the edge, and an arbitrary list of key-value attribute pairs. A temporal graph can also be equivalently described by a set of changes to the graph over time. We call an atomic change at a specific timepoint in the graph an *event*. The changes could be structural, such as the addition or the deletion of nodes or edges, or be related to attributes such as an addition or a deletion or a change in the value of a node or an edge attribute. For instance, a new user joining the Facebook social network corresponds to an event of node creation; connecting to another user is an event of edge creation; changing location or posting an update are events of change and creation of attribute values, respectively. These approaches specified here as well as certain hybrids have been used in the past for the physical and logical modeling of temporal data. Our approach to temporal processing in this paper is best described using a *node-centric* logical model, i.e., the historical graph is seen as a collection of evolving vertices over time; the edges are considered as attributes  $E$  of the nodes. This abstraction helps in our design of distributed storage of the graph and parallel execution of the analytical tasks.

### 3.2 Challenges

The nature of data management tasks in historical graph analytics can be categorized based on the scope of analysis using the dual dimensions of *time* and *entity* as illustrated with examples in Figure 1. The temporal scope of an analysis task can range from a single point in time to a long interval; the entity scope can range from a single node to the entire graph. While the diversity of analytical tasks provides a potential for a rich set of insights from historical graphs, it also poses several challenges in constructing a system that can perform those tasks. To the best of our knowledge, none of the existing systems address a majority of those challenges that are described below:

**Compact storage with fast access:** A natural tradeoff between index size and access latencies can be seen in the Log and Copy approaches for snapshot retrieval. Log requires minimal information

to encode the graph’s history, but incurs large reconstruction costs. Copy, on the other hand, provides direct access, but at the cost of excessive storage. The desirable index should consume space of the order of Log index but provide near direct access like Copy.

**Time-centric versus entity-centric indexing:** For *point* access such as past snapshot retrieval, a time-centric indexing such as DeltaGraph or Copy+Log is suitable. However, for version retrieval tasks such as retrieving a *node’s history*, entity-centric indexing is the correct choice. Neither of the indexing approaches, however, are feasible in the opposite scenarios. Given the diversity of access needs, we require an index that works well with both styles of lookup at the same time.

**Optimal granularity of storage for different queries:** Query latencies for a graph also depend on the size of chunks in which the data is indexed. While larger granularities of storage incur wasteful data read for “node retrieval”, a finely chunked graph storage would mean higher number of lookups and aggregation for a 2-hop neighborhood lookup. The physical and logical arrangement of data should take care of access needs at all granularities.

**Coping with changing topology in a dynamic graph:** It is evident that graph partitioning is inevitable in the storage and processing of large graphs. However, finding the appropriate strategy to maintain workable partitioning on a constantly *changing* graph is another challenge while designing a historical graph index.

**Systematically expressing temporal graph analytics:** A platform for expressing a wide variety of historical graph analytics requires an appropriate amalgam of temporal logic and graph theory. Additionally, utilizing a vast body of existing tools in network science is an engineering challenge and opportunity.

**Appropriate abstractions for distributed, scalable analytics:** Parallelization is key to scale up analytics for large graph datasets. It is essential that the underlying data-representations and operators in the analytical platform be designed for parallel computing.

### 3.3 System Overview

Figure 2 shows the architecture of our proposed Historical Graph Store. It consists of two main components:

**Temporal Graph Index (TGI)** records the entire history of a graph compactly while enabling efficient retrieval of several temporal graph primitives. It encodes various forms of differences (called *deltas*) in the graph, such as atomic events, changes in subgraphs over intervals of time, etc. It uses specific choices of graph partitioning, data replication, temporal compression and data placement to optimize the graph retrieval performance. TGI uses Cassandra, a distributed key-value store for the deltas. In Section 4, we describe the design details of TGI and the access algorithms.

**Temporal Graph Analytics Framework (TAF)** provides a *temporal node-centric* abstraction for specifying and executing complex temporal network analysis tasks. It helps the user analyze the history of the graph by means of simple yet expressive *temporal operators*. The abstraction of temporal graph through a *set of (temporal) nodes (SoN)* allows the framework to achieve computational scalability through distribution of tasks by node and time. TAF is built on top of Apache Spark to utilize its support for scalable, in-memory, cluster computation; TAF provides an option to utilize GraphX for static graph computation. We provide a Java and Python based library to specify the retrieval, computation and analysis tasks. In Section 5, we describe the details of the data and computational models, query processing, parallel data fetch aspects of the system, the analytical library along with a few examples.

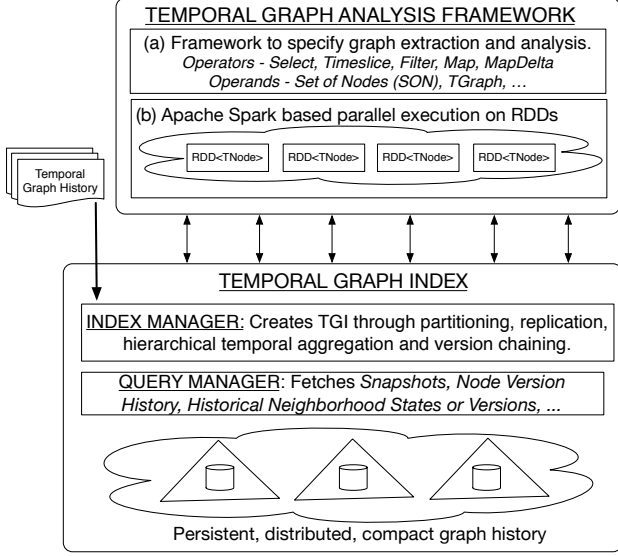


Figure 2: System Overview

## 4. TEMPORAL GRAPH INDEX

In this section, we investigate the issue of indexing temporal graphs. First, we introduce a *delta framework* to define any temporal index as a set of different changes or *deltas*. Using this framework, we are able to qualitatively compare the access costs and sizes of different alternatives for temporal graph indexing, including our proposed approach. We then present the Temporal Graph Index (TGI), that stores the entire history of a large evolving network in the cloud, and facilitates efficient parallel reconstruction for different graph primitives. TGI is a generalization of both entity and time-based indexing approaches and can be tuned to suit specific workload needs. We claim that TGI is the minimal index that provides efficient access to a variety of primitives on a historical graph, ranging from past snapshots to versions of a node or neighborhood. We also describe the key partitioning strategies instrumental in scaling to large datasets across a cloud storage.

### 4.1 Preliminaries

We start with a few preliminary definitions that help us formalize the notion of the delta framework.

**DEFINITION 1 (STATIC NODE).** A static node refers to the state of a vertex in a network at a specific time, and is defined as a set containing: (a) node-id, denoted  $I$  (an integer), (b) an edge-list, denoted  $E$  (captured as a set of node-ids), (c) attributes, denoted  $A$ , a map of key-value pairs.

A static edge is defined analogously, and contains the node-ids for the two endpoints and the edge direction in addition to a map of key-value pairs. Finally, a static graph component refers to either a static edge or a static node.

**DEFINITION 2 (DELTA).** A Delta ( $\Delta$ ) refers to either: (a) a static graph component (including the empty set), or (b) a difference, sum, union or intersection of two deltas.

Such a definition of delta helps express the change in a wider context than merely difference of graph states at two points. It helps us articulate several temporal graph indexes including TGI and Delta-Graph in a single framework.

**DEFINITION 3 (CARDINALITY AND SIZE).** The cardinality and the size of a delta are the unique and total number of static node or edge descriptions within it, respectively.

**DEFINITION 4 ( $\Delta$  SUM).** A sum (+) over two deltas,  $\Delta_1$  and  $\Delta_2$ , i.e.,  $\Delta_s = \Delta_1 + \Delta_2$  is defined over graph components in the two deltas as follows: (1)  $\forall gc_1 \in \Delta_1$ , if  $\exists gc_2 \in \Delta_2$  s.t.  $gc_1.I = gc_2.I$ , then we add  $gc_2$  to  $\Delta_s$ , (2)  $\forall gc_1 \in \Delta_1$  s.t.  $\nexists gc_2 \in \Delta_2$  s.t.  $gc_1.I = gc_2.I$ , we add  $gc_1$  to  $\Delta_s$ , and (3) analogously the components present only in  $\Delta_2$  are added to  $\Delta_s$ .

Note that:  $\Delta_1 + \Delta_2 = \Delta_2 + \Delta_1$  is not necessarily true due the order of changes. We also note that:  $\Delta_1 + \emptyset = \Delta_1$ , and  $(\Delta_1 + \Delta_2) + \Delta_3 = \Delta_1 + (\Delta_2 + \Delta_3)$ . Analogously, difference(-) is defined as a set difference over different components of the two deltas.  $\Delta_1 - \emptyset = \Delta_1$  and  $\Delta_1 - \Delta_1 = \emptyset$ , are true, while,  $\Delta_1 - \Delta_2 = \Delta_2 - \Delta_1$ , does not necessarily hold.

**DEFINITION 5 ( $\Delta$  INTERSECTION).** An intersection of two deltas is defined as a set intersection over the the components of two deltas.  $\Delta_1 \cap \emptyset = \emptyset$ , is true for any delta. Similarly, union of two deltas  $\Delta_\cup = \Delta_1 \cup \Delta_2$ , consists of all elements from  $\Delta_1$  and  $\Delta_2$ . The following is true for any delta:  $\Delta_1 \cup \emptyset = \Delta_1$ .

Next we discuss and define some specific types of deltas:

**DEFINITION 6 (EVENT).** An event is the smallest change that happens to a graph, i.e., addition or deletion of a node or an edge, or a change in an attribute value. An event is described around one time point. As a delta, an event concerning a graph component  $c$ , at time point  $t_e$ , is defined as the difference of state of  $c$  at and before  $t_e$ , i.e.,  $\Delta_{event}(c, t_e) = c(t_e) - c(t_e - 1)$ .

**DEFINITION 7 (EVENTLIST).** An eventlist delta is a chronologically sorted set of event deltas. An eventlist's scope may be defined by the time duration,  $(t_s, t_e]$ , during which it defines all the changes that happened to the graph.

**DEFINITION 8 (EVENTLIST PARTITION).** An eventlist partition delta is a chronologically sorted set of event deltas pertaining to a set of nodes,  $P$ , over a given time duration,  $(t_s, t_e]$ .

**DEFINITION 9 (SNAPSHOT).** A snapshot,  $G^{t_a}$  is the state of a graph  $G$  at a time point  $t_a$ . As a delta, it is defined as the difference of the state of the graph at  $t_a$  from an empty set,  $\Delta_{snapshot}(G, t_a) = G(t_a) - G(-\infty)$ .

**DEFINITION 10 (SNAPSHOT PARTITION).** A snapshot partition is a subset of a snapshot. It is identified by a subset  $P$  of all nodes in graph,  $G$  at time,  $t_a$ . It consists of all nodes in  $G$  at  $t_a$  and all the edges whose at least one end-point lies in  $P$  at time,  $t_a$ .

### 4.2 Prior Techniques

The prior techniques for temporal graph indexing use changes or differences in various forms to encode time-evolving datasets. We can express them in the  $\Delta$ -framework as follows. The **Log** index is equivalent to a set of all event deltas (equivalently, a single eventlist delta encompassing the entire history). The **Copy+Log** index can be represented as combination of: (a) a finite number of distinct snapshot deltas, and (b) eventlist deltas to capture the change between successive snapshots. Although we are not aware of a specific proposal for a **vertex-centric** index, however, a natural approach would be to maintain a set of eventlist partition deltas,



one for each node (with edge information replicated with the endpoints). The **DeltaGraph** index, proposed in our prior work, is a tunable index with several parameters. For a typical setting of parameters, it can be seen as equivalent to taking a Copy+Log index, and replacing the *snapshot* deltas in it with another set of deltas constructed hierarchically as follows: for every  $k$  successive *snapshot* deltas, replace them with a single delta that is the intersection of those deltas and a set of difference deltas from the intersection to the original snapshots, and recursively apply this till you are left with a single delta.

Table 1 estimates the cost of fetching different graph primitives as the number and the cumulative size of deltas that need to be fetched for the different indexes. The first column shows an estimate of index storage space, which varies considerably across the techniques. For proofs, please refer to the extended version [22].

### 4.3 Temporal Graph Index: Definition

Given the above formalism, a Temporal Graph Index for a graph  $G$  over a time period  $T = [0, t_c]$  is described by a collection of different deltas as follows:

- (a) Eventlist Partitions: A set of eventlist partition deltas,  $\{E_{tp}\}$ , where  $E_{tp}$  captures the changes during the time interval  $t$  belonging to partition  $p$ .
- (b) Derived Snapshot Partitions: Consider  $r$  distinct time points,  $t_i$ , where  $1 \leq i \leq r$ ,  $t_i \in T$ . For each  $t_i$ , we consider  $l$  partition deltas,  $P_j^i$ ,  $1 < j < l$ , such that  $\cup_j P_j^i = G^{t_i}$ . There exists a function that maps any node-id(I) in  $G^{t_i}$  to a unique partition-id( $P_j^i$ ),  $f_i: I \rightarrow P_j^i$ . With a collection of  $P_j^i$  over  $T$  as leaf nodes, we construct a hierarchical tree structure where a parent is the intersection of children deltas. The difference of each parent from its child delta is called as a *derived snapshot partition* and is explicitly stored. Note that  $P_j^i$  are not explicitly stored. This is the same as DeltaGraph, with the exception of partitioning.
- (c) Version Chain: For all nodes  $N$  in the graph  $G$ , we maintain a chronologically sorted list of pointers to all the references for that node in the delta sets described above (a and b). For a node  $I$ , this is called a *version chain* (denoted  $VC_I$ ).

In short, the TGI stores *deltas* or *changes* in three different forms, as follows. The first one is the atomic changes in a chronological order through eventlist partitions. This facilitates direct access to the changes that happened to a part or whole of the graph at specified points in time. Secondly, the state of nodes at different points in time is stored indirectly in form of the derived snapshot partition deltas. This facilitates direct access to the state of a neighborhood or the entire graph at a given time. Thirdly, a meta index stores node-wise pointers to the list of chronological changes for each node. This gives us a direct access to the changes occurring to individual nodes. Figure 3(a) shows the arrangement of eventlist, snapshot and derived snapshot partition deltas. Figure 3(b) shows a sample version chain.

TGI utilizes the concept of temporal consistency which was optimally utilized by DeltaGraph. However, it differs from DeltaGraph in two major ways. First, it uses a partitioning for eventlists, snapshots or deltas instead of large monolithic chunks. Additionally, it maintains a list of version chain pointers for each node. The combination of these two novelties along with DeltaGraph’s temporal compression generalizes the notion of entity-centric and time-centric indexing approaches in an efficient way. This can be seen by the qualitative comparison shown in Table 1 as well as empirical results in Section 6. Note that the particular design of TGI in the form of eventlist partitions and deltas, and version chain is not equivalent to two separate indexes, one with snapshots and eventlists and

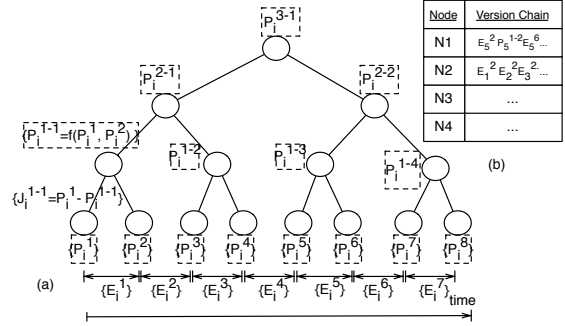


Figure 3: Temporal Graph Index representation: (a) TGI deltas partitions - eventlists, snapshots and derived snapshots. The (dashed) bounded deltas are not stored; (b) Version Chains.

the other with chronologically organized events per node. For instance, the latter is fairly inefficient to fetch temporal subgraphs or neighborhoods over time intervals.

### 4.4 TGI: Design and Architecture

In the previous subsection, we presented the logical description of TGI. We now describe the strategies for physical storage on a cloud which enables high scalability. In a distributed index, we desire that all graph retrieval calls achieve maximum parallelization through equitable distribution. A distribution strategy based on pure node-based key is good idea for snapshot style access, however, it is bad for a subgraph history style of access. A pure time-based key strategy on the other hand, has complementary qualities and drawbacks. An important related challenge for scalability is dealing with two different skews in a temporal graph dataset – temporal and topological. They refer to the uneven density of graph activity over time and the uneven edge density across regions of the graph, respectively. Another important aspect to note is that for a retrieval task, it is desirable that all the deltas needed for a fetch operation that are present on a particular machine be proximally located to minimize latency of lookups<sup>5</sup>. Based on the above constraints and desired properties, we describe the physical layout of TGI as follows:

1. The entire history of the graph is divided into *time spans*, keeping the number of changes to the graph consistent across different time spans,  $f_t: e.time \rightarrow tsid$ , where  $e$  is the event and  $tsid$  is the unique identifier for the time span.
2. A graph at any point is horizontally partitioned into a fixed number of *horizontal partitions* based upon a random function of the node-id,  $f_h: nid \rightarrow sid$ , where  $nid$  is the node-id and  $sid$  is unique identifier of for the horizontal partition.
3. The partition deltas (including eventlists) are stored as a key-value pairs, where the delta-key is composed of  $\{tsid, sid, did, pid\}$ , where  $did$  is a delta-id, and  $pid$  is the partition-id of the partition.
4. The placement-key is defined as a subset of the composite deltas key described above, as  $\{tsid, sid\}$ , which defines the chunks in which data is placed across a set of machines on a cluster. A combination of the  $tsid$  and  $sid$  ensure that a large fetch task, whether snapshot or version oriented, seeks data distributed across the cluster and not just one machine.

<sup>5</sup>In general, this depends on the underlying storage mechanism. The physical placement of deltas is irrelevant for an in-memory store, but significant for an on-disk store due to seek times.

	Index Size	Snapshot		Static Vertex		Vertex versions		1-hop		1-hop Versions	
		$\sum_{\Delta}  \Delta $	$\sum_{\Delta} 1$	$\sum_{\Delta}  \Delta $	$\sum_{\Delta} 1$	$\sum_{\Delta}  \Delta $	$\sum_{\Delta} 1$	$\sum_{\Delta}  \Delta $	$\sum_{\Delta} 1$	$\sum_{\Delta}  \Delta $	$\sum_{\Delta} 1$
Log	$ G $	$ G $	$\frac{ G }{ E }$	$ G $	$\frac{ G }{ E }$	$ G $	$\frac{ G }{ E }$	$ G $	$\frac{ G }{ E }$	$ G $	$\frac{ G }{ E }$
Copy	$ G ^2$	$ S $	1	$ S $	1	$ S  G $	$ G $	$ S $	1	$ S  G $	$ G $
Copy+Log	$\frac{ G ^2}{ E }$	$ S  +  E $	2	$ S  +  E $	2	$ G $	$\frac{ G }{ E }$	$ S  +  E $	2	$ G $	$\frac{ G }{ E }$
Node Centric	$2 G $	$2 G $	$ N $	$ C $	1	$ C $	1	$ R  \cdot  V $	$ R $	$ R  \cdot  V $	$ R $
DeltaGraph	$ G (h+1)$	$h \cdot  S  +  E $	$2h$	$h \cdot  S  +  E $	$2h$	$ G $	$\frac{ G }{ E }$	$h \cdot ( S  +  E )$	$2h$	$ G $	$\frac{ G }{ E }$
TGI	$ G (2h+3)$	$h \cdot  S  +  E $	$2h$	$\frac{h \cdot  S }{p} + \frac{ E }{p}$	$2h$	$ V (1 + \frac{ S }{p})$	$ V  + 1$	$\frac{h \cdot ( S  +  E )}{p}$	$2h$	$ V (1 + \frac{ S }{p})$	$ V  + 1$

Table 1: Comparison of access costs for different retrieval queries and index storage for various temporal indexes.  $|G|$  =number of changes in the graph;  $|S|$  =size of a snapshot;  $h$  = height and  $|E|$  = eventlist size;  $|V|$  =number of changes to a node;  $|R|$ =numbers of neighbors of a node;  $p$ = number of partitions in TGI. The metrics used are, sum of delta cardinalities ( $\sum_{\Delta} |\Delta|$ ) and number of deltas ( $\sum_{\Delta} 1$ ).

- The partitioned deltas are clustered by the delta key. The given order of delta-key along with the placement-key implies that all partitions of a delta are stored contiguously, which makes it efficient to scan and read all partitions belonging to a delta in a snapshot query. Also, if the order of *did* and *pid* is reversed, it makes fetching a partition across different deltas more efficient.

**Implementation and Architecture:** TGI uses Cassandra for its delta storage as well as metadata regarding partitioning, time-spans, etc. TGI consists of a *Query Manager* (QM) is responsible for planning, dividing and delegating the query to one or more *Query Processors* (QP). Multiple QPs query the datastore in parallel and process the raw deltas into the required result. Depending on the query specification, the distributed result is either aggregated at a particular QP (the QM) or returned to the client which made the request without aggregation. An *Index Manager* is responsible for the construction and maintenance activities of the index. We omit further details and refer the reader to the extended version [22].

## 4.5 Dynamic Graph Partitioning

Partitioning of the deltas is an essential aspect of TGI and provides cheaper access to subgraph elements when compared to DeltaGraph or similar indexes. The two traditional approaches to partitioning a static graph are random (node-id hash-based) or locality-based (min-cut, max-flow) partitioning. Random partitioning is simpler and involves minimal bookkeeping. However, since it loses locality, it is unsuitable for neighborhood-level granularity access. Locality-aware partitioning, on the other hand, preserves locality but incurs extra bookkeeping in form of a {node-id:partition-id} map. TGI is designed to work with either configuration as desired, as well as different partition size specifications. TGI also supports replication of edge-cuts for further speed up of 1-hop neighborhoods. It uses a separate *auxiliary delta partition* besides each delta partition to store the replication, thereby preventing extra read cost for snapshot or node centric queries. More details on this can be found in the extended manuscript.

Locality-aware partitioning, however, faces an additional challenge with time-evolving graphs. With the change in size and topology of a graph, a partitioning deemed good (with respect to locality) at an instant may cease to be good at a later time. A probable approach of frequent repartitioning over time would maintain partitioning quality, but leads to excessive amounts of bookkeeping, which in turn leads to degradation of performance while accessing different node or neighborhood versions.

Our approach of dealing with this dilemma is described as follows. For a time-evolving graph,  $G(t)$ , we update the partitioning once at the beginning of each *time span*. The partitioning valid during a time-span  $\tau$ , is decided as the collectively best partitioning for the graph during time  $\tau$ ,  $G^\tau$ . Now, the best-suited partition-

ing for a graph over a time-interval  $G^\tau$  is performed by projecting it to a static graph using a function,  $\Omega(G^\tau)$ , followed by a static-graph partitioning.  $\Omega$  could be defined in various ways, depending on the best-deemed interpretation of a representative static graph. Any definition, however, must retain all and only the nodes that appeared in  $G^\tau$ . In TGI, the default choice of  $\Omega$  is called *Union-Mean* and includes all edges that appeared in  $G^\tau$  with the edge-weights computed as a function of time-fraction of existence. We refer the reader to the extended manuscript for further details on different choices of  $\Omega$ , contrast of this technique with other alternatives, and comments on the associated problem of finding the appropriate boundaries of time-spans.

## 4.6 Fetching Graph Primitives

We briefly describe how the different types of retrieval queries are executed. The details of the algorithms can be found in the extended version of the paper.

**Snapshot Retrieval:** In snapshot retrieval, the state of a graph at a time point is retrieved. Given a time  $t_s$ , the query manager locates the appropriate time span  $T$  such that  $t_s \in T$ , within which, it figures out the path from the root of the TGI to the leaf closest to the given time point. All the snapshot deltas,  $\Delta_{s1}, \Delta_{s2}, \dots, \Delta_{sm}$ , (i.e., all the corresponding partitions) along that path from root to the leaf, and the eventlists from the leaf node to the time point,  $\Delta_{e1}, \Delta_{e2}, \dots, \Delta_{en}$  are fetched and merged appropriately as:  $\sum_{i=1}^m \Delta_{si} + \sum_{i=1}^n \Delta_{ei}$  (notice the order). This is performed across different query processors covering the entire set of horizontal partitions. This is conceptually similar to the DeltaGraph snapshot reconstruction with the addition of the aspect of partitions.

**Node's history:** Retrieving a node's history during time interval,  $[t_s, t_e]$  involves finding the state of the graph at point  $t_s$ , and all changes during the time range  $(t_s, t_e)$ . The first one is done in a similar manner to snapshot retrieval except the fact that we look up only a specific delta partition in a specific horizontal partition, that the node belongs to. The second part happens through fetching the node's version chain to determine its points of changes during the given range. The respective eventlists are fetched and filtered for the given node.

**k-hop neighborhood (static):** In order to retrieve the k-hop neighborhood of a node, we can proceed in two possible ways. One of them is to fetch the whole graph snapshot and filter the required subgraph. The other is to fetch the given node, and then determine its neighbors, fetch them, and recurse. It is easy to see that the performance of the second method will deteriorate fast with growing  $k$ . However for lower values, typically  $k \leq 2$ , the latter is faster or at least as good, especially if we are using neighborhood replication as discussed in a previous subsection. In case of a neighborhood

fetch, the query manager automatically fetches the auxiliary portions of deltas (if they exist), and if the required nodes are found, further lookup is terminated.

**Neighborhood evolution:** Neighborhood evolution queries can be posed in two different ways. First, requesting all changes for a described neighborhood, in which case the query manager fetches the initial state of the neighborhood followed by the events indicating the change. Second, requesting the state of the neighborhood at multiple specific time points. This translates to the retrieval of multiple single neighborhoods fetch tasks.

## 5. ANALYTICS FRAMEWORK

In this section, we describe the *Temporal Graph Analysis Framework (TAF)*, that enables programmers to express complex analytical tasks on time-evolving graphs and execute them in a scalable, parallel, in-memory manner. We present details of the novel computational model, including a set of operators and operands. We also present the details of implementation on top of Apache Spark, as well as the user API (exposed through Python and Java). Finally, we describe TAF’s coordination with TGI, particularly the parallel data transfer protocol, that provides a complete ecosystem for historical graph management and analysis.

### 5.1 Data and Computational Model

At the heart of this analytics framework is an abstraction with the view of historical graph as a *set of nodes (or subgraphs) evolving over time*. The choice of temporal nodes as a primitive is instrumental in enabling us to express a wide range of fetch and compute operations in an intuitive manner. More significantly, it provides us with the appropriate basis for the parallelizing computation of arbitrary analysis tasks. The *temporal nodes* and *set of temporal nodes* bear a correspondence to *tuples* and *tables* of the relational algebra, as the basic unit of data and the prime operand, respectively. The two central data types are defined below:

**DEFINITION 11 (TEMPORAL NODE).** A *temporal node* ( $NodeT$ ),  $N^T$ , is defined as a sequence of all and only the states of a node  $N$  over a time range,  $T = [t_s, t_e)$ . All the  $k$  states of the node must have a valid time duration  $T_i$ , such that  $\cup_i^k T_i = T$  and  $\cap_i^k T_i = \phi$ .

**DEFINITION 12 (SET OF TEMPORAL NODES).** A *SoN*, is defined as a set of  $r$  temporal nodes  $\{N_1^T, N_2^T \dots N_r^T\}$  over a time range,  $T = [t_s, t_e)$ , as depicted in Figure 4.

The *NodeT* class provides a range of methods to access the state of the node at various time points, including: `getVersions()` which returns the different versions of the node as a list of static nodes (*NodeS*), `getVersionAt()` which finds a specific version of the node given a timepoint, `getNeighborIDsAt()` which returns IDs of the neighbors at the specified time point, and so on.

A *Temporal Subgraph (SubgraphT)* generalizes *NodeT* and captures a sequence of the states of a subgraph (i.e., a set of nodes and edges among them) over a period of time. Typically the subgraphs correspond to  $k$ -hop neighborhoods around a set of nodes in the graph. An analogous `getVersionAt()` function can be used to retrieve the state of the subgraph as of a specific time point as an in-memory *Graph* object (the user program must ensure that any graph object so created can fit in the memory of a single machine). A *Set of Temporal Subgraphs (SoTS)* is defined analogously to *SoN* as a set of temporal subgraphs.

### 5.2 Temporal Graph Analysis Library

The important temporal graph algebra operators supported by our system are described below.

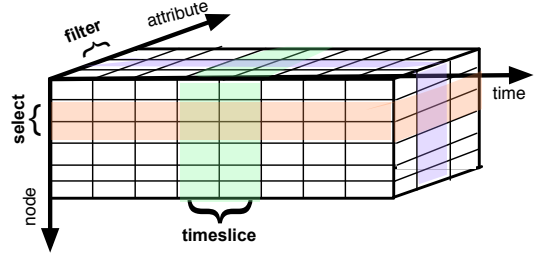


Figure 4: SoN: A set of nodes can be abstracted as a 3 dimensional array with temporal, node and attribute dimensions.

1. **Selection** accepts an SoN or an SoTS along with a boolean function on the nodes or the subgraphs, and returns an SoN or SoTS. It performs *entity-centric filtering* on the operand, and does not alter temporal or attribute dimensions of the data.
2. **Timeslicing** accepts an SoN or an SoTS along with a timepoint (or time interval)  $t$ , finds the state of each of individual nodes or subgraphs in the operand as of  $t$ , and returns it as another SoN or SoTS, respectively (SoN/SoTS can represent sets of static nodes or subgraphs as a well). The operator can accept a list of timepoints as input and return a list.
3. **Graph** accepts an SoN and returns an in-memory *Graph* object containing the nodes in the SoN (with only the edges whose both endpoints are in the SoN). An optional parameter,  $t_p$ , may be specified to get a *GraphS* valid at time  $t_p$ .
4. **NodeCompute** is analogous to a *map* operation; it takes as input an SoN (or an SoTS) and a function, and applies the function to all the individual nodes (subgraphs) and returns the results as a set.
5. **NodeComputeTemporal**. Unlike *NodeCompute*, this operator takes as input a function that operates on a static node (or subgraph) in addition to an SoN (or an SoTS); for each node (subgraph), it returns a sequence of outputs, one for each different state (version) of that node (or subgraph). Optionally, the user may specify another function (*NodeComputeDelta*, described next) that operates on the delta between two versions of a node (subgraph). Another optional parameter is a method describing points of time at which computation needs to be performed; in the absence of it, the method will be evaluated at all the points of change.
6. **NodeComputeDelta** operator takes as input: (a) a function that operates on a static node (or subgraph) and produces an output quantity, (b) an SoN (or an SoTS), (c) a function that operates on the following: a static node (or subgraph), some auxiliary information pertaining to that state of the node (or subgraph), the value of the quantity at that state, and an *update* (event) to it. This operator returns a sequence of outputs, one for each state of the node (or subgraph), similar to *NodeComputeTemporal*. However, it computes the required quantity for each version incrementally instead of computing it afresh. An optional parameter is the method describing points of time at which to base the comparison. An optional parameter is a method describing points of time at which computation needs to be performed; in the absence of it, the method will be evaluated at all the points of change.
7. **Compare** operator takes as input two SoNs (or two SoTSs) and a scalar function (returning a single value), computes the function value over all the individual components, and returns the differences between the two as a set of (*node-id, difference*) pairs. This operator tries to abstract the common operation of

comparing two different snapshots of a graph at different time points. A simple variation of this operator takes a single SoN (or SoTS) and two timepoints as input, and does the compare on the timeslices of the SoN as of those two timepoints. An optional parameter is the method describing points of time at which to base the comparison.

8. **Evolution** operator samples a specified quantity (provided as a function) over time to return evolution of the quantity over a period of time. An optional parameter is the method describing points of time at which to base the evolution.
9. **TempAggregation** abstractly represents a collection of temporal aggregation operators such as *Peak*, *Saturate*, *Max*, *Min*, and *Mean* over a scalar timeseries. The aggregation operations are performed over a specified quantity for an SoN or SoTS. For instance, finding “times at which there was a *peak* in the network density” is used to find eventful timepoints of high interconnectivity such as conversations in a cellular network, or high transactional activity in a financial network.

### 5.3 System Implementation

The library is implemented in Python and Java and is built on top of the Spark API. The choice of Spark provides us with an efficient in-memory cluster compute execution platform, circumventing dealing with the issues of data partitioning, communication, synchronization, and fault tolerance. We provide a GraphX integration for utilizing the capabilities of the Spark based graph processing system for static graphs. Note that while we use Spark for implementation, the concepts presented as a part of the TAF are general and can be implemented over other distributed frameworks such as DryadLINQ<sup>6</sup>.

The key abstraction in Spark is that of an RDD, which represents a collection of objects of the same type, stored across a cluster. SoN and SoTS are implemented as RDDs of NodeT and SubgraphT respectively (i.e., as RDDTG<NodeT> and RDDTG<SubgraphT>, where RDDTG extends RDD class). Note that the in-memory graph objects may be implemented using any popular graph representation, specially the ones that support useful libraries on top. We now describe in brief the implementation details for NodeT and SubgraphT, followed by details of the incremental computational operator, and the parallel data fetch operation.

Figure 5 shows sample code snippets for three different analytical tasks – (a) finding the node with the *highest clustering coefficient* in a historical snapshot; (b) *comparing different communities* in a network; (c) finding the *evolution of network density* over a sample of ten points.

**NodeT and SubgraphT:** A set of temporal nodes is represented with an RDD of NodeT (temporal node). A temporal node contains the information for a node during a specified time interval. The question of the appropriate physical storage of the NodeT (or SubgraphT) structure is quite similar to storing a temporal graph on disk such as the one using a DeltaGraph or a TGI, however, in-memory instead of disk. Since NodeT is fetched at query time, it is preferable to avoid creating a complicated index, since the cost to create the index at query time is likely to offset any access latency benefits due to the index. Upon observing several analysis tasks, we noticed that the common access pattern is mostly in chronological order, i.e., the query requesting the subsequent versions or changes, in order of time. Hence, we store NodeT (and SubgraphT) as an initial snapshot of the node (or subgraph), followed by a list of chronologically sorted events. It provides methods such as `GetStartTime()`, `GetEndTime()`, `GetStateAt()`,

<sup>6</sup><http://research.microsoft.com/en-us/projects/DryadLINQ/>

```

tgiH = TGIHandler(tgiconf, "wiki", sparkcontext)
sots = SOTS(k=1, tgiH).Timeslice("t = July 14,2002").fetch()
nm = NodeMetrics()
nodeCC = sots.NodeCompute(nm.LCC, append = True, key="cc")
maxLCC = nodeCC.Max(key="cc")

```

(a) Finding node with highest local clustering coefficient

```

tgiH = TGIHandler(tgiconf, "snet", sparkcontext)
son = SON(tgiH).Timeslice('t >= Jan 1,2003 and t < Jan 1, '
    \',2004').Filter("community")
sonA=son.Select("community =\A\").fetch()
sonB=son.Select("community =\B\").fetch()
compAB = SON.Compare(sonA, sonB, SON.count())
print('Average membership in 2003,')
print(A=%s\tB=%s'%(mean(compAB[0]), mean(compAB[1])))

```

(b) Comparing two communities in a network

```

tgiH = TGIHandler(tgiconf, "wiki", sparkcontext)
son = SON(tgiH).Select("id < 5000").Timeslice("t >= oct"
    \',24, 2008').fetch()
gm = GraphMetrics()
evol = son.GetGraph().Evolution(gm.density, 10)
print('Graph density over 10 points=%s'%evol)

```

(c) Evolution of network density

Figure 5: Examples of analytics using the TAF Python API.

`GetIterator()`, `Iterator.GetNextVersion()`, `Iterator.GetNextEvent()`, and so on. We omit their details as their functionality is apparent from the nomenclature.

**NodeComputeDelta:** `NodeComputeDelta` evaluates a quantity over each NodeT (or SubgraphT) using two supplied methods,  $f()$  which computes the quantity on a state of the node or subgraph, and  $f_{\Delta}()$ , which updates the quantity on a state of the node or subgraph for a given set of event updates. Consider a simple example of computing the fraction of all nodes that contain a specific attribute value in a given SubgraphT. If this was performed using `NodeComputeTemporal`, the quantity will be computed afresh on each new version of the subgraph, which would cost  $\mathcal{O}(N.T)$  operations where  $N$  is the size of the operand (number of nodes) and  $T$  is the number of versions. However, using incremental computation, each new version after the first snapshot can be processed in constant time, which adds up to  $\mathcal{O}(N+T)$ . While performing incremental computation, the corresponding  $f_{\Delta}()$  method is expected to be defined so as to evaluate the nature of the event – whether it brings about any change in the output quantity or not, i.e., a scalar change value based upon the actual event and the concerned portions of the state of the graph, and also update the auxiliary structure, if used. Code snippet in Figure 6 illustrates the usage of `NodeComputeTemporal` and `NodeComputeDelta` in a similar example.

Consider a somewhat more intricate example, where one needs to find counts of a small pattern *over time* on an SoTS, such as finding the occurrence of a subgraph pattern in the data graph’s history. In order to perform such pattern matching over long sequences of subgraph versions, it is essential to maintain certain inverted indexes which can be looked up to answer in constant time whether an event has caused a change in the answer from a previous state or caused a change in the index itself, or both. Such inverted indexes, quite common to subgraph pattern matching, are required to be updated with every event; otherwise, with every new event update, we would need to look up the new state of the subgraph afresh which would simply reduce it to performing non-indexed subgraph pattern matching over new snapshots of a subgraph at each time point, which is a fairly expensive task. In order to utilize a constantly updated set of indices, the auxiliary information, which is a parameter and a return type for  $f_{\Delta}()$ , can be utilized. Note that such an incremental computational operator opens up possibilities of utiliz-

```

tgiH = TGIHandler(tgiConf, "dblp", sparkContext)
sots = SOTS(k=2, tgiH).Timeslice('t >= Nov 1,2009 and t < Nov 30,\'
2009').fetch()
labelCount = sots.NodeComputeTemporal(fCountLabel)
labelCount = sots.NodeComputeDelta(fCountLabel, fCountLabelDel)

def fCountLabel(g):
    labCount = 0
    for node in g.GetNodes():
        if node.GetPropValue('EntityType') == 'Author':
            labCount += 1
    return labCount

def fCountLabelDel(gPrev, valPrev, event):
    valNew = valPrev
    if event.Type == EType.AttribValAlter:
        if event.Attribkey == 'EntityType':
            if event.PrevVal == 'Author':
                valNew = valPrev - 1
            else if event.NextVal == 'Author':
                valNew = valPrev + 1
    return valNew

```

Figure 6: Incremental computation using different options: NodeComputeTemporal and NodeComputeDelta to compute counts of nodes with a specific label in subgraphs over time.

ing a large body of algorithmic work in online and streaming graph query evaluation for the purpose of graph analytics.

**Specifying interesting time points:** In the map-oriented version operators on an SoN or an SoTS, the time points of evaluation, by default, are all the points of change in the given operand. However, a user may choose to provide a definition of which points to select. This can be as simple as returning a constant set of time-points, or based on a more complex function of the operand(s). Except the Compare operator, which accepts two operands, other operators allow an optional function, which works on a single temporal operand; the compare accepts a similar function that operates on two such operands. Two such examples can be seen in Figure 7.

```

tgiH = TGIHandler(tgiConf, "wiki", sparkContext)
son = SON(tgiH).select("id < 5000").Timeslice("t >= oct"
"24, 2008").fetch()
gm = GraphMetrics()
evol = son.GetGraph().Evolution(gm.density,
\selectTimepointsMinimal)
print('Graph density over 3 points=%s'%evol)

def selectTimepointsMinimal(son):
    time_arr = []
    st = son.GetStartTime()
    et = son.GetEndTime()
    time_arr.append(st)
    time_arr.append((st + et)/2)
    time_arr.append(et)
    return time_arr

```

Figure 7: Using the optional timepoint specification function for an Evolution query with the start, middle and endpoint of SON.

**Data Fetch:** In a temporal graph analysis task, we first need to instantiate a TGI connection handler instance. It contains configurations such as address and port of the TGI query manager host, graph-id, and a SparkContext object. Then, a SON (or SOTS) object is instantiated by passing the reference to the TGI handler, and any query specific parameters (such as k-value for fetching 1-hop neighborhoods with SOTS). The next few instructions specify the semantics of the graph to be fetched from the TGI. This is done through the commands explained in Section 5.1, such as the Select, Filter, Timeslice, etc. However, the actual retrieval from the index doesn't happen until the first statement following the specification instructions. A fetch() command can be used explicitly to tell the system to perform the fetch operation. Upon the fetch() call, the analytics framework sends the combined instructions to the query planner of the TGI, which translates those instructions into an optimal retrieval plan. This prevents the system from retrieving large amounts of data from the index that is a superset of the required information and prune it later.

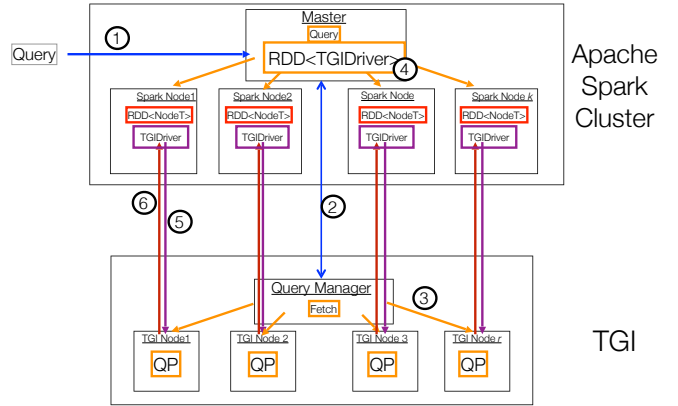


Figure 8: A flow diagram of the parallel fetch operation between the TGI and TAF clusters. The numbers in circles indicate the relative order of events and arrowheads indicate the direction of flow.

The analytics engine runs in parallel on a set of machines, so does the graph index. The parallelism at both places speeds up and scales both the tasks. However, if the retrieval graph at the TGI cluster was aggregated at the Query Manager and sent serially to the master of the analytical framework engine after which it was distributed to the different machines on the cluster, it would create a space and time bottleneck at the Query Manager and the master, respectively, for large graphs. In order to bypass this situation, we have designed a parallel fetch operation, in which there is a direct communication between the nodes of the analytics framework cluster and the nodes of the TGI cluster. This happens through a protocol that can be seen in Figure 8 and summarized below:

1. Analytics query containing fetch instructions is received by the TAF master.
2. A handshake between the TAF master and TGI query manager is established. The latter receives fetch instructions and the former is made aware of the active TGI query processors.
3. Parallel fetch starts at the TGI cluster.
4. The TAF master instantiates a TGIDriver instance at each of its cluster machines wrapped in a RDD.
5. Each node at the TAF performs a handshake with one or more of the TGI nodes.
6. Upon completion of fetch at TGI, the individual TGI nodes transfer the SoN to an RDDs on the corresponding TAF nodes.

More details on the TGI-TAF integration can be found in the longer version of the paper [22].

## 6. EXPERIMENTAL EVALUATION

In this section, we empirically evaluate the efficiency of TGI and TAF. To recap, TGI is a persistent store for entire histories of large graphs, that enables fast retrieval for a diverse set of graph primitives – snapshots, subgraphs, and nodes at past time points or across intervals of time. We primarily highlight the performance of TGI across the entire spectrum of retrieval primitives. We are not aware of a baseline that may compete with TGI across all or a substantial subset of these retrieval primitives. Specialized alternatives such as DeltaGraph for snapshot retrieval is highly unsuitable for node or neighbor version retrieval; a version centric index may be specialized for node-version retrieval but is highly unsuitable for snapshot

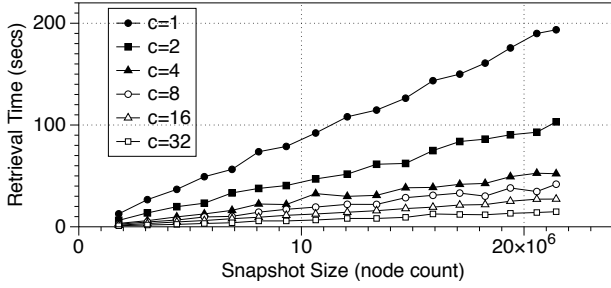


Figure 9: Snapshot retrieval times for varying parallel fetch factor ( $c$ ), on Dataset 1;  $m = 4$ ;  $r = 1$ ,  $ps = 500$ .

or neighborhood-version style retrieval. Also note that TGI generalizes all the known approaches including those two; using appropriate parameter configurations, it can even converge to any specific alternative. Secondly, we demonstrate the scalability of TGI design through experiments on parallel fetching for large and varying data sizes. Finally, we also report experiments demonstrating computational scalability of the TAF for a graph analysis task, as well as the benefits of our incremental computational operator.

**Datasets and Notation:** We use four datasets: (1) Wikipedia citation network consisting of 266,769,613 edge addition or modification events from Jan 2001 to Sept 2010. At its largest point, the graph consists of 21,443,529 nodes and 122,075,026 edges; (2) We augment Dataset 1 by adding around 333 million synthetic events which randomly add new edges or delete existing edges over a period of time, making a total of 700 million events; (3) Similarly, we add 733 million events, making the total around 1 billion events; (4) Using a Friendster gaming network snapshot, we add synthetic dates at uniform intervals to 500 million events with a total of approximately 37.5 million nodes and 500 million edges.

Following key parameters that are varied in the experiments: data store machine count ( $m$ ), replication across dataset ( $r$ ), number of parallel fetching clients ( $c$ ), eventlist size ( $l$ ), snapshot or eventlist partition size ( $ps$ ), and Spark cluster size ( $m_a$ ).

We conducted all experiments on an Amazon EC2 cluster. Cassandra ran on machines containing 4 cores and 15GB of available memory. We did not use row caching and the actual memory consumption was much lower than the available limit on those machines. Each fetch client ran on a single core with up to 7.5GB available memory. The machines with TAF nodes running Spark workers ran on a single core and 7.5GB of available memory each.

**Snapshot retrieval:** Figure 9 shows the snapshot retrieval times for Dataset 1 for different values of the parallel fetch factor,  $c$ . We observe that the retrieval cost is directly proportional to the size of the output. Further, using multiple clients to retrieve the snapshots in parallel gives near-linear speedup, especially with low parallelism. This demonstrates that TGI can exploit available parallelism well. We expect that with higher values of  $m$  (i.e., if the index were distributed across a more machines), linear speedup would be seen for larger values of  $c$  (this is corroborated by the next set of experiments). Figure 11c shows snapshot retrieval times for Dataset 4.

Figure 10 shows snapshot retrieval performance for three different sets of values for  $m$  and  $r$ . We can see that while there is no considerable difference in performance across the different configurations, using two storage machines slightly decreases the query latency over using one machine, in the case of a single query client,  $c = 1$ . For higher  $c$  values, we see that  $m = 2$  has a slight edge over  $m = 1$ . Also, the behavior for the two  $m = 1$  and  $m = 2$ ;  $r = 2$  cases are quite similar for same  $c$  values. However, we observed that the

latter case allows a higher possibility of  $c$  value whereas the former peaks out at a lower  $c$  value. Further, compression for deltas is negligible for TGI. We omit the detailed points of our investigation, but Figure 11a is representative of the general behavior.

In the special case of  $ps \rightarrow \infty$ , TGI becomes structurally equivalent to a DeltaGraph. While DeltaGraph provides the most efficient way of performing snapshot retrieval, we show that using lower values of  $ps$  in TGI only has a marginal impact on the performance of snapshot retrieval (Figure 11b). This occurs due to the TGI design policy of storing all the partitions of a delta contiguously in a cluster and avoiding any additional seek costs. Hence, DeltaGraph is subsumed as a part of TGI and we omit further comparisons in this respect. Also note that the internals of snapshot retrieval through DeltaGraph have been thoroughly explored in our prior work [21].

**Node History Retrieval:** Smaller eventlists or partition sizes provide a lower latency time for retrieving different versions of a node, which can be seen in Figure 12a and Figure 12c, respectively. This is primarily due to the reduction in effort for fetching and deserialization. A higher parallel fetch factor is effective in reducing the latency for version retrieval (Figure 12b). Note that the performances of version and snapshot retrieval for varying partition sizes are opposite. However, smaller eventlist sizes benefit both version and snapshot retrieval. Node version retrieval for Dataset 4 shows a similar behavior, which can be seen in Figure 14.

**Neighborhood Retrieval:** We compared the performance of retrieving 1-hop neighborhoods, both static and specific versions, using different graph partitioning and replication choices. A topological, flow-based partitioning accesses fewer graph partitions compared to a random partitioning scheme, and a 1-hop neighborhood replication restricts the access to a single partition. This can be seen in Figure 13a for 1-hop neighborhood retrieval latencies. As discussed in Section 4, the 1-hop replication does not affect other queries involving snapshots or individual nodes, as the replicated portion is stored separately from the original partition. In case of a 2-hop neighborhood retrieval, there are similar performance benefits over random partitioning.

**Increasing Data Over Time:** We observed the fetch performance of TGI with an increasing size of the index. We measured the latencies for retrieving certain snapshots upon varying the time duration of the graph dataset, as shown in Figure 13b. Datasets 2 and 3 contain additional 333 million and 733 million events over dataset 1, respectively. Only a marginal difference in snapshot retrieval performance demonstrates TGI’s scalability for large datasets.

**Conducting Scalable Analytics:** We examined TAF’s performance through an analytical task for determining the highest local clustering coefficient in historical graph snapshot. Figure 13c shows compute times for the given task on different graph sizes, as well as varying size of the Spark cluster. Speedups due to parallel execution can be observed, especially for larger datasets.

**Temporal Computation:** Earlier in the chapter, we presented two separate ways of computing a quantity over changing versions of a graph (or node). Those include, evaluating the quantity on different versions of the graph separately, and alternatively, performing it in an incremental fashion, utilizing the result for the previous version and updating it with respect to the graph updates. This can be seen for a simple node label counting task in Figure 6. the benefits due to the incremental (`NodeComputeDelta` operator) computation over a version-based computation (`NodeComputeTemporal` operator) can be seen in Figure 15.

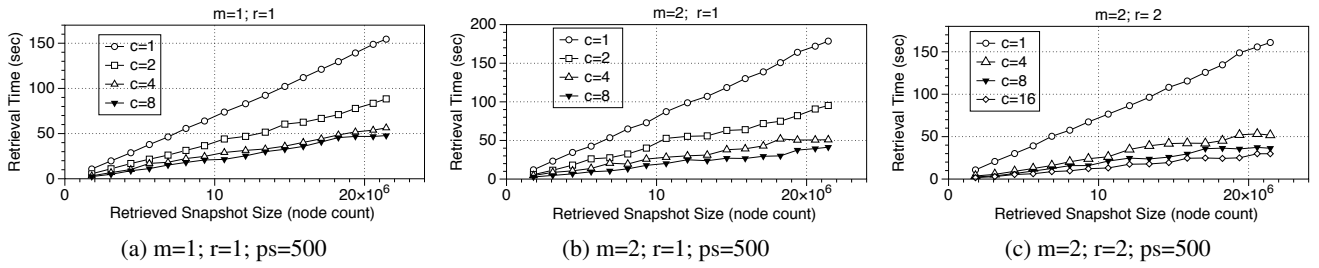


Figure 10: Snapshot retrieval times across different  $m$  and  $r$  values on Dataset 1.

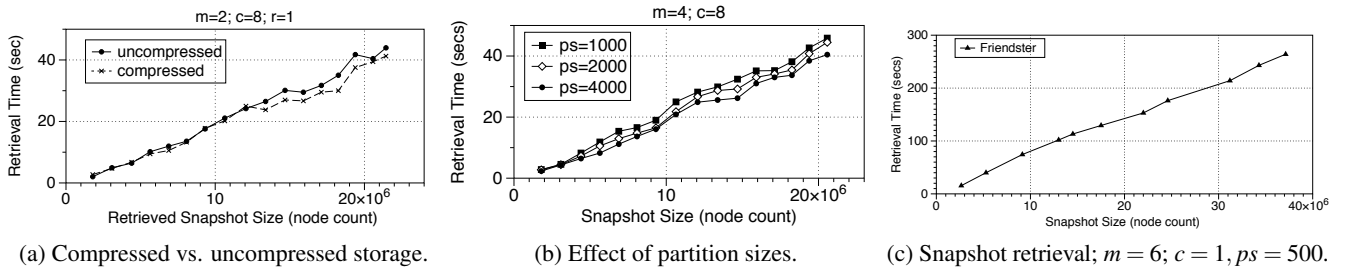


Figure 11: Snapshot retrieval across various parameters.  $r=1$ ; Dataset 1 for (a) and (b); Dataset 4 for (c).

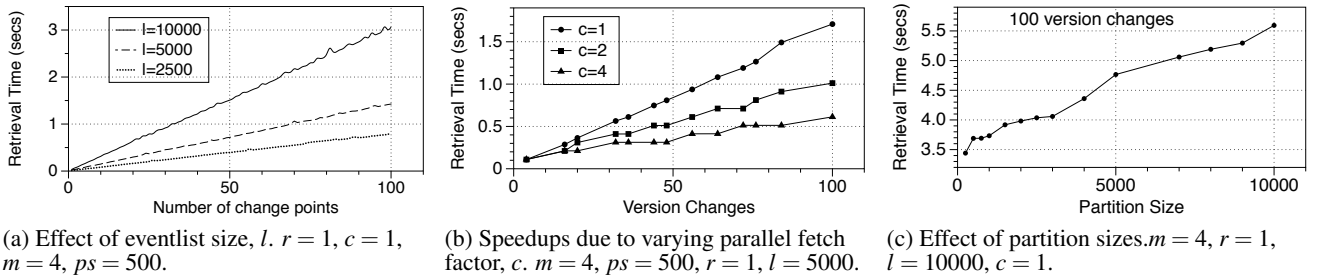


Figure 12: Node version retrieval across various parameters.

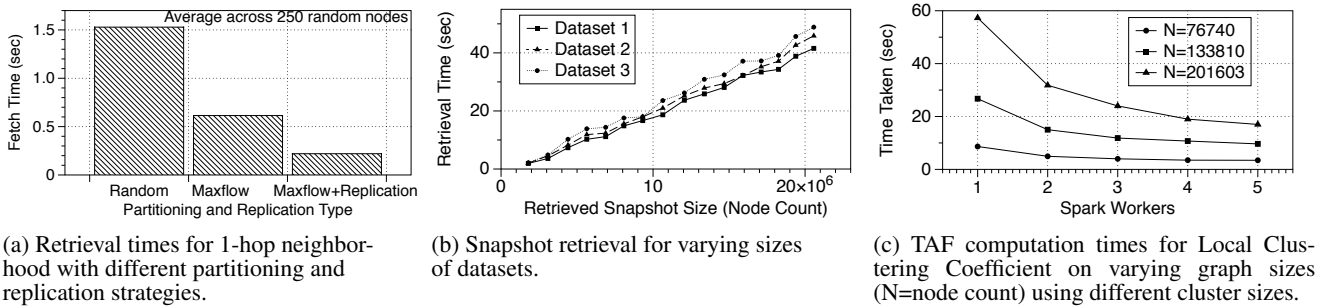


Figure 13: Experiments for partitioning strategies and growing data size ( $m=4, r=2, c=4, ps=500$ ); TAF analytics computation.

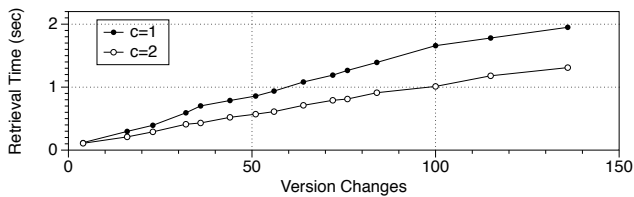


Figure 14: Node versions; Dataset 4;  $m=6, r=1, c=1, ps=500$ .

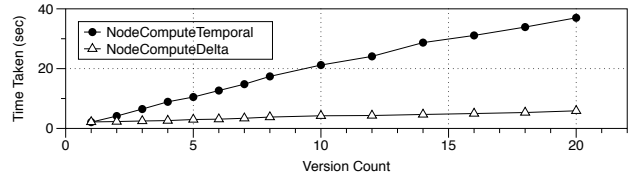


Figure 15: Label counting for 2-hop neighborhoods using (NodeComputeTemporal) and (NodeComputeDelta), respectively. We report cumulative time taken (excluding fetch time) for varying version counts on Dataset 4 with 2 Spark workers.

## 7. CONCLUSION

Graph analytics are increasingly considered crucial in obtaining insights about how interconnected entities behave, how information spreads, what are the most influential entities in the data, and many other characteristics. Analyzing the history of a graph's evolution can provide significant additional insights, especially about the future. Most real-world networks however, are large and highly dynamic. This leads to creation of very large histories, making it challenging to store, query, or analyze them. In this paper, we presented a novel Temporal Graph Index that enables compact storage of very large historical graph traces in a distributed fashion, supporting a wide range of retrieval queries to access and analyze only the required portions of the history. Our experiments demonstrate its efficient retrieval performance across a wide range of queries, and can effectively exploit parallelism in a distributed setting. We also presented a distributed analytics framework, built on top of Apache Spark, that allows analysts to quickly write complex temporal analysis tasks and execute them scalably over a cluster.

**Acknowledgments:** This work was supported by NSF under grant IIS-1319432, an IBM Collaborative Research Award, and an Amazon AWS in Education Research grant.

## 8. REFERENCES

- [1] J.-w. Ahn, C. Plaisant, and B. Shneiderman. A task taxonomy for network evolution analysis. *IEEE Transactions on Visualization and Computer Graphics*, 2014.
- [2] L. Arge and J. Vitter. Optimal dynamic interval management in external memory. In *FOCS*, 1996.
- [3] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized pagerank. *VLDB*, 2010.
- [4] A. Barrat, M. Barthelemy, and A. Vespignani. *Dynamical processes on complex networks*. 2008.
- [5] T. Y. Berger-Wolf and J. Saia. A framework for analysis of dynamic social networks. In *SIGKDD*, 2006.
- [6] G. Blankenagel and R. Guting. External segment trees. *Algorithmica*, 1994.
- [7] Z. Cai, D. Logothetis, and G. Siganos. Facilitating real-time graph mining. In *CloudDB*, 2012.
- [8] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EUROSYS*, 2012.
- [9] D. Eisenberg, E. M. Marcotte, I. Xenarios, and T. O. Yeates. Protein function in the post-genomic era. *Nature*, 2000.
- [10] B. Gedik and R. Bordawekar. Disk-based management of interaction graphs. *TKDE*, 2014.
- [11] A. Ghrab, S. Skhiri, S. Jouili, and E. Zimányi. An analytics-aware conceptual model for evolving graphs. In *Data Warehousing and Knowledge Discovery*. 2013.
- [12] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: graph processing in a distributed dataflow framework. In *OSDI*, 2014.
- [13] F. Grandi. T-SPARQL: A TSQL2-like temporal query language for RDF. In *ADBIS*, 2010.
- [14] D. Greene, D. Doyle, and P. Cunningham. Tracking the evolution of communities in dynamic social networks. In *ASONAM*, 2010.
- [15] T. Gross, C. J. D. D'Lima, and B. Blasius. Epidemic dynamics on an adaptive network. *Physical review*, 2006.
- [16] R. Gulati and M. Gargiulo. Where do interorganizational networks come from? *American journal of sociology*, 1999.
- [17] H. He and A. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, 2008.
- [18] W. Huo and V. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *SSDBM*, 2014.
- [19] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In *ACM SIGKDD*, 2011.
- [20] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May. Timeline index: A unified data structure for processing queries on temporal data in SAP HANA. In *ACM SIGMOD*, 2013.
- [21] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *IEEE ICDE*, 2013.
- [22] U. Khurana and A. Deshpande. Storing and analyzing historical graph data at scale. *CoRR*, abs/1509.08960, 2015.
- [23] G. Koloniari and E. Pitoura. Partial view selection for evolving social graphs. In *GRADES workshop*, 2013.
- [24] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.
- [25] A. Labouseur, J. Birnbaum, J. Olsen, P., S. Spillane, J. Vijayan, J. Hwang, and W. Han. The G\* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, 2014.
- [26] K. Lerman and R. Ghosh. Information contagion: An empirical study of the spread of news on digg and twitter social networks. *ICWSM*, 2010.
- [27] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *VLDB*, 2012.
- [28] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient Graph Analytics Using Large Multiversioned Arrays. In *ICDE*, 2015.
- [29] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD*, 2010.
- [30] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, et al. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM TOS*, July 2015.
- [31] G. Ozsoyoglu and R. Snodgrass. Temporal and real-time databases: a survey. *IEEE TKDE*, 1995.
- [32] R. K. Pan and J. Saramäki. Path lengths, correlations, and centrality in temporal networks. *Physical Review E*, 2011.
- [33] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *The Semantic Web*. 2006.
- [34] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. In *VLDB*, 2011.
- [35] B. Salzberg and V. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 1999.
- [36] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *ACM SIGMOD*, 2013.
- [37] R. Snodgrass and I. Ahn. A taxonomy of time in databases. In *SIGMOD*, 1985.
- [38] I. W. Taylor, R. Linding, D. Warde-Farley, Y. Liu, C. Pesquita, D. Faria, S. Bull, T. Pawson, Q. Morris, and J. L. Wrana. Dynamic modularity in protein interaction networks predicts breast cancer outcome. *Nature biotechnology*, 2009.
- [39] V. Tsotras and N. Kangelaris. The snapshot index: an I/O-optimal access method for timeslice queries. *Inf. Syst.*, 1995.
- [40] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX conference on Hot topics in cloud computing*, 2010.



# Providing Serializability for Pregel-like Graph Processing Systems

Minyang Han

David R. Cheriton School of Computer Science  
University of Waterloo  
m25han@uwaterloo.ca

Khuzaima Daudjee

David R. Cheriton School of Computer Science  
University of Waterloo  
kdaudjee@uwaterloo.ca

## ABSTRACT

There is considerable interest in the design and development of distributed systems that can execute algorithms to process large graphs. Serializability guarantees that parallel executions of a graph algorithm produce the same results as some serial execution of that algorithm. Serializability is required by many graph algorithms for accuracy, correctness, or termination but existing graph processing systems either do not provide serializability or cannot provide it efficiently. To address this deficiency, we provide a complete solution that can be implemented on top of existing graph processing systems. Our solution formalizes the notion of serializability and the conditions under which it can be provided for graph processing systems. We propose a novel partition-based synchronization approach that enforces these conditions to efficiently provide serializability. We implement our partition-based technique into the open source graph processing system Giraph and demonstrate that our technique is configurable, transparent to algorithm developers, and provides large across-the-board performance gains of up to  $26\times$  over existing techniques.

## 1. INTRODUCTION

Graph data processing has become ubiquitous due to the large quantities of data collected and processed to solve real-world problems. For example, Facebook processes massive social graphs to compute popularity and personalized rankings, find communities, and propagate advertisements for over 1 billion monthly active users [16]. Google processes web graphs containing over 60 trillion indexed webpages to determine influential vertices [19].

Graph processing solves real-world problems through algorithms that are implemented and executed on *graph processing systems*. These systems provide programming and computation models for graph algorithms as well as correctness guarantees that algorithms require.

One key correctness guarantee is *serializability*. Informally, a graph processing system provides serializability if it

can guarantee that parallel executions of an algorithm, implemented with its programming and computation models, produce the same results as some serial execution of that algorithm [18].

Serializability is required by many algorithms, for example in machine learning, to provide both theoretical and empirical guarantees for convergence or termination. Parallel algorithms for combinatorial optimization problems experience a drop in performance and accuracy when parallelism is increased without consideration for serializability. For example, the Shotgun algorithm for  $L_1$ -regularized loss minimization parallelizes sequential coordinate descent to handle problems with high dimensionality or large sample sizes [11]. As the number of parallel updates is increased, convergence is achieved in fewer iterations. However, after a sufficient degree of parallelism, divergence occurs and *more* iterations are required to reach convergence [11]. Similarly, for energy minimization on  $NK$  energy functions (which model a system of discrete spins), local search techniques experience an abrupt degradation in the solution quality as the number of parallel updates is increased [32]. Some algorithms also require serializability to prevent unstable accuracy [27] while others require it for statistical correctness [17]. Graph coloring requires serializability to terminate on dense graphs [18] and, even for sparse graphs, will use significantly fewer colors and complete in only a single iteration (rather than many iterations) when executed serializably.

Providing serializability in a graph processing system is fundamentally a system-level problem that informally requires: (1) vertices see up-to-date data from their neighbors and (2) no two neighboring vertices execute concurrently. The general approach is to pair an existing system or computation model with a *synchronization technique* that enforces conditions (1) and (2). Despite this, of the graph processing systems that have appeared over the past few years, few provide serializability as a configurable option. For example, popular systems like Pregel [28], Giraph [1], and GPS [31] pair a vertex-centric programming model with the bulk synchronous parallel (BSP) computation model [34] but do not provide serializability.

Giraphx [33] provides serializability by pairing the asynchronous parallel (AP) model, which is an asynchronous extension of the BSP model, with the single-layer token passing and vertex-based distributed locking synchronization techniques. However, it implements these synchronization techniques as part of specific user algorithms rather than within the system, meaning algorithm developers must re-implement the techniques into every algorithm that they

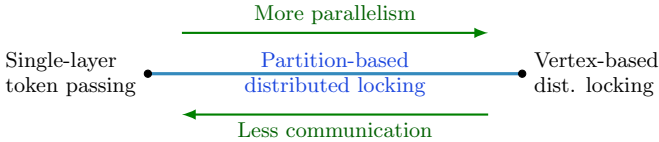


Figure 1: Spectrum of synchronization techniques.

write. Consequently, Giraphx unnecessarily couples and exposes internal system details to user algorithms, meaning serializability is neither a configurable option nor transparent to the algorithm developer. Furthermore, its implementation of vertex-based distributed locking unnecessarily divides each superstep, an iteration of computation, into multiple sub-supersteps in which only a subset of vertices can execute. This exacerbates the already expensive communication and synchronization overheads associated with the global synchronization barriers that occur at the end of each superstep [20], resulting in poor performance.

GraphLab [27], which now subsumes PowerGraph [18], takes a different approach by starting with an asynchronous implementation of the Gather, Apply, Scatter (GAS) computation model. This asynchronous mode (GraphLab async) avoids global barriers by using distributed locking. GraphLab async provides the option to execute with or without serializability and uses vertex-based distributed locking as its synchronization technique. However, GraphLab async suffers from high communication overheads [22, 20] and scales poorly with this technique. Moreover, neither GraphLab nor Giraphx provide a theoretical framework for proving the correctness of their synchronization techniques.

Irrespective of the specific system, synchronization techniques used to enforce conditions (1) and (2) fall on a spectrum that trades off parallelism with communication overheads (Figure 1). In particular, single-layer token passing and vertex-based distributed locking fall on the extremes of this spectrum: token passing uses minimal communication but unnecessarily restricts parallelism, forcing only one machine to execute at a time, while vertex-based distributed locking uses a dining philosopher algorithm to maximize parallelism but incurs substantial communication overheads due to every vertex needing to synchronize with their neighbors.

To overcome these issues, we first formalize the notion of serializability in graph processing systems and establish the conditions under which it can be provided. To the best of our knowledge, no existing work has presented such a formalization. To address the shortcomings of the existing techniques, we introduce a fundamental design shift towards *partition aware* synchronization techniques, which exploit graph partitions to improve performance. In particular, we propose a novel *partition-based distributed locking* solution that allows control over the coarseness of locking and the resulting trade-off between parallelism and communication overheads (Figure 1). We implement all techniques at the system level in the open source graph processing system Giraph so that they are performant, configurable, and transparent to algorithm developers. We demonstrate through experimental evaluation that our partition-based solution substantially outperforms existing techniques.

Our **contributions** are hence threefold: (i) we formalize the notion of serializability in graph processing systems and establish the conditions that guarantee it; (ii) we intro-

duce the notion of partition aware techniques and our novel partition-based distributed locking technique that enables control over the trade-off between parallelism and communication overheads; and (iii) we implement and experimentally compare the techniques with Giraph and GraphLab to show that our partition-based technique provides substantial across-the-board performance gains of up to  $26\times$  over existing synchronization techniques.

This paper is organized as follows. In Section 2, we provide background on the BSP, AP, and GAS models. In Section 3, we formalize serializability and, in Sections 4 and 5, describe both existing techniques and our partition-based approach. In Section 6, we detail their implementations in Giraph. We present an extensive experimental evaluation of these techniques in Section 7 and describe related work in Section 8 before concluding in Section 9.

## 2. BACKGROUND AND MOTIVATION

In this section, we introduce the computation models and give a concrete motivation for serializability.

### 2.1 BSP Model

Bulk synchronous parallel (BSP) [34] is a computation model in which computations are divided into a series of (BSP) *supersteps* separated by global barriers. Pregel (and Giraph) pairs BSP with a vertex-centric programming model, where vertices are the units of computation and edges act as communication channels.

Graph computations are specified by a user-defined compute function that executes, in parallel, on all vertices in each superstep. The function specifies how each vertex processes its received messages, updates its vertex value, and who to send messages to. Importantly, messages sent in one superstep can be consumed/processed by their recipients only in the next superstep. Vertices can vote to halt to become inactive but are reactivated by incoming messages. The computation terminates when all vertices are inactive and no more messages are in transit.

Pregel and Giraph use a master/workers configuration. The master machine partitions the input graph across worker machines, coordinates all global barriers, and performs termination checks based on the two aforementioned conditions. The graph is partitioned by *edge-cuts*: each vertex belongs to a single worker while an edge can span two workers. Finally, BSP is *push-based*: messages are pushed by the sender and buffered at the receiver.

As a running example, consider the greedy graph coloring algorithm. Each vertex starts with the same color (denoted by its vertex value) and, in each superstep, selects the smallest non-conflicting color based on its received messages, broadcasts this change to its neighbors, and votes to halt. The algorithm terminates when there are no more color conflicts. Consider an undirected graph of four vertices partitioned across two worker machines (Figure 2). All vertices broadcast the initial color 0 in superstep 1 but the messages are not visible until superstep 2. Consequently, in superstep 2, all vertices update their colors to 1 based on stale data. Similarly for superstep 3. Hence, vertices collectively oscillate between 0 and 1 and the algorithm never terminates. However, if we could ensure that only  $v_0$  and  $v_3$  execute in superstep 2 and only  $v_2$  and  $v_1$  execute in superstep 3, then this problem would be avoided. As we will show in Section 4.3, serializability provides precisely this solution.

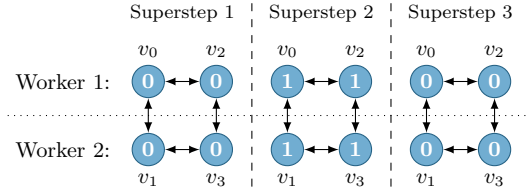


Figure 2: BSP execution of greedy graph coloring. Each graph is the state at the end of that superstep.

## 2.2 AP Model

The asynchronous parallel (AP) model improves on the BSP model by reducing staleness: instead of delaying all messages until the next superstep, vertices can immediately process any received messages (including ones sent in the same superstep). The AP model retains global barriers to separate supersteps, so messages that arrive too late to be seen by a vertex in superstep  $i$  (because the vertex was already executed) will be processed in the next superstep  $i+1$ . We use a more efficient and performant version of the AP model, described in [20], and its implementation in Giraph, which we will refer to as Giraph async.

Like BSP, the AP model can also fail to terminate for the greedy graph coloring algorithm. Consider again the undirected graph (Figure 3) and suppose that workers  $W_1$  and  $W_2$  execute their vertices sequentially as  $v_0$  then  $v_2$  and  $v_1$  then  $v_3$ , respectively. Furthermore, suppose the pairs  $v_0, v_1$  and  $v_2, v_3$  are each executed in parallel. Then the algorithm fails to terminate. Specifically, in superstep 1,  $v_0$  and  $v_1$  initialize their colors to 0 and broadcast to their neighbors. Due to the asynchronous nature of AP,  $v_2$  and  $v_3$  are able to see this message 0 and select the color 1. Similarly, in superstep 2,  $v_0$  and  $v_1$  now see each other’s message 0 (sent in superstep 1) and also the message 1 from  $v_2$  and  $v_3$ , respectively, so they update their colors to 2. Similarly for  $v_2$  and  $v_3$ , who now update their colors to 0. Ultimately, the graph’s state at superstep 4 returns to that at superstep 1, so the vertices are collectively cycling through three graph states in an infinite loop.

However, if we can force  $v_0$  to execute concurrently with  $v_3$  instead of  $v_1$  (and  $v_2$  with  $v_1$ ), then neighboring vertices will not simultaneously pick the same color. Furthermore, if we ensure that  $v_2$  and  $v_1$  wait for the messages from  $v_3$  and  $v_0$  to arrive before they execute, then they will have up-to-date information on all their neighbors’ colors. With these two constraints, graph coloring will terminate in just two supersteps. In Section 3, we present a theoretical framework for serializability that formalizes and incorporates these constraints as correctness criteria.

## 2.3 GAS Model

The Gather, Apply, and Scatter (GAS) model is used by GraphLab for both its synchronous and asynchronous modes, which we refer to as GraphLab sync and GraphLab async. These two system modes use the sync GAS and async GAS models, respectively. In GAS, each vertex pulls information from its neighbors in the gather phase, applies the accumulated data in the apply phase, and updates and activates neighboring vertices in the scatter phase.

Like Pregel and Giraph, GraphLab pairs GAS with a vertex-centric programming model. However, as evidenced by the Gather phase, GAS is *pull-based* rather than push-

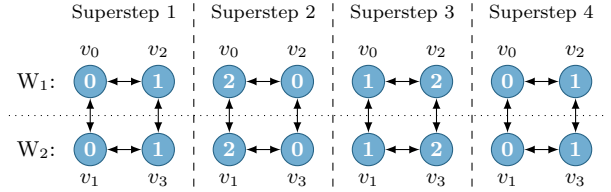


Figure 3: AP execution of greedy graph coloring. Each graph is the state at the end of that superstep.

based. Furthermore, GraphLab partitions graphs by *vertex-cut*: for each vertex  $u$ , one worker owns the *primary copy* of  $u$  while all other workers owning a neighbor of  $u$  get a local read-only replica of  $u$ .

Sync GAS is similar to BSP: vertices are executed in supersteps separated by global barriers and the effects of apply and scatter of one superstep are visible only to the gather of the next superstep. Async GAS, however, is different from AP as it has no notion of supersteps. To execute a vertex  $u$ , each GAS phase individually acquires a write lock on  $u$  and read locks on  $u$ ’s neighbors to prevent data races [3]. However, this does not provide serializability because GAS phases of different vertex computations can interleave [18]. To provide serializability, a synchronization technique must be added on top of async GAS. This technique prevents neighboring computations from interleaving by performing distributed locking over all three GAS phases.

Async GAS can similarly fail to terminate for graph coloring [18]. For example, for the graph in Figure 3, suppose both  $W_1$  and  $W_2$  each have two threads for their two vertices and that all four threads execute in parallel. Then, as described above, the GAS phases of different vertices will interleave, which causes vertices to see stale colors and so the execution is not guaranteed to terminate: it can become stuck in an infinite loop. In contrast, executing in async GAS with serializability will always terminate successfully.

## 3. SERIALIZABILITY

In this section, we present a theoretical framework that formalizes key conditions under which serializability can be provided for graph processing systems. Later, we show how serializability can be enforced efficiently in these systems.

### 3.1 Preliminaries

Since popular graph processing systems use a vertex-centric programming model, where developers specify the actions of a single vertex, we focus on vertex-centric systems. The formalisms that we will establish apply to all vertex-centric systems, irrespective of the computation models they use.

Existing work [18, 33] considers serializability for vertex-centric algorithms where vertices communicate only with their direct neighbors, which is the behaviour of the majority of algorithms that require serializability. For example, the GAS model supports only algorithms where vertices communicate with their direct neighbors [27, 18]. Thus, we focus on this type of vertex-centric algorithms. Our goal is to provide serializability transparently within the graph processing system, independent of the particular algorithm being executed.

In vertex-centric graph processing systems, there are two levels of parallelism: (1) between multiple threads within a single worker machine and (2) between the multiple worker

machines. Due to the distributed nature of computation, the input graph must be partitioned across the workers and so data replication will occur. To better understand this, let *neighbors* refer to both in-edge and out-edge neighbors.

**DEFINITION 1.** *A vertex  $u$  is a machine boundary vertex, or m-boundary for short, if at least one of its neighbors  $v$  belongs to a different worker machine from  $u$ . Otherwise,  $u$  is a machine internal, or m-internal, vertex.*

**DEFINITION 2.** *A replica is local if it belongs to the same worker machine as its primary copy and remote otherwise.*

Systems keep a read-only replica of each vertex on its owner’s machine and of each m-boundary vertex  $u$  on each of  $u$ ’s out-edge neighbor’s worker machines. This is a standard design used, for example, in Pregel, Giraph, and GraphLab. Remote replicas (of m-boundary vertices) exist due to graph partitioning: for vertex-cut partitioning,  $u$ ’s vertex value is explicitly replicated on every out-edge neighbor  $v$ ’s worker machine; for edge-cut,  $u$  is implicitly replicated because the message it sends to  $v$ , which is a function of  $u$ ’s vertex value, is buffered in the message store of  $v$ ’s machine. This distinction is unimportant for our formalism as we care only about whether replication occurs. Local replicas occur in push-based systems because message stores also buffer messages sent between vertices belonging to the same worker. In pull-based systems, local replicas are required for implementing synchronous computation models like sync GAS. For asynchronous models, pull-based systems may not always have local replicas (such as in GraphLab async) but we will consider the more general case in which they do (if they do not, then reads of such vertices will always trivially see up-to-date data).

**DEFINITION 3.** *A read of a replica is fresh if the replica is up-to-date with its primary copy and stale otherwise.*

An execution is *serializable* if it produces the same result as a serial execution in which all reads are fresh. Formally, this is one-copy serializability (1SR) [5]. Informally, we will say a system provides serializability if all executions conform to 1SR. In terms of traditional transaction terminology, we define a site as a worker machine, an item as a vertex, and a transaction as the execution of a single vertex. We detail such transactions next.

### 3.2 Transactions

We define a transaction to be the single execution of an arbitrary vertex  $u$ , consisting of a read on  $u$  and the replicas of  $u$ ’s in-edge neighbors followed by a write to  $u$ . The read acts only on  $u$  and its in-edge neighbors because  $u$  receives messages (or pulls data) from only its in-edge neighbors—it has no dependency on its out-edge neighbors. Denoting the read set as  $N_u = \{u, u$ ’s in-edge neighbors $\}$ , any execution of  $u$  is the transaction  $T_i = r_i[N_u]w_i[u]$ , or simply  $T_i(N_u)$  as all transactions are of the same form.

Any  $v \in N_u$  with  $v \neq u$  is also annotated to distinguish it from the other read-only replicas of  $v$ . For example, if  $u$  belongs to worker  $A$ , we annotate the read-only replica as  $v_A \in N_u$ . However, the next two sections will show how we can drop these annotations.

Our definition relies only on the fact that the system is vertex-centric and not on the nuances of specific computation models. For example, although BSP and AP have

a notion of supersteps, the  $i$  for a transaction  $T_i(N_u)$  has no relation to the superstep count. The execution of  $u$  in two different supersteps is represented by two different transactions  $T_i(N_u)$  and  $T_j(N_u)$ . Our definitions also work when there is no notion of supersteps, such as in async GAS, or when there are per-worker logical supersteps (supersteps that are not globally coordinated), such as proposed in [20]. Thus, the notion of a transaction that follows from our above definition is consistent with the standard notion of a transaction [5]: it captures, for graph processing, the atomic unit of operation that acts on shared data (the graph state).

### 3.3 Our Approach

In contrast to traditional database systems, graph processing systems present unique constraints that need to be taken into account for providing serializability.

First, Pregel-like graph processing systems such as Giraph and GraphLab do not natively support transactions: they are not database systems and thus have no notion of commits or aborts. The naive solution is to implement transaction support into all graph processing systems. However, this requires a fundamental redesign of each system, which is neither general nor reusable. Moreover, such a solution fails to be modular: it introduces performance penalties for graph algorithms that do not require serializability.

Second, an abort in a graph processing system can result in prohibitively expensive (and possibly cascading) rollbacks on the distributed graph state: a transaction often involves sending messages to vertices of different worker machines, the effects of which are difficult to undo. Consequently, solutions relying on optimistic currency control are a poor fit for graph processing due to the high cost of aborts.

However, for graph processing systems, a *write-all* approach [5] can be used to keep replicas up-to-date because graph processing systems replicate only for distributed computation and not for availability. When a worker machine fails, we lose a portion of the input graph and so cannot proceed with the computation. Indeed, failure recovery requires all machines to rollback to a previous checkpoint [1, 27, 28], meaning the problem of pending writes to failed machines never occurs. In contrast, a write-all approach is very expensive for traditional database systems because they replicate primarily for better performance and/or availability.

Furthermore, as detailed in Section 3.2, the read and write sets of each transaction are known a priori ( $N_v$  and  $v$ , respectively, for a transaction  $T_i(N_v)$ ), which means pessimistic concurrency control can be used to avoid costly aborts.

Our approach, at a high level, is to pair graph processing systems with a *synchronization technique*, which uses (1) a write-all approach to avoid data staleness and (2) pessimistic concurrency control to prevent conflicting transactions from starting. For the graph processing systems, (1) means vertices will always read from fresh replicas and so the system need not reason about versioning, while (2) means all transactions that start will commit, so aborts never occur and hence the system can treat all operations as final without needing explicit support for commits and aborts. Furthermore, this solution enables us to use transactions to formally reason about correctness without the burden of fundamentally redesigning each system to support transactions. Since aborts cannot occur, we also avoid the expensive penalties of distributed cascading rollbacks.

Using the definitions introduced in Section 3.2, we can

formalize our requirements into the following two conditions:

CONDITION C1. *Before any transaction  $T_i(N_u)$  executes, all replicas  $v \in N_u$  are up-to-date.*

CONDITION C2. *No transaction  $T_i(N_u)$  is concurrent with any transaction  $T_j(N_v)$  for all copies of  $v \in N_u$ ,  $v \neq u$ .*

Next, we will prove that 1SR can be provided by enforcing these two conditions.

### 3.4 Correctness

We first prove, in Lemma 1, that enforcing condition C1 simplifies the problem of providing 1SR to that of providing standard serializability on a single logical copy of each vertex (i.e., without data replication).

LEMMA 1. *If condition C1 is true, then it suffices to use standard serializability theory where operations are performed on a single logical copy of each vertex.*

PROOF. Condition C1 ensures that before every transaction  $T_i(N_u)$  executes, the replicas  $v \in N_u$  are all up-to-date. Then all reads  $r_i[N_u]$  see up-to-date replicas and are thus the same as reading from the primary copy of each  $v \in N_u$ . Hence, there is effectively only a single logical copy of each vertex, so we can apply standard serializability theory.  $\square$

Theorem 1 then establishes the relationship between 1SR and conditions C1 and C2.

THEOREM 1. *All executions are serializable for all input graphs if and only if conditions C1 and C2 are both true.*

PROOF SKETCH. Due to space constraints, we will briefly sketch only the key ideas of the proof. The full proof is provided in [21].

(IF) Since condition C1 is true, by Lemma 1 we can apply standard serializability theory [5]. It can then be shown that, for all input graphs, if condition C2 is true then it is impossible for the read and write sets of two arbitrary transactions to overlap. Thus, transactions never conflict and so the serialization graph [5] is always acyclic.

(ONLY IF) We prove the inverse (if either condition is false then there exists a non-serializable execution for some input graph) by considering an input graph with two vertices connected by an undirected edge. When C1 is true and C2 is false, we can construct a non-serializable history with two parallel but conflicting transactions. When C2 is true and C1 is false, replicas are no longer kept up-to-date and so, by placing each vertex on a different worker, we can construct a serial history that violates 1SR.  $\square$

### 3.5 Enforcing Serializability

The computation models from Section 2 do not enforce conditions C1 and C2 and therefore, by Theorem 1, do not provide serializability. Consequently, graph processing systems that implement these models also do not provide serializability. Moreover, these models do not guarantee fresh reads even under serial executions (on a single machine or under the sequential execution of multiple machines). For example, BSP effectively updates replicas lazily<sup>1</sup> because messages sent in one superstep, even if received, cannot be

<sup>1</sup>The “synchronous” in BSP refers to the global communication barriers, *not* the method of replica synchronization.

read by the recipient in the same superstep. Thus, both m-boundary *and* m-internal vertices (Definition 1) suffer stale reads under a serial execution. While AP reduces this staleness and can update local replicas eagerly, it propagates messages to remote replicas lazily without the guarantees of condition C1 and so stale reads can again occur under a serial execution of multiple machines.

As mentioned in Section 3.3, to provide serializability, we enforce conditions C1 and C2 by adding a synchronization technique (Sections 4 and 5) to the systems that implement the above computation models. These synchronization techniques implement a write-all approach for updating replicas, which is required for enforcing condition C1. They also ensure that a vertex  $u$  does not execute concurrently with any of its in-edge *and* out-edge neighbors. At first glance, this appears to be stronger than what condition C2 requires. However, suppose  $v$  is an out-edge neighbor of  $u$  and  $v$  is currently executing. Then if  $u$  does not synchronize with its out-edge neighbors, it will erroneously execute concurrently with  $v$ , violating condition C2 for  $v$ . Alternatively, if  $v$  is an out-edge neighbor of  $u$  then  $u$  is an in-edge neighbor of  $v$ , so they must not execute concurrently.

## 4. EXISTING SYNCHRONIZATION TECHNIQUES

Token passing and distributed locking are the two general approaches for implementing synchronization techniques that enforce conditions C1 and C2. In this section, we review two existing synchronization techniques: single-layer token passing and vertex-based distributed locking.

### 4.1 Preliminaries

How a synchronization technique implements a write-all approach (Section 3.3) depends on whether the computation model is synchronous or asynchronous.

In asynchronous computation models (AP and async GAS), replicas immediately apply received updates. Thus, local replicas can be updated eagerly, since there is no network communication (Section 6). Remote replicas, however, are updated lazily in a just-in-time fashion to provide communication batching. This lazy update is possible because all vertices are coordinated by a synchronization technique: any vertex  $v$  must first acquire a shared resource (e.g., a token or a fork) from its neighbor  $u$  before it can execute. Consequently, for an m-boundary vertex  $u$  with a replica on its neighbor  $v$ 's worker,  $u$ 's worker can buffer remote replica updates until  $v$  wants to execute (i.e., requests the shared resource)—at which point  $u$ 's machine will flush all pending remote replica updates (and ensure their receipt) before handing over the shared resource that allows  $v$  to proceed.

In contrast, synchronous computation models (BSP and sync GAS) hide updates from replicas until the next superstep. That is, replicas can be updated only after a global barrier. This means systems with synchronous models are limited to specialized synchronization techniques that keep replicas up-to-date by dividing each superstep into multiple sub-supersteps. This is significantly less performant than synchronization techniques for systems with asynchronous computation models, as detailed further in Section 6.

### 4.2 Single-Layer Token Passing

Single-layer token passing, considered in [33], is a simple technique that passes an exclusive global token in a round-

robin fashion between workers arranged in a logical ring. Each worker machine must execute with only one thread.

The worker machine holding the global token can execute both its m-internal and m-boundary vertices (Definition 1), while workers without the token can execute only their m-internal vertices. This prevents neighboring vertices from executing concurrently since each m-internal vertex and its neighbors are executed by a single thread, so there is no parallelism, while an m-boundary vertex can execute only when its worker machine holds the exclusive token.

To enforce condition C1, local replicas must be updated eagerly, while remote replicas of a worker’s m-boundary vertices can be updated in batch before a worker passes along the global token (as updates will arrive before the token). Per Section 4.1, this is possible with asynchronous computation models. Thus, for asynchronous models, single-layer token enforces conditions C1 and C2 and, by Theorem 1, provides serializability. However, this technique does not provide serializability for synchronous computation models as they cannot update local replicas eagerly.

### 4.3 Vertex-based Distributed Locking

Vertex-based distributed locking, unlike token passing, pairs threads with individual vertices to allow all vertices to attempt to execute in parallel. As motivated in Section 2.1, the key idea is to coordinate these vertices such that neighboring vertices do not execute concurrently, while also addressing issues such as deadlock and fairness.

This coordination is achieved using the Chandy-Misra algorithm [13], which solves the hygienic dining philosophers problem, a generalization of the dining philosophers problem. In this problem, each philosopher is either thinking, hungry, or eating and must acquire a shared fork from each of its neighbors to eat. Philosophers can communicate with their neighbors by exchanging forks and request tokens for forks. The “dining table” is effectively an undirected graph where each vertex is a philosopher and each edge is associated with a shared fork: a philosopher  $u$  must acquire  $\deg(u)$  forks to eat. The Chandy-Misra algorithm ensures no neighbors eat at the same time, guarantees fairness (no philosopher can hog its forks), and prevents deadlocks and starvation [13]. Hence, condition C2 is enforced.

To enforce condition C1, local replicas are updated eagerly and, for remote replicas, each worker flushes its pending remote replica updates before any m-boundary vertex relinquishes a fork to a vertex of another worker. Then, per Section 4.1, vertex-based distributed locking provides serializability for asynchronous computation models. As we mentioned in Section 4.1, this solution is incompatible with synchronous models (BSP and sync GAS) because these models do not allow local replicas to be updated eagerly. However, applying the theory developed in Section 3, Proposition 1 shows that a constrained vertex-based locking solution can provide serializability for systems with synchronous models. We omit the proof due to space constraints. It can be found in the longer version of this paper [21].

**PROPOSITION 1.** *Vertex-based distributed locking enforces conditions C1 and C2 for synchronous computation models when the following two properties hold: (i) all vertices act as philosophers and (ii) fork and token exchanges occur only during global barriers.*

## 5. PARTITION AWARE TECHNIQUES

In this section, we show how partition aware synchronization techniques can address severe limitations of existing techniques. We then present our partition-based solution to demonstrate its significant performance advantages.

### 5.1 Preliminaries

Existing graph processing systems provide parallelism at each worker machine by pairing computation threads with either graph partitions or individual vertices.

For example, both Giraph and Giraph async (Section 2.2) assign multiple graph partitions to each worker machine and pair threads, each roughly equivalent to a CPU core, with available partitions. This allows multiple partitions to execute in parallel, while vertices in each partition are executed sequentially. We call such systems *partition aware*.

In contrast, GraphLab async uses over-threading to pair lightweight threads (called fibers) with individual vertices. Thus, it has no notion of partitions. The large number of fibers provides a high degree of parallelism and ensures that CPU cores are kept busy even when some fibers are blocked on communication.

Systems like GraphLab async are well-suited for very fine-grained synchronization techniques such as vertex-based distributed locking (Section 4.3). Partition aware systems like Giraph async are able to support partition aware techniques that, as we will show, take advantage of partitions to significantly improve performance. Since GraphLab async is not partition aware, it is unable to support such techniques.

Lastly, as we will show in the following sections, it is important for synchronization techniques implemented in partition aware systems to distinguish between p-internal and p-boundary vertices, defined as follows.

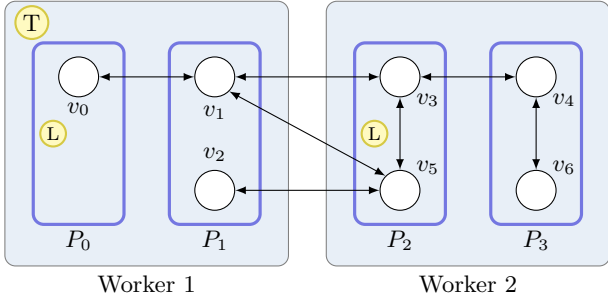
**DEFINITION 4.** *A vertex  $u$  is a partition boundary vertex, or p-boundary for short, if at least one of its neighbors  $v$  belongs to a different partition from  $u$ . Otherwise,  $u$  is a partition internal, or p-internal, vertex.*

### 5.2 Motivation

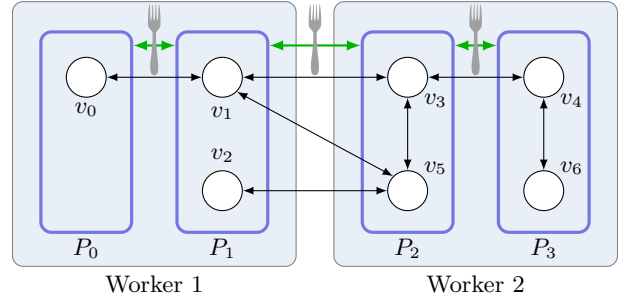
The two existing synchronization techniques described in Section 4 suffer from several major performance issues.

Token passing has minimal communication overheads but very limited parallelism (Figure 1): only one worker machine can execute its m-boundary vertices at any time. Having only one global token also results in poor scalability, because the size of the token ring increases with the number of workers, which leads to longer wait times. Moreover, the token ring is fixed: workers that are finished must still receive and pass along the token, which adds unnecessary overheads. This is especially evident in algorithms such as SSSP, where workers may dynamically halt or become active depending on the state of their constituent vertices. Thus, as we show in Section 7.3, single-layer token passing is too coarse-grained, which negatively impacts performance.

On the other hand, vertex-based distributed locking maximizes parallelism, by allowing all vertices to execute in parallel, but suffers significant communication overheads. Vertex-based locking requires, in the worst case,  $O(|\mathcal{E}|)$  forks, where  $|\mathcal{E}|$  is the number of edges in the graph ignoring directions (i.e., counting undirected edges once). This leads to significant communication overheads due to the forks and corresponding request tokens that must be sent between individual vertices. Furthermore, it is difficult to form large batches



**Figure 4: Dual-layer token passing, with the global token T at worker 1 and the local tokens L at partitions 0 and 2.**



**Figure 5: Partition-based distributed locking.**

of messages (remote replica updates) as messages must be flushed very frequently, whenever an m-boundary vertex releases its forks. Although systems such as GraphLab async can use fibers to try to mask communication latency with additional vertex computations, it does not fully mitigate the communication overheads, which results in poor performance and scalability as we demonstrate in Section 7.3.

A key deficiency of these techniques is that they are not partition aware: given a partition aware system, they are unable to exploit partitions to improve performance. For example, single-layer token passing would pass the global token between the partitions rather than workers, with p-boundary vertices requiring the token to execute (Definition 4). This increases the size of the token ring and does not solve the existing performance problems. Similarly, vertex-based distributed locking (for asynchronous models) would require only p-boundary vertices to act as philosophers, since p-internal vertices are executed sequentially. However, although this reduces the number of forks, the heavy-weight threads will block an entire CPU whenever a vertex blocks on communication. Consequently, it is unable to mask communication latency and performs worse than GraphLab async’s pairing of fibers with individual vertices (Section 5.1).

We address these performance deficiencies by considering *partition aware* synchronization techniques. Adding partition awareness enables us to devise either a more fine-grained token passing technique to increase parallelism, or a more coarse-grained distributed locking technique to reduce communication overheads. We present these approaches next.

### 5.3 Dual-Layer Token Passing

We propose dual-layer token passing, which, unlike single layer token passing, supports multithreading by being partition aware. This enables more vertices to execute in parallel while ensuring condition C2 is enforced.

Dual-layer token passing uses two layers of tokens and a more fine-grained categorization of vertices. Let  $u$  be a vertex of partition  $P_u$  of worker  $W_u$ . Then an m-internal vertex  $u$  is now either a p-internal vertex, if all its neighbors belong to  $P_u$ , or a *local boundary* vertex otherwise. An m-boundary vertex  $u$  is either *remote boundary*, if its neighbors are only on partitions of other workers, or *mixed boundary* otherwise (i.e., its neighbors belong to partitions of both  $W_u$  and other workers). For example, in Figure 4,  $v_6$  is a p-internal vertex,  $v_0$  and  $v_4$  are local boundary vertices,  $v_2$  is a remote boundary vertex, and  $v_1$ ,  $v_3$ , and  $v_5$  are mixed boundary vertices.

A global token is passed in a round-robin fashion between

the workers. Each worker also has its own local token passed between its partitions in a round-robin fashion (Figure 4). A p-internal vertex can execute without tokens, while a local boundary vertex requires its partition to hold the local token. A global boundary vertex requires its worker to hold the global token and a mixed boundary vertex requires both tokens to be held. To ensure that every mixed boundary vertex gets a chance to execute, each worker must hold the global token for a number of iterations equal to the number of partitions it owns. Like single-layer token passing, local replicas are updated eagerly while remote replicas are updated before a worker relinquishes the global token. Hence, dual-layer token passing enforces conditions C1 and C2 for asynchronous computation models. Then, by Theorem 1, it provides serializability for asynchronous models.

Although dual-layer token passing improves parallelism by adding support for multithreading, it still suffers from the same performance issues as single-layer token passing. It again uses only one global token, has a fixed token ring, and scales poorly when the number of workers and/or partitions are increased. Having only one local token per worker also means local boundary vertices cannot execute in parallel.

While these problems may be solved via more sophisticated schemes, such as using multiple global tokens for more parallelism or tracking additional state to support a dynamic ring, it becomes much harder to guarantee correctness (no deadlocks and no starvation) while also ensuring fairness. Thus, rather than make token passing even more complex and fine-grained, we propose an inherently partition-based, coarse-grained distributed locking approach next.

### 5.4 Partition-based Distributed Locking

We propose partition-based distributed locking by building on the Chandy-Misra algorithm and treating partitions as the philosophers. Two partitions share a fork if an edge connects their constituent vertices. For example, in Figure 5, partitions  $P_0$  and  $P_1$  share a fork due to the edge between their vertices  $v_0$  and  $v_1$ , respectively. Alternatively, forks are associated with the virtual partition edges (in green), created based on the edges between each partition’s vertices.

Condition C2 is enforced for p-boundary vertices because neighboring partitions never execute concurrently, while p-internal vertices do not need coordination as each partition is executed sequentially. As an optimization, we can avoid unnecessary fork acquisitions by skipping the partitions for which all vertices are halted and have no more messages. To enforce condition C1, local replicas are updated eagerly and, for remote replicas, each worker flushes its pending remote replica updates before any partition (with an m-boundary vertex) relinquishes a fork to a partition of another worker.

Since both conditions are enforced, Proposition 2 follows immediately.

**PROPOSITION 2.** *Partition-based distributed locking enforces conditions C1 and C2 for asynchronous computation models.*

Hence, by Theorem 1, partition-based distributed locking provides serializability for asynchronous computation models. Synchronous models are not supported as they cannot update local replicas eagerly (Section 4.1), which is required due to the sequential execution of p-internal vertices.

Partition-based distributed locking needs at most  $O(|P|^2)$  forks, where  $|P|$  is the total number of partitions. By controlling the number of partitions, we can control the granularity of parallelism. On one extreme,  $|P| = |V|$  can give vertex-based distributed locking (Section 4.3). On the other extreme, we can have exactly one partition per worker. This latter extreme still provides better parallelism than single-layer token passing because any pair of non-neighboring workers can execute in parallel, with a negligible increase in communication. In general,  $|P|$  is set such that each worker can use multithreading to execute multiple partitions in parallel.

Due to this flexibility, partition-based locking is both more general and more performant than vertex-based locking: any choice of  $|P| \ll |V|$  significantly reduces the number of forks and hence communication overheads. Moreover, partition-based locking enables messages (remote replica updates) of an entire partition of vertices to be batched, substantially reducing communication overheads. Compared to token passing, partition-based locking enables more parallelism: forks are required only between partitions that cannot execute in parallel, removing the need for a token ring, and halted partitions do not need their forks and will not perform unnecessary communication with their neighbors. These factors result in partition-based locking’s superior performance and scalability over both vertex-based distributed locking and token passing.

Hence, partition-based distributed locking leverages the best of both worlds: the increased parallelism of distributed locking and the minimal communication overheads of token passing. It scales better than vertex-based locking and token passing, due to its lower communication overheads and the absence of a token ring, and offers flexibility in the number of partitions to allow for a tunable trade-off between parallelism and communication overheads.

## 6. IMPLEMENTATION

We now describe our implementations for dual-layer token passing and partition-based distributed locking in Giraph, an open source graph processing system. Each technique is an option that can be enabled and paired with Giraph async to provide serializability. We show in Section 6.5 that providing serializability for AP does not impact usability. We do not consider the constrained vertex-based locking for BSP (Proposition 1) as it further exacerbates BSP’s already expensive communication and synchronization overheads [20].

We use Giraph because it is a popular and performant system used, for example, by Facebook [14]. It is partition aware and thus can support partition aware synchronization techniques. We do not implement token passing and partition-based distributed locking in GraphLab async because, as described in Section 5.1, GraphLab async is optimized for vertex-based distributed locking and is *not*

partition aware. Adding partitions would require substantial changes to the architecture, design, and functionality of GraphLab async, which is not the focus of this paper.

### 6.1 Giraph Background

As described in Section 5.1, Giraph assigns multiple graph partitions to each worker. During each superstep, each worker creates a pool of compute threads and pairs available threads with uncomputed partitions. Each worker maintains a message store to hold all incoming messages, while each compute thread uses a message buffer cache to batch outgoing messages to more efficiently utilize network resources. These buffer caches are automatically flushed when full but can also be flushed manually. In Giraph async, messages between vertices of the same worker skip this cache and go directly to the message store so that they are immediately available for their recipients to process.

Since Giraph is implemented in Java, it avoids garbage collection overheads (due to millions or billions of objects) by serializing vertex, edge, and message objects when not in use and deserializing them on demand. For each vertex  $u$ , Giraph stores only  $u$ ’s out-edges in  $u$ ’s vertex object. Thus, in-edges are not explicitly stored within Giraph.

### 6.2 Dual-Layer Token Passing

For dual-layer token passing, each worker uses three sets to track the vertex ids of local boundary, remote boundary, and mixed boundary vertices that it owns. p-internal vertices are determined by their absence from the three sets. We keep this type information separate from the vertex objects so that token passing is a modular option. Moreover, augmenting each vertex object with its type adds undesirable overheads since vertex objects must be serialized and deserialized many times throughout the computation. Having the type information in one place also allows us to update a vertex’s type without deserializing its object.

To populate the sets, we intercept vertices during input loading and scan the partition ids of its out-edge neighbors to determine its type. This is sufficient for undirected graphs but not for directed graphs: a vertex  $u$  has no information about its in-edge neighbors. Thus, we have each vertex  $v$  send a message to its out-edge neighbors  $u$  that belong to a different partition. Then  $u$  can correct its type based on messages received from its in-edge neighbors. This all occurs during input loading and thus does not impact computation time. We also batch all dependency messages to minimize network overheads and input loading times.

As per Section 5.3, the global and local tokens are passed in a round-robin fashion. Each local token is passed among its worker’s partitions at the end of each superstep. Since local messages (between vertices of the same worker) are not cached, local replicas are updated eagerly. For remote replicas, workers flush and await delivery confirmations for their remote messages before passing along the global token.

### 6.3 Partition-based Distributed Locking

For partition-based distributed locking, each worker tracks fork and token states for its partitions in a dual-layer hash map. For each pair of neighboring partitions  $P_i$  and  $P_j$ , we map  $P_i$ ’s partition id  $i$  to the id  $j$  to a byte whose bits indicate whether  $P_i$  has the fork, whether the fork is clean or dirty, and whether  $P_i$  holds the request token. Since partition ids are integers in Giraph, we use hash maps optimized



for integer keys to minimize memory footprint.

In the Chandy-Misra algorithm, forks and tokens must be placed such that the precedence graph, whose edge directions determine which philosopher has priority for each shared fork, is initially acyclic [13]. A simple way to ensure this is to assign each philosopher an id and, for each pair of neighbors, give the token to the philosopher with the smaller id and the dirty fork to the one with the larger id. This guarantees that philosophers with smaller ids initially have precedence over all neighbors with larger ids, because a philosopher must give up a dirty fork upon request (except while it is eating). Partition ids naturally serve as philosopher ids, allowing us to use this initialization strategy.

For directed graphs, two neighboring partitions may be connected by only a directed edge, due to their constituent vertices. Since partitions must be aware of both its in-edge and out-edge dependencies, workers exchange dependency information for their partitions during input loading. Like in token passing, dependency messages can be batched to ensure a minimal impact on input loading times.

Partitions acquire their forks synchronously by blocking until all forks arrive. This is because even if all forks are available, it takes time for them to arrive over the network, so immediately returning is wasteful and may prevent other partitions from executing (a partition cannot give up clean forks: it must first execute and dirty them). Finally, per Section 5.4, each worker flushes its remote messages before a partition sends a shared fork to another worker’s partition.

Using the insights from our implementation of partition-based distributed locking, we can also implement vertex-based distributed locking, which is the special case where  $|P| = |V|$  (Section 5.4). Each worker tracks fork and token states for its p-boundary vertices and uses vertex ids as the keys for its dual-layer hash map. Keeping this data in a central per-worker data structure, rather than at each vertex object, is even more important than in token passing: forks and tokens are constantly exchanged so their states must be readily available to modify. Storing this data at each vertex object would incur significant deserialization overheads. Fork and token access patterns are also fairly random, which would further incur an expensive traversal of a byte array to locate the desired vertex.

Like the partition-based approach, for directed graphs, each vertex  $v$  broadcasts to its out-edge neighbors  $u$  so that  $u$  can record the in-edge dependency into the per-worker hash map. This occurs during input loading and all messages are batched. Vertices acquire their forks synchronously and each worker flushes its remote messages before any m-boundary vertex forfeits a fork to a vertex of another worker. However, as we show in Section 7, these batches of remote messages are far too small to avoid significant communication overheads.

## 6.4 Fault Tolerance

For fault tolerance, we use the existing checkpointing mechanism of Giraph. In addition to the data that Giraph already writes to disk at each synchronous checkpoint, we change Giraph to also record the relevant data structures (hash sets or hash maps) that are used by the synchronization techniques. For dual-layer token passing, each worker also records whether they have the global token and the id of the partition holding the local token. Checkpoints occur after a global barrier and thus capture a consistent state: there are no vertices executing and no in-flight messages.

Thus, neither token passing’s global token nor distributed locking’s fork and request tokens are in transit.

## 6.5 Algorithmic Compatibility and Usability

A system can provide one computation model for algorithm developers to code with and use a different computation model to execute user algorithms. For example, Giraph async allows algorithm developers to code for the BSP model and transparently execute with an asynchronous computation model to maximize performance [20]. Thus, with respect to BSP, the more efficient AP model implemented by Giraph async does not negatively impact usability.

When we pair Giraph async with partition-based or vertex-based distributed locking, it remains backwards compatible with (i.e., can still execute) algorithms written for the BSP model. To take advantage of serializability, algorithm developers can now code for a serializable computation model. Specifically, this is the AP model with the additional guarantee that conditions C1 and C2 hold. For example, our graph coloring algorithm is written for this serializable AP model rather than for BSP (Section 7.2.1).

However, not all synchronization techniques provide this clean abstraction. Token passing fails in this regard because only a subset of vertices execute in each superstep. That is, token passing cannot provide the guarantee that all vertices will execute some code in superstep  $i$ , because only a subset of the vertices will execute at superstep  $i$ . The same issue arises for the constrained vertex-based distributed locking solution for BSP and sync GAS (Proposition 1), because it relies on global barriers for the exchange of forks and tokens. In contrast, our implementations of partition-based and vertex-based locking ensure that all vertices are executed exactly once in each superstep and thus provide superior compatibility and usability.

## 7. EXPERIMENTAL EVALUATION

We compare dual-layer token passing and partition-based distributed locking using Giraph async and vertex-based distributed locking using GraphLab async. We exclude Giraph async for vertex-based locking because it is much slower than GraphLab async, up to 44× slower on OR (Table 1). As discussed in Section 5.1, this is because GraphLab async is specifically tailored for the vertex-based technique whereas Giraph async is not. On the other hand, unlike Giraph async, GraphLab async is not partition aware and thus cannot support token passing or partition-based distributed locking. Hence, our evaluation focuses on the most performant combinations of systems and synchronization techniques.

### 7.1 Experimental Setup

To evaluate the different synchronization techniques, we use 16 and 32 EC2 r3.xlarge instances, each with four vCPUs and 30.5GB of memory. All machines run Ubuntu 12.04.1 with Linux kernel 3.2.0-70-virtual, Hadoop 1.0.4, and jdk1.7.0\_65. We implement our modifications in Giraph 1.1.0-RC0 and compare against GraphLab 2.2, which is the latest version that provides serializability.

We use large real-world datasets<sup>2,3</sup>[8, 7, 6], which are stored on HDFS as text files and loaded into each system using the default random hash partitioning. We use hash par-

<sup>2</sup><http://snap.stanford.edu/data/>

<sup>3</sup><http://law.di.unimi.it/datasets.php>

**Table 1: Directed datasets. Parentheses give values for the undirected versions used by graph coloring.**

Graph	$ V $	$ E $	Max Degree
com-Orkut ( <b>OR</b> )	3.0M	117M (234M)	33K (33K)
arabic-2005 ( <b>AR</b> )	22.7M	639M (1.11B)	575K (575K)
twitter-2010 ( <b>TW</b> )	41.6M	1.46B (2.40B)	2.9M (2.9M)
uk-2007-05 ( <b>UK</b> )	105M	3.73B (6.62B)	975K (975K)

tioning as it is the fastest method of partitioning datasets across workers and, importantly, does not favour any particular synchronization technique. Alternative partitioning algorithms such as METIS [24] are impractical as they can take several hours to partition large datasets [22, 30].

Table 1 lists the four graphs we use: **OR** and **TW** are social network graphs while **AR** and **UK** are web graphs.  $|V|$  and  $|E|$  of Table 1 denote the number of vertices and directed edges for each graph, while the maximum degree gives a sense of how skewed the graph’s degree distribution is. All graphs have large maximum degrees because they follow a power-law degree distribution.

For partition-based distributed locking, we use Giraph’s default setting of  $|W|$  partitions per worker, where  $|W|$  is the number of workers. Increasing the number of partitions beyond this does not improve performance: more edges become cut, which increases inter-partition dependencies and hence leads to more forks and tokens. Smaller partitions also mean smaller message batches and thus greater communication overheads. However, using too few partitions restricts parallelism for both compute threads and communication threads: the message store at each worker is indexed by separate hash maps for each partition, so more partitions enables more parallel modifications to the store while fewer partitions restricts parallelism and degrades performance.

## 7.2 Algorithms

We use graph coloring, PageRank, SSSP, and WCC as our algorithms. Our choice is driven by the requirements exhibited by graph processing algorithms that need serializability. As described in Section 1, many machine learning algorithms require serializability for correctness and convergence. SSSP is a key component in reinforcement learning while WCC is used in structured learning [29, 10]. Both algorithms are used with extensive parallelism, making convergence a crucial criterion that serializability can provide. Similarly, as established in Section 2, graph coloring falls into yet another class of algorithms where serializability ensures successful termination. Finally, PageRank is a good comparison algorithm for two reasons: first, existing systems that have considered serializability also implement PageRank [33, 18] and second, the simple computation and communication patterns of PageRank are identical to other more complex algorithms [4], which allows us to better understand the performance of the synchronization techniques without being hindered by algorithmic complexity.

### 7.2.1 Graph Coloring

We use a greedy graph coloring algorithm (Algorithm 1) that requires serializability and an undirected input graph. Each vertex  $u$  initializes its value/color as `NO_COLOR`. Then, based on messages received from its (in-edge) neighbors,  $u$  selects the smallest non-conflicting color as its new color and broadcasts it to its (out-edge) neighbors.

**Algorithm 1** Graph coloring pseudocode.

```

1 procedure COMPUTE(vertex, incoming messages)
2   if superstep == 0 then
3     vertex.setValue(NO_COLOR)
4     return
5   if vertex.getValue() == NO_COLOR then
6      $c_{min} \leftarrow$  smallest non-conflicting color
7     vertex.setValue( $c_{min}$ )
8     Send  $c_{min}$  to vertex’s out-edge neighbors
9   voteToHalt()

```

In theory, the algorithm requires only one iteration since serializability prevents conflicting colors. In practice, because Giraph async is push-based, it requires three iterations: initialization, color selection, and handling extraneous messages. The extraneous messages occur because vertices indiscriminately broadcast their current color, even to neighbors who are already complete. This wakes up vertices, leading to an additional iteration. GraphLab async, which is pull-based, has each vertex gather its neighbors’ colors rather than broadcast its own and thus completes in a single iteration.

### 7.2.2 PageRank

PageRank is an algorithm that ranks webpages based on the idea that more important pages receive more links from other pages. Each vertex  $u$  starts with a value of 1.0. At each superstep,  $u$  updates its value to  $pr(u) = 0.15 + 0.85x$ , where  $x$  is the sum of values received from  $u$ ’s in-edge neighbors, and sends  $pr(u)/\text{deg}^+(u)$  along its out-edges. The algorithm terminates after the PageRank value of every vertex  $u$  changes by less than a user-specific threshold between two consecutive execution of  $u$ . The output  $pr(u)$  gives the expectation value for a vertex  $u$ , which can be divided by the number of vertices to obtain the probability value.

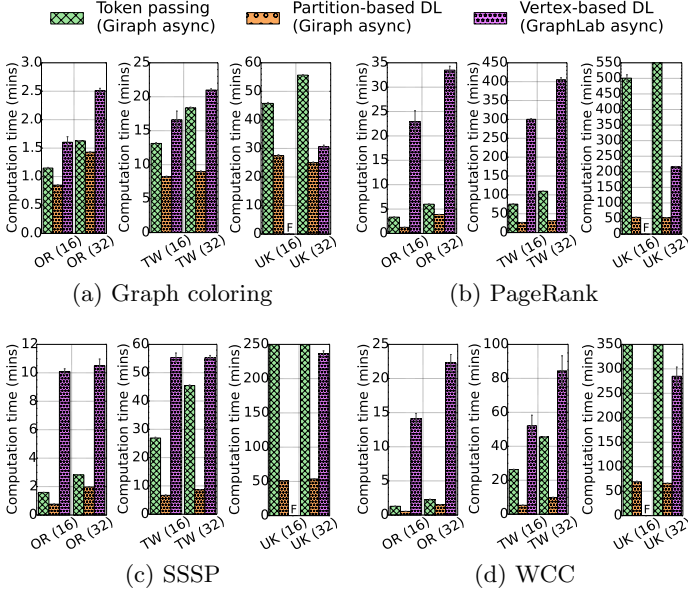
We use a threshold of 0.01 for **OR** and **AR** and 0.1 for **TW** and **UK** so that experiments complete in a reasonable amount of time. Using the same threshold ensures that all systems perform the same amount of work for each graph.

### 7.2.3 SSSP

Single-source shortest path (SSSP) finds the shortest path between a source vertex and all other vertices in its connected component. We use the parallel variant of the Bellman-Ford algorithm [15]. Each vertex initializes its distance (vertex value) to  $\infty$ , while the source vertex sets its distance to 0. Vertices update their distance using the minimum distance received from their neighbors and propagate any newly discovered minimum distance to all neighbors. We use unit edge weights and the same source vertex to ensure that all systems perform the same amount of work.

### 7.2.4 WCC

Weakly connected components (WCC) is an algorithm that finds the maximal weakly connected components of a graph. A component is weakly connected if all constituent vertices are mutually reachable when ignoring edge directions. We use the HCC algorithm [23], which starts with all vertices initially active. Each vertex initializes its component ID (vertex value) to its vertex ID. When a smaller component ID is received, the vertex updates its vertex value to that ID and propagates the ID to its neighbors.



**Figure 6: Computation times for graph coloring, PageRank, SSSP, and WCC. Missing bars are labelled with ‘F’ for unsuccessful runs.**

### 7.3 Results

For our results, we report computation time, which is the total time of running an algorithm minus the input loading and output writing times. This also captures any communication overheads that the synchronization techniques may have: poor use of network resources translates to longer computation times. For each experiment, we report the mean and 95% confidence intervals of five runs (three runs for experiments taking over 3 hours). Due to space constraints, we exclude the results for AR. They can be found in [21].

For graph coloring, partition-based locking is up to 2.3× faster than vertex-based locking for TW with 32 machines (Figure 6a). This is despite the fact that Giraph async performs an additional iteration compared to GraphLab async (Section 7.2.1). Similarly, partition-based locking is up to 2.2× faster than token passing for UK on 32 machines. Vertex-based locking fails for UK on 16 machines because GraphLab async runs out of memory.

As detailed in Section 5.4, these performance gains arise from significantly reducing the communication overheads, which is achieved by sharing fewer forks between larger partitions instead of millions or billions of forks between individual vertices. Moreover, unlike vertex-based locking, partition-based locking is able to support message batching, which further reduces communication overheads.

For PageRank, partition-based distributed locking again outperforms the other techniques: up to 18× faster than vertex-based locking on OR with 16 machines (Figure 6b). Vertex-based locking again fails for UK on 16 machines due to GraphLab async exhausting system memory. Token passing takes over 12 hours (720 mins) for UK on 32 machines due to its limited parallelism (Section 5.3), making partition-based locking over 14× faster than token passing.

For SSSP and WCC on UK, token passing takes over 7 hours (420 mins) for 16 machines and 9 hours (540 mins) for 32 machines, while GraphLab async fails on 16 machines due to running out of memory (Figures 6c and 6d). For SSSP,

partition-based locking is up to 13× faster than vertex-based locking for OR on 16 machines and over 10× faster than token passing for UK with 32 machines. For WCC, partition-based locking is up to 26× faster than vertex-based locking for OR on 16 machines and over 8× faster than token passing for UK with 32 machines. These performance gains are larger because these algorithms, like many machine learning algorithms, require multiple iterations to complete: the per-iteration performance gains, described earlier, are further multiplied by the number of iterations executed.

Partition-based distributed locking also scales better when going from 16 to 32 machines. For example, partition-based locking achieves a speedup with graph coloring on UK, whereas token passing suffers a slowdown (Figure 6a). In the cases where partition based-locking also experiences slowdown, which occurs because serializability trades off performance for stronger guarantees, its performance does not degrade as quickly as token passing and vertex-based locking and its computation time remains the shortest.

Lastly, Giraphx implements its synchronization techniques only for graph coloring, so we can compare against only this algorithm. As discussed previously, Giraphx implements its techniques as part of user algorithms rather than within the system, resulting in poor usability as they must be re-implemented in every user algorithm. A key advantage of our techniques is that, because they are implemented at the system level, serializability is automatically provided for all user algorithms. For graph coloring on OR with 16 machines, Giraphx with single-layer token passing is 30× and 41× slower than Giraph async with dual-layer token passing and partition-based distributed locking, respectively. With vertex-based locking, Giraphx is 55× slower than GraphLab async with vertex-based locking and 103× slower than Giraph async with partition-based locking. On TW and UK, Giraphx fails to run due to exhausting system memory. Giraphx’s poor performance is due to its less efficient techniques, the fact that it uses a much older and less performant version of Giraph and, unlike Giraph async, does not implement the more performant version of the AP model.

## 8. RELATED WORK

To the best of our knowledge, this paper is the first to formulate the important notion of serializability for graph processing systems and to incorporate it into a foundational framework that has been implemented in a real system to deliver an end-to-end solution. Only Giraphx [33] and GraphLab [27, 18] provide serializability but, as we showed in this paper, our techniques significantly outperform their designs. Moreover, neither of their proposals provide a formal framework for reasoning about serializability nor do they show correctness for their synchronization techniques. Giraphx considers single-layer token passing and vertex-based distributed locking but its implementations are a part of user algorithms rather than within the system: each technique must be re-implemented in every user algorithm, which negatively impacts performance and usability. GraphLab async uses vertex-based distributed locking and is tailored for this synchronization technique. However, it is not partition aware and thus cannot support the more efficient partition-based distributed locking technique.

We mention several other vertex-centric graph processing systems next, however, they neither consider nor provide serializability. Apache Hama [2] is a general BSP system that,

unlike Giraph, is not optimized for graph processing. GPS [31] and Mizan [25] are BSP systems that consider dynamic workload balancing, but not serializability, while GRACE [35] is a single-machine shared memory system that implements the AP model. GraphX [36] is a system built on the data parallel engine Spark [37], and considers graphs stored as tabular data and graph operations as distributed joins. GraphX’s primary goal is to provide efficient graph processing for end-to-end data analytic pipelines implemented in Spark. Pregelix [12] is a BSP graph processing system implemented in Hyracks [9], a shared-nothing dataflow engine. Pregelix stores graphs and messages as data tuples and uses joins to implement message passing. GraphChi [26] is a single-machine disk-based graph processing system for processing graphs that do not fit in memory.

## 9. CONCLUSION

We presented a formalization of serializability for graph processing systems and proved that two key conditions must hold to provide serializability. We then showed the need for partition aware synchronization techniques to provide serializability more efficiently. In particular, we introduced a novel partition-based distributed locking technique that, in addition to being correct, is more efficient than existing techniques. We implemented all techniques in Giraph to provide serializability as a configurable option that is completely transparent to algorithm developers. Our experimental evaluation demonstrated that our partition-based technique is up to 26× faster than existing techniques that are implemented by graph processing systems such as GraphLab.

## 10. REFERENCES

- [1] Apache Giraph. <http://giraph.apache.org>.
- [2] Apache Hama. <http://hama.apache.org>.
- [3] GraphLab: Distributed Graph-Parallel API. [http://docs.graphlab.org/classgraphlab\\_1\\_1\\_async\\_consistent\\_engine.html](http://docs.graphlab.org/classgraphlab_1_1_async_consistent_engine.html), 2014.
- [4] M. Balassi, R. Palovics, and A. A. Benczur. Distributed Frameworks for Alternating Least Squares. In *RecSys '14*.
- [5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.
- [6] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.
- [7] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW '11*, pages 587–596, 2011.
- [8] P. Boldi and S. Vigna. The Webgraph Framework I: Compression Techniques. In *WWW '04*, pages 595–602.
- [9] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *ICDE '11*, pages 1151–1162, 2011.
- [10] A. Boularias, O. Krömer, and J. Peters. Structured Apprenticeship Learning. In *ECML PKDD '12*, pages 227–242, 2012.
- [11] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel Coordinate Descent for L1-Regularized Loss Minimization. In *ICML*, 2011.
- [12] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) Graph Analytics on A Dataflow Engine. *PVLDB*, 8(2):161–172, 2015.
- [13] K. M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, 1984.
- [14] A. Ching. Scaling Apache Giraph to a trillion edges. <http://www.facebook.com/10151617006153920>, 2013.
- [15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition.
- [16] J. Edwards. ‘Facebook Inc.’ Actually Has 2.2 Billion Users Now. <http://www.businessinsider.com/facebook-inc-has-22-billion-users-2014-7>, 2014.
- [17] J. Gonzalez, Y. Low, A. Gretton, and C. Guestrin. Parallel Gibbs Sampling: From Colored Fields to Thin Junction Trees. In *AISTATS*, volume 15, pages 324–332, 2011.
- [18] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI '12*, pages 17–30.
- [19] Google. How search works. <http://www.google.com/insidesearch/howsearchworks/thestory/>, 2014.
- [20] M. Han and K. Daudjee. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *PVLDB*, 8(9):950–961, 2015.
- [21] M. Han and K. Daudjee. Providing Serializability for Pregel-like Graph Processing Systems. Technical Report CS-2016-01, University of Waterloo, 2016.
- [22] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An Experimental Comparison of Pregel-like Graph Processing Systems. *PVLDB*, 7(12):1047–1058, 2014.
- [23] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *ICDM '09*, pages 229–238, 2009.
- [24] Karypis Lab. METIS and ParMETIS. <http://glaros.dtc.umn.edu/gkhome/views/metis>.
- [25] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *EuroSys '13*, pages 169–182, 2013.
- [26] A. Kyrola, G. Blleloch, and C. Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In *OSDI '12*, pages 31–46, 2012.
- [27] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 5(8):716–727, 2012.
- [28] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD/PODS '10*, pages 135–146, 2010.
- [29] Mausam and A. Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.
- [30] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen. Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases. In *EDBT*, pages 25–36, 2015.
- [31] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM '13*, pages 22:1–22:12, 2013.
- [32] A. G. Siapas. *Criticality and Parallelism in Combinatorial Optimization*. PhD thesis, Massachusetts Institute of Technology, 1996.
- [33] S. Tasci and M. Demirbas. Giraphx: Parallel Yet Serializable Large-scale Graph Processing. In *Euro-Par '13*, pages 458–469, 2013.
- [34] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, 1990.
- [35] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR '13*, 2013.
- [36] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *GRADES '13*, pages 2:1–2:6, 2013.
- [37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud '10*, 2010.

# DBExplorer: Exploratory Search in Databases

Manish Singh  
Indian Institute of Technology  
Hyderabad, India  
msingh@iith.ac.in

Michael J. Cafarella  
University of Michigan  
Ann Arbor, USA  
michjc@umich.edu

H.V. Jagadish  
University of Michigan  
Ann Arbor, USA  
jag@umich.edu

## ABSTRACT

A traditional relational database can evaluate complex queries but requires users to precisely express their information need. But users often do not know what information is available in a database, and hence cannot correctly express their information need. Traditional databases do not provide convenient means for users to gain familiarity with the data.

In this paper, we study the problem of exploratory search, which a user may wish to perform to get an understanding of the data set. We note that users often have some decisions already made, so what they need is not an overall database summary, but rather a summary “in context” of the relevant portion of the database. Towards this end, we devise a novel data summarization technique called the Conditional Attribute Dependency (CAD) View, which shows the conditional dependencies between attribute values conditioned on applied selections. The CAD View can help users to gain familiarity with structured datasets in an attribute-wise manner.

To evaluate the CAD View, we perform a user study comprising three complex exploratory tasks on a real dataset. Our studies show that users are able to do all the tasks about 4-5 times faster and with better accuracy using the CAD View compared to the data summary shown in faceted navigation, which is currently the most popular search interface for e-commerce and has support for exploratory search.

## 1. INTRODUCTION

Users today have access to many large databases, yet find it difficult to access the records they want. In some cases, the challenge is to write correct SQL. But databases today often come with easy-to-use query interfaces. Users still find it difficult to specify the precise query conditions, due to limited familiarity with the data. Consider, for example, a user on a travel web site looking to book a hotel in a big city. If she knows her preferences for price, location, star rating, and other such relevant attributes, she can easily specify a query that will pull out a few good choices for

her to consider from among the hundreds of hotels in the city. But, if she is unfamiliar with the city, she may not understand what typical prices are in the city or how all the 5-star hotels are clustered in the financial district or how there is a tradeoff between location and price. Without this knowledge of the data in the database, she is forced to depend on other data sources, such as advice from friends and relatives, social media, web documents, etc., to gain data familiarity and pose the right queries. In consequence, even after hours of effort she may be left with various doubts: “Did I make a good choice?” “Did I explore all my options?” “Did I spend more than I needed to?”

Our goal in this paper is to develop database facilities to support exploratory search. There are two types of search: *lookup* and *exploratory* [26, 24, 19]. In lookup search users have a specific well-defined search goal. In contrast, in exploratory search, the users’ goal is to gain a comprehensive understanding of data that will enable them to pose more informed lookup queries.

Supporting data exploration is difficult because: (a) Datasets are complex and heterogeneous, and (b) Users have diverse needs. It is easy, for example, to provide the user with some simple summary statistics, such as average price for a hotel room. However, this number is of only limited value to the user, perhaps because there is huge variance between different parts of the city or perhaps because the user is a backpacker looking for youth hostels whose price is poorly correlated with those at fancy hotels. What the user needs is a characterization of a portion of the data (which she has identified to the system) along dimensions that are of interest to her.

Let us consider an example task to better appreciate our problem. For variety, we describe a car purchase task rather than a hotel room booking task. For specificity, we write all queries in SQL, even though we expect any real implementation to have a user-friendly interface layer on top the query language.

**EXAMPLE 1.** Consider a used car database, which contains a single table  $D$  with  $n$  attributes where each tuple represents a car for sale. The table has numerous attributes that describe details of the car, such as Price, Make, Model, BodyType, Drivetrain, Mileage, EngineSize, NumCylinders, Color, FuelEconomy, Power, Year, etc.

Consider a user Mary who is unfamiliar with cars and wants to buy a relatively new SUV car. She has five initial Make preferences (Ford, Chevrolet, Toyota, Honda and Jeep), because she has friends who drive these Makes, but she is open to explore any similar option. She starts her exploration

with an initial query:  $\mathcal{R} = \text{SELECT } * \text{ FROM } \mathcal{D} \text{ WHERE Mileage BETWEEN } 10K \text{ AND } 30K \text{ AND Transmission = Automatic AND BodyType = SUV}$ . This query leads to a large result set with thousands of tuples. Mary has to specify more constraints to get to a smaller result set that she can explore in depth. She thinks a good place to start may be to reduce the number of **Makes** she considers. Let's look at her difficulties in choosing between **Makes**.

**Limitation 1. Understanding Attribute Values** —

The attribute **Make** has more than 50 values. Even to choose among the 5 **Makes** she has initially chosen, she needs to understand what is the main difference between SUVs from any pair of manufacturers, such as **Jeep** and **Chevrolet**. Furthermore, Mary knows that her initial set of 5 **Makes** is just a rough starting point, so before she narrows it down further she may also want to understand what other **Makes** are similar, and therefore belong in her consideration set. For example, she may want to know who else makes SUVs very similar to those made by **Chevrolet**.

Comparison can be of two types: *independent* and *conditional*. An independent comparison would be comparing the general characteristics of **Chevrolet** vs. **Jeep** cars. A conditional comparison would be based on the user's already made selections. For example, Mary might want to compare the five **Makes**: **Ford**, **Chevrolet**, **Toyota**, **Honda** and **Jeep**, given the following choices:  $\text{BodyType} = \text{SUV}, 10K \leq \text{Mileage} \leq 30K, \text{Transmission} = \text{Automatic}$ . Conditional comparisons are difficult even for users who are quite familiar with the domain. For example, Mary might know what **Make** she would prefer if there are no constraints on other attributes. However, if there are constraints in other attributes, such as **Price**, **Year**, **Color**, etc., it is hard to find **Makes** that will lead to cars that maximally satisfy her preferences across all the attributes.

With traditional relational database, result sets are presented as sets of tuples. To compare **Chevrolet** SUVs with **Jeep** SUVs, Mary has to look at hundreds of instances in each set. This is very difficult to do. Perhaps these tuples could be sorted on some important attributes, such as **Price**, so that corresponding tuples can be compared. But this requires knowing enough of the database to choose important attributes, and even so provides only limited assistance in understanding.

Finding similarities and differences are two complementary aspects of comparison. We looked at the difference case in the preceding paragraph. To find additional similar **Makes** is even more difficult, because now we need to check the hundreds of **Chevrolet** instances with the thousands of instances from dozens of other manufacturers. We note further Mary has a desired mileage range initially specified. As she explores the data set, she may decide to change this. If she is comparing **Makes** conditioned on her mileage selection, then she has a whole new comparison. The conditional comparisons change with every change in the given query condition.

**Limitation 2. Querying Hidden Attributes** —

Often, there are characteristics of the data item that are important to the user but not explicitly recorded as an attribute in the database. For example, Mary wants to choose a certain car body look, but this field is not encoded anywhere in the database. There may be a way to express her preference as a selection on available attributes (perhaps as a combination

of low height, large wheel diameter and four doors). But Mary does not know how to express her desired look in terms of these other attributes.

Even worse, many database interfaces, for the sake of simplicity, may limit the number of queryable attributes. The number of cylinders in the engine may be an attribute recorded in the database, but it is not available to Mary through her forms-based interface for querying the database. It is possible that queryable attributes, such as fuel efficiency, can be used as surrogates to express her preference for a 4 cylinder engine. However, such cross-attribute relationships are completely opaque to Mary, and she is unable to substitute the surrogate for her desired attribute.

In exploring a database, users have two problems: (a) Choosing attributes that will enable them to efficiently and precisely reach to their desired result set, and (b) Choosing attribute values for each chosen attribute. These choices are challenging because there is complex dependency between attribute values within and across attributes, and users also have an unspecified, complicated preference function that spans across multiple attributes. Moreover, their preference function changes on seeing the comparison between available choices.

In short, while exploring a data set, users often make choices in sequence (with some backtracking where needed). They need help to understand the fragment of the database that is currently selected, and they would like to see this fragment of the data set characterized in terms of the choices (of attributes and attribute values) that the user is contemplating next. The alternation of browsing and querying in user interaction with data has been well-documented, where the purpose of browsing is mostly exploration. When data sets become large, unfortunately, browsing is no longer effective because of the very large number of tuples to be viewed. Therefore, this understanding of selected database fragment is best provided as a context-sensitive summary that supports the user's exploration need.

In this paper, we present a novel data summarization technique called the *Conditional Attribute Dependency* (CAD) View, which allows users to systematically explore the conditional dependencies between attribute values both within and across attributes. It thereby lifts the two limitations described in the motivating example above. Our proposed CAD View can be integrated with any structured data presentation system.

Our key contributions are as follows:

- We identify two limitations that users face in exploring databases due to limited data familiarity (Section 1).
- We propose a query model and a data summarization technique called the CAD View that can help users gain familiarity with structured datasets (Section 2).
- We present the algorithms and techniques necessary to create and present the relevant CAD Views (Sections 3 and 4).
- We integrate CAD View with faceted navigation to make the exploratory search process user-friendly. Moreover, this also leads to a novel search interface that can support both exploratory and lookup search. (Section 5).
- We evaluate the CAD View on real data with a detailed user study. We find that users, on average, can perform

Make	Compare Attrs.	IUnit 1	IUnit 2	IUnit 3
Chevrolet	Model Engine Price Drivetrain Year	[Traverse LT] [Equinox LT] [V6] [25K-30K] [20K-25K] [AWD] [2011-2012]	[Suburban 1500 LT, Tahoe LT] [V8] [35K-40K] [40K-45K] [4WD] [2WD] [2011-2012]	[Captiva LS, Equinox LT] [V4] [15K-20K, 20K-25K] [2WD] [2011-2012]
Ford	Model Engine Price Drivetrain Year	[Escape XLT] [Escape Ltd.] [V6, V4] [20K-25K, 15K-20K] [2WD][4WD] [2011-2012] [2010-2011]	[Explorer XLT] [Explorer Ltd.] [V6] [V8] [30K-35K] [25K-30K] [4WD] [2WD] [2011-2012]	[Edge Ltd.] [Edge SEL] [V6] [25K-30K] [AWD, 2WD] [2011-2012] [2010-2011]
Honda	...	...	...	...
Toyota	...	...	...	...
Jeep	Model Engine Price Drivetrain Year	[Wrangler Unlimited] [V6] [V8] [25K-30K] [30K-35K] [4WD] [2011-2012] [2010-2011]	[Compass Sport, Patriot Sport] [V4] [15K-20K] [4WD] [2WD] [2011-2012]	[Liberty Sport] [V6] [15K-20K] [4WD] [2WD] [2011-2012] [2010-2011]

**Table 1:** This table shows a sample Conditional Attribute Dependency (CAD) View for comparing five different car manufacturers. The first column Make is the Pivot Attribute. The second column Compare Attributes shows the top-5 attributes that are most informative for comparing the five Makes. The last three columns shows the top-3 IUnits for each Make. The user has selected BodyType = SUV,  $10K \leq \text{Mileage} \leq 30K$ , Transmission = Automatic. Each IUnit is a cluster label that summarizes a group of similar SUVs.

tasks that require data understanding with 4-5 times greater efficiency and accuracy using our CAD View as compared to faceted interface (Section 6).

Finally, we conclude with Section 8 after a discussion of related work in Section 7.

## 2. SOLUTION ARCHITECTURE

In this section we describe our solution to the exploratory search problem in complex databases — the *Conditional Attribute Dependency* (CAD) View. We also identify algorithmic challenges that must be solved for the CAD View to fulfill its goals.

### 2.1 The CAD View

The CAD View is a novel data summarization technique that shows the conditional relationship between values in a given attribute with values in other attributes. It is best introduced by example. A more formal treatment follows in the next subsection.

#### 2.1.1 Overview

Table 1 shows a sample CAD View, obtained from a real dataset, for the example query discussed in Section 1. Mary’s goal was to explore automatic transmission SUV cars that have Mileage between 10K-30K from five different Makes. The CAD View has several important components:

- 1. The Pivot Attribute** organizes the information that is shown in the CAD View. A user explicitly chooses one of the attributes as *Pivot Attribute*  $f_p$  and requests the system to create a CAD View that facilitates comparison among attribute values selected from the Pivot Attribute by showing their relationship with values across other attributes. In Table 1, Mary has chosen Make as the Pivot Attribute.

- 2. Compare Attributes** are data attributes that interact with the Pivot Attribute in “interesting” ways. All the

values in the Pivot Attribute are compared using the same set of Compare Attributes. These attributes can be automatically determined based on the result set and the Pivot Attribute or explicitly provided by the user. For example, one can use correlation to quantify interesting interaction. In Table 1, the system has given five Compare Attributes: Model, Engine, Price, Drivetrain, and Year.

- 3. An IUnit** (Interaction Unit) is an “interesting” group of values for the Compare Attributes. In Table 1, each IUnit is described using the five Compare Attributes mentioned above. Each IUnit is chosen to be relevant to a Pivot Attribute value: Chevrolet, Ford, Honda, etc. The top-left IUnit in this table (containing Traverse LT and Equinox LT) identifies a set of mid-sized Chevrolet SUVs: they share an engine size and a drivetrain, and have similar prices. One can think of an IUnit as a cluster of database values with two special differences: it is a cluster on a partition of the database determined by each Pivot Attribute value, and the cluster is labeled using the chosen Compare Attribute labels and Compare Attribute values.

**The Overall CAD View** is a tabular combination of the above three components. It displays one row for each value of the user-selected Pivot Attribute. In the second column the system shows an ordered list of Compare Attributes, one for each row of the table. The rest of the table shows each row’s top IUnits, sorted left-to-right in descending order of relevance to the row’s Pivot Attribute value. If an IUnit cluster can be represented equally well by multiple values in a single Compare Attribute, then an IUnit will show multiple attribute values in square brackets (e.g. Traverse LT and Equinox LT).

Note that there are competing ways to rank IUnits from left-to-right within each row. They can be ranked left to right in order of their salience for the row’s Pivot Attribute value. Or we could try to ensure that all of the IUnits in a single column can be compared across all Pivot Attribute

values so that, *e.g.*, the IUnit 1 for Chevrolet is similar to IUnit 1 for Ford (and thereby addressing Limitation 1).

However, not all Pivot Attribute values may share comparable IUnits, forcing our system into an impossible tradeoff between IUnit quality and columnar IUnit “comparability.” Thus, we chose to rank IUnits strictly by their relevance to the row’s Pivot Attribute value. We use other means to satisfy the comparability goal as described below in Section 2.1.3.

### 2.1.2 Query Model

We use the following extension of SQL to express an exploratory search query:

```
CREATE CADVIEW cadview_name AS
SET pivot = pivot_attr
SELECT attr1, attr2,...,attrN
FROM table1, table2...
[WHERE Clause]
[LIMIT COLUMNS M] [IUNITS K]
[ORDER BY attr_name, attr_name ASC|DESC]
```

In the above expression, the list of attributes shown in the SELECT clause are the attributes that the user has explicitly selected as Compare Attributes. The LIMIT COLUMNS clause is used to limit the number of Compare Attributes. The CAD View will have total of M columns as Compare Attributes, in which N are explicitly provided by the user and the remaining (M-N) are automatically selected based on the query result and the Pivot Attribute. The number of IUnits per row K is determined using the keyword IUNITS. The ORDER BY keyword can be used to sort the IUnits by one or more columns.

```
CREATE CADVIEW CompareMakes AS
SET pivot = Make
SELECT Price
FROM UsedCars
WHERE Mileage BETWEEN 10K AND 30K AND
Transmission = Automatic AND BodyType = SUV AND
(Make = Jeep OR Make = Toyota OR Make = Honda OR
Make = Ford OR Make = Chevrolet)
LIMIT COLUMNS 5 IUNITS 3
```

For example, Mary’s query can be expressed as above. The Price attribute has been explicitly selected as a Compare Attribute, and the remaining four attributes (Model, Engine, Drivetrain and Year) are automatically determined.

### 2.1.3 Finding Similar Information

If there are  $\mathcal{V}$  values in the Pivot Attribute and the user has requested  $k$  IUnits per attribute value, then the CAD View will have  $k|\mathcal{V}|$  IUnits. As discussed in Section 1, one of the primary goal of exploratory search is comparison, which includes finding similarities and differences. To facilitate comparison, we support the following two search operations within the CAD View: (i) Finding similar top ranked IUnits, and (ii) Finding similar attribute values within the Pivot Attribute.

For example, if a user likes a particular IUnit from one of the selected Pivot Attribute values (*e.g.*, Chevrolet), then the user may want to efficiently locate similar top-ranked IUnits that belong to other Pivot Attribute values. Similarly, if the user likes multiple IUnits of a particular Pivot Attribute value, then the user might be interested to find out other Pivot Attribute values that have similar IUnits.

Let’s say Mary likes IUnit 3 of Chevrolet. She can create a new CAD View where all the IUnits that are similar to this IUnit gets highlighted by using the following query:

```
HIGHLIGHT SIMILAR IUNITS
IN CompareMakes
WHERE SIMILARITY(Chevrolet, 3) > 3.5
```

In the similarity function the user gives the Pivot Attribute value and the IUnit ID. The above query will highlight all the IUnits (*e.g.*, IUnit 1 of Ford, IUnit 2 of Jeep) in the CAD View CompareMakes with similarity score greater than 3.5. As discussed later, for five Compare Attributes the max similarity score can be 5.0.

Similarly, to find Makes that are similar to Chevrolet, one can reorder the rows of the CAD View such that the Pivot Attribute values are ordered in terms of decreasing similarity with respect to Chevrolet. The similarity between two Pivot Attribute values can be measured by measuring the similarity between their IUnits. This query can be expressed as follows:

```
REORDER ROWS
IN CompareMakes
ORDER BY SIMILARITY(Chevrolet) DESC
```

### 2.1.4 Design Goals

We can now examine the extent to which the CAD View addresses the limitations described in the motivating example above:

**Limitation 1. Understanding Attribute Values** — With the traditional tuplewise presentation of result set, it was difficult for Mary to find the Makes that are similar to Chevrolet, or see the difference between Chevrolet and Jeep. However, using the CAD View it is easy to see that IUnits of Chevrolet and Ford are quite similar, and thus one can infer that both Chevrolet and Ford offer SUVs at roughly similar capacities and price points. One can also see that SUVs from Chevrolet and Jeep are quite different, and they primarily differ in Price and Drivetrain. Moreover, the CAD View can show conditional comparisons. Since Mary had selected Mileage between 10K and 30K, the CAD View shows her comparison between SUVs in Year range 2011-2012.

**Limitation 2. Querying Hidden Attributes** — Also recall that Mary was unable to choose cars with V4 engines, because the interface did not expose Engine type as an option in the query panel even though the information was contained in the database (*i.e.*, Engine was a non-queriable attribute). Moreover, she was not familiar enough with the database to indirectly find V4 engines by selecting values in the *queriable* attributes. In contrast, the CAD View identifies V4 engines as a characteristic of specific IUnits for each body style. Mary can select the desired tuples using the corresponding queriable attributes.

## 2.2 Problem Definition

### 2.2.1 Assumptions

The CAD View is a tabular structure whose size must be small enough for the summary information to be absorbed effectively by the user. For example, the width must be small enough not to require horizontal scrolling when displayed on the user’s screen. We reflect this constraint on the width by limiting the number of IUnits we can show for each attribute



value. Let this number be  $k$ . We assume that  $k$  is given to us, either by the user explicitly, or through the system gaining knowledge of the user’s set up.

The length of the table must also be constrained for the same reason. There are two variables that control table length. The first is the number of distinct values for the pivot attribute. By default, we will show all of them. If the user is focused on only specific values, these can be listed explicitly in the CAD View specification. Mary has chosen 5 specific Makes in the example above. The second variable affecting table length is the number of Compare Attributes in each row. We assume that this number  $c$  is given to us, just as  $k$  is. Furthermore, if the user is interested in specific attributes, she can insist that these be included in the Compare Attributes that the system selects.

For categorical attributes or attributes with small discrete numerical domain, the attribute values are directly obtained from the domain. Where the number of values is very large, such as for most numerical domains, ranges of values are binned together to create a small number of discrete attribute values. Such attribute value cardinality reduction is necessary for effective summarization. However, this cardinality does not itself play a role in the CAD View generation algorithm. Therefore, we mention it here as a pre-processing step, but do not go into details of exactly how this binning is done. We suggest following the well-developed techniques in histogram construction[17] for this purpose.

In this section we describe the problems that needs to be solved to create the CAD View. Our goals are (i) to populate this structure effectively, making the most of limited screen real-estate available, and (ii) to arrange and present the information populated in this structure to maximize its value to the user.

For the first goal, we have to find the best (i.e., most informative) Compare Attributes, the best IUnit clusters, and (for each IUnit) the best value labels to describe the IUnit’s data.

The CAD View structure already lays out IUnits in rows, one per attribute value for the Pivot Attribute. For the second goal, the system must further decide how to order IUnits within each row, how to indicate similarity between IUnits in different rows, and how to indicate similarities and differences between rows as a whole.

### 2.2.2 Creating the CAD View

The CAD View is created for a given result data set  $\mathcal{R}$  and a Pivot Attribute. Populating the CAD View entails one main task: obtaining the  $k$  IUnits of interest for each value of the pivot attribute. This task can be written formally as:

**Problem 1 (Generate IUnits):** *Given a result set  $\mathcal{R}$ , a Pivot Attribute  $f_p$ , a set of attribute values  $\mathcal{V}$  selected from  $f_p$ , and a threshold value  $k$ , find for each attribute value  $v \in \mathcal{V}$  a list of  $k$  IUnits  $S^v$ , where  $S^v = \{s_1^v, s_2^v, \dots, s_k^v\}$  and  $s_j^v$  is the  $j^{\text{th}}$  IUnit for attribute value  $v$ .*

The first task could be accomplished as finding  $k$  clusters with our favorite clustering algorithm. However, we observe that our goal is to explain the main structure of this fragment of the data set to the user. Therefore, there are two important ways in which we deviate from the basic problem statement above. The first is that we restrict the clustering to be on the basis of only the attributes selected as *Compare Attributes*. These are the attributes that will be displayed in

the CAD View. In other words, these are the attributes that will be used to label each cluster (IUnit). Therefore, it is the values of these attributes that we wish to have clustered together in each IUnit rather than some other attributes not shown to the user. The second point we note is that we are under no obligation to cover all points in the data set with the clusters produced. We do not want outliers to distort the clusters. To this end, we choose to solve the clustering problem with a larger number  $l$ , and then choose the top- $k$  IUnits from among these  $l$  clusters.  $l$  can be chosen by iterating through all plausible  $l$  values and evaluating the quality of the resulting CAD View for each. Or it could be obtained as a system tuning parameter, such as  $l = 1.5k$ .

We can then restate the CAD View generation problem as the following sequence of sub-problems:

**Problem 1.1 (Compare Attributes):** *Given a result set  $\mathcal{R}$ , a Pivot Attribute  $f_p$ , and set of attribute values  $\mathcal{V}$  selected in  $f_p$ , find a subset of Compare Attributes  $\mathcal{I}$  s.t.  $\mathcal{I}$  generate the most contrast among values in  $\mathcal{V}$ .*

Choosing Compare Attributes is a feature selection problem [12, 22] with a specialized way of evaluating the quality of a feature: good features (that is, Compare Attributes) yield sharply contrasting IUnits across the different Pivot Attribute values. One can discriminate among Compare Attributes as follows: Given a multi-class problem, a feature X is preferred to another feature Y if X induces a greater contrast between the multi-class conditional probabilities than Y. X and Y are indistinguishable if they induce the same amount of contrast.

**Problem 1.2 (Generate Candidate IUnits):** *Given a result set  $\mathcal{R}$ , a Pivot Attribute  $f_p$ , a set of attribute values  $\mathcal{V}$  selected from  $f_p$ , a set of Compare Attributes  $\mathcal{I}$ , and a threshold value  $l$ , find for each attribute value  $v \in \mathcal{V}$  a list of  $l$  candidate IUnits  $S^v$ , where  $S^v = \{s_1^v, s_2^v, \dots, s_l^v\}$  and  $s_j^v$  is the  $j^{\text{th}}$  candidate IUnit for attribute value  $v$ .*

Problem 1.2 is now stated as a clustering problem, with each resulting cluster being a candidate IUnit. We finally need to choose  $k$  IUnits from among these  $l$  candidates.

**Problem 2 (Top-k IUnits):** *Given a list of IUnits  $S^v$  for attribute value  $v$ , and a preference function  $P$ , find the top- $k$  IUnits  $T^v$  in  $S^v$  according to preference  $P$ , where  $T^v = \{t_1^v, t_2^v, \dots, t_k^v\}$  and  $T^v \subseteq S^v$ .*

The IUnits could be ranked based on a function that is rooted in the clustering algorithm; for example, we could prefer “tight” clusters by ranking them in ascending order of minimum pairwise similarity. However, we can pursue some application-specific goals by ranking IUnit clusters in a manner that is distinct from the IUnit creation mechanism. For example, our car navigation interface might, by default, rank clusters in ascending order of cluster price. In contrast, the fleet manager for a taxi company might have a different preference function that ranks IUnits in descending order of car mileage. Therefore, we have defined this ranking in terms of a specific preference function. If no function is specified by the user, we can use a simple system default, such as cluster size.

### 2.2.3 Finding Similar Information

The two search operations within the CAD View can be stated as following two problems:

**Problem 3 (Similar IUnits):** *Given two attribute values*

$x$  and  $y$  from the Pivot Attribute, and an IUnit  $t_i^x$  from  $T^x$ , find all IUnits  $t_j^y$  s.t.  $t_j^y \in T^y$  and  $\text{sim}(t_i^x, t_j^y) \geq \tau$ .

We can use any similarity function for this purpose, and any user or system specified threshold  $\tau$ . We describe the specifics of the similarity function in Section 4.1.

**Problem 4 (Similar Attribute Value):** *Given two attribute values  $x$  and  $y$  and their top- $k$  list of IUnits  $T^x$  and  $T^y$ , find the similarity between  $x$  and  $y$  by measuring the similarity of their top- $k$  IUnits.*

If a user shows preference for a particular attribute value, it implies that the user has liked most of the top- $k$  IUnits that has been shown for that attribute value. The user would be interested to see other attribute values that have similar IUnits both in terms of content and rank. We describe the specifics in Section 4.2.

### 3. CREATING THE CAD VIEW

In this section, we describe how we create and sort IUnits (problems 1 and 2 above) for the CAD View.

#### 3.1 Generating Candidate IUnits

Generating uniformly labeled IUnits consists of two steps: finding good Compare Attributes  $\mathcal{I}$  that can show contrast in Pivot Attribute values  $\mathcal{V}$ ; and then generating  $l$  IUnits for each value  $v \in \mathcal{V}$ .

##### 3.1.1 Finding Compare Attributes

The problem of finding Compare Attributes is similar to feature selection in a multi-class classification problem. To provide efficient user interaction and understanding, we use a feature selection algorithm that is computationally efficient and returns all the relevant features.

To determine the number of Compare Attributes we consider two factors: the available screen space and the relevance score of each informative facet. The user’s available screen space determines the maximum number  $c$  of Compare Attributes that can be shown for any Pivot Attribute. However, all Pivot Attribute may not have  $c$  informative facets that have relevance greater than a required minimum threshold relevance score. A relevant Compare Attribute always provides additional useful information. However, if a Compare Attribute is not informative about the Pivot Attribute, including it will lower the quality of generated IUnits and waste valuable screen space.

We use the ChiSquare feature selection algorithm [23]. ChiSquare evaluates the worth of an attribute by computing the value of the chi-squared statistic with respect to the class. For ChiSquare test one can determine the threshold relevance using p-values, such as significance level equal to 0.01, 0.05, or 0.10. Even with this simple technique, ranking Compare Attributes in order of decreasing relevance yields a few interesting observations that a typical user might not know. For example, it might seem that `Mileage` should be the best Compare Attribute when distinguishing among different `Year` values: older cars will naturally accrue more miles. However, it turns out that `Model` is better, as specific car models (`Suburban 1500 LT`, not simply `Suburban`) are released frequently, and a specific model is prominent in the database for only a short period of time.

##### 3.1.2 Finding Important Attribute Interactions

To create IUnits for a Pivot Attribute value  $v \in \mathcal{V}$ , we take all tuples from the result set  $\mathcal{R}$  that contain the given value  $v$ , and allocate those tuples to  $l$  clusters. We derive an IUnit from each of these  $l$  clusters. We cluster the tuples using only the above-chosen Compare Attributes.

Since the CAD View is a user-facing application, we want to create it within interactive time limits, well under 1 sec. There are various factors that can slow down a clustering algorithm: (i) clustering a dataset with large numbers of tuples or dimensions, (ii) trying to infer the ideal number of clusters using the clustering algorithm, and (iii) clustering with large numbers of cluster centers. Since there are standard existing techniques to address each of these factors, we defer their discussion to experimental evaluation (Section 6.3).

The quality of IUnits depends on the quality of the clustering algorithm. Since both efficiency and quality are major concerns of our system, we use standard k-means algorithm. Our main contribution in the clustering step is the dynamic variation of system parameters to achieve real-time performance, as discussed later in Section 6.3.

Our key contribution in creating the IUnits is the post-clustering step of cluster labeling, which is often ignored in clustering research. Although clustering is a very nice data-categorization technique, it is very hard for most users to understand the large amount of information that is contained in each cluster, or be able to compare multiple clusters.

There are existing systems to visually explore clusters of structured data [5, 21, 29]. Some of these systems are not easy to explore when the data is high-dimensional or categorical. For normal end-users, the commonly used cluster labeling technique is to show the centroid of each cluster, which is useful when all clusters are spherical. For complex shaped clusters, it is considered more informative to show multiple tuples that can show the whole cluster boundary [5]. It is very hard to understand a high-dimensional cluster by seeing just one centroid or some boundary points. When a user sees a high-dimensional representative tuple, it is not easy to infer the dimensions that are most significant. We need to label the clusters in such a way that we can convey large amount of information in a summarized manner and also emphasize the important information.

The way we label the clusters has many benefits. We label all IUnits uniformly and use ranking at all levels. We rank the Compare Attributes to highlight the attributes that are most significant. Similarly, in each IUnit we rank the Compare Attribute values and show only the most important representatives. Instead of showing few representative tuples from each cluster, we try to summarize statistical distribution of each Compare Attribute. To label both categorical and numerical attributes in uniform manner, we discretize the numerical attributes. We rank attribute values based on frequency count and then group multiple values if they have similar frequency count. We use two thresholds — max display count and statistical difference between frequency counts — to determine the representative Compare Attribute values for each cluster.

#### 3.2 Top- $k$ IUnits

Without an explicit user preference function, we choose a preference function that depends on the size of the IUnit’s underlying cluster, as well as overall result diversity. IUnits that represent large clusters are desirable because they

summarize attribute interactions for larger number of tuples. Moreover, they may give more reliable insight than smaller outlier-prone clusters. However, when we select the top- $k$  IUnits based purely on the cluster size, many are quite similar and appear redundant to the user.

Thus we use the generic top- $k$  algorithm proposed in [25] to compute diversified top- $k$  IUnits. It requires the following measures: preference score of each IUnit  $s_i^x$ , denoted as  $\text{score}(s_i^x)$ ; similarity between two IUnits  $s_i^x$  and  $s_j^y$ , denoted as  $\text{sim}(s_i^x, s_j^y)$ ; and a user defined threshold similarity value  $\tau$ . Two IUnits  $s_i^x$  and  $s_j^y$  are considered similar, denoted as  $s_i^x \approx s_j^y$ , if  $\text{sim}(s_i^x, s_j^y) \geq \tau$ .

**Diversified Top- $k$  IUnits:** Given a list of IUnits  $S^v = \{s_1^v, s_2^v, \dots\}$  for a attribute value  $v$ , and an integer  $k$ , the diversified top- $k$  IUnits for  $v$ , denoted as  $T^v = \{t_1^v, t_2^v, \dots\}$ , is the list of IUnits that satisfy the following conditions:

- 1)  $T^v \subseteq S^v$  and  $|T^v| \leq k$
- 2) For any two different IUnits  $s_m^v$  and  $s_n^v$ , if  $s_m^v \approx s_n^v$  then  $\{s_m^v, s_n^v\} \not\subseteq T^v$
- 3)  $\sum_{t_i^v \in T^v} \text{score}(t_i^v)$  is maximized.

To create the CAD View, we need to compute the diversified top- $k$  IUnits for each attribute value  $v$ . The diversified top- $k$  problem can be reduced to the NP-Hard maximum independent set problem on graphs [25]. Greedy solutions often lead to good approximate results in many NP-Hard problems, but for this problem a greedy algorithm can lead to arbitrarily bad solutions, with no bounded constant factor solution [25]. Because in our problem the size  $|S^v|$  is generally not large, Qin, *et al.*'s basic **div-astar** algorithm works well.

## 4. FINDING SIMILAR INFORMATION

In this section, we describe how to find similar components in the CAD View. These are solutions to Problems 3 and 4.

### 4.1 Finding Similar IUnits

If a user likes one of the IUnits, say IUnit  $t_i^x$  from  $T^x$ , the user can find all the IUnits  $t$  in the CAD View s.t.  $t_i^x \approx t$  (in other words,  $\text{sim}(t_i^x, t) \geq \text{tau}$ ). This approach allows us to address the IUnit sorting problem mentioned in Section 2.1.1; we can now sort IUnits from left-to-right by order of salience to the row's Pivot Attribute value, while still allowing the user to compare similar IUnits.

Computing similarity between IUnits is equivalent to computing similarity between clusters. For a numerical dataset, one can compute cluster distance by measuring the distance (such as Euclidean distance) between cluster centroids. For a categorical dataset, one can use any distance measure that is used in existing categorical clustering algorithms to compute cluster distance [11]. However, things become more complicated when we have a mixed dataset, having both numerical and categorical attributes. The distance measure that is used in categorical datasets is quite different compared to those used in numerical datasets. To compute similar IUnits, we propose a new distance measure that can treat both numerical and categorical attributes in a uniform manner. We use discretization to convert numerical attributes into categorical attributes. Then we use a modified form of *cosine-similarity* to compute IUnit similarity.

Let  $t_i^x$  and  $t_j^y$  be two top- $k$  IUnits for selected attribute values  $x$  and  $y$ , and  $\mathcal{I}$  be their set of Compare Attributes. We measure the similarity of  $t_i^x$  and  $t_j^y$  by summing their cosine

---

### Algorithm 1 IUnit Pair Similarity

---

*Input:*  $t_i^x$ : IUnit 1  
 $t_j^y$ : IUnit 2  
 $\mathcal{I}$ : set of informative dimensions  
*Output:*  $s$ : similarity between the two IUnits  
*Method:*

- 1:  $s \leftarrow 0$
- 2: **for all**  $d \in \mathcal{I}$  **do**
- 3:    $s \leftarrow s + \text{cosine-similarity}(t_i^x.d, t_j^y.d)$
- 4: **end for**
- 5: **return**  $s$

---

similarities along each dimension  $d$  s.t.  $d \in \mathcal{I}$ . We use the frequency count of each attribute value in the corresponding cluster as the attribute value's term frequency. Since the range of *cosine-similarity* function is  $[0, 1]$ , the range of the above similarity function is  $[0, |\mathcal{I}|]$ . Based on the specific data domain, one can choose the IUnit similarity threshold value  $\tau$  as some  $\alpha \cdot |\mathcal{I}|$ , where  $\alpha \in (0, 1)$ .

---

### Algorithm 2 Attribute-value Pair Similarity

---

*Input:*  $T^x = \{t_1^x, t_2^x, \dots, t_k^x\}$  top- $k$  IUnits for attribute value  $x$   
 $T^y = \{t_1^y, t_2^y, \dots, t_k^y\}$  top- $k$  IUnits for attribute value  $y$   
*Output:*  $d$ : distance between  $T^x$  and  $T^y$   
*Method:*

- 1:  $d \leftarrow 0$
- 2: **for all**  $t_i^x \in T^x$  **do**
- 3:   **if**  $\exists t \in T^y$  s.t.  $t \approx t_i^x$  **then**
- 4:      $index \leftarrow j$  s.t.  $t_i^x \approx t_j^y$  and  $\text{argmin}_j |j - i|$
- 5:   **else**
- 6:      $index \leftarrow |T^y| + 1$
- 7:   **end if**
- 8:    $d \leftarrow d + |i - index|$
- 9: **end for**
- 10: **for all**  $t_j^y \in T^y$  **do**
- 11:   **if**  $\exists t \in T^x$  s.t.  $t \approx t_j^y$  **then**
- 12:      $index \leftarrow i$  s.t.  $t_i^x \approx t_j^y$  and  $\text{argmin}_i |j - i|$
- 13:   **else**
- 14:      $index \leftarrow |T^x| + 1$
- 15:   **end if**
- 16:    $d \leftarrow d + |j - index|$
- 17: **end for**
- 18: **return**  $d$

---

### 4.2 Finding Similar Attribute Values

If a user has preference for a Pivot Attribute value, the user can create a CAD View where the first row contains IUnits for the preferred value, and the remaining Pivot Attribute values are shown in decreasing order of similarity to the preferred value. Two attribute values are considered similar if their top- $k$  IUnits lists are similar. Two ranked IUnit lists  $T^x$  and  $T^y$  should be similar if they have similar IUnits, and similar IUnits have similar rank.

To the best of our knowledge, there is no existing distance metric to compute similarity between two ranked lists having a disjoint set of items. In Algorithm 2, we propose a distance measure that can compute distance between two given ranked lists by taking into consideration the similarity between their items both in terms of information content and rank.

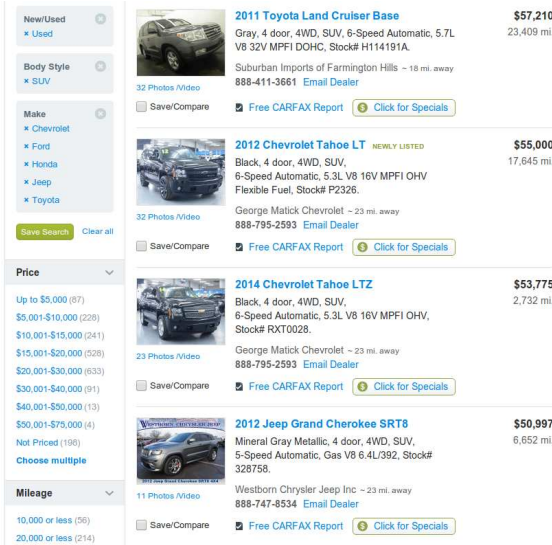


Figure 1: This screen capture from cars.com represents an example of a faceted navigation interface.

In lines 2-9, we compare how IUnits in  $T^x$  compare to IUnits in  $T^y$ . In line 3, we check whether the list  $T^y$  has some IUnit which is similar to IUnit  $t_i^x$  from  $T^x$ . If there is no similar IUnit (line 6), we assume that  $t_i^x$  is similar to the IUnit that has highest rank amongst non-selected IUnits (i.e.,  $S^y \setminus T^y$ ), and thus has rank  $|T^y| + 1$ . In lines 3-4, if there are multiple IUnits in  $T^y$  that are similar to  $t_i^x$ , then we take the IUnit whose rank is closest to rank of  $t_i^x$  in  $T^x$ , which is  $i$ . In line 8, we sum the rank differences for all IUnits in  $T^x$ . Lines 10-16 show the same steps for list  $T^y$ .

## 5. FACETED SEARCH WITH CAD VIEW

The CAD View defined above can be used with any relational database, independent of any front ends used. In fact, we have even suggested small SQL extensions to capture this concept. Nevertheless, we recognize that most end-users are unlikely to be SQL programmers, and are likely to be accessing relational data through some user-friendly interface. In this section, we consider one such popular interface, and describe how we have integrated CAD View with it.

Shoppers in e-commerce applications are a major target for our work: they are often exploring unfamiliar web sites before they actually buy. Since most e-commerce web sites use faceted navigation, that is the interface that we chose to integrate CAD View into. Figure 1 is a screenshot of a typical faceted interface for browsing a database of cars. In this section, we describe a novel two-phased faceted interface, called *TPFacet*, which integrates CAD View with faceted browsing.

A basic faceted interface has two main component panels: a query panel and a results panel. The latter typically occupies the majority of the screen real estate and shows the set of currently selected items. The former is usually on the left side, and offers both user interface controls as well as a summary digest of the current query and result set. This summary digest typically comprises all the attribute values (attribute values) that appear in the selected items, grouped by their corresponding attribute (attribute). The tuple count for each attribute value may also be included.

To fit the CAD View within users' limited screen space,

we propose a slightly changed interaction model for faceted navigation: at any one time, the interface will display either the results panel *or* the CAD View. The user explicitly toggles between them, though it is easy to imagine a system that intelligently chooses a default view, based on the size of query results. We imagine the user will interact with the system in two distinct phases: the *query revision* phase focuses on the CAD View, while the *result set* phase focuses on the results panel, with the user exploring individual items of interest in the result set.

Faceted navigation is an interaction based search system. We need to modify the faceted search interface so that users can create the CAD View or find similar components within the CAD View using interactive search techniques. We made the following three modifications: (i) Make each queryable attribute selectable (using html radio buttons) so that users can select them as Pivot Attribute, (ii) When users click on an IUnit in the CAD View we highlight all the other similar IUnits, and (iii) When users click a Pivot Attribute value in the CAD View we reorder all the rows in the CAD View in decreasing order of similarity w.r.t. the clicked attribute value. We call the faceted interface with these changes as TPFacet system.

## 6. EVALUATION

The goal of the CAD View is to facilitate exploratory search in complex datasets. As such, the primary evaluation of the CAD View is by means of a user study. In particular, we compare the use of the CAD View with a standard faceted interface for three exploratory search tasks. As a baseline for comparison, we use Apache Solr [2], which is a popular open source enterprise search platform. Apache Solr has support for faceted navigation and is used by many e-commerce sites. Apache Solr has many configuration settings. We chose a setting that is closest to the CAD View query model. We discuss the user study in depth in Section 6.2.

A secondary question is one of performance. Since the summaries shown in the CAD View are quite complex, we have to make sure that they can be computed in reasonable (interactive) time for the data set complexities and sizes that we expect. We discuss this issue in Section 6.3.

### 6.1 Implementation and Environment

We integrated the CAD View with Apache Solr to design the TPFacet system (see Section 5). We input the users' query from faceted interface, compute the CAD View and all similarity scores in the backend server, and return the resulting CAD View and similarity information using HTML and Javascript. To do feature selection and clustering, we use ChiSquare and SimpleKMeans algorithm respectively. Both algorithms are available in Weka [13].

We used two real datasets—YAHOOUSED CAR and MUSHROOM [9]—to do the evaluation. We scraped Yahoo's used car site [1] to create a table comprising 40,000 tuples with 11 attributes. The MUSHROOM dataset has 8124 tuples with 23 attributes. These numbers are at the lower end of what one sees in a typical e-commerce dataset. The CAD View will become more valuable in datasets that have more number of attributes or tuples. The MUSHROOM dataset is very popular in machine learning. It is simple to understand for a non-expert, since it describes familiar properties, such as color and smell, but has data that most of us (and all of our

users) have no knowledge of, forcing us to learn patterns by examining the data set afresh without reliance on previous knowledge.

## 6.2 User Study

We devised a diverse set of carefully specified information exploration tasks, described in the subsections that follow, each of which tests (some aspects of) the users’ understanding of the database. These tasks roughly correspond to the two motivating limitations discussed in Section 1. The first two tasks correspond to Limitation 1, where we evaluate users’ ability to perform comparisons in the form of finding differences and similarities respectively. The third task corresponds to Limitation 2, where we test users’ ability to query non-queriable attributes using available queriable attributes. We used the MUSHROOM data set, which was unfamiliar to all our users.

We compare TPFacet and Solr in terms of their usability in users’ task completion time and quality of response to given tasks. For all the tasks we report the results using statistical analysis.

We performed our user study using eight graduate students from our university. As we will see in the following subsections, statistical analysis show that the conclusions we draw from these eight users are statistically significant.

We gave all the users a demo explaining all features of the TPFacet and the steps to do the tasks using both the interfaces. We allowed the users to do the tasks remotely to minimize effect of any environmental factors. We created 3 matched pairs of tasks, one pair for each type described below. We divided the eight users into two equal groups. We indicate each user by their user id  $U1-U8$ . Users with id  $U1-U4$  were assigned to group 1 and  $U5-U8$  to group 2. For a task pair  $(A, B)$  we asked one of the groups to do task  $A$  using TPFacet and task  $B$  using Solr. We reversed the task assignment for the other group. In other words, if a task was done by group 1 users using Solr, then the same task was done by group 2 users using TPFacet, and vice versa.

For all the three tasks we have performed linear mixed model statistical analysis [28]. We use Display type as fixed effect and User ID as random effect. Computing p-values for mixed models aren’t as straightforward as they are for linear models. The most popular way to obtain p-value is to use the *Likelihood Ratio* test as a means to attain p-values. The logic of the likelihood ratio test is to compare the likelihood of two models with each other. First, the model without the factor that one is interested in (the null model) and then the model with the factor that one is interested in. By comparing these two models, one can determine whether the factor one is considering is significant or not. We use ANOVA to compare the two models.

### 6.2.1 Simple Classifier

This task illustrates the benefits of the CAD View in finding differences between attribute values. We asked users to build a simple classifier. Classification is an important machine learning problem where given a training data with multiple class labels, one builds a classification model by which one can find the set of classes (categories) a new test observation belongs. In this task, we build a classifier for binary class data. We assume a simple classification model that consists of selecting at most two attribute values that maximizes the number of tuples retrieved from a given

target class, and minimizes the number of tuples from the other class. Although problems like classification are rarely done manually for large datasets, human ability in this task demonstrates an understanding of crucial database themes. We evaluate the goodness of the classifier using standard F1 accuracy score. A sample task was to build a classifier for target class **Bruises = true**, where the given classes were  $Bruises = \{true, false\}$ .

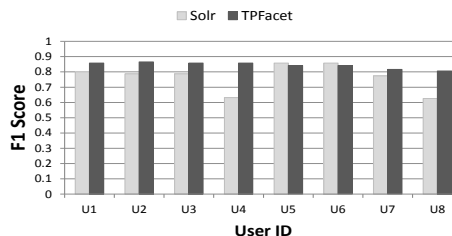


Figure 2: Simple Classifier

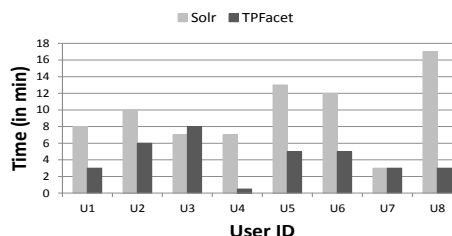


Figure 3: Simple Classifier

Figure 2 shows the F1 scores for the classifiers that users got for this task. Statistical analysis shows that TPFacet affects the quality of classifier by  $(\chi^2(1) = 5.572, p = 0.018)$ , increasing the F1 score by about  $0.078 \pm 0.0285$ . Moreover, the variation in F1 score is much lower when users use the TPFacet system as compared to Solr because the exploration using TPFacet is more methodical. In Figure 3, we show the time taken by the users to build the classifiers. Statistical analysis shows that TPFacet affects the time taken by  $(\chi^2(1) = 8.54, p = 0.003)$ , lowering it by about  $5.44 \pm 1.56$  minutes.

### 6.2.2 Most Similar Facet Value Pair

This task illustrates the benefits of the CAD View in finding similar (or equivalent) attribute values. In this task, we gave users a list of four attribute values from an attribute and asked them to find the two most similar attribute values. For example, given attribute = **GillColor** and attribute values = **{buff, white, brown, green}**, find the two most similar gill colors.

In the traditional faceted interface, users can Compare Attribute values by comparing their summary digest. We gave users a cosine-similarity based distance metric to compare the summary digests. We asked users to select each of the given attribute values, one at a time, and compare their summary digest. In the CAD View, we didn’t show the computed similarity scores, but allowed users to use interactive effects to find similar IUnits and attribute values.

Figure 4 shows users response quality for this task. Since there are four attribute values, there are 6 possible attribute value pairs. We computed the defined similarity score for each pair and ranked them from 1 to 6, with the most similar pair being ranked as 1. Since computing exact similarity score is very hard for humans, we purposely chose attributes

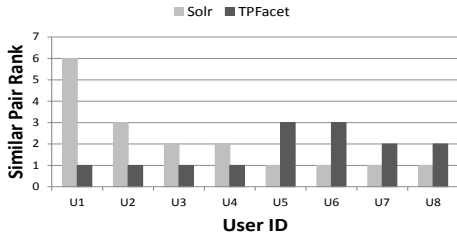


Figure 4: Most Similar Attribute Value Pair

and attribute values that would make the task humanly feasible in Solr. The similarity between gill colors brown and white was so high as compared to other choices that all the eight users got correct answer for this task. Group 1 users (U1-U4) did this task using TPFacet and group 2 users using Solr. However, the other similarity task was slightly harder. For the other task, users U7 and U8 got the most similar attribute value pair according to attribute value similarity we defined in Section 4, but according to the metric defined in this task, they turned out to be second most similar pair. Statistical analysis shows that there is no difference in users response quality by using the two types of interface.

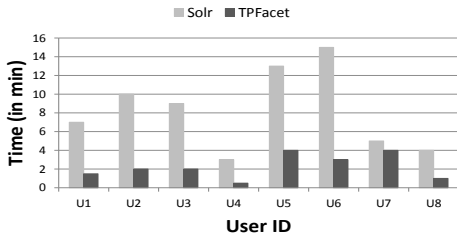


Figure 5: Most Similar Attribute Value Pair

Figure 5 shows the time users took to finish this task. Statistical analysis shows that TPFacet affects the time taken by ( $\chi^2(1) = 12.04$ ,  $p = 0.0005$ ), lowering it by about  $6.00 \pm 1.23$  minutes. All users, except user U7, finished the task around four times faster using TPFacet as compared to Solr. Since the users were doing this task for the first time, some of them were trying to manually compare the IUnits. Users could have got the desired answer for this task much faster by just using the interactive effects, as seen in case of users U4, U8 and U1.

### 6.2.3 Alternative Search Condition

This task illustrates the benefits of the CAD View in querying non-queriable attributes using queriable attribute values. In this task, we gave users a set of selection conditions that lead to some result set  $\mathcal{R}$ . We asked users to find another set of selection conditions that would lead to same result set  $\mathcal{R}$ , but not using any of the already given selection conditions. One can see the given selection conditions as selection conditions on non-queriable attributes that the users cannot query. Only an informed user can precisely access the desired result set using an alternate option. A sample task was to find an alternative selection condition using at most two attribute values that would lead to the same result as selecting: `StalkShape = enlarged` and `SporePrintColor = chocolate`.

To evaluate users response quality, we checked the similarity between the query result obtained from the given selection condition and the users alternate selection condition. To measure similarity between the two results, we measured the similarity between their faceted summary digest.

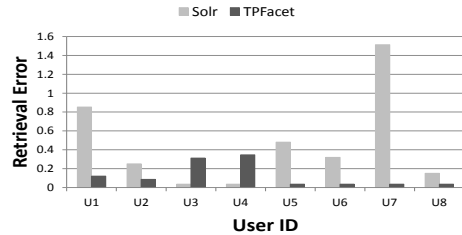


Figure 6: Alternative Search Condition

Figure 6 shows users response quality for this task. Statistical analysis shows that TPFacet affects the users alternative search condition by ( $\chi^2(1) = 3.28$ ,  $p = 0.07$ ), lowering the retrieval error by about  $0.329 \pm 0.172$ . Using TPFacet most users were able to do the task with five times lower retrieval error. In this task pair, the task that group 1 users did using Solr, turned out to be quite easier compared to the one they had to do using TPFacet. We can see this difference by seeing that users U3 - U8 have very low and similar error for this task. For the easier task, just one attribute value was sufficient to get to the desired result set. All the users in group 2 had come up with slightly variant solutions, but exactly the same retrieval error (48 missing tuples out of 1344). Since TPFacet allows users to do this type of task in more methodical approach compared to Solr, we find much lower variation in users response quality. For the slightly harder task, we see slight variation in retrieval error among group 1 users who did this task using TPFacet, but the error variation is much higher for group 2 users who did it using Solr. Group 2 users, such as U5, U6 and U8, who had much lower error compared to user U7 had to spend significantly more amount of time as seen in Figure 7.

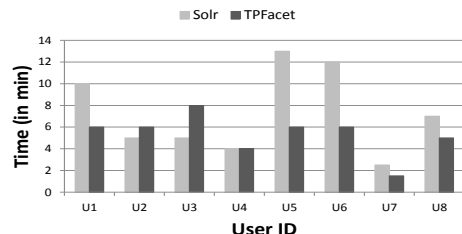


Figure 7: Alternative Search Condition

Figure 7 shows the time users took to finish this task. Statistical analysis shows that TPFacet affects the time taken by ( $\chi^2(1) = 2.58$ ,  $p = 0.108$ ), lowering it by about  $2.00 \pm 1.14$  minutes. Most users were able to do the task 1.5 to 2 times faster using TPFacet as compared to Solr. This task required more time because users had to manually differentiate the IUnits. The main benefit of TPFacet was that users didn't have to try various options using hit-and-trial. They had to look through the IUnits to find the discriminating attribute values, but then it was just trying very few possible alternate choices to see which one gives the best result.

## 6.3 Performance

Computational time is a crucial constraint for all user facing applications because users expect almost instantaneous response. In this subsection, we evaluate whether TPFacet can provide interactive responses. We performed all our performance experiments on the YAHOOUSED CAR dataset with 40K tuples and 11 attributes. When users browse over e-commerce sites, they rarely deal with result size that is more than 30K-40K tuples and 5-10 queriable attributes. Thus

we evaluate our system using all the tuples of our used-car dataset as query result, with all its attributes being used as queryable attributes.

Our experiments show that TPFacet can give acceptable performance by just using computationally efficient feature selection and clustering algorithms. Each of our experimental graphs are based on average readings of 50 simulations, where for each simulation we generate a different query result by randomly selecting a subset of tuples and/or attributes. The default parameters in these experiments are: the number of Compare Attribute  $\mathcal{I} = 11$ , the number of generated IUnits  $l = 10$ , the number of IUnits shown  $k = 6$ , and the number of attribute-values selected in the Pivot Attribute  $\mathcal{V} = 5$ . In these experiments, we assume that if the total size of the query result set is  $|\mathcal{R}|$ , then each attribute value  $v \in \mathcal{V}$  has  $|\mathcal{R}|/|\mathcal{V}|$  tuples.

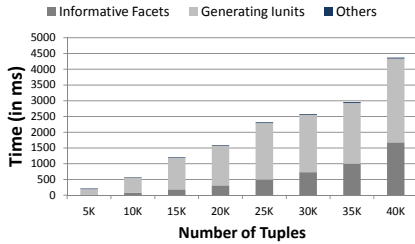


Figure 8: Worst Case System Performance

Figure 8 shows the total time to compute the CAD View for different sizes of query result. In the shown graph, we do not do any optimizations, except using a computationally efficient feature selection and clustering algorithm, that can lead to better system performance. Moreover, we chose system parameter values to demonstrate the worst-case performance of our system. For example, we kept  $|\mathcal{I}| = 11$  and  $l = 15$ . When we consider interaction between many attributes (large  $|\mathcal{I}|$ ) or try to compute many interactions (large  $l$ ), then it decreases system performance, as shown in later experiments. We divide the total time into three parts: time to compute Compare Attribute, time to generate IUnits and time for all remaining steps, such as top-k ranking, and similarity between IUnits and attribute-values, that we represent collectively as others. We can see that the most computationally intensive parts of TPFacet is computing the top Compare Attribute and generating candidate IUnits. Total time for all other steps is negligible because of the small values of  $k$  and  $|\mathcal{V}|$  established due to user’s display constraint. We can see that even this naive solution is acceptable when the result size is less 15K. But as we increase the result set size, we can see that the time to compute CAD View increases and becomes almost 4.5 secs for 40K tuples. Since the result set size is likely to be the largest in the initial stages of exploration, and since this is also likely to be when the user really needs interactive response to freely try alternatives, a multi-second response time is too slow. To alleviate this problem, we developed several optimizations.

**Optimization 1. Sampling** — Sampling can improve both feature selection and clustering. For all our attributes, when we computed the set of top ranked Compare Attribute using a small random sample of size 5K-10K, we always got almost the same set, as we got from any larger sample size, including the full dataset. As shown in Figure 8, computing

Compare Attribute takes only 20-50 ms for 5K-10K tuples, as compared to 1700 ms for 40K tuples. Quality of Compare Attribute is more crucial when users are towards the end of their exploration, and at that time even the exact computation will take very short time due to small result size. Even if there were some degradation in quality due to sampling, it may not matter much in the initial stages. Similarly, we can also reduce the time for generating IUnits by generating IUnits from a small sample.

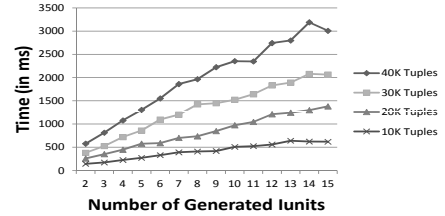


Figure 9: Number of Generated IUnits vs Time

**Optimization 2. Varying Generated IUnits** — Figure 9 shows the effect of number of generated IUnits  $l$  on computation time for different result sizes. We observed that as we increased the number of generated IUnits, it increases computation time due to increased time for clustering. For small 10K result size, computation time is small, less than 500 ms, even when we generate 15 IUnits per attribute value. However, if the result set is large and we generate large number of IUnits, as shown in Figure 8 for 40K tuples with  $l = 15$ , then it slows down system performance. When users are in their beginning stages of exploration, it is hard to know their preference because their query is too broad. Generating more IUnits and finally ranking is meaningful when we know users’ preference more precisely, which typically happens near the end-stages of exploration. Thus we generate fewer IUnits when the result set is very large, so that we can provide a good summary of all options. As users narrow down their exploration, we increase the number of generated IUnits and return better top- $k$  IUnits.

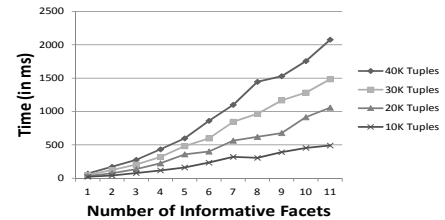


Figure 10: Number of Compare Attributes vs Time

**Optimization 3. Fewer Compare Attributes** — Figure 10 shows the effect of number of Compare Attributes on computing clusters for different result sizes. As we increase the number of Compare Attributes, it increases computation time because we need to look at the interaction between larger number of attributes. By showing few Compare Attributes we can cluster even 40K tuples in less than 500 ms.

By combining all the above optimizations in creating the CAD View, we can greatly increase the performance of TPFacet system. For example, we can get an CAD View for 40K tuples in less than 500 ms.

## 7. RELATED WORK

Exploratory search [26, 27, 24, 19] has recently become an important research problem in IR, HCI and database communities. We defined a new exploratory search problem in databases. In evaluating exploratory search systems we cannot separate human behavior from the search system. Since users have diverse background knowledge and information need, it is difficult to evaluate exploratory search systems. Designing evaluation metrics and methodologies for exploratory search system is a challenging research problem [26]. We presented a detailed user-study, based on explicit exploration/understanding tasks with quantitative measures, to evaluate our system.

The CAD View is a summary of important interactions between attributes. Measuring attribute interactions is a part of broader feature selection problem [12, 22, 18] in machine learning. In databases, attribute interactions are often measured in form of functional dependencies [8, 16] and referential integrities. Although standard feature selection can find the interaction between attributes, a Bayesian network [15] can provide a more accurate description of attribute interactions by giving probabilistic dependencies between attributes. These techniques can be used to create CAD Views with other types of data summaries.

Large volumes of relational data are often summarized using data warehousing and OLAP technology [10]. There are also many data mining techniques, such as clustering [20, 3] and decision trees [4, 6], that can group data into meaningful groups according to some user given notion of similarity. A central property of these algorithms is that they depend on the data and are independent of the user's interest. Therefore, the results are often not related to the user's specific exploratory goal. In this paper, we presented a context dependent summarization technique.

Faceted categorization and clustering are both grouping techniques. Hearst [14] presents a nice comparison of how these two techniques complement each other. Various usability studies have shown that users prefer the predictable faceted categorization over clustering [7]. In this paper, we combined faceted browsing with clustering to build the TP-Facet system that has benefits of both faceted navigation and clustering.

## 8. CONCLUSION

In this paper, we presented an exploratory search system for relational databases. Our solution relies on a novel data summarization technique called the CAD View, which provides a context dependent summary of relational result set. We showed through an extensive user study that the CAD View can help users gain quick data familiarity with complex datasets. Although computing the CAD View is computationally intensive, we provided optimizations that enable it to be easily integrated with existing search interfaces, without compromising system performance.

## 9. REFERENCES

- [1] Yahoo used-car. [http://autos.yahoo.com/used\\_cars.html](http://autos.yahoo.com/used_cars.html).
- [2] Apache solr. <http://lucene.apache.org/solr/>, 2014.
- [3] P. Berkhin. A survey of clustering data mining techniques. In *GMD*, pages 25–71. Springer, 2006.
- [4] K. Chakrabarti, S. Chaudhuri, and S. Hwang. Automatic categorization of query results. In *SIGMOD*, pages 755–766. ACM, 2004.
- [5] K. Chen and L. Liu. Clustermap: Labeling clusters in large datasets via visualization. In *CIKM*, pages 285–293. ACM, 2004.
- [6] Z. Chen and T. Li. Addressing diverse user preferences in sql-query-result navigation. In *SIGMOD*, pages 641–652. ACM, 2007.
- [7] J. C. Fagan. Usability studies of faceted browsing: A literature review. *ITL*, 29(2):58–66, 2013.
- [8] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *TKDE*, 23(5):683–698, 2011.
- [9] A. Frank and A. Asuncion. UCI Machine Learning Repository, 2010.
- [10] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.
- [11] S. Guha, R. Rastogi, and K. Shim. Rock: A robust clustering algorithm for categorical attributes. In *ICDE*, pages 512–521. IEEE, 1999.
- [12] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *JMLR*, 3:1157–1182, 2003.
- [13] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.
- [14] M. A. Hearst. Clustering versus faceted categories for information exploration. *Communications of the ACM*, 49(4):59–61, 2006.
- [15] D. Heckerman. *A tutorial on learning with Bayesian networks*. Springer, 2008.
- [16] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647–658. ACM, 2004.
- [17] H. Jagadish and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, pages 275–286, 1998.
- [18] I. Kononenko. Estimating attributes: analysis and extensions of relief. In *ECML*, pages 171–182. Springer, 1994.
- [19] G. Koutrika, L. V. Lakshmanan, M. Riedewald, and K. Stefanidis. Exploratory search in databases and the web. In *EDBT/ICDT Workshops*, pages 158–159, 2014.
- [20] C. Li, M. Wang, L. Lim, H. Wang, and K. Chang. Supporting ranking and clustering as generalized order-by and group-by. *SIGMOD*, 2007.
- [21] B. Liu and H. Jagadish. Using trees to depict a forest. *VLDB*, 2009.
- [22] H. Liu and L. Yu. Toward integrating feature selection algorithms for classification and clustering. *TKDE*, 17(4):491–502, 2005.
- [23] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, 2008.
- [24] G. Marchionini. Exploratory search: from finding to understanding. *Communications of the ACM*, 49(4):41–46, 2006.
- [25] L. Qin, J. X. Yu, and L. Chang. Diversifying top-k results. *VLDB Endowment*, 5(11):1124–1135, 2012.
- [26] R. W. White and R. A. Roth. Exploratory search: Beyond the query-response paradigm. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 1(1):1–98, 2009.
- [27] M. L. Wilson, R. W. White, et al. Evaluating advanced search interfaces using established information-seeking models. *ASIST*, 60(7):1407–1422, 2009.
- [28] B. Winter. A very basic tutorial for performing linear mixed effects analyses, 2001.
- [29] T. Wu, X. Li, D. Xin, J. Han, J. Lee, and R. Redder. DataScope: viewing database contents in Google Maps' way. *VLDB*, 2007.



# Refinement Driven Processing of Aggregation Constrained Queries

Manasi Vartak<sup>1,a</sup>, Venkatesh Raghavan<sup>2,b</sup>, Elke Rundensteiner<sup>3,c</sup>, Samuel Madden<sup>4,a</sup>

<sup>a</sup>Massachusetts Institute of Technology, <sup>b</sup>Pivotal Inc., <sup>c</sup>Worcester Polytechnic Institute

<sup>1</sup>mvartak@mit.edu, <sup>2</sup>vraghavan@pivotal.io, <sup>3</sup>rundenst@cs.wpi.edu, <sup>4</sup>madden@csail.mit.edu

## ABSTRACT

Although existing database systems provide users an efficient means to select tuples based on attribute criteria, they however provide little means to select tuples based on *whether they meet aggregate requirements*. For instance, a requirement may be that the cardinality of the query result must be 1000 or the sum of a particular attribute must be  $< \$5000$ . In this work, we term such queries as “Aggregation Constrained Queries” (ACQs). Aggregation constrained queries are crucial in many decision support applications to maintain a product’s competitive edge in this fast moving field of data processing. The challenge in processing ACQs is the unfamiliarity of the underlying data that results in queries being either too strict or too broad. Due to the lack of support of ACQs, users have to resort to a frustrating trial-and-error query refinement process. In this paper, we introduce and define the semantics of ACQs. We propose a refinement-based approach, called ACQUIRE, to efficiently process a range of ACQs. Lastly, in our experimental analysis we demonstrate the superiority of our technique over extensions of existing algorithms. More specifically, ACQUIRE runs up to 2 orders of magnitude faster than compared techniques while producing a 2X reduction in the amount of refinement made to the input queries.

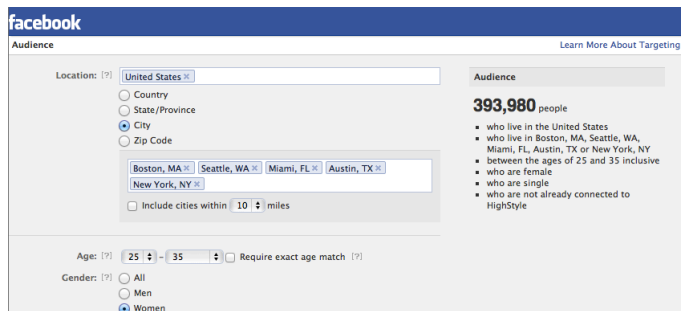
## 1. INTRODUCTION

Databases provide a number of ways to efficiently select tuples of interest to the user by constraining attributes of individual tuples, for instance, return tuples that meet the criteria  $price < \$50$ , join results between tuples in table A and table B that match on attribute “id,” etc. However, little effort has been focused on a means of selecting tuples based on *whether they satisfy aggregate constraints*. For instance, select tuples with average  $price < \$10$ , number of tuples = 1000, etc. The ability to apply aggregate constraints along with constraints on tuples’ individual attribute values is important in many applications as illustrated below.

- In advertising campaigns (such as Example 1), *the budget restricts the number of users that can be reached* [4]; as a result, the campaign manager must select users based not only on demographics but also whether the total number of users (i.e. the COUNT) is within the budget limit.

- In a supply chain application, *a requirement on the total number of parts to be ordered from suppliers translates to a constraint on the sum of the number of parts available with each supplier* (Example 2). As a result, queries must place constraints not only on part specifications but also the SUM of the parts available.
- When analyzing large data sets through aggregates [15], users often want to identify what input tuples produced outliers in aggregate values (e.g. select patients who had extremely high average cost). In this case, the user would like to place constraints on the AVG aggregate.

**Example 1.** *HighStyle Designers would like to run a Facebook<sup>1</sup> ad campaign to get more users to “like” their page. The campaign budget of \$10,000 will allow HighStyle to reach 1 million customers. Therefore, when the campaign manager, Alice, selects target users, she must not only constrain her search based on customer demographics but also based on the total number of customers who must be reached. This situation thus calls for an “Aggregation Constrained Query” (ACQ).*



**Figure 1: Facebook Ad Creation Interface: Allows specifying demographic criteria and view estimate of audience size.**

Figure 1 shows the Facebook’s Advertising Interface<sup>2</sup> that allows campaign manager Alice to select target users for her ad. In terms of SQL, Alice has to run the following query:

```
Q1: SELECT * FROM Users
WHERE location in ('Boston', 'New York',
'Seattle', 'Miami', 'Austin') AND
(gender = 'Women') AND (25 <= age <= 35)
AND (education = 'CollegeGrad')
```

<sup>1</sup><http://www.facebook.com>

<sup>2</sup><https://www.facebook.com/ads/create/>

```
AND (relationshipStatus = 'Single')
AND (interests IN {'Retail', 'Shopping'})
```

**Need for Query Refinement.** For Alice’s above query, Facebook estimates the reach to be 393,980 users, i.e. only 40% of the required 1 million users. While the results of query Q1 precisely satisfy Alice’s selection predicates, they are far from meeting her aggregate constraints. In fact, selection and aggregation constraints are orthogonal in most cases. As a result, we need to refine various query predicates in order to meet the aggregate constraints.

**Current Approach.** In existing systems Alice has to manually alter her criteria to encompass more users while ensuring that the semantics of her query are not altered. While some selection criteria (e.g. gender and shopping interest) may be fixed, Alice can try potentially infinite refinements of her predicates such as target consumers in additional cities; alter age range; relax relationship status; or any combination of the above. Repeatedly altering the original query and having its size estimated is not only inefficient for the backend, but the process is tedious and frustrating for Alice.

**Desired User Experience.** A much better user experience can be provided if Alice was allowed to specify her (1) demographic criteria (query), and (2) aggregate constraints, and the database engine can then execute variations of the input query such that the aggregate constraints are met. The output of such a search would be a set of refined queries that change Q1 as little as possible while meeting the aggregate constraints (in our case the audience size). Alice would then simply pick the query that best meets her selection criteria.

In this paper, we encode ACQs by introducing two SQL keywords CONSTRAINT and NOREFINE, where CONSTRAINT captures the aggregate constraint and NOREFINE specifies whether the predicate should not be refined. The encoded Query Q1 is:

```
Q1': SELECT * FROM Users
CONSTRAINT COUNT(*)=1M
WHERE location in ('Boston', 'New York',
'Seattle', 'Miami', 'Austin') AND
(gender = 'Women') NOREFINE AND (25 <=age<=35)
AND (education = 'CollegeGrad')
AND (relationshipStatus = 'Single') AND
(interests IN {'Retail', 'Shopping'}) NOREFINE;
```

Running Q1' will automatically generate alternate queries that produce 1M customers and alter Q1 as little as possible.

**Example 2.** *HybridCars Co. would like to place an order for 100,000 units of a burnished steel part having specific size, wholesale price less than \$1000, and from suppliers who have a low account balance. On the TPC-H benchmark, HybridCars runs query Q2 to find the suppliers with whom to place the order.*

```
Q2: SELECT * FROM supplier, part, partsupp
WHERE (s_suppkey = ps_suppkey) AND
(p_partkey = ps_partkey) AND
(s_acctbal < 2000)
AND (p_retailprice < 1000) AND (p_size = 10)
AND (p_type = 'SMALL BURNISHED STEEL')
```

As in Example 1, this situation calls for an ACQ as we would like to constrain the total number of available parts, i.e. *sum of the number of parts available per supplier* (i.e. SUM(ps\_availqty)) in addition the select predicates. We can encode the ACQ as Q2' to produce alternate refined queries. As before, the NOREFINE keyword associated with p\_type and p\_size indicate that these predicates cannot be altered.

```
Q2': SELECT * FROM supplier, part, partsupp
CONSTRAINT SUM(ps_availqty) >= 0.1M
WHERE (s_suppkey = ps_suppkey) NOREFINE AND
(p_partkey = ps_partkey) NOREFINE AND
(p_retailprice < 1000) AND (s_acctbal < 2000)
AND (p_size = 10) NOREFINE AND
(p_type = 'SMALL BURNISHED STEEL') NOREFINE
```

Building a system to execute ACQs is challenging because the number of possible refined queries is exponential in the number of predicates. Hence an exhaustive search of all possible queries is prohibitively expensive. Moreover even for aggregates such as COUNT, finding a query that meets its constraint is an NP-Hard problem [1]. In this paper, we limit ourselves to ACQs with numerical select and join predicates, and aggregates that satisfy the *optimal substructure* property (Section 2). Additionally, we focus on the problem of *expanding predicates* to meet constraints, rather than the inverse problem of shrinking queries returning too many tuples.

**Contributions.** We propose a technique to efficiently execute ACQs and our contributions are summarized as follows:

- We introduce and define semantics of a new class of queries called an **Aggregation Constrained Query (ACQ)**. These special purpose queries are of value in real-world applications and are amenable to clever execution techniques.
- We propose a technique called ACQUIRE to execute ACQs via query refinement. ACQUIRE auto-generates alternative refined queries that minimize changes to the original query while meeting aggregate constraints.
- We combine the building blocks of breadth-first-search and dynamic programming in a novel way to elegantly and efficiently re-use query results. We call this *Incremental Aggregate Computation* (Section 5).
- We propose sensible default query refinement scoring and aggregate error functions. The design principle of ACQUIRE is general and therefore we allow user defined predicate refinement scoring and aggregate error functions. The functions used in this work are merely sensible defaults.
- Our experimental analysis on TPC-H dataset demonstrates that ACQUIRE consistently out-performs extensions to current techniques by up to 2 orders of magnitude. Moreover, queries recommended by ACQUIRE are on average closer to the original query by a factor of 2X more than the compared techniques (Section 8).

## 2. PRELIMINARIES

### 2.1 SQL extension for ACQs

We propose to capture ACQs by using two keywords: CONSTRAINT to describe the aggregate constraint and NOREFINE to indicate that a predicate should not be refined. By default, we assume that all predicates can be refined.

```
SELECT * FROM Table1, Table2 ...
CONSTRAINT AGG(attribute) Op X
WHERE Predicate1 AND Predicate2 ...
AND Predicate_i NOREFINE AND Predicate_j
AND ...Predicate_n NOREFINE
```

The aggregate constraint is of the form  $AGG(attribute) Op X$ , where AGG is a standard (COUNT, SUM, MIN, MAX, AVG) or user defined aggregate function,  $X$  is a positive number and  $Op$  is a comparison operator ( $=, \leq, <, \geq, \text{and } >$ ). In this work, we focus on the problem of expanding predicates to meet constraints, rather than the inverse problem of shrinking queries returning too many tuples, we therefore limit the comparison operation to  $=, \geq, \text{and } >$ . Henceforth for illustrative purposes only we assume that the aggregate constraint has an equality condition.

## 2.2 Query Representation

In this work, we focus on queries with numeric select, project and join predicates of the form  $Q = P_1 \wedge \dots \wedge P_d$ , where  $P_i$ 's is a predicates on relations  $R_1 \dots R_k$ . To illustrate consider query Q3 with one select and one join predicate.

```
Q3: SELECT * FROM A, B
     WHERE A.x=B.x AND B.y < 50
```

For a given query  $Q$ , we divide each predicate  $P_i$  into two parts: the *predicate function* ( $P_i^{\mathcal{F}}$ ) and the *predicate interval* ( $P_i^{\mathcal{I}}$ ).  $P_i^{\mathcal{F}}$  is a monotonic function on attributes of relations  $R_1 \dots R_k$  while  $P_i^{\mathcal{I}}$  denotes the interval of acceptable values for  $P_i^{\mathcal{F}}$ , that is,  $P_i^{\mathcal{I}} = (min_i^{\mathcal{I}}, max_i^{\mathcal{I}})$ . To illustrate, if the minimum value of  $B.y$  is 0, the predicate  $(B.y < 50)$ , in Q3 is decomposed into  $P_i^{\mathcal{F}} = B.y$  and  $P_i^{\mathcal{I}} = (0, 50)$ . Range predicates like  $(10 < B.y < 50)$  are rewritten as two one-sided predicates,  $(B.y > 10) \wedge (B.y < 50)$ . This enables the refinement of one or both sides of the range predicate. For equi-joins  $(A.x = B.x)$  and non-equi joins  $(2 * A.x < 3 * B.x)$ , the form of  $P_i^{\mathcal{I}}$  is unchanged; however,  $P_i^{\mathcal{F}}$  takes the form  $\Delta((P_i^{\mathcal{F}})_1, (P_i^{\mathcal{F}})_2)$ , where  $(P_i^{\mathcal{F}})_1$  and  $(P_i^{\mathcal{F}})_2$  are separate predicate functions and  $\Delta$  is the function measuring distance between them. Therefore, join predicate  $A.x = B.x$  in Q3 is decomposed into  $(P_i^{\mathcal{F}})_1 = A.x$  and  $(P_i^{\mathcal{F}})_2 = B.x$ .  $P_i^{\mathcal{I}} = (0, 0)$  signifies that values of the two functions must match exactly. For each predicate  $P_i$ , we also store a boolean value indicating whether the predicate can be refined. Recall that ACQ's contain an aggregate function that specifies the target value of an aggregate over the output result. We denote the target, or expected, aggregate value as  $A_{exp}$  and actual aggregate value returned by the query as  $A_{actual}$ .

## 2.3 Measuring Query Refinement Quality

We define a query refinement score to measure the change that has been made to the original query to obtain the refined query. A query  $Q=(P_1 \wedge \dots \wedge P_d)$  is refined to  $Q'$  by refining one or more predicates  $P_i \in Q$  to predicates  $P_i' \in Q'$ . The refinement of  $Q'$  along  $P_i$ , called the **predicate refinement score**, denoted as  $PScore_i(Q, Q')$ , is measured as the percent departure of  $(P_i^{\mathcal{I}})'$  from  $P_i^{\mathcal{I}}$  (Equation 1). Note that if  $(P_i^{\mathcal{I}})_{min} = (P_i^{\mathcal{I}})_{max}$ ,  $PScore_i(Q, Q') = 0$ . For equality join predicates, the denominator is set to 100. Measuring relative change as opposed to absolute change in predicate intervals, compensates for the differing scales of query attributes. While percent refinement is the default predicate refinement metric used in this work, a user can override the metric with custom (monotonic) functions without changes to our algorithm. By computing the refinement score for each query predicate, a refined query  $Q'$  can be represented as a d-dimensional vector of predicate refinement scores, called the **predicate refinement vector** or  $\overline{PScore}(Q, Q')$  (Equation 2).

$$PScore_i(Q, Q') =$$

$$\frac{|(P_i^{\mathcal{I}})_{min} - (P_i^{\mathcal{I}})'_{min}| + |(P_i^{\mathcal{I}})_{max} - (P_i^{\mathcal{I}})'_{max}|}{|(P_i^{\mathcal{I}})_{max} - (P_i^{\mathcal{I}})_{min}|} \cdot 100 \quad (1)$$

$$\overline{PScore}(Q, Q') = (PScore_1(Q, Q') \dots PScore_d(Q, Q')) \quad (2)$$

The **query refinement score** of  $Q'$ , denoted by  $QScore(Q, Q')$  is defined as a monotonic function  $f : \mathcal{R}^d \rightarrow \mathcal{R}$  used to measure the magnitude of  $\overline{PScore}(Q, Q')$ . We use the popular weighted vector p-norms [7] to calculate  $QScore(Q, Q')$ . Equation 3 shows the calculation of  $QScore(Q, Q')$  using the default  $L_1$  norm.

$$L_1 : QScore(Q, Q') = \left( \sum_{i=1}^d PScore_i(Q, Q') \right) \quad (3)$$

**Example 3.** Consider the following refinement to Q3.

```
Q3': SELECT * FROM A, B
      WHERE A.x = B.x AND B.y < 60
```

The refined query  $Q3'$  expands the range of acceptable values for  $B.y$  from  $(0, 50)$  to  $(0, 60)$ . Therefore,  $Q3'$  is represented as  $\overline{PScore}(Q3, Q3') = (0, \frac{60-50}{50-0} \cdot 100)$  and has  $QScore(Q3, Q3')=20$  for the  $L_1$  norm.

## 2.4 Refining Join Predicates

The advantage of representing predicates as functions ( $P_i^{\mathcal{F}}$ ) and intervals ( $P_i^{\mathcal{I}}$ ), and defining refinement as the change in the predicate interval, is that join refinement can be expressed and operated on in the same way as select predicates. For instance, a query with  $\overline{PScore}(Q3, Q3'') = (10, 20)$  indicates that the join predicate in Q3 has been refined by 10 to become  $\|A.x - B.x\| \leq 10$  and that the  $B.y$  predicate has been refined by 10 units. Thus, the algorithm can be applied unchanged for select as well as join queries.

## 2.5 Measuring Aggregate Error

To measure the difference between the expected aggregate value  $A_{exp}$  and the actual aggregate value  $A_{actual}$ , we use a relative error measure defined as:

$$Err_A = \frac{\|A_{exp} - A_{actual}\|}{A_{exp}} \quad (4)$$

This measure is appropriate for aggregates such as COUNT or AVG; however, a hinge-function that only penalizes errors on one side is appropriate for SUM, MIN and MAX.

$$Err_A = \begin{cases} (A_{exp} - A_{actual}) & \text{if } A_{exp} > A_{actual} \\ 0 & \text{otherwise} \end{cases}$$

## 2.6 Optimal Substructure Property

In this work, we limit ourselves to aggregate functions that either (a) have the **optimal substructure property** (OSP), or (b) can be broken down into functions that satisfy the OSP. Consider any two queries Q1 and Q2 such that all the results of query Q2 are also results of query Q1 (Q1 *contains* Q2). An aggregate is said to satisfy the OSP if the value of the aggregate for the results of Q1 can be computed without re-executing part or whole of the query Q2.

For instance, the COUNT aggregate is said to satisfy the OSP because given queries Q1 and Q2 as defined above, the value of COUNT for Q1 can be computed by adding the value of COUNT for Q2 to the value of COUNT for the query (Q1-Q2). SUM, MIN

and MAX similarly satisfy the OSP, and can be addressed by our technique. AVG, another common aggregate, can be broken down into two aggregates SUM and COUNT which have the optimal substructure property in turn, and therefore AVG can also be addressed by our technique. STDDEV, on the other hand, does not satisfy the OSP because even if the STDDEV for Q2’s results are known, the results of Q2 must be re-analyzed to compute STDDEV for Q1.

## 2.7 Problem Definition

Given a query  $Q$  and a desired aggregate value  $A_{exp}$ , the problem of **Aggregation Constrained Query Execution** consists of refining  $Q$  to produce alternate queries  $Q'$  that produce the aggregate value  $A_{exp}$  while changing  $Q$  as little as possible. Formally, we can state it as follows:

**DEFINITION 1.** Given database  $\mathcal{D}$ , query  $Q$ , desired aggregate value  $A_{exp}$ , an aggregate error threshold  $\delta$ , and refinement threshold  $\gamma$ , ACQ finds a set of refined queries  $Q'$  s.t. (a) the actual aggregate value for  $Q'$ ,  $A_{actual}$ , satisfies:  $Err_A \leq \delta$ , and (b)  $\|QScore(Q, Q_i) - QScore_{opt}\| \leq \gamma$ , where  $Q_i \in Q'$ ,  $QScore_{opt} = \min \{QScore(Q, Q'_j) \mid \forall \text{ valid query refinements } Q'_j \text{ s.t. } (Err_A \leq \delta)\}$ .

Since the problem of attaining the required aggregate value is NP-hard, we cannot provide formal guarantees about constraint (a) in Definition 1. However, as demonstrated in our experiments (Section 8), our algorithm ensures that the constraint is met practically every time. Our proximity-driven refinement technique guarantees that ACQUIRE will always meet constraint (b) in Definition 1.

We now turn our attention to evaluating these ACQs. As noted in the introduction, in this paper, our major focus is on queries that undershoot the aggregate constraint, however, we show in Section 7 how ACQUIRE can be extended to handle queries that overshoot the constraint. Furthermore, although we use COUNT as the aggregate of choice for all discussions, it is straightforward to support other aggregates with our technique and we note any changes to the algorithm that are required for doing so.

## 3. ACQUIRE: AN OVERVIEW

Given a query, the desired aggregate value, and acceptable result thresholds, ACQUIRE produces a set of refined queries that minimize changes to the original query but also satisfy the aggregate constraint. In formulating this set of refined queries, ACQUIRE adopts the strategy of *Expand and Explore* to iteratively *expand* the original query and to *explore* refined queries with respect to aggregate values. The *expand* phase and that queries with smaller refinements are produced before those with larger refinements. Thus, once ACQUIRE finds a query satisfying the aggregate constraint, it need not examine queries with larger refinements. The *explore* phase on the other hand efficiently computes aggregate values for refined queries via an incremental aggregate computation algorithm. We delegate all actual query execution tasks to an *evaluation layer*, which in this case is Postgres. However, the evaluation layer is modular and can be replaced with other techniques such as estimation, and/or sampling. Our incremental aggregate computation algorithm exploits dependencies between refined queries and the optimal substructure property so that for each query, ACQUIRE must only execute a small sub-query and then simply use our recursive model to combine results from previous queries. Together, these two techniques ensure that once a query  $Q$  has been executed, any query  $Q'$  that *contains*  $Q$  will not have to re-execute  $Q$ . As a result, ACQUIRE can evaluate a large number of refined

queries at a cost that is a fraction of the execution time for a single query. Figure 2 shows the system architecture described above.

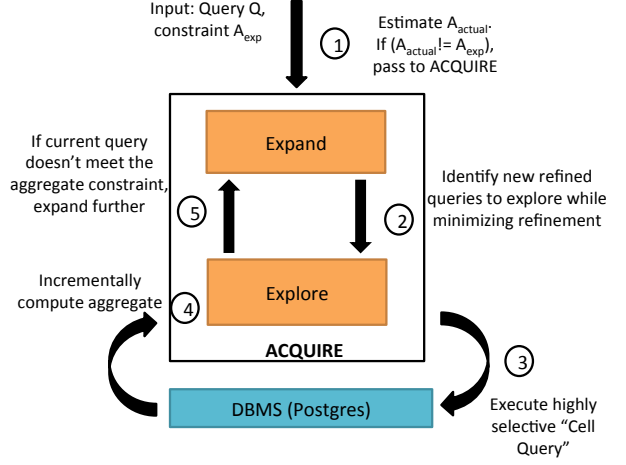


Figure 2: System Architecture of ACQUIRE

## 4. PHASE I: EXPAND

As described in the previous section, the *Expand* phase of ACQUIRE is responsible for iteratively generating refined queries that meet two criteria: (1) they satisfy the proximity threshold, and (2) their refinement scores ( $QScore$  values) are greater or equal to the scores of previously generated queries.

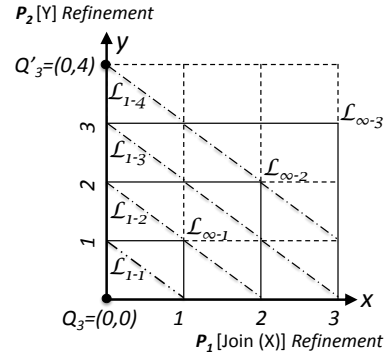


Figure 3: Refined Space and Generation of Refined Queries

To meet the above query generation goals, ACQUIRE uses an abstraction called the **Refined Space** to represent all refined queries. Given an original query  $Q$  having  $d$  predicates, the *Refined Space*, denoted henceforth by  $RS(Q)$ , is a  $d$ -dimensional space, where the origin represents  $Q$  and the axes measure individual predicate refinement. To illustrate, consider a refined query  $Q'$  and assume that the  $L_1$  norm is used to compute  $QScore$ .  $Q'$  would then be represented in  $RS(Q)$  as  $(u_1, u_2, \dots, u_d)$  where  $u_i = (PScore_i(Q, Q')) \forall i = 1, \dots, d$ , making  $QScore(Q, Q') = (\sum_{i=1}^d u_i)$ . Conversely, every point in the refined space  $(u_1, u_2, \dots, u_d)$  corresponds to some query  $Q'$  with  $PScore_i(Q, Q') = u_i$ . Therefore, any  $d$ -dimensional hyper-rectangle on  $RS(Q)$  also corresponds to a query. ACQUIRE divides  $RS(Q)$  into a multi-dimensional grid with step-size  $\frac{\gamma}{d}$  to avoid an exhaustive search of  $RS(Q)$  and to stay within the proximity threshold, as illustrated by Theorem 1. Each query on the multi-dimensional grid is called a *grid query*.

**Theorem 1.** Suppose the original query is  $Q$  and  $Q_{opt}$  is the optimal query meeting the aggregate constraint and having minimum refinement. Let  $RS(Q)$  be a multi-dimensional grid with step-size on each axis equal to  $\frac{\gamma}{d}$ . Then at least one refined query  $Q'$  lying on the  $RS(Q)$  grid will satisfy the proximity constraint w.r.t. to  $Q_{opt}$ .

*Proof:* Let  $Q_{opt} = \{u_1, u_2, \dots, u_d\}$  lie in some grid cell  $G$  in  $RS(Q)$ . Since the refined space grid has step-size  $\frac{\gamma}{d}$ , any query  $Q' = \{u'_1, u'_2, \dots, u'_d\}$  on  $G$  satisfies:

$$|u_1^p - u_1'^p| + |u_2^p - u_2'^p| + \dots + |u_d^p - u_d'^p| \leq \frac{\gamma}{d} \cdot d = \gamma$$

$$\Rightarrow |(u_1^p + u_2^p + \dots + u_d^p) - (u_1'^p + u_2'^p + \dots + u_d'^p)| \leq \gamma$$

$$\Rightarrow |QScore(Q_{opt}, Q)^p - QScore(Q', Q)^p| \leq \gamma$$

$$\Rightarrow QScore(Q_{opt}, Q)^p - QScore(Q', Q)^p \leq \gamma$$

(assume  $QScore(Q_{opt}, Q)^p \geq QScore(Q', Q)^p$ )

$$\Rightarrow (QScore(Q_{opt}, Q) - QScore(Q', Q)) \cdot (QScore(Q_{opt}, Q)^{p-1} + QScore(Q_{opt}, Q)^{p-2} \cdot QScore(Q', Q) + \dots + QScore(Q', Q)^{p-1}) \leq \gamma$$

$$\Rightarrow (QScore(Q_{opt}, Q) - QScore(Q', Q)) \leq \gamma (\gamma > 1) \quad \blacksquare$$

Figure 3 depicts the refined space abstraction for query Q3 assuming  $\gamma = 10$ . Since Q3 has two predicates, step-size=5 and  $RS(Q3)$  is a 2-dimensional space with the axes respectively measuring the refinements along the select and join predicates. A refined query like Q3' having  $PScore(Q3, Q3') = (0, 20)$  is represented as (0, 4) in  $RS(Q3)$ .

The second goal of the *Expand* phase is to generate refined queries in order of increasing refinement. ACQUIRE achieves this goal by producing queries close to the origin in  $RS(Q)$  before those far from it. In particular, the *Expand* phase uses breadth-first search to generate refined queries in layers where queries in a given *query-layer* have the same  $QScore$ . Consequently, for all  $L_p$  norms except  $L_\infty$ , query-layers take the form of d-dimensional planes corresponding to  $QScore = k \Rightarrow QScore^p = k^p \Rightarrow (\sum_{i=1}^d u_i^p) = k^p$ . For  $L_\infty$ , however, query-layers are L-shaped and intersect each axis at  $k^p$ . Figure 3 shows query-layers for Q3 assuming the  $L_1$  and  $L_\infty$  norms. Beginning with the query-layer with refinement 0, ACQUIRE generates all grid queries in the current query-layer. If no query from the current layer satisfies the aggregate constraint, ACQUIRE proceeds to the next query-layer having  $QScore$  increased by  $\frac{\gamma}{d}$ . Since this iterative expansion model examines queries in order of increasing refinement, ACQUIRE can stop immediately after a query is found to meet the required constraint, thus reducing the number of queries examined by ACQUIRE. Algorithms 1 and 2 respectively describe the pseudo code for generating queries using the  $L_p$  and  $L_\infty$  norms. The  $L_p$  algorithm generates query-layers using a breadth-first search while the  $L_\infty$  norm sequentially enumerates queries in the given layer.

---

**Algorithm 1** GetNextQuery(Queue queryQue)

---

```

1: int[] Qcurr = queryQue.Pop() //Indexed from 1
2: for i = 1, ..., d do
3:   Qnext ← GetNextNeighbor(i) //Increment i-th dimension
   of Qcurr by stepsize
4:   if (!queryQue.Contains(Qnext)) then
5:     queryQue.Push(Qnext)
6: return Qcurr

```

---



---

**Algorithm 2** GetNextQuery(Queue queryQue, int currRef)

---

```

1: if (!queryQueue.Empty()) then
2:   return queryQueue.Pop()
3: else
4:   Query Qnew = 0
5:   for i = 1, ..., d do
6:     Qnew[i] = currRef; queryQueue.Push(Qnew)
7:   while Qnew != null do
8:     IncrementQuery(Qnew, i, currRef) // enumerate
   queries with i-th dim fixed at currRef and others < currRef
9:     queryQueue.Push(Qnew)

```

---

**Theorem 2.** A grid query  $Q'_i$  with  $QScore(Q, Q'_i) = k$  is investigated after all grid queries with  $QScore(Q, Q'_i) = (k - 1)$  have been investigated.

*Proof:* Consider the refined space to be a directed graph with the origin as the root and every grid query as a node. Every grid query is connected to  $d$  queries obtained by incrementing one dimension by the unit step-size. These connections form the graph's edges. Then *GetNextQuery* for the  $L_p$  norm performs a breadth-first search on the refined space grid, guaranteeing that all queries at distance  $k - 1$  from the root are investigated before those at distance  $k$ . The result is trivially true for  $L_\infty$  norm since our algorithm explicitly generates queries in each query layer.  $\blacksquare$

**Time Complexity.** The worst case complexity of the *Expand* phase is  $O(V + E)$  where  $V$  is maximum number of refined queries in the grid and  $|E| = d \cdot |V|$ .

## 5. PHASE II: EXPLORE

The *Explore* phase of ACQUIRE is responsible for efficiently computing the aggregate values of queries produced in the *Expand* phase. For this purpose, we introduce a light-weight query execution methodology based on a novel, efficient incremental query execution algorithm that exploits dependencies between refined queries using a specialized recursive model. For each query, our model requires execution of only one sub-query and computes the overall aggregate by intelligently combining partial results from previous queries. ACQUIRE guarantees that a query is executed at most once, irrespective of how many queries contains it.

### 5.1 Incremental Aggregate Computation

The principle underlying our query execution algorithm is that refined queries often share results. Therefore, once a query result has been evaluated it must never be re-evaluated for any other query.

*Query Containment.* A refined query  $Q' = (u'_1, u'_2, \dots, u'_d)$  is said to be *contained* within another refined query  $Q'' = (u''_1, u''_2, \dots, u''_d)$  if  $(u'_i \leq u''_i) \forall i = 1 \dots d$ .

**Theorem 3.** If refined query  $Q'$  is contained within refined query  $Q''$ : (1) all results of  $Q'$  also satisfies  $Q''$ . (2)  $Q'$  is guaranteed to be generated before  $Q''$  in the *Expand* phase.

*Proof:* Let tuple  $\tau$  satisfy  $Q'$ . (1) By Equation 2:  
 $PScore_i(\tau, Q) \leq PScore_i(Q', Q) \forall i = 1, \dots, d$   
 $\Rightarrow PScore_i(\tau, Q)^p \leq PScore_i(Q', Q)^p = u_i^p \forall i = 1, \dots, d$   
 $(PScore \geq 0)$   
 $\Rightarrow PScore_i(\tau, Q)^p \leq u_i^p$   
 $\Rightarrow PScore_i(\tau, Q) \leq PScore_i(Q'', Q)$ .

Consequently, all the query results of  $Q'$  also satisfy  $Q''$ . For

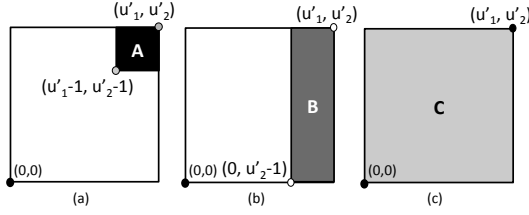


Figure 4: Sub-queries of a 2-D query

(2), from the definition of contained queries,  $QScore(Q', Q) \leq QScore(Q'', Q)$ . Therefore, by Theorem 2, the *Expand* phase will produce  $Q'$  before  $Q''$ . ■

Since all contained queries are produced and executed before those containing them, ACQUIRE can extensively use previously-generated query results. In particular, ACQUIRE exploits the concept of query containment by constructing contained queries, called *sub-queries* henceforth, that are used as units of query execution and result sharing. We now describe the sub-queries used.

### 5.1.1 Query Decomposition

Consider query  $Q'$  with  $d$  predicates, represented as point  $(u'_1, \dots, u'_d)$  in the refined space. In addition to  $Q'$ , ACQUIRE defines  $d$  specialized sub-queries contained within it, giving  $d + 1$  queries in all. Figure 4 shows these queries for a 2-predicate query. The first sub-query (A) corresponds to the unit square in  $RS(Q)$  with its upper-right corner at  $Q'=(u'_1, u'_2)$ , the second sub-query (B) corresponds to a unit-width rectangle in  $RS(Q)$  with  $Q'$  at its upper-right corner, and the third sub-query is the entire query (C). Similarly, for a 3-predicate query as in Figure 5, the first sub-query (A) is the unit cube, the second (B) is a unit length and width parallelepiped, the third (C) is a unit width parallelepiped, and the fourth (D) is the entire query sub-query. For ease of exposition, we refer to the first sub-query as **cell**, the second as **pillar**, the third as **wall**, and the fourth as **block**, respectively.

In a  $d$ -dimensional refined space, the  $d + 1$  sub-queries, called  $O_1, O_2, \dots, O_{d+1}$ , can be formally defined as shown in Equations 5-8. All  $d + 1$  sub-queries have the same upper bound ( $Q' = (u'_1, \dots, u'_d)$ ), but different lower bounds. For instance, the cell sub-query  $O_1$  has a lower bound which is a unit length away from  $(u'_1, \dots, u'_d)$  on all dimensions (Equation 5). The cell sub-query corresponds to the cell in the refined space grid having  $(u'_1, \dots, u'_d)$  as its upper bound. Similarly, the pillar sub-query has a lower bound with the first dimension equal to 0 and all remaining dimensions  $j$  ( $j = 2, \dots, d$ ) unit length away from  $u'_j$  (Equation 6). In general, the lower bound of the  $j^{th}$  sub-query  $O_j$  is  $(0, \dots, 0, u'_j - 1, \dots, u'_d - 1)$ . For simplicity, we will refer to an sub-query  $O_i$  corresponding to query  $(u'_1, \dots, u'_d)$  as  $O_i(u'_1, \dots, u'_d)$ .

$$O_1 = ((u'_1 - 1, \dots, u'_d - 1), (u'_1, \dots, u'_d)) \quad (5)$$

$$O_2 = ((0, u'_2 - 1, \dots, u'_d - 1), (u'_1, \dots, u'_d)) \quad (6)$$

$$O_j = ((0, 0, \dots, 0, u'_j - 1, \dots, u'_d - 1), (u'_1, \dots, u'_d)) \quad (7)$$

$$O_{d+1} = ((0, \dots, 0), (u'_1, \dots, u'_d)) \quad (8)$$

By decomposing a query into the sub-queries defined above, we can reuse previously obtained results. To illustrate, consider Figure 6.a where the 2-D query is decomposed into 3 sub-queries. We observe that sub-query A is the *Cell* $(u'_1, u'_2)$ , B is the *Pillar* $(u'_1, u'_2 - 1)$ , and C is the *Wall* $(u'_1 - 1, u'_2)$ . Similarly, Figure 6.b shows the decomposition of a 3-predicate query into the four sub-queries A, B, C and D which are respectively the *Cell* $(u'_1, u'_2, u'_3)$ , the

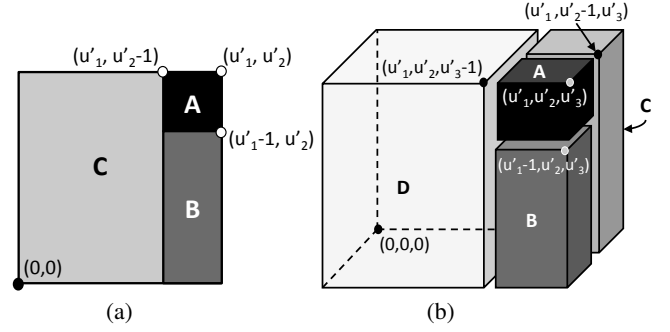


Figure 6: Query Decomposition: (a) 2-D (b) 3-D

*Pillar* $(u'_1 - 1, u'_2, u'_3)$ , the *Wall* $(u'_1, u'_2 - 1, u'_3)$ , and the *Block* $(u'_1, u'_2, u'_3 - 1)$ . In general, a  $d$ -predicate query can be decomposed into the previously defined  $(d + 1)$  sub-queries:

**2 – predicate Query :** (9)

$$O_3(u'_1, u'_2) = O_1(u'_1, u'_2) + O_2(u'_1 - 1, u'_2) + O_3(u'_1, u'_2 - 1)$$

**3 – predicate Query :** (10)

$$O_4(u'_1, u'_2, u'_3) = O_1(u'_1, u'_2, u'_3) + O_2(u'_1 - 1, u'_2, u'_3) + O_3(u'_1, u'_2 - 1, u'_3) + O_4(u'_1, u'_2, u'_3 - 1)$$

**d – predicate Query :** (11)

$$O_{d+1}(u'_1, u'_2, \dots, u'_d) = O_1(u'_1, u'_2, \dots, u'_d) + O_2(u'_1 - 1, u'_2, \dots, u'_d) + O_3(u'_1, u'_2 - 1, u'_3, \dots, u'_d) + \dots + O_{d+1}(u'_1, u'_2, \dots, u'_d - 1)$$

Thus, if the aggregates for the  $(d + 1)$  sub-queries have been pre-computed, the aggregate of query  $Q'$  is the mere addition<sup>1</sup> of these sub-aggregates. We must store only the aggregate *values* for the  $d + 1$  sub-queries. The corresponding result tuples can either be stored in main memory or paged to disk. The above sub-query decomposition also leads to two crucial observations: (1) **The only part of a query unique to itself is the cell**; all remaining parts of the sub-query are shared with other queries. (2) **The  $d + 1$  sub-queries defined above belong to queries completely contained in  $Q'$** . Therefore, Theorem 3 guarantees that these queries would have been produced and hence executed before investigating  $Q'$ . As a consequence, ACQUIRE must only execute the cell sub-query and can directly reuse aggregates of the remaining sub-queries.

### 5.1.2 Recursive Aggregate Computation

Query decomposition assumes that the aggregates for the  $d + 1$  sub-queries have already been computed. But independently determining aggregates of these sub-queries is redundant. Instead, we present a recursive strategy to calculate the aggregates of the sub-queries in constant time. Reconsider Figure 6 focusing now on the relationship between sub-queries. We observe that for 2-predicate sub-queries (Figure 6.a) the *Pillar* $(u'_1, u'_2)$  is equivalent to *Cell* $(u'_1, u'_2)$  and *Pillar* $(u'_1-1, u'_2)$  combined. Similarly, the *Wall* $(u'_1, u'_2)$ , which is the entire query is equal to the sum of *Pillar* $(u'_1, u'_2)$  and *Wall* $(u'_1, u'_2 - 1)$ . For the 3-predicate query, in Figure 6.b, we have three similar recurrences as shown below.

<sup>1</sup>For aggregates like MIN/MAX, addition is replaced by the corresponding MIN/MAX function, while AVERAGE = SUM/COUNT. SUM and COUNT aggregates are computed and stored separately. AVERAGE is computed from these values as required.

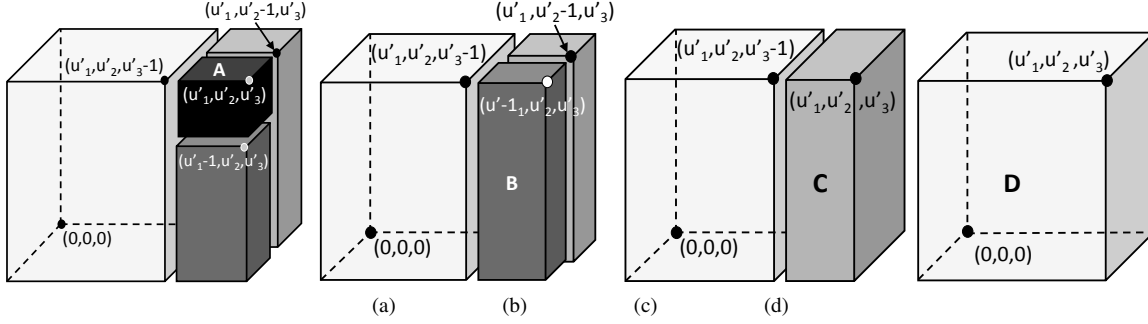


Figure 5: Sub-queries of a 3-predicate query

**2 – Recurrences :** (12)

$$Pillar(u'_1, u'_2) = Cell(u'_1, u'_2) + Pillar(u'_1 - 1, u'_2)$$

$$Wall(u'_1, u'_2) = Pillar(u'_1, u'_2) + Wall(u'_1, u'_2 - 1) \quad (13)$$

**3 – Recurrences :** (14)

$$Pillar(u'_1, u'_2, u'_3) = Cell(u'_1, u'_2, u'_3) + Pillar(u'_1 - 1, u'_2, u'_3)$$

$$Wall(u'_1, u'_2, u'_3) = Pillar(u'_1, u'_2, u'_3) + Wall(u'_1, u'_2 - 1, u'_3) \quad (15)$$

$$Block(u'_1, u'_2, u'_3) = Wall(u'_1, u'_2, u'_3) + Block(u'_1, u'_2, u'_3 - 1) \quad (16)$$

In general, this recursion for a d-predicate query is:

$$O_i(u'_1, \dots, u'_d) = O_{i-1}(u'_1, \dots, u'_d) + \quad (17)$$

$$O_i(u'_1, u'_2, \dots, u'_{i-1} - 1, \dots, u'_d) \text{ where } i = 2, \dots, d + 1$$

Since the sub-query  $O_1$  has no recurrences, its aggregate must be computed by executing the query. However, once the aggregate of  $O_1$  is determined, it takes  $d$  (constant) steps to calculate the total aggregate for query  $Q'$ .

### 5.1.3 Aggregate Computation Algorithm

Algorithm 3 takes as input the query  $Q'(u'_1, \dots, u'_d)$  being investigated and produces its aggregate. For this, Algorithm 3 first computes the aggregate of the  $Cell(u'_1, \dots, u'_d)$ , and then iteratively applies the recurrence in Equation 17 to compute aggregates of the remaining sub-queries. The function `ExecuteCellQuery` is used to compute the aggregate over a single input cell by issuing a query to the evaluation layer.

---

**Algorithm 3** `ComputeAggregate(Query  $Q_{curr}$ , int  $d$ )`

---

```

1: int[ $d + 1$ ]  $A_{curr}$  // All arrays are indexed from 1
2:  $A_{curr}[1] = \text{ExecuteCellQuery}(Q_{curr})$ 
3: for  $i = 2, \dots, d + 1$  do
4:    $Q_{prev} \leftarrow \text{GetPreviousNeighbour}(i-1)$  // decrement the  $(i - 1)^{th}$  dimension of  $Q_{curr}$  by stepsize
5:   int[]  $A_{prev} = \text{GetAllAggregates}(Q_{prev})$ 
6:    $A_{curr}[i] = A_{curr}[i - 1] + A_{prev}[i]$ 
7: StoreAllAggregates( $Q_{curr}$ ,  $A_{curr}$ )
8: return  $A_{curr}[d + 1]$ 

```

---

## 6. PUTTING IT ALL TOGETHER

Algorithm 4 presents the pseudo code for the ACQUIRE framework. Given an initial query  $Q$  and the refinement threshold  $\gamma$ , ACQUIRE begins to iteratively *Expand* and *Explore* refined queries,

starting at the origin of the refined space and sequentially traversing queries in subsequent layers. For each refined query, ACQUIRE calculates the aggregate using the Incremental Aggregate Computation technique described in Algorithm 3. Once the aggregate value  $A_{actual}$  has been determined, it is compared to  $A_{exp}$ . If the aggregate is within the error threshold  $\delta$ , the query is stored in the answer list ( $\mathcal{A}$ ). In this case, query search terminates with the exploration of all queries in the current layer, i.e., all alternate queries with the same refinement score. If all queries in a layer undershoot the constraint by more than  $\delta$ , ACQUIRE explores the next higher layer. Lastly, if any query overshoots the expected aggregate value by more than  $\delta$ , we repartition the cell corresponding to the given query and examine queries lying within. We repeat the repartitioning process for  $b$  iterations, where  $b$  is a tunable parameter. If, at the end of repartitioning, no query is found to satisfy the aggregate constraint, ACQUIRE returns the query attaining the closest aggregate value.

---

**Algorithm 4** `ACQUIRE(Query  $Q_{original}$ , double  $A_{exp}$ , int  $\delta$ , double  $\gamma$ )`

---

```

1:  $\mathcal{A} = []$  // Set of refined Queries
2: Queue  $queryQueue = []$  // Data structure for traversal
3:  $d \leftarrow$  Flexible predicates in  $Q_{original}$ 
4: int[ $d$ ]  $Q_{curr} = \{0, \dots, 0\}$  // Origin represents  $Q_{original}$ 
5:  $queryQueue.push(Q_{curr})$ 
6: int  $minRefLayer = \text{MAX\_INTEGER\_VALUE}$ 
7: int  $currRefLayer = 0$ 
8: while ( $currRefLayer \leq minRefLayer$ ) do
9:   double  $A_{actual} = \text{ComputeAggregate}(Q_{curr}, d)$  // Algorithm 3
10:  if ( $|A_{exp} - A_{actual}| \leq \delta$ ) then
11:     $\mathcal{A}.add(Q_{curr})$ 
12:     $minRefLayer = currRefLayer$ 
13:  else if ( $A_{exp} > A_{actual}$ ) then
14:     $\mathcal{A}.add(\text{Repartition}(Q_{curr}))$ 
15:     $Q_{curr} = \text{GetNextQuery}(queryQueue)$  // Algorithm 1
16:     $currRefLayer = \text{QScore}(Q_{curr})$ 
17: return  $\mathcal{A}$ 

```

---

## 7. EXTENSIONS

In this section, we present extensions to the framework that accommodates some of the limitations of our approach.

### 7.1 Preferences in Refinement

Along with the NOREFINE keyword used to identify and preserve rigid constraints, ACQUIRE allows users to set preferences

on which predicates should be refined. This can be easily done by specifying a  $L_{W_p}$  norm which sets appropriate weights on various predicates. Similarly, users can also supply maximum refinement limits on predicates. While we provide several avenues for user control, user intervention is not required and each tunable parameter is provided an appropriate default setting.

## 7.2 Contracting Queries With Too Many Results

ACQUIRE with minor modifications handles queries that generate *too many results*. This is achieved by constructing a query  $Q'_{min}$  with each predicate of the original query  $Q$  set to its minimum value. Since  $Q'_{min}$  will produce too few results, we can now construct a refined space bounded by  $Q$  and  $Q'_{min}$ . ACQUIRE now traverses the refined space to find queries that meet the cardinality constraint, this time minimizing refinement with respect to  $Q$  instead of  $Q'_{min}$ .

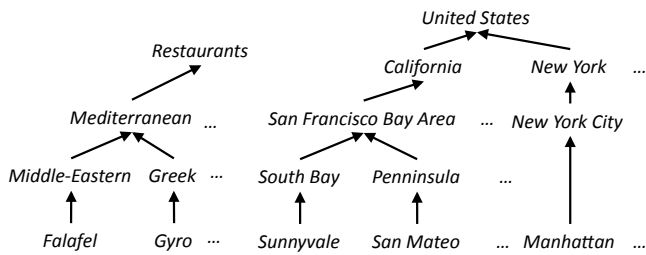


Figure 7: Ontology for Categorical Data

## 7.3 Non-numeric Predicates

The focus of this work is to handle numeric predicates. Measuring refinement distance between categorical data points is in itself a challenging problem, requiring the analysis of taxonomy information. However, ACQUIRE can be extended to support categorical predicates by plugging in the appropriate means for measuring the distance between any two categorical values. For example, Figure 7 depicts sample ontology trees related to *food* preferences and *location*. The refinement distance between the original query desiring places that serve *Gyro* to restaurants that have any *Mediterranean cuisine* may be defined based on the relative depths of the two nodes. In general, the roll-up operation on an ontology tree corresponds to making the predicate less selective, i.e., relaxation. While the drill down operation translates to query contraction. Given this meta-information from the ontology tree and a distance metric, the ACQUIRE framework can be used to refine categorical predicates.

## 7.4 Exploiting Indexes and Data Distribution

The algorithms discussed so far make no assumptions about the underlying data distribution or presence of indexes on the data. Moreover, experiments in Section 8 indicate that ACQUIRE is already 2 orders of magnitude faster than the state-of-the-art techniques. However, if required, we can further boost the efficiency of ACQUIRE by employing a specialized bitmap-like index structure on the tables. To construct this index, we divide each attribute dimension into equi-width parts and create a multi-dimensional grid on the table. We then examine the records in the table to determine which grid cell each record belongs to. In our index, each cell is assigned a corresponding bit, which is set to 1 if the cell contains some tuple and 0 otherwise (storing the number of tuples may be easier for keeping the index up-to-date but requires more space). Once constructed, this simple index structure can be used

in the *Explore* phase to determine if a given cell query is empty without actually executing the query. If the query is found to be empty, we can safely skip it and proceed to the next, thus avoiding unnecessary query execution costs.

## 8. EXPERIMENTAL EVALUATION

### 8.1 System Implementation

The ACQUIRE framework is built on top of Postgres. ACQUIRE sits outside the DBMS where it performs the tasks of exploring the refinement space, formulating queries and applying our aggregate computation algorithm. To make ACQUIRE portable across multiple database systems, and to aid in proper comparison with competing techniques, all query execution tasks are delegated to the DBMS. We similarly implemented the compared techniques on top of Postgres.

### 8.2 Alternative Techniques

We compare ACQUIRE to three extensions of existing techniques that address the ACQ problem to varying degrees. First, we compare it to Top-k which, although unable to produce *refined queries*, is suited to ranking tuples in order of refinement. While it is straightforward to translate a COUNT constraint to Top-k, translating other aggregate constraints (e.g. AVERAGE) is difficult if not impossible. As a result, we only study Top-k ranking for COUNT constraints. We use existing DBMS capabilities of ORDER BY and LIMIT to implement Top-k, as demonstrated on generic queries (Q and corresponding Top-k-Q) below.

```
Q = SELECT COUNT(*) from table1
    WHERE x <= 10 and y <= 20;
```

```
Top-k-Q = SELECT * FROM table1 ORDER BY
(case when (x <= 10) then 0
else (x - 10)/(x.max - x.min)) +
(case when (y <= 20) then 0
else (y - 20)/(y.max - y.min)) LIMIT A_exp
```

We also compare ACQUIRE to the TQGen [11] and a simple binary search (BinSearch) technique [11]. Our experiments uses the TQGen parameters reported in [11]. To allow for uniform comparisons across all methods, we do not employ sampling techniques for TQGen. However, our experiments demonstrate that our results hold even for small sample-size datasets (see Figure 10.a). The final point to note is that, unlike ACQUIRE, (a) none of the above techniques addresses aggregates other than COUNT, and (b) even for COUNT, none of the above techniques are capable of refining join predicates.

### 8.3 Methodology

To study the robustness of ACQUIRE we vary (1) *dimensionality* of refinement space, i.e., number of refinable predicates, and combination of attributes in these predicates, (2) *magnitude of aggregate value discrepancy*, i.e., ratio  $\mathcal{A}_{actual}/\mathcal{A}_{exp}$  between the actual aggregate value and the desired aggregate value, (3) *dataset size*, (4) *aggregate types*, and (5) *data distributions*. To study the efficiency gained by the ACQUIRE system, we evaluated the net decrease in query execution time for various data sizes and dimensionality. Finally, we evaluated the performance of ACQUIRE under various settings of refinement and aggregate thresholds as well as presence of join refinement. For each experimental setting, we measure the time needed to return the set of refined queries,  $Q_F$ , amount of refinement (refinement score), and relative aggregate error =  $Err_A$ .



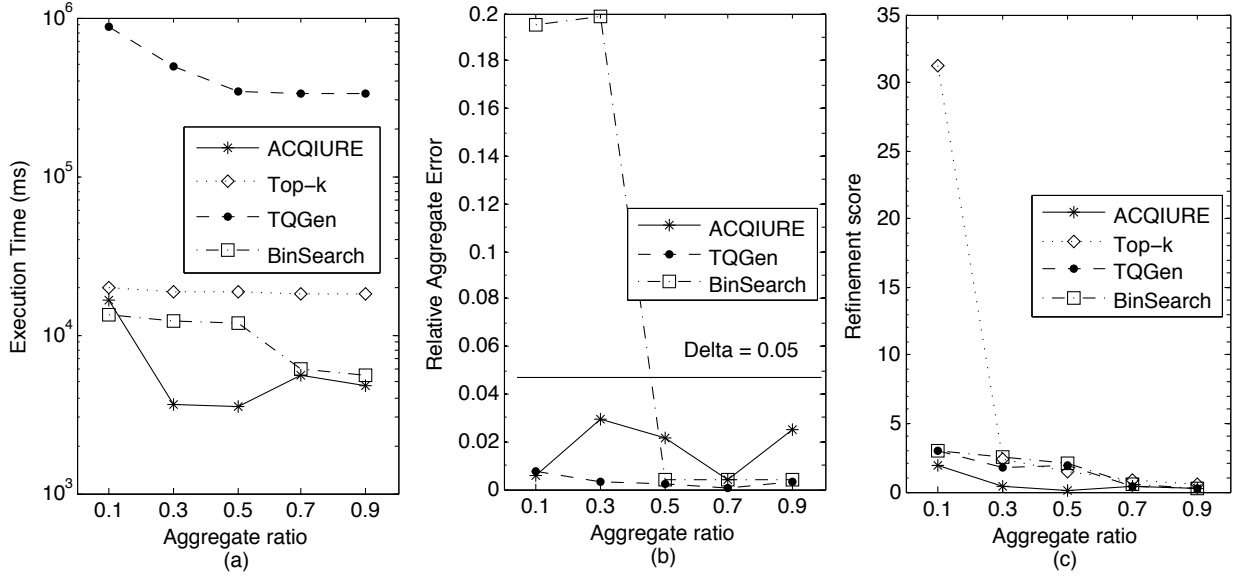


Figure 8: Performance Comparison Under Varying Aggregate Ratios: ACQUIRE, Top-k, TQGen and BinSearch

All algorithms were implemented in Java. Measurements were obtained on AMD 2.6GHz Dual Core CPUs, and Java heap of 2GB. We utilized the TPC-H datasets of varying sizes (1K - 10M tuples). Since the standard TPC-H data is uniformly distributed (i.e.,  $Z = 0$ ), we used [3] to also generate skewed data with  $Z = 1$ . Our test queries are TPC-H queries which have been adapted to include only numeric range and join predicates. Query Q2 (in Example 2) provides skeleton query that was used to evaluate the SUM aggregate. For each dataset, query, and ACQUIRE settings, we define the original aggregate  $A_{actual}$  and the aggregate ratio  $\frac{A_{actual}}{A_{exp}}$ .

## 8.4 Performance Comparisons

### 8.4.1 Effect of $\frac{A_{actual}}{A_{exp}}$ Ratio

We first examine the effect of aggregate ratio,  $\frac{A_{actual}}{A_{exp}}$ , on the execution time, error rate and refinement scores. A small  $\frac{A_{actual}}{A_{exp}}$  ratio implies that the original query is highly selective and needs large refinements, while a large  $\frac{A_{actual}}{A_{exp}}$  implies that the original query is close to the desired query and needs only small refinements. These experiments were carried out on a 1 million tuple dataset and a query with 3 flexible predicates. The aggregate ratio was varied between 0.1 - 0.9.

As shown in Figure 8.a, the execution time for ACQUIRE increases with decreasing expansion ratio, i.e., the greater the need to expand the query, longer it takes for ACQUIRE to reach the required aggregate ratio. While Top-k requires the same execution time (the ranking function is unchanged and all records need to be sorted), its execution time however is on average 3.7X more than ACQUIRE. TQGen and BinSearch both need to explore the same number of queries each time and hence their execution time remains constant. ACQUIRE does consistently as well as all the other methods, and is on average 2X faster than BinSearch and 2 orders of magnitude faster than TQGen (Y-axis is in log scale). Although BinSearch shows promise with respect to execution time, we show next that it is not robust with respect to aggregate errors.

Figure 8.b shows the relative error (average relative error for BinSearch) for each of the queries with changing aggregate ratio. We do not compare Top-k because a Top-k query explicitly specifies

the number of tuples to return and hence has no aggregate error by definition. The BinSearch line in the graph shows that BinSearch is extremely unstable and has high variance in aggregate errors. The underlying reason is that BinSearch is very sensitive to the order in which predicates are refined; even a single change to the order can change the error by a factor of 100. To illustrate, one ordering of predicate refinement in BinSearch produces a refinement error of 0.19 or 20% whereas another ordering produces an error of 0.002 or 0.2%. Attempting to refine the query by attempting all orderings of predicates is computationally expensive. ACQUIRE, on the other hand, not only produces queries consistently within the threshold ( $\delta = 0.05$ ), but also does so efficiently. TQGen, in fact, produces lower error rates than ACQUIRE. However, this reduction comes at the cost of a 100X increase in execution time. Since both error rates are acceptable, we prefer ACQUIRE. Lastly, in Figure 8.c we compare the refinement scores obtained by each method. We see that the refinement score for queries generated by other methods are 2-3X larger than those from ACQUIRE.

### 8.4.2 Effects of Dimensionality

Next, we discuss the effects of increasing dimensionality, i.e. increase in the number of query predicates. We used the same dataset as before, used expansion ratio = 0.3 and varied the number of predicates in the query. In execution time, we see the same trend as before where the execution time increases with increasing dimensionality of the query. However, for ACQUIRE, the increase is largely linear and not exponential. For Top-k, the execution time remains largely constant since only the ranking function changes. For TQGen, we see an exponential increase in the execution time (as number of queries executed is exponential in number of dimensions) with the method taking 500X more time than ACQUIRE for high dimensional queries. Thus, ACQUIRE is a much better alternative to the state-of-the-art on queries of varying dimensions. Figure 9.b once again demonstrates that BinSearch is extremely unstable with respect to aggregate error. While some queries obtain an error rate of 0.6%, some obtain an unacceptable error rate of 45%. This large variance in error values produced by BinSearch indicates that the method is unpredictable and not-robust. As a result, it cannot guarantee any threshold on the error rate.

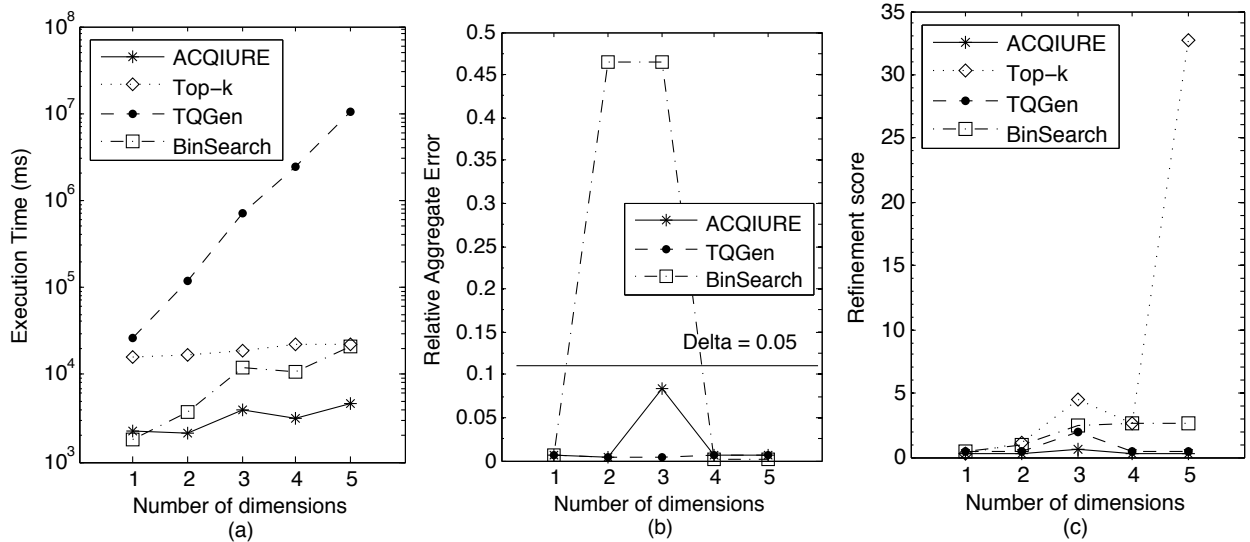


Figure 9: Performance Comparison Under Varying Number of Predicates Ratios: ACQUIRE vs. Top-k vs. TQGen vs. BinSearch

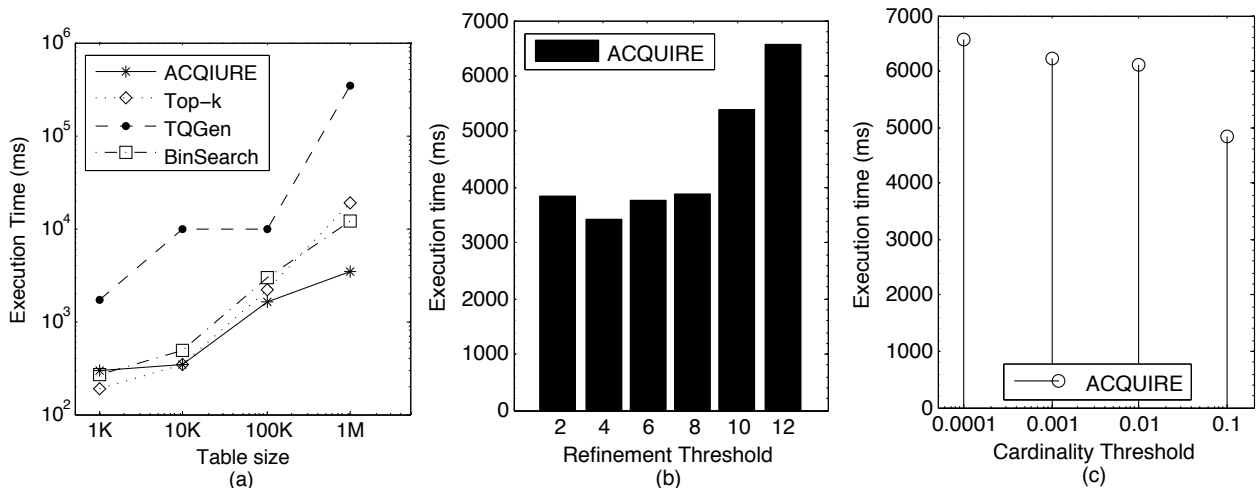


Figure 10: Performance Comparison Under Varying (a) Table Size, (b) Refinement Threshold and (c) Cardinality Threshold

In Figure 8.c exemplifies the trends in query refinement score seen with all methods. The refinement scores of ACQUIRE are consistently the lowest across all methods – meaning fewer changes to the original user query and therefore more desirable. Top-k produces higher refinement than ACQUIRE. This figure also shows that TQGen and BinSearch can have high variance in refinement scores. Since the goal of these techniques is only to meet the aggregate constraint and not to minimize refinement, this is expected. BinSearch queries have, on average, 4.8X more refinement than ACQUIRE queries.

### 8.4.3 Varying Table Size

For datasets of varying size, beginning with a 1k-tuple dataset (to mimic a sample based approach) to a 1M-tuple dataset. As shown in Figure 10.a the execution time for ACQUIRE and all compared techniques increases proportionally to the size of the dataset. Relative error and refinement scores show the same trends as before.

### 8.4.4 Effect of Varying Data Distributions

To study the robustness of our method, we re-ran experiments on data with Zipfian skew = 1. Trends in results were same as above.

### 8.4.5 ACQUIRE Parameter Studies

In Figure 10.a and Figure 10.c, we report the performance of ACQUIRE with respect to its internal parameters, namely the aggregate threshold, the number of steps in the grid and the depth of the search. As expected, a stringent cardinality and refinement threshold produces proportional increases in the ACQUIRE execution time as more queries need to be explored.

### 8.4.6 Varying Aggregate Types

ACQUIRE framework is general and can be applied to different types of aggregates satisfying the optimal substructure from Section 2.6. We tested the technique for other aggregates too. Figure 11 shows the results for the SUM, COUNT and MAX aggregates. We omit MIN since this can be written as the MAX(-1 \*

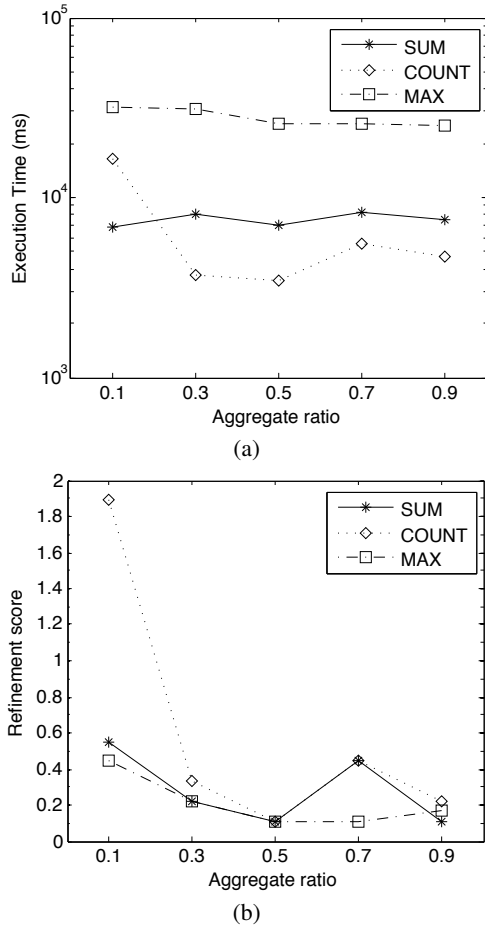


Figure 11: ACQUIRE’s Performance on Different Aggregates

attribute). We find that ACQUIRE successfully minimizes refinement and reaches the aggregate thresholds in all the above aggregates.

## 8.5 Summary of Experimental Conclusions

1. ACQUIRE is consistently 2 orders of magnitude faster than TQGen and on average 2X faster than *BinSearch*.
2. In all experimental conditions, ACQUIRE’s aggregate error is well below the aggregate error threshold. In contrast, *BinSearch* has very high variance in error rates, reaching up to 45% of the expected aggregate value.
3. Although, Top-k can be efficient at small-sized datasets, it quickly becomes inefficient as data size increases. In general Top-k is about 3.7 times slower than ACQUIRE.
4. ACQUIRE generates queries that on average have 2X better refinement scores than a query produced by either TQGen or *BinSearch*.

## 9. RELATED WORK

In this section, we discuss two areas of related work namely, (1) *set-based queries* [6, 5, 16, 14, 13] and (2) solving the *empty result problem* [9, 1, 11]. Existing set-based query evaluation techniques

differ from our work fundamentally because they are solving a different problem than the one addressed in this work. For instance, techniques proposed in [13] address the problem of recommending “satellite items” (e.g., car charger, case) for a given item that the customer is currently shopping for (e.g. smart phones). Alternatively, [16, 5] solve the generalized Knapsack problem [6] of making composite recommendations of a set of items. That is, recommend the Top-k sets of items with the total cost below a given budget and preferring the set with higher ratings. In contrast, [14] focused on finding users (e.g. tourists) sets of results (e.g. a set of places of interest) given a set of constraints (e.g. budget). This is identical to the current behavior of the Facebook Ad Creation Interface [4]. However, this approach is less than desirable (as described in Section 1) as it would force Alice to go through hundreds of iterations to find a meaningful query that meet the aggregate constraints. To summarize, techniques for set-based queries focus on returning tuples or sets of tuples that meet a constraint. In large scale database systems since the users are mostly unfamiliar with the characteristics of the underlying data, they usually construct queries that are either too strict or too broad [9]. In such scenarios, execution techniques designed for set-based queries could potentially return no results or all tuples in the database.

To the best of our knowledge, we are the first work to address the question of recommending refined user queries that meets their aggregate constraints. Existing query refinement techniques can be classified into two categories namely, (1) *tuple-oriented approaches*, and (2) *query-oriented approaches*. Table I summarizes the key related work, and whether they support all aggregate constraints and / or a proximity criteria.

Techniques	Aggregates Supported	Proximity	Card.	Query
Tuple-Oriented: Skyline [8], Top-k [2],	COUNT	✓	✓	
Query-Oriented: BinSearch [11], IQR [10]	COUNT			✓
Query-Oriented: TQGen [11], Hill-Climbing [1]	COUNT		✓	✓
ACQUIRE	COUNT, SUM, MIN, MAX, AVG, UDA <sup>2</sup>	✓	✓	✓

Table 1: Summary of the Related Work

**Tuple-Oriented Techniques.** Result refinement techniques [12, 8] focus only on generating the required number of results and ignore the problem of generating refined queries that explain how the result tuples were selected. The refinement criteria are crucial in scientific and business applications. Similarly Top-k algorithms, such as [2], while useful in many instances cannot correctly address the ACQ problem since they can only handle COUNT aggregates. To illustrate, consider a query that selects patients based on income, blood pressure, and the amount of weekly exercise. A Top-k based approach will obtain the required number of patients, but these patients will likely be skewed in certain predicate dimensions and will not be representative of the population. Thus pure Top-k and its variations are inadequate to address the ACQ problem; clearly demonstrated in our experiments (see Section 8).

**Query-Oriented Approach.** More recently in the context of database testing [1, 11, 10] have started to focus on the problem of

<sup>2</sup>User Defined Aggregates that either satisfy the optimal substructure property (OSP) or can be broken into functions that satisfy OSP

generating refined predicates. [10] proposed a framework that iteratively narrows the bounds on each selection predicate in a query and asks the user to manually refine the predicate within the constrained dimensions. This approach however cannot be extended to support the refinement of join predicates as ACQUIRE does. For select-only queries, [11] seeks only to attain the desired cardinality and disregards proximity. Consequently, it cannot guarantee that the refined query has the least refinement. The BinSearch algorithm [11] is heavily influenced by the order in which predicates are refined; some orders produce accurate results whereas others produce large errors. Unlike ACQUIRE, these techniques don't generate a set of alternative refined queries for the user to choose from. To summarize, ACQUIRE is the first technique to refine select and join queries to meet the dual constraints of proximity to the original query and the desired aggregate constraint.

## 10. CONCLUSION

We introduce *Aggregation Constrained Queries* that constrain not only the tuples produced by the query, but also aggregates on these tuples. We argue that algorithms targeting ACQs must combine efficient query execution and query refinement. We propose ACQUIRE to tackle ACQs. ACQUIRE adopts the *Expand and Explore* strategy where it iteratively *expands* the original query to minimize refinement and efficiently *explores* refined queries via a novel incremental aggregate computation technique. The general principle of ACQUIRE allows us to support user defined predicate refinement scoring and aggregate error functions. ACQUIRE guarantees that each query is executed at most once, regardless of the number of queries it is contained within thereby exploiting work sharing. This enables ACQUIRE to consistently perform up to 2 orders of magnitude faster and produce queries with 2X smaller refinement than extensions to existing techniques.

## 11. REFERENCES

- [1] N. Bruno, S. Chaudhuri, and D. Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE TKDE*, 18(12):1721–1725, 2006.
- [2] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *VLDB*, pages 397–410, 1999.
- [3] S. Chaudhuri and V. Narasayya. Program for tpc-d data generation with skew.
- [4] V. Goel. How facebook sold you krill oil. *The New York Times*, August 2014.
- [5] S. Guha, D. Gunopulos, N. Koudas, D. Srivastava, and M. Vlachos. Efficient approximation of optimization queries under parametric aggregation constraints. In *VLDB*, pages 778–789, 2003.
- [6] H. Kellerer, U. Pfersch, and D. Pisinger. *Knapsack problems*. Springer, 2004.
- [7] J. Koliha. *Metrics, Norms and Integrals: An Introduction to Contemporary Analysis*. World Scientific Publishing Company, 2008.
- [8] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.
- [9] G. Luo. Efficient detection of empty-result queries. In *VLDB*, pages 1015–1025, 2006.
- [10] C. Mishra and N. Koudas. Interactive query refinement. In *EDBT*, pages 862–873, 2009.
- [11] C. Mishra, N. Koudas, and C. Zuzarte. Generating targeted queries for database testing. In *SIGMOD*, pages 499–510, 2008.
- [12] I. Muslea. Online query relaxation. In *SIGKDD*, pages 246–255, 2004.
- [13] S. B. Roy, S. Amer-Yahia, A. Chawla, G. Das, and C. Yu. Constructing and exploring composite items. In *SIGMOD*, pages 843–854, 2010.
- [14] Q. T. Tran, C.-Y. Chan, and G. Wang. Evaluation of set-based queries with aggregation constraints. In *CIKM*, pages 1495–1504, 2011.
- [15] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8):553–564, 2013.
- [16] M. Xie, L. V. S. Lakshmanan, and P. T. Wood. Breaking out of the box of recommendations: from items to packages. In *RecSys*, pages 151–158, 2010.

# Reverse Engineering Top-k Database Queries with PALEO\*

Kiril Panev  
 TU Kaiserslautern  
 Kaiserslautern, Germany  
 panev@cs.uni-kl.de

Sebastian Michel  
 TU Kaiserslautern  
 Kaiserslautern, Germany  
 smichel@cs.uni-kl.de

## ABSTRACT

Ranked lists are an essential methodology to succinctly summarize outstanding items, computed over database tables or crowdsourced in dedicated websites. In this work, we address the problem of reverse engineering top-k queries over a database, that is, given a relation  $R$  and a sample top-k result list, our approach, named PALEO<sup>1</sup>, aims at determining an SQL query that returns the provided input result when executed over  $R$ . The core problem consists of finding predicates of the where clause that return the given items, determining the correct ranking criteria, and to evaluate the most promising candidate queries first. To capture cases where only a sample of  $R$  is available or when  $R$  is different to the relation that indeed generated the input, we put forward a probabilistic model that allows assessing the chance of a query to output tuples that are resembling or are somewhat close to the input data. We further propose an iterative candidate query execution to further eliminate unpromising queries before being executed. We report on the results of a comprehensive performance evaluation using data and queries of the TPC-H and SSB [14] benchmarks.

## 1. INTRODUCTION

Reverse engineering database queries describes the task of obtaining an SQL query that is able to generate a specified input table, when executed over a given database instance. This generic problem has various important application scenarios, specifically for top-k database queries that often yield valuable analytical insights. Consider, for instance, business analysts who are interested in determining alternative queries that yield the same or similar query result tuples, data scientists who try to find explanatory SQL queries for crowd-sourced top-k rankings, or to find the data-generating query of a sample input in order to re-execute it on current or future database instances in cases

\*This work has been supported by the German Research Foundation (DFG) under grant MI 1794/1-1.

<sup>1</sup>PALEO is approximately the reverse of the word OLAP and also emphasizes the goal of assembling queries based on their data footprints (results), much like paleontologists reconstruct and study fossils.

Name	City	State	Plan	Month	Minutes	SMS	Data
John Smith	SF	CA	XL	June	654	87	1,230
John Smith	SF	CA	XL	July	175	22	900
...	...	...	...	...	...	...	...
Jane O'Neal	LA	CA	XL	April	699	15	2,300
Jane O'Neal	LA	CA	XL	June	334	10	1,900
...	...	...	...	...	...	...	...
Richard Fox	Oakland	CA	XL	June	596	23	1,272
...	...	...	...	...	...	...	...
Jack Stiles	San Jose	CA	XL	March	429	42	1,192
Jack Stiles	San Jose	CA	XL	April	586	8	1,275
...	...	...	...	...	...	...	...
Lara Ellis	San Diego	CA	XL	May	784	11	2,107

Table 1: Sample relation of telecommunications traffic data

where the original query has not been saved or has not been made public, for one or another reason. The discovered queries can reveal interesting properties of the input, most importantly the constraints to tuples expressed in the “where clause” of the query and how tuples are ranked. The last years have brought up various research results [17, 12, 19] on reverse engineering database queries. Compared to existing approaches that operate on input in form of full tables, reverse engineering top-k queries adds two complex ingredients to the re-engineering task. First, it is the rather small input, consisting of only a few (as  $k$  is usually quite short) ranked tuples and, second, the various ways top-k SQL queries can be formulated, given various sorting orders and aggregation functions.

Consider a relation *Traffic*, illustrated in Table 1, containing cellphone-traffic data. The relation contains textual attributes like name of the customer, the city and state the customer lives in, and the tariff plan and the month for which the traffic was realized. In addition, there are numerical attributes that measure the customer’s traffic, like number of minutes talked, the number of text messages (SMS) sent, and the number of spent megabytes of data.

Lara Ellis	784
Jane O'Neal	699
John Smith	654
Richard Fox	596
Jack Stiles	586

Table 2: Example input list

Table 2 shows a top-k list with two columns and five rows. The input list does not have attribute names (or if it does, are not correlated to the attribute names in the database table). The first attribute is the customer’s name, while the second is the performance attribute according to which the customer ranking was produced. Note that there are no empty cells in the list, all values are specified. Considering the *Traffic* relation of Table 1, we can see that the input ranking list can perhaps be generated using the following query:

```

SELECT name, max(minutes) FROM traffic
WHERE state = 'CA'
GROUP BY name ORDER BY max(minutes) DESC
LIMIT 5

```

This query computes the top 5 customers of the telecommunications company, living in the state of California, ranked by the number of minutes talked in a single month. In general, there can be several different queries that produce the same results; consider for instance augmenting the above query  $Q$  with an additional constraint to customers with the tariff plan “XL”, it would leave the result unchanged (including the order among tuples).

## 1.1 Problem Statement

Given a database  $D$  with a single relation  $R$  with schema  $\mathcal{R} = \{A_1, A_2, \dots\}$  and an input relation  $L$  that represents a ranked list of items with their values. **The task we consider in this paper is to efficiently and effectively determine queries  $Q_i$  that output tuples that resemble  $L$  when executed over  $R$ .**

We focus on top-k select-project queries over relation  $R$  of the form shown in Figure 1(left). We specifically focus on a single relation to emphasize on the intrinsic characteristics of top-k queries, instead of considering the reverse engineering of joins, too, which has been addressed by Zhang et al. [19] in their recent work on reverse engineering complex join queries.

```

SELECT id, agg(value)
FROM table
WHERE  $P_1$  and  $P_2$  and ...
GROUP BY id
ORDER BY agg(value) LIMIT k

```

$L$	
$L.e$	$L.v$
e	100
f	90
g	80
m	70
o	60

Figure 1: Query template (left) and example input  $L$  (right)

The problem has two properties that can be relaxed or tightened. First, it can either demand determining only one, multiple, or all input-generating queries. Second, the notion of a query being valid in the sense that it resembles the input can be relaxed to a notion of approximately resembling the input.

The problem is challenging for the following reasons: (i) The size of the input list is rather small, it is difficult to derive meaningful (statistical) properties in order to identify valid predicates and ranking criteria, (ii) the relevant subset of  $R$  that features all tuples of the entities in  $L$  can become very large, and (iii) false positive and false negative candidate queries deteriorate system performance due to many necessary query evaluations and limit the chance to successfully determine a valid query that generates the input.

The presented approach, coined PALEO, is not limited to finding exact matches, but can almost directly be applied to finding queries that compute a ranking  $L'$  over  $R$ , with  $L'$  being similar to  $L$ . We get back to this generalization in Section 3.3. We refer to the specific attribute in  $R$  that contains the entities the table reports on as  $A_e$  and assume it is known a priori.

As already indicated in the template query, we focus on predicates  $P$  of the form  $P_1 \wedge P_2 \cdots \wedge P_m$ , where  $P_i$  is an atomic equality predicate of the form  $A_i = v$  (e.g., state = “CA”). Furthermore, we denote with *size* of a predicate  $|P|$  the number of atomic predicates  $P_i$  in the conjunctive clause.

The input top-k list  $L$  has two columns;  $L.e$  and  $L.v$  denote the entity column and the numeric score column, respectively. Note that  $L$  does not contain the name of the

column  $L.v$  or the column name of  $L.v$  is named for human consumption (e.g., “Total traffic”, which can be total number of minutes, SMS, or data), i.e., not corresponding to the ones present in the database. Hence, referencing to the appropriate attribute in  $R$  cannot be done by name. Table 3 shows a summary of the most important notations used throughout this paper.

## 1.2 Sketch of the Approach

A naïve approach would enumerate all possible queries, say with a limited complexity of the predicate in the where clause, evaluate the queries one-by-one against the database and check whether the returned results resemble the input list. This is clearly beyond hope, even for relatively small databases and schemas.

Our approach, conceptually, loads all tuples from  $R$  that contain any of the entities in  $L$ . This table is called  $R'$  and is used in two subsequent steps, first, to determine the query predicate and, second, to find the right attribute(s) and aggregation function. In case  $R'$  is completely given, our approach is extremely effective in determining the individual building blocks of the desired query. When working on a subset of  $R'$ , we show how to handle large amounts of potential candidate queries by introducing a suitability-driven order among them, in order to find the desired query early.

## 1.3 Contributions and Outline

With this paper we make the following contributions:

- To the best of our knowledge, this work is the first to consider the problem of reverse engineering top-k OLAP queries. We present an efficient and effective solution to it, in a flexible and extensible framework.
- We show how to efficiently compute promising predicates using an apriori-style algorithm over  $R'$  and how to augment them with ranking criteria using data samples and statistics obtained from the base relation  $R$ .
- We present a probabilistic reasoning that allows ordering candidate queries by the likelihood that they compute the input ranking  $L$ . This, together with a method to skip unpromising queries dynamically at validation time, allows finding the desired valid queries very efficiently.
- We report on the results of a carefully conducted experimental evaluation using data and queries from the TPC-H [16] and SSB [14] benchmarks.

This paper is organized as follows. Section 2 discusses related work. Section 3 presents the framework and key ideas behind our approach, followed by the specific sub-problems of identifying query predicates in Section 4, and determining the ranking attributes and aggregation function, in Section 5. Section 6 considers handling changed data in  $R$ , and proposes a probabilistic model to rank queries by their expected suitability to generate the input. Section 7 introduces an incremental strategy to eliminate unpromising candidate queries based on observed results of already executed candidates. Section 8 reports on the results of the experimental evaluation and presents lessons learned. Section 9 concludes the paper.

## 2. RELATED WORK

The problem of reverse engineering queries was considered by Tran et al. [17] in their data-driven approach called *Query by Output* (QBO). Given a database  $D$  and a query output  $Q(D)$  produced by a query  $Q$ , they try to find an instance-equivalent query  $Q'$ . They focus on identifying the selection predicates in select-project-join queries and formulate this

$R$	Base table in the database
$A_i$	Attribute in $R$
$A_e$	Entity attribute in $R$
$L$	Top-k input list
$L.e$	Entity column in $L$
$L.v$	Ranking column in $L$
$e_i$	Entities in $A_e$ or $L.e$
$v$	Values in $A_i$
$P$	Predicate (atomic or conjunctive)
$Q$	Query
$Q(R)$	Result set of $Q$ when querying $R$

Table 3: Overview of Notations

problem as a data classification task. For generating the selection conditions they use a decision tree classifier that is constructed in a top-down manner in a greedy fashion by determining a “good” predicate according to which the tuples are split into two classes. These two classes would then form the root nodes of two decision trees (constructed recursively).

Sarma et al. [12] explore the *View Definitions Problem* (VDP) which is a subproblem of QBO in that it considers only one relation  $R$  and there are no joins and projections. Thus, they only try to find the selection condition of the view  $V$  and do this looking at the problem as an instance of the set cover problem. From the families of queries that they cover, we focus on conjunctive queries with a single equality predicate and conjunctive queries with any number of equality predicates. For both types they propose naïve algorithms that utilize the size of the attribute domains in the view. Zhang et al. [19] compute a generating *join query* that produces a table  $Q(D)$  from the tables in  $D$ . The generated join query does not have selection conditions and they focus mostly on identifying the joins using graph structures following foreign/primary-key links.

Shen et al. [13] study the problem of discovering a minimal project-join query that contains given example tuples in its output and do not consider selections. They only handle text columns with keyword search allowed on them and introduce a candidate generation-verification framework to discover all valid queries. By using common sub-join trees of the candidate queries as filters they manage to improve the efficiency of their approach.

Psallidas et al. [10] propose a candidate-enumeration and evaluation framework for discovering project-join queries. Their system handles only text columns and establish a query relevance score based evaluation of candidate queries. The system returns the PJ queries with the top-k highest scores and it discovers not only the queries that exactly match the given example tuples. Moreover, they propose a caching-evaluation scheduler, where they dynamically cache common sub-expressions that are shared among the PJ queries. Join queries are orthogonal to our work and none of the above approaches handle top-k aggregation queries.

In keyword search over databases [2], the input is a single tuple with specified keywords as fields. The works of [5, 15] interpret the query intent behind the keywords and compute aggregate SQL queries. Blunski et al. [5] use patterns that interpret and exploit different kinds of metadata, while Tata et al. [15] discovers aggregate SQL expressions that describe the intended semantics of the keyword.

The principle of reverse query processing is studied in [3, 4, 6, 9], however their objectives and techniques are different. Binning et al. [3, 4] discuss the problem of generating a test database  $D$  such that given a query  $Q$  and a desired result  $R$ ,  $Q(D) = R$ . Bruno et al. [6] and Mishra et al. [9] study the problem of generating test queries to meet certain cardinality constraints on their subexpressions.

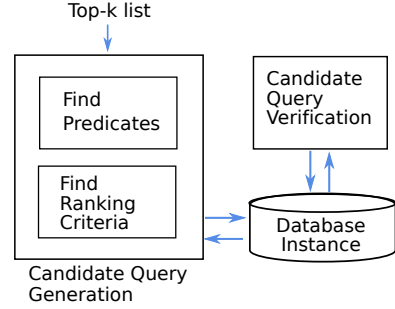


Figure 2: System task steps

A reverse top-k query [18] returns for a point  $q$  and a positive integer  $k$ , the set of linear preference functions (in terms of weighting vectors) for which  $q$  is contained in their top-k result. For example, finding all customers who treat the given query product  $q$  as one of their top-k favorite elements. In such cases, each customer is described as a vector of weights. Although it appears related given the name, this research area is not directly related to our work.

### 3. APPROACH

The task of reverse engineering top-k queries is split into the following three steps, illustrated in Figure 2:

- **Step 1:** find the predicate  $P$  in the where clause of  $Q$
- **Step 2:** find the ranking criteria
- **Step 3:** validate queries

As the basis of further computation, we first retrieve from relation  $R$  all tuples whose entity column contains one of the entities of the input table  $L$ ; we call the resulting table  $R'$ .

#### 3.1 Table $R'$

Consider a top-k list  $L$  as shown in Figure 1. Let  $e_i \in \{e, f, g, m, o\}$  denote the entities in the column  $L.e$ .

By using a standard database index, such as a B+ tree, on the entity attribute of  $R$ , we can efficiently retrieve  $R'$  (shown in Table 4) containing all tuples from  $R$  matching any of the entities  $e_i \in L.e$ . Whether the index is actually used or the query optimizer decides to perform a table scan is not a concern here. In any case, in this example, the query to compute  $R'$  is

```
SELECT * FROM R
WHERE  $A_e$  IN [e, f, g, m, o]
```

For the purpose of efficient access of its data, PALEO stores  $R'$  in-memory in a **column oriented** fashion, with columns being represented as arrays, allowing fast evaluation of aggregate queries over  $R'$ . The relation  $R'$  has  $k' \geq k$  number of tuples, since it contains all tuples without (potentially) being filtered by predicates. In fact, it is reasonable to assume, without prior knowledge, that  $k' \gg k$ , as each distinct entity  $e_i$  can appear many times in  $R$ . We will allow to work on a subset (samples) of  $R'$  in Section 6, and study the consequences, but for now we assume  $R'$  in fact covers *all* tuples of *any* entity of the input.

#### 3.2 The Three Steps

**Candidate Predicates Identification.** Using the tuples in  $R'$  we create a set of *candidate predicates* that are subsequently augmented with ranking criteria to make up full-fledged candidate queries.

**DEFINITION 1. Candidate Predicate**

We say a predicate  $P$  is a candidate predicate iff for each entity that appears in  $L$  there is a tuple  $t$  in  $R'$  that fullfils the predicate. Formally,

$$\forall e_i \in L.e \exists \text{tuple } t \in R' : P(t) = \text{true} \wedge t.e = e_i$$

It is easy to see that a candidate predicate can potentially produce the top-k input list. In other words, having a candidate predicate in the where clause is a *necessary criterion* of a query to be a valid query, but it is *not a sufficient criterion*. This is because a candidate predicate can still “let through” tuples of other entities (that are not in the input table  $L$ ) that can be ranked higher than the tuples in  $L$ , hence, the query is not a valid query as the output does not match the input.

**COROLLARY 1. Downward-closure (anti-monotone) property of the candidate predicate criterion.** *Given a predicate  $P_1$  that is not a candidate predicate, then a predicate  $P_2$  such that  $P_1 \subseteq P_2$  (that is, all sub-predicates in  $P_1$  are also present in  $P_2$ ) can not be a candidate predicate.*

The corollary follows immediately from the definition of candidate predicates: any predicate  $P_i$  with  $P_1 \subseteq P_i$  for another predicate  $P_1$  evaluates to true for a subset of tuples for which  $P_1$  evaluates to true. This property is used to prune the searchspace in Section 4, similar to what the apriori algorithm [1] does for the support measure.

**Ranking Criteria Identification.** In the *second step* of our approach, we identify the ranking criteria according to which the entities in the top-k list are ranked. For this purpose we need to find a suitable numeric attribute (or multiple ones) including an aggregation function—or decide if one is used at all.

**DEFINITION 2. Candidate Ranking Criterion**

We say a ranking criterion, consisting of one or multiple numerical attributes and, if existing, an aggregation function is a candidate iff, when executed on  $R'$  together with a candidate predicate, it returns a result identical to the input list  $L$ .

This definition is very reasonable but similar to the criterion to identify candidate predicates it is only a necessary condition to a valid ranking criteria for a query when executed over the entire relation  $R$ . It is, however, not a sufficient condition, as when executed on  $R$  there can be still other entities, not in  $L$ , that are disturbing the “correct” order. The case of partial matches is discussed below.

**Candidate Queries Identification and Evaluation.** Using the candidate predicates and the valid ranking criteria we can form candidate queries. Each candidate query is executed on  $R$  and the results are compared with the input top-k list. The queries that produce instance-equivalent results with the original query are the valid queries.

**3.3 Allowing Partial Matches**

Like other approaches on reverse engineering queries, this approach can be relaxed to allow finding also partially matching queries. This can be useful for cases where the input  $L$  has been obtained from an older instance of the database or in cases where  $L$  has been generated in the extreme, through crowdsourcing top-k rankings. Our approach can be adapted to such partial match scenarios as follows. First, the condition to accept a query during the validation phase needs to be switched to accepting partial match. For comparing rankings, there exist several ways, most prominently Spearman’s Footrule distance and Kendall’s Tau. Fagin et al. [7]

t.id	R'								
	E	A	B	C	...	N <sub>1</sub>	N <sub>2</sub>	N <sub>3</sub>	...
1	e	a <sub>1</sub>	b <sub>9</sub>	c <sub>3</sub>	...	75	4	5	...
2	e	a <sub>1</sub>	b <sub>8</sub>	c <sub>1</sub>	...	100	8	7	...
3	e	a <sub>3</sub>	b <sub>1</sub>	c <sub>6</sub>	...	45	15	1	...
4	f	a <sub>1</sub>	b <sub>8</sub>	c <sub>1</sub>	...	90	16	2	...
5	f	a <sub>5</sub>	b <sub>4</sub>	c <sub>6</sub>	...	35	23	3	...
...	...	...	...	...	...	...	...	...	...
10	g	a <sub>1</sub>	b <sub>8</sub>	c <sub>3</sub>	...	80	42	14	...
...	...	...	...	...	...	...	...	...	...
20	m	a <sub>1</sub>	b <sub>8</sub>	c <sub>4</sub>	...	70	29	10	...
...	...	...	...	...	...	...	...	...	...
30	o	a <sub>1</sub>	b <sub>8</sub>	c <sub>4</sub>	...	60	31	7	...
...	...	...	...	...	...	...	...	...	...

Table 4: Example of a Relation  $R'$  for Input L in Figure 1

show how these measures can be applied to top-k lists. In our case of ranking with two columns (entity and value) we would compute such methods on the entity column; and can additionally compute a distance measure like L1 or L2 on the values if numerical or otherwise use a distance like the set-based Jaccard distance. Second, not taking for granted that we cannot precisely reverse engineer the input  $L$  implies that even a fully known  $R'$  would behave exactly like being a sample, with the consequences described in Section 6. That means, we can directly apply the reasoning on query suitability explained there.

**4. CANDIDATE PREDICATES**

The task we consider in this section is to find all  $k$ -sized candidate predicates  $P_i$ . Each predicate can be simple atomic equality predicate like  $(A = a_1)$  or conjunctions of atomic equality predicates, e.g.,  $(A = a_1) \wedge (B = b_8)$ . Candidate predicates are determined over the table  $R'$ , as described above. From Definition 1 we know that in order to be a candidate predicate, a predicate  $P$  has to have for each entity in the input  $L.e$  at least one tuple in  $R'$  with  $P(t) = \text{true}$ .

This criteria is anti-monotone (aka. downward-closed), i.e., a predicate  $P_i$  with size  $k$  can be considered a candidate predicate if and only if all its sub-predicates are also candidate predicates. This problem is similar to frequent itemset mining for which the apriori principle and algorithm [1] is widely known. In data mining terminology, itemsets resemble the values that are used to form the candidate predicates.

The method to compute candidate predicates in PALEO is described in Algorithm 1. In the first step,  $k = 1$ , we start by identifying all atomic candidate predicates, i.e., the predicates with size  $|P_i| = 1$  (Lines 2–6 in Algorithm 1). For this purpose for each column  $A_i$  we identify values  $v$  such that the predicate  $P_i := (A_i = v)$  is a candidate predicate (Lines 3–4 in Algorithm 1). Furthermore, for each such created  $P_i$  we keep a set  $\mathcal{I}_{P_i}$  containing the tuple ids (aka. row ids) that this predicate selects, i.e.,  $\mathcal{I}_{P_i} = \{t.id | P_i(t) = \text{true}\}$ . In each additional step, conjunctive predicates of size  $k$  are created, by adding atomic predicates from the set  $\mathcal{P}_1$  to the predicates created in the previous iteration (Lines 7–14 in Algorithm 1). The algorithm does not create a predicate multiple times. The conjunctive predicate  $P_{ij}$  whose tuple ids set  $\mathcal{I}_{P_{ij}}$  covers all entities in the input list is added to the set of candidate predicates with size  $k$  (Lines 12–13) and will be used in creating candidate predicates of size  $k + 1$  in the next iteration.

*Example:* Considering Table 4 and the input list in Figure 1, we create atomic predicates starting with column  $A$  as we iterate over its values  $a_i$ . Note that the entities in  $E$  are sorted. The set of atomic candidate predicates is created,  $\mathcal{P}_1 = \{P_1 := (A = a_1), P_4 := (B = b_8)\}$ . These two predicates are candidates, since the tuples that fulfill the predicates cover all entities in the input list  $L$ . Furthermore, the set tuple ids that the predicates select are kept, e.g.,  $\mathcal{I}_{P_1} = \{1, 2, 4, 10, 20, 30\}$ . If added as a selection condition,



method: **findPredicates**

input: top-k list  $L$

relation  $R'$

output: a set of candidate predicates  $\mathcal{P}$

```

1   $\mathcal{P} = \emptyset; k = 1; \mathcal{P}_k = \emptyset$ 
2  for each  $A_i$  in  $R'$ 
3    find  $P_i := (A_i = v)$  with  $|P_i| = 1$  s.t.
4     $\forall e_i \in L.e \exists \text{tuple } t \in R' : P_i(t) = \text{true} \wedge t.e = e_i$ 
5    add  $P_i$  to  $\mathcal{P}_k$ 
6    for each  $P_i$  keep  $\mathcal{I}_{P_i} = \{t.id | P_i(t) = \text{true}\}$ 
7  repeat
8     $k = k + 1$ 
9     $\mathcal{P}_k = \emptyset$ 
10   for each  $P_i \in \mathcal{P}_1$  and  $P_j \in \mathcal{P}_{k-1}$  and  $P_i \cap P_j = \emptyset$ 
11     create  $\mathcal{I}_{P_{ij}} = \mathcal{I}_{P_i} \cap \mathcal{I}_{P_j}$ 
12     if  $\mathcal{I}_{P_{ij}}$  covers all  $e_i \in L.e$ 
13       add  $P_{ij} := P_i \wedge P_j$  to  $\mathcal{P}_k$ 
14 until  $\mathcal{P}_k = \emptyset$ 
15 return  $\mathcal{P} = \bigcup_k \mathcal{P}_k$ 

```

**Algorithm 1:** Finding candidate predicates

these candidate atomic predicates would result in a candidate query.

In each next step, we try to produce conjunctive clauses of size  $k$  from the predicates in  $\mathcal{P}_1$  and  $\mathcal{P}_{k-1}$ . Thus, for  $k = 2$ , we test if the predicate  $P_{14} := (A = a_1) \wedge (B = b_8)$  qualifies as a candidate by intersecting the corresponding sets of tuple ids. Since the intersected tuple ids in  $\mathcal{I}_{P_1} \cap \mathcal{I}_{P_4} = \{2, 4, 10, 20, 30\}$  cover all entities in  $L.e$ , the predicate  $P_{14}$  is a candidate predicate. Recall that  $R'$  is held in memory and that we can, via tuple ids, very efficiently access the full tuple to check whether or not it matches the predicate.

**Properties of the Algorithm:**

- (i) The algorithm is **correct** with respect to  $R'$ , that is, predicates returned by the algorithm are guaranteed to be candidate predicates, following Definition 1. Further, the algorithm is **complete**, that is, it finds all possible candidate predicates over  $R'$ .
- (ii) When predicates are applied in  $R$  instead of  $R'$  they can also let tuples with entities that are not in  $L$  pass, which leads to **false positive** candidate queries.

The difference to the apriori algorithm that operates on the support measure is that apriori counts the frequency of *all* itemsets and then determines the ones above the specified threshold. In our algorithm, we eliminate a predicate as soon as we find that it does not cover a certain entity. The same happens in each additional pass, since apriori will generate all the pairs of frequent items and count their appearance. Thus, all pairs that contain a false positive singleton will also be false positives.

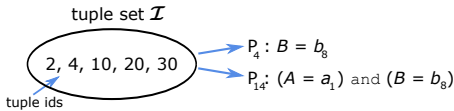


Figure 3: Mapping from tuple set to predicates.

## 4.1 Tuple Sets and Predicates

Some of the created candidate predicates have identical tuple sets  $\mathcal{I}_{P_i}$ . These predicates select the same tuples in  $R'$  and share the same data characteristics regarding to  $R'$ . Thus, candidate predicates are grouped according their tuple

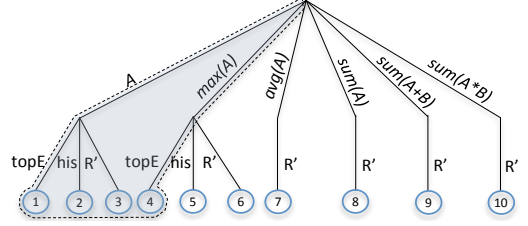


Figure 4: Order of looking for the ranking criteria

sets, i.e., if  $\mathcal{I}_{P_i} = \mathcal{I}_{P_j}$ , then  $P_i$  and  $P_j$  would belong to the same group.

Figure 3 depicts a tuple set mapped to a group of candidate predicates created from the tuples in Table 4. The predicates  $P_4$  and  $P_{14}$  cover the same tuples in Table 4. Thus, for these predicates, it is enough to examine the data characteristics of the tuples in the tuple set  $\mathcal{I}$ .

## 5. RANKING CRITERIA

In order to find the ranking criteria according to which the ranking in the top-k list is done, PALEO operates on the distinct tuple sets, determined in the algorithm above. If relation  $R$ , and hence also  $R'$ , is identical to the database state when the input data was once generated, it is guaranteed that PALEO is able to determine the valid ranking criteria.

The actual size of  $R'$  depends naturally on the size  $k$  of the input list  $L$  and also on the data characteristics, i.e., how many tuples  $R$  contains for a single entity. We expect  $R'$  to be holding a factor of  $k/n$  less tuples than  $R$ , where  $n$  is the number of distinct entities in  $R$ , and that this allows to load  $R'$  entirely in main memory. While it might be reasonably cheap to execute a query on this  $R'$  in memory, note that we have to possibly do so very many times to identify suitable ranking criteria. That is, depending on the size of  $R'$  we can potentially reduce the runtime of our algorithm if it can be avoided to work on  $R'$  directly.

The idea is to harness small data samples, histograms, or simple descriptive statistics computed upfront from the base relation  $R$  in order to select a subset of potentially useful columns without touching  $R'$ . However, there might be invalid criteria identified or potentially also no criteria at all, given the limited coverage of data samples and the impreciseness of histograms. Therefore, identified candidate ranking criteria are validated on  $R'$  and in case no heuristic is applicable or was not successful, the whole ranking criteria identification is executed on  $R'$ .

Depending on the aggregation function we aim at checking for suitability, we can or cannot use some of these techniques. For instance, comparing the entities in  $L$  with the top entities stored for each column of  $R$  can be applied to queries with *max* aggregate function, but not directly to queries using *sum* as the aggregation function. Figure 4 summarizes this observation. Traversing the tree pre-order depth-first is the way PALEO looks for the ranking criteria, with the leaf nodes showing the order in which the techniques are applied. The system tries to identify the ranking criteria with smaller search space first. Thus, for instance, if the valid ranking criteria is *max(A)* and comparing the top entities produces valid results, only the shaded part of Figure 4 will be processed.

## 5.1 Top Entities

The most apparent first attempt to identify an attribute according to which tuples are sorted in  $L$  is to store for each attribute in  $R$  the topmost entries, when sorted by the specific attribute. Then, we intersect the input entity set

method: **topEntities**

**input:** top-k list  $L$   
relation  $R'$   
**output:** a set of candidate numerical columns  $\mathcal{A}_C$

```
1 for each  $A_i$  in  $R'$ 
2   if  $A_i$  not numerical, then skip  $A_i$ 
3   if  $\max(v \in A_i) < \max(v \in L.v)$ , then skip  $A_i$ 
4   if  $\min(v \in A_i) > \min(v \in L.v)$ , then skip  $A_i$ 
5   if  $|A_i| < |L.v|$ , then skip  $A_i$ 
6   if  $TopE(A_i) \cap L.v \neq \emptyset$ , add  $A_i$  to  $\mathcal{A}_C$ 
7 return set of candidates  $\mathcal{A}_C$ 
```

**Algorithm 2:** Finding candidate columns with top entities

from  $L$  with these top entries. More than just the  $k$  top values are stored to increase the chance that these entities do overlap with the entities in  $L$ . Clearly, it should also not be too large such that each numeric column appears promising. The exact way of how this idea is applied is shown in Algorithm 2, line 6.

Before this is done, PALEO filters out attributes by applying three simple checks: it compares the max (min) values of the input list and the column and if the column’s value is smaller (greater) than the max value of the input list it does not intersect the entities (Algorithm 2, line 3 and 4). Additionally, the number of distinct values is compared: If the column has less distinct values than the input list, we skip this column (Algorithm 2, line 5).

The numerical columns that result in a *non-empty intersection* are considered as candidate numerical columns. Thus, using  $R'$  and the tuple sets created in finding candidate predicates, they are checked whether they can match the ranking in the top-k input list.

## 5.2 Querying Histograms

In the case no candidate numerical columns have been identified with the above intersection of top entities, PALEO employs histograms describing an attribute’s frequency distribution in order to find candidate attributes that appear suitable for ranking. As we consider only numeric attributes to be used as the bases of ranking criteria, such a histogram describes how frequent a specific numeric value appears in the attribute’s column in relation  $R$ . One idea is comparing the value-frequency distributions of the histogram of the input list with the histograms of the numerical columns in  $R$ , by using histograms distance measures such as Earth Mover’s Distance [11]. However, a top-k list is inherently small and does not contain enough elements to provide a meaningful distribution. Hence, PALEO samples each attribute’s histogram and calculate the L1 distance between its top-k values and the input values. Similar to using top entities of each column, we draw samples following the distribution described in the histogram. PALEO uses equi-width histograms having 1000 cells each.

This is done for each attribute, which allows ordering all attributes by the L1 distance of the sampled data to the data in  $L$ . Depending on the data in the table, if there is a column with similar values and distribution as the column we are looking for, it is possible that the correct column does not have lowest L1 distance. In order to account for this, we consider the top 30% of the columns in the list as candidate attributes.

## 5.3 Validation over $R'$

As a validation for the possible ranking criteria identified above, we use the tuples in  $R'$ . In the case when we have successfully identified candidate attributes with the previous techniques, we first check if any of these candidate attributes

can produce the ranking. For this purpose, we go through the distinct tuple sets  $\mathcal{I}_i$  we computed in Section 4 and check which of the candidate numerical columns, i.e., their sorted aggregated values exactly match the input  $L$ .

Some of the supported ranking criteria cannot be identified by the above mentioned techniques, requiring more complicated statistics and this is beyond the scope of this paper. For instance, with the *avg* and *sum* aggregate functions, the top entities for a column depend heavily on the predicate, since the values are aggregated over multiple tuples. Similarly, harnessing histograms with *sum* would involve convolutions of the histograms of the pairs of columns.

As a fall back, if none of the candidate attributes can produce the ranking criteria, we revert to checking the remaining numerical columns in  $R'$  that were not found as candidates. We still use only the tuples with tuple ids found in the tuple sets  $\mathcal{I}_i$ . For each tuple set and each numerical attribute in  $R'$  that passes our (three) simple checks (i.e., min, max comparison, and number of distinct items), we compute whether the tuples in  $\mathcal{I}_i$  if sorted according to the specific attribute and aggregate function are identical to  $L$ . After identifying the appropriate numerical attribute and aggregate function, we can filter out some of the *candidate predicates*. If a certain tuple set does not contain the input numerical values, we **remove** this tuple set and all the candidate predicates that correspond to it from the candidate predicates.

## 6. HANDLING VARIATIONS OF $R$

The techniques behind PALEO discussed so far are based on the assumption that exactly the same relation  $R$  that produced the input list  $L$  is available and that it is feasible to operate on it directly. However, it might appear that tuples in  $R$  have changed, for instance, because of inserts, updates, and deletes, due to slowly changing dimensions [8] in data warehousing scenarios, or only a subset (sample) is available. In this section, we describe how PALEO deals with situations when only subset of the original tuples in  $R$  is available.

This assumption has direct consequences on PALEO’s ability to accurately identify suitable predicates and ranking criteria. As we have discussed above, determining query predicates with the proper table  $R$  at hand *only* leads to obtaining **false positives** in the candidate predicates, introduced by additional entities outside  $R'$  that qualify for the predicate. The changed data further introduces **false negatives**. That is, the query that generated the input might not be found at all, although such a query exists. This is caused by missing or modified tuples in  $R$  that would be required to unveil a predicate to be fulfilled by all of the  $k$  entities. False negatives are synonym to *loss in recall*, i.e., the fraction of found queries to all existent queries that generate the input.

We address this by

- Reasoning about likelihood of being a successful query.
- Smart evaluation to skip unpromising queries.

Variations in  $R$  means also variations in  $R'$ . Let us denote the table stemming from the modified base table as  $R''$ . It can happen that  $R''$  does not contain tuples from all entities from the input list, for instance if all tuples for a certain entity  $e_i \in L.e$  have been deleted from  $R$ . Recall that the method for finding predicates, described in Algorithm 1 demands that a predicate must cover all entities of the input list  $L.e$ .

Now, it is possible that the tuples containing the valid predicate for a certain entity have changed in the columns that comprise the predicate. Then, it is impossible to precisely validate or invalidate the predicate using the method in Algorithm 1: Being strict, missing the tuples with the

valid predicate for a certain entity will lead to evicting the valid predicate even though the majority of entities in  $R''$  contain tuples with it, thus resulting in false negatives. To avoid that, the condition of evicting a predicate is relaxed. Instead of demanding that a predicate is considered as a candidate predicate if it covers all distinct entities in  $R''$ , we ask for it to cover the *majority of the entities*, thus taking into account that some entities can have tuples with the valid predicate missing. Another possible approach is not to evict predicates at all, i.e., form all the predicates that we encounter in  $R''$  while not demanding any entity covering. This might, however, result in very many candidate predicates with too many false positives. Executing candidate queries for all such predicates will drastically decrease the overall efficiency of PALEO.

We describe a probabilistic model of assessing candidate predicates when the data in the base table has changed and how uncertainty in finding ranking criteria can be handled.

## 6.1 Assessing Candidate Predicates

Changes in  $R$  introduce uncertainty in finding the valid predicates. To account for such changes, the condition of evicting a predicate is relaxed. As a result, our methods identify more candidate predicates that need to be assessed whether or not they are likely to be indeed a valid predicate. This assessment is later used when executing queries in the final step such that queries can be executed in increasing order of the likelihood to be in fact a valid query.

A candidate predicate  $P_i$  identified from the table  $R''$  is a false positive if:  $\exists e_i \nexists t$  s.t.  $P(t) = true$ . In other words, if for a certain entity  $e_i$  there is no tuple for which the predicate  $P$  is valid, then this predicate is a false positive. This means that a query with this predicate would return a top-k list without the entity  $e_i$ .

Consider a predicate  $P$  over the attributes  $A_1, \dots, A_m$ . The probability that a tuple exists in relation  $R$  is given by the number of distinct entries of the columns  $A_i$  (i.e.,  $|A_i|$ ) as

$$P[\text{tuple exists}] = \prod_i \frac{1}{|A_i|}$$

Consider an entity  $e_j$  for which we did not find a tuple that matches the predicate and let  $unseen(e_j)$  be the number of changed tuples of entity  $e_j$ , then

$$P[\text{won't see for } e_j] = (1 - P[\text{tuple exists}])^{unseen(e_j)}$$

The probability that at least one entity is rendering this predicate to be a false positive (by not providing a matching tuple) is thus given as

$$P[\text{false positive}] = 1 - \prod_j (1 - P[\text{won't see for } e_j])$$

## 6.2 Approximating Ranking Criteria

Operating on  $R''$  also introduces uncertainty in finding ranking criteria. Since not all tuples for each entity  $e_i$  are the same, the ranking criterion cannot exactly match the numerical values in the input top-k list. This is why there is a need of measuring the suitability of each candidate ranking criteria to the input list. For this purpose, we compute the distance between the input values and the candidate attribute(s) values. We use the L1 distance (aka. Manhattan distance) that is simply the sum of absolute differences in the numeric values.

**Queries without sum:** The changes in the tuples for an entity  $e_i$  renders the **topEntities** method (Section 5.1) not directly applicable. Without the identical tuples, it is difficult to match the candidate numerical columns with the input ranking values. Using the L1 distance and the column

values in  $R''$  (Section 5.3) provides the possibility to compute the suitability of the candidate ranking columns. That way, each candidate column has a corresponding L1 distance that is used in ranking the candidate queries.

**Queries with sum:** The **sum** aggregate function sums up all values for a certain entity  $e_i$ . Since with changed data some of the tuples for an entity are missing, they need to be approximated. We do this by using the column values for the column(s) in  $R'$ . Using this approximation, the L1 distance to the input ranking values is calculated and then used for ranking the suitability of the column(s).

The approximation of the sum for each entity is done using the tuple id sets. We take a look at the more complicated case of having a sum of two columns  $A_i$  and  $A_j$ . Thus, for a predicate  $P$  with a corresponding tuple set  $\mathcal{I}_P$ , for each entity  $e_i$  let  $sum_{A_{ij}}(\mathcal{I}_P)$  denote the sum of the values, of the columns  $A_i$  and  $A_j$ , of the tuples in  $R''$  with tuple ids in  $\mathcal{I}_P$  that have an entity  $e_i$ , i.e.:

$$sum_{A_{ij}}(\mathcal{I}_P) = A_i(\mathcal{I}_P) \text{ op } A_j(\mathcal{I}_P) \text{ s.t. } t.e = e_i, \text{ op} \in \{+, *\}$$

Additionally, let  $\#v$  denote the number of tuple ids in the tuple set  $\mathcal{I}_P$  of the entity  $e_i$ , i.e., the number of tuples that the predicate  $P$  selects with  $e_i$ . We approximate the sum as:

$$approxSum_{e_i}(\mathcal{I}_P) = \frac{sum_{A_{ij}}(\mathcal{I}_P)}{\#v} \times (\#v \times \frac{|e_i|_{R''}}{|e_i|_{R''} - unseen(e_i)})$$

where  $|e_i|_{R''}$  is the number of tuples in  $R''$  for the entity  $e_i$ . Thus, for each entity  $e_i$ , the average summed value from the sampled tuples is multiplied with the approximated selectivity of the predicate  $P$ . The sorted list  $approxSum(\mathcal{I})$  formed from the sums for each entity  $approxSum_{e_i}(\mathcal{I}_P)$  is then used for calculating the L1 distance  $d$  to the input list and ranking the candidate column pairs.

## 6.3 Combined Model

The queries formed from the combination of candidate predicates and ranking criteria need to be validated by executing them on  $R$ . The order of execution is done ordered by a suitability value for each candidate query  $Q_c$ . The suitability is computed as:

$$s(Q_c) = (1 - P[\text{false positive}]) \times (1 - d)$$

where  $P[\text{false positive}]$  is the probability of the predicate in the candidate query of being a false positive and  $d$  is the max normalized L1 distance between the ranking criteria in  $Q_c$  and the numerical values in the input list  $L$ .

## 6.4 Working with Samples of $R'$

Consider a scenario where it is impossible or unfeasible to work on the complete relation  $R'$  (the subset of  $R$  of all tuples that contain any of the entities in  $L$ ). This relation  $R'$  can be very large, potentially as big as  $R$ , if there are many tuples for each distinct entity—a typical case in data-warehousing applications that often aggregate large amounts of observations of a specific entity. The probabilistic model for assessing candidate predicates together with the approximation of the ranking criteria can also be applied to such a scenario as well.

We consider two approaches of sampling. First, we sample by retrieving all tuples for a certain (e.g., randomly selected) subset of the entities in  $L.e$ . In this way, we do not get any false negatives and the candidate predicates set is a superset of the valid predicates. This is because having all tuples in  $R''$  for a certain entity is guaranteed to contain the tuples with valid predicates. As a result, our algorithm will create the predicate as a candidate. However, the drawback of this approach is having too many false positives. This can

especially happen if for a sampled entity there are too many tuples in the base table  $R$ . This will lead to creating a large amount of false positives which impairs efficiency.

Sampling uniformly from all entities mediates this problem, thus sampling a certain percentage of the tuples from each entity. This way, possibility of false positives is decreased, at the price of an increased possibility of false negatives. We encounter the same problem as if we would sample by tuple: it can happen that tuples that contain a valid predicate are not sampled for a certain entity. Relaxing the condition of evicting a predicate mediates this problem.

We can draw a parallel between the scenarios of having modified data in  $R$  and sampling. The tuples that are sampled in  $R''$  correspond to the tuples in the base table that have the columns comprising the valid predicate **unmodified**. Hence, the not sampled tuples are analogous to the ones that are modified.

Consider a predicate  $P_i$  that is a valid predicate for an entity  $e_i$ . The probability that  $k$  tuples with the valid predicate are sampled in  $R''$  has a hypergeometric distribution, i.e.,

$$P[\text{k tuples sampled}] = \frac{\binom{K}{k} \binom{N-K}{n-k}}{\binom{N}{n}}$$

where  $K$  is the total number of tuples with the predicate in  $R'$ ,  $N$  is the total number of tuples to sample from, i.e.,  $N = |R'|$ , and  $n = |R''|$  is the number of sampled tuples.

The probability of sampling at least one tuple with the valid predicate  $P_i$  for an entity  $e_i$ :

$$P[\text{one tuple sampled}] = 1 - \frac{\binom{K}{0} \binom{N-K}{n-0}}{\binom{N}{n}}$$

Considering an input top-k list with  $m$  distinct entities  $e_i$  and assuming independence in the sampling from the different entities, the probability of seeing a tuple with the valid predicate is:

$$P[\text{all } e_i] = P[\text{one tuple sampled}]^m$$

Intuitively, this probability describes that increasing the sampling size increases the probability of sampling a tuple with the valid predicate for each distinct entity. Consequently, making the condition of evicting a predicate more strict as the sample size increases is needed, i.e., increasing the number of entities  $e_i$  that are covered by a predicate so it can qualify as a candidate predicate. This would eliminate the creation of too many false positives with larger sample sizes.

## 7. SMART QUERY VALIDATION

Ordering candidate queries by their expected suitability to answer the input  $L$  promises to find a valid query early—ideally at the first query execution. Even if more than one valid query is to be found, such an order is accelerating the discovery process immensely. We will show in the experimental evaluation that this is indeed the fact.

Now, instead of purely trusting the order, it would be careless to simply execute queries sequentially in the given order, without trying to benefit from information learned while executing them. Consider a candidate query  $Q_c$  that is executed and yields a result  $Q_c(R)$  that is very similar to the input list  $L$ , but is still not an exact match. It would be preferable to continue validating queries that are similar to  $Q_c$  and skip those in the ordered query candidate list  $\mathcal{C}$  that are not.

It is clear that the similarity (overlap) of the results of a candidate query when executed over  $R$  and input list  $L$  can be directly computed, using Jaccard similarity for instance. But for the not-yet-executed queries we do not have direct insight on their result, but we can “speculate” about it: We model this similarity between  $Q_1$  and  $Q_2$  by two means; first,

```

method: resultDrivenValidation
input: ordered list of candidate queries  $\mathcal{C}$ ;
      Jaccard similarity threshold  $\tau$ 
output: a valid query  $Q_v$ 
1   $Q_c := \mathcal{C}.first$ 
2  /* search for first query with results overlapping  $L^*$ /
3  while  $J(Q_c(R).e, L.e) < \tau$ 
4      $Q_c := \mathcal{C}.next$ 
5  /* keep this first match query */
6   $Q_{fm} := Q_c$ 
7  foundR := false
8  foundR := true if  $J(Q_c.v, L.v) > \tau$ 
9  while( $\mathcal{C}.hasNext$ )
10      $Q_c = \mathcal{C}.next$ 
11     /* skip query  $Q_c$ ? */
12     if  $(P(Q_c) \cap P(Q_{fm}) = \emptyset$  or
        (foundR and  $R(Q_{fm})! = R(Q_c)))$ 
13         continue
14     execute  $Q_c$ 
15     return if found valid
16 resultDrivenValidation(skipped  $Q_c$ )

```

Algorithm 3: Result driven candidate query validation

by the common atomic equality predicates in the conjunctive where clause, and second, by the use of the same (or not) ranking criteria. For this, with  $R(Q)$  we denote the ranking criterion of a query and with  $P(Q)$  the set of its atomic predicates.

For each executed query we check if its output matches the input list. In the first part of the algorithm presented in Algorithm 3, we sequentially test the candidate queries until we have found for which the entities in its results are similar to the entities in  $L.e$ . This query is denoted  $Q_{fm}$ , for “first match query”. We also check if the numeric values of the query result are similar to the numeric values  $L.v$  of the input list  $L$ , again, using the Jaccard similarity. If they are sufficiently similar, we mark the ranking criteria of query  $Q_{fm}$  as valid. In the second while loop (line 9–13 in Algorithm 3), we iterate over the remaining candidate queries and skip those queries whose predicates are not at all overlapping with the predicates in  $Q_{fm}$ . We further skip queries that have a different ranking criterion to the one of  $Q_{fm}$  (line 12 in Algorithm 3), in case this was found as valid.

If by the end of the query list  $\mathcal{C}$  a valid query is not found, the algorithm is called for the previously skipped queries, until all queries are evaluated or one valid query is found.

## 8. EXPERIMENTAL EVALUATION

We have implemented the approach described above in Java. Experiments are conducted on a 2× Intel Xeon 6-core machine, 256GB RAM, running Debian as an operating system, using Oracle JVM 1.7.0\_45 as the Java VM (limited to 20GB memory). The base relation  $R$  is stored in a PostgreSQL 9.0 database, with a B+ tree index on  $R$ ’s entity column.

**Datasets.** We evaluate our approach of computing instance-equivalent queries using data and queries of two benchmarks, **TPC-H** [16] and the **SSB** [14]. For this, we created a scale factor 1 instance of both TPC-H and SSB data and materialized a single table  $R$  by joining all tables from their respective schema. The table  $R$  results in 57 and 60 columns, for TPC-H and SSB, respectively. The column  $c\_name$  (from the customer table) acts as the entity column. We obtain tables with the characteristics described in Table 5.

	TPC-H	SSB
# Tuples	5,313,609	6,001,171
# Entities	171,753	20,000
# Textual columns	27	28
# Non-key numerical columns	13	20
# Avg tuples per entity	31	300
Highest # tuples per entity	187	579

Table 5: Table  $R$  characteristics

	Query	sel.
T P C - H	$\gamma_{c\_name, MAX(o\_totalprice)}$ $(\sigma_{p\_type='MEDIUM POLISHED STEEL'})$ $\wedge r\_name='AMERICA'(R)$	0.001
	$\gamma_{c\_name, SUM(ps\_supplycost+ps\_availqty)}$ $(\sigma_{n\_name='JAPAN'})$ $\wedge p\_container='JUMBO BAG'$ $\wedge l\_shipmode='TRUCK'(R)$	0.0001
	$\gamma_{c\_name, AVG(lo\_revenue)}$ $(\sigma_{s\_nation='UNITED STATES'})$ $\wedge p\_category='MFGR\#14'(R)$	0.002
S S B	$\gamma_{c\_name, SUM(lo\_extendedprice*lo\_discount)}$ $(\sigma_{p\_brand='MFGR\#2221'})$ $\wedge s\_region='ASIA'$ $\wedge d\_year=1995(R)$	0.00003

Table 6: Example queries and their selectivity

We examine the applicability of our approach with variation of data in  $R$ , by performing experiments with sampling. As described in Section 6, operating on a sample of  $R$  has similar characteristics as working on a table  $R$  with modified data.

**Queries.** There are 13 and 22 queries available in the TPC-H and SSB benchmark, respectively. We adjusted the original queries by creating different query types ( $max(A)$ ,  $avg(A)$ ,  $sum(A)$ ,  $sum(A+B)$ ,  $sum(A*B)$ , and no aggregation), supported by PALEO (cf., Figure 4). We only write the ranking criteria when discussing the different query types. In order to examine the effects of the predicate size and selectivity factor, in each query, we vary the predicate size  $|P|$ , with  $|P| \in \{1, 2, 3\}$ . Queries with larger predicates have higher selectivity. Furthermore, all queries have the column  $c\_name$  as an entity column. Example queries and their selectivity are shown in Table 6.

We execute each query  $Q$  over the table  $R$  to produce the top- $k$  lists  $L$ . Using the LIMIT clause, we create top- $k$  lists with  $k \in \{5, 10, 20, 50, 100\}$ . Then, we execute PALEO with inputs  $L$  and the table  $R$ . For the experiments involving sampling, we perform the experiments three times for each input list  $L$  and report on the **median** performance. In order to examine the effects of different sample sizes, we created experiments with sample size of 5%, 10%, 20%, and 30%. We keep the 1,000 top entities for each numerical column.

Using the B+ tree on  $R$ , for each input list we retrieve (a sample of)  $R'$  and store it in memory. Thus, identifying the candidate predicates and ranking criteria are in-memory processes. Without using any compression techniques, the memory consumption of  $R'$  in our experiments was around 500MB. The query validation step is done by issuing queries to the underlying PostgreSQL database that resides on disk. Finally, queries show similar results depending on the number of columns in the aggregate function. Thus, for the sake of brevity, we discuss the results of  $max(A)$  queries as representative of single column queries and  $sum(A+B)$  for the two column queries. Finally, although PALEO discovers all valid queries for an input list, we focus on the efficiency of discovering the first valid query in the presented results.

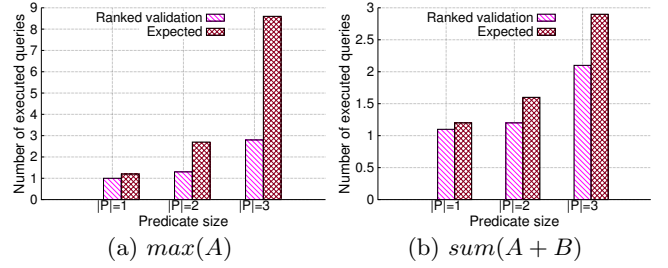


Figure 5: Number of query executions until first valid query with all tuples for TPC-H dataset

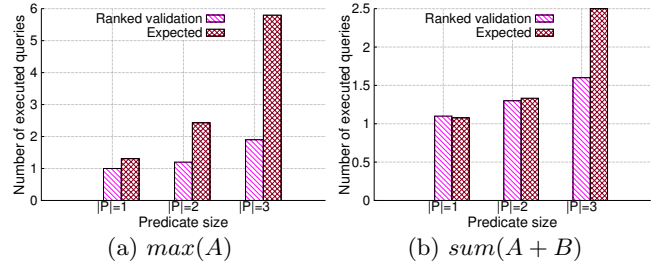


Figure 6: Number of query executions until first valid query with all tuples for SSB dataset

**Valid query discovery.** PALEO **always** discovers all valid queries for any of the supported query types when having available the entire table  $R'$ . The availability of all tuples ensures that false negatives are avoided, and introduces only (a small number of) false positives.

We observe that with all tuples from  $R'$  available, our system requires very few query executions in order to identify a valid query. Thus, for  $sum(A+B)$  queries and the TPC-H dataset, the average number of query validations amounts to only 1.1 for  $|P|=1$ , 1.3 for  $|P|=2$ , and 2.1 for  $|P|=3$ . In fact, for both TPC-H and SSB, **only a single query validation** is required for 76% of the top- $k$  lists that stem from  $sum(A+B)$  queries, while only two query executions are required for 14% of the top- $k$  lists. Similarly, 65% and 70% of the top- $k$  lists from  $max(A)$  queries are found after a single candidate query is executed, while 26.6% and 16% after two query executions, for TPC-H and SSB respectively. Moreover, as shown in Figures 5 and 6, ranked validation outperforms the expected unordered validation and the benefit increases with predicate size. The expected number of query validations reflects the case of executing candidate queries in random order. Assuming a uniform probability of the location of the valid queries in the candidate list, we compute the number of expected validations with dividing the number of candidate queries with the number of valid queries.

**Query discovery efficiency.** We study the efficiency of the different steps from our system. Figure 7 shows the runtime of each step of our approach. As expected, the total runtime is dominated by the database-related operation, i.e., the candidate query validation (Step 3). Note that Figure 7 shows the runtime of finding the first valid query. We observe that for the TPC-H dataset the runtime of Step 3 is orders of magnitude higher than that of Step 1 and 2. Thus, for  $max$  queries the average runtime of candidate query validation is 3.6 *seconds*, while the average runtime of identifying candidate predicates and ranking criteria is 12.4 and 3.9 *milliseconds*, respectively. With the SSB dataset and the same

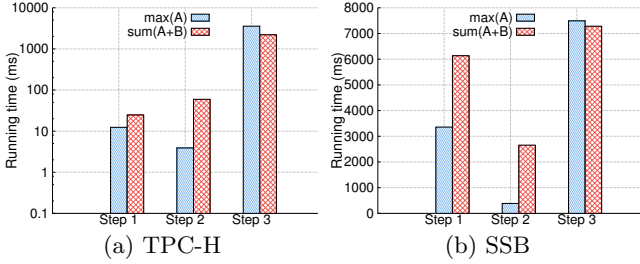


Figure 7: Running times by step

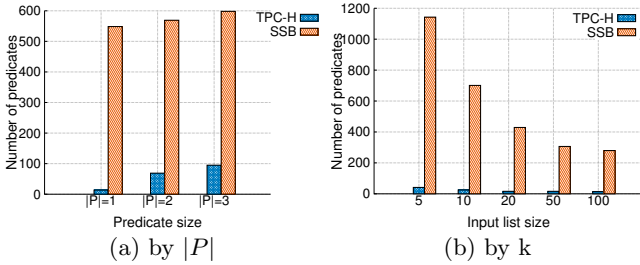


Figure 8: Number of candidate predicates for  $\max(A)$  queries

type of queries, Step 3 needs 7.5 seconds, while the runtime of Step 1 and 2 amounts to 3.3 and 0.3 seconds respectively. The table  $R$  from the SSB dataset has more tuples per entity, which leads to having a larger  $R'$  and more data to process with our algorithms.

**Identifying candidate predicates.** We study the effect of predicate size and the length of the input top-k lists on creating candidate predicates. Figure 8 shows the number of created candidate predicates with different predicate and input list size. We observe that for the TPC-H data, the average number of created candidate predicates increases from 13.8 with  $|P| = 1$ , to 69 with  $|P| = 2$ , and to 95 with  $|P| = 3$ . We observe the same trend with the SSB dataset. Larger predicate size leads to generating more candidate predicates. The reason for this is that for a valid predicate with size  $|P|$  we create as candidate predicates all sub-predicates with size smaller than  $|P|$  as well. The number of shared tuple sets is smaller than the one of created predicates.

Figure 8(b) shows the average number of created predicates with different length of the top-k input lists. We observe that the number of candidate predicates decreases with larger  $k$ . For TPC-H, the number of created predicates decreases from 41.3 for  $k = 5$  to 14.3 for  $k = 100$ . With SSB, the average number of candidate predicates decreases from 1142.9 for  $k = 5$  to 279.7 for  $k = 100$ . Larger  $k$  reduces the number of false positives in the candidate predicates. A predicate needs to select tuples with the distinct entities from the input list in order to qualify as a candidate predicate. With larger lists the number of entities increases, thus making it more difficult for a predicate to qualify as a candidate. Furthermore, we observe that a significantly larger number of predicates is created with the SSB data. This is due to the characteristics of the dataset, with SSB having more tuples per entity and more variety in data.

## 8.1 Evaluation with Sampling

The TPC-H generator creates uniform column distributions, thus the generated instance does not contain enough tuples per entity, with 14 tuples for an entity, at most. The

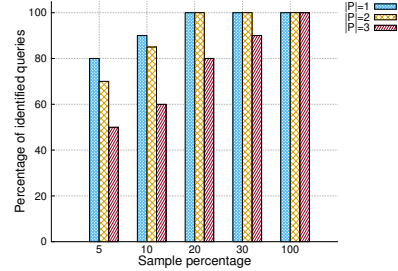


Figure 9: Valid query discovery with  $\text{sum}(A+B)$  queries

SSB data has many tuples per entity, however these are extremely diverse in terms of predicates, i.e., the predicates found in the SSB queries often cover only a single tuple per entity. We thus focus on TPC-H data when employing sampling. There, for each tuple  $t$  in  $R$  we add  $n$  additional tuples, where  $n$  is a random number following the Gaussian distribution  $\mathcal{N}(200, 50)$ . These  $n$  tuples have the same values in the textual columns as  $t$ , but with non-key numerical values:  $v = v + v \times \text{abs}(m)$ , where  $m \in [0, 1]$  is a random number following  $\mathcal{N}(0.5, 0.5)$ .

We study the effect of the sample size on the successful discovery of valid queries. We observe that a valid query is successfully discovered for **all** top-k lists that stem from single column queries, regardless of sample and predicate size. Figure 9 shows results for the discovery of  $\text{sum}(A+B)$  queries. The discovery of valid queries depends on both the sample and predicate size. Having larger sample size enables better query discovery. For  $|P| = 2$  and a sample size of 5% our system successfully manages to discover a valid query for 70% of the top-k lists. With a sample size of 10% the percentage of discovered queries increases to 85%, while with a sample size of 20% and larger, we manage to discover 100% of the queries with  $|P| = 2$ . Furthermore, we observe that discovering queries with larger predicate size is more difficult. With a sample size of 10% we successfully discover a valid query for 90% of the top-k lists with  $|P| = 1$ , 85% with  $|P| = 2$ , and 60% with  $|P| = 3$ . Queries with larger predicates are very selective, hence the probability of sampling tuples with a valid predicate is lower, which leads to false negatives. Sampling more tuples for these queries mediates this problem.

**Smart Query Validation.** Validating the created candidate queries is the bottleneck of our approach; executing (aggregated) queries on the database is expensive. We study the effects of our candidate query validation in terms of the computed query suitability and our result driven optimization. In addition, we investigate the effects of the predicate and sample size. Table 7 shows the average number of query executions needed using the two approaches for candidate query validation: smart result driven validation and ranked validation by query suitability. Furthermore, it shows the average number of created candidate queries  $Q_c$  for each query type and the average number of valid queries identified when having all tuples from  $R'$  available.

Figure 10 compares average number of executed queries with our two approaches to validation with the expected number of query validations if the candidate queries are not ordered. For  $\max(A)$  queries, we observe that smart validation outperforms unordered validation by a factor of 7.3 with  $|P| = 1$ , 4.2 with  $|P| = 2$ , and 3.3 with  $|P| = 3$ . Furthermore, smart validation performs 26% query executions less than ranked validation with  $|P| = 2$  and 33% less executions for  $|P| = 3$ . The benefits with discovering  $\text{sum}(A+B)$  queries are even greater. Thus, smart validation in average reduces the number of expected query executions by a fac-

P	Sample %	select $A_e, \max(A)$				select $A_e, \text{sum}(A + B)$			
		Smart	Ranked	# candidates	# valid $Q$	Smart	Ranked	# candidates	# valid $Q$
1	5	20.6	24.6	163.7		16.6	32.1	11621.9	
1	10	13.7	12.4	185.1		24.9	28.2	10919.9	
1	20	5.1	3.4	144.7		9.8	16.2	10330.7	
1	30	3.6	2.0	105.1		4.3	6.6	7287.4	
1	100	1.0	<b>1.0</b>	4.8	4.0	1.1	<b>1.1</b>	4.8	4.0
2	5	<b>33.1</b>	69.8	161.3		<b>100.9</b>	1379.0	6540.4	
2	10	<b>23.4</b>	40.4	219.3		<b>47.4</b>	958.4	6991.2	
2	20	9.3	12.8	155.5		<b>20.4</b>	362.8	6605.4	
2	30	6.4	8.7	130.1		10.5	49.4	4820.4	
2	100	1.3	<b>1.3</b>	12.9	4.8	1.2	<b>1.2</b>	5.8	3.7
3	5	<b>59.4</b>	121.0	219.4		<b>199.0</b>	2510.6	3802.5	
3	10	<b>49.5</b>	129.4	282.0		<b>133.8</b>	982.4	4524.0	
3	20	<b>24.8</b>	56.4	224.0		<b>22.7</b>	61.4	3263.0	
3	30	20.3	31.8	203.5		15.4	38.5	4457.1	
3	100	2.8	<b>2.8</b>	25.7	3.0	2.1	<b>2.1</b>	4.4	1.5

Table 7: Number of candidate query validations with the different approaches by sample and predicate size for  $\max(A)$  and  $\text{sum}(A + B)$  queries

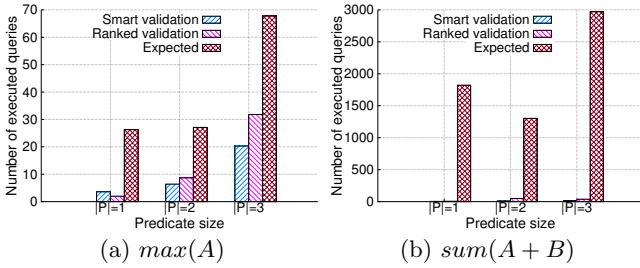


Figure 10: Number of query executions until first valid query with 30% sample for TPC-H data

tor of 424.7 with  $|P| = 1$ , 124.7 with  $|P| = 2$ , and 192.6 with  $|P| = 3$ . The greater benefit with this type of queries stems from the fact that identifying the ranking criteria involves different combinations of columns, which significantly increases the number of candidate queries.

Furthermore, smart validation significantly outperforms ranked validation with smaller sample size. Thus, with sample size of 5% smart validation reduces the number of query executions for discovering  $\text{sum}(A+B)$  queries over the rank-based validation by a factor of 13.7 and 12.6, with  $|P| = 2$  and  $|P| = 3$  respectively. Similarly, with a sample size of 10% smart validation reduces the average number of executions by a factor of 20.2 with  $|P| = 2$  and 7.3 with  $|P| = 3$ . We observe that smart candidate query validation improves over rank-based validation for  $\max(A)$  queries as well, albeit with smaller but still significant effect. The greater benefit with  $\text{sum}(A + B)$  queries stems from the fact that identifying the ranking criteria is more complex with this type of queries, thus making the query suitability less precise.

Larger sample size improves the candidate query suitability and reduces the number of candidate queries, thus resulting in less query validations. Smart validation reduces the number of validations for discovering  $\max(A)$  queries with a sample size of 30% by an average factor of 4.6 over a sample of 5%. Less candidate queries are created with larger sample size, since the availability of more tuples leads to better generation of candidate predicates and we discuss this later using Figure 11. Larger sample size significantly improves the approximation in finding the ranking criteria with  $\text{sum}(A + B)$  queries and the factor of improvement amounts to 8.8 for the same sample sizes.

Larger predicate size increases the number of needed query validations. We observe that with a sample size of 30% discovering  $\max(A)$  queries requires 3.6 candidate query val-

idations with  $|P| = 1$ , 6.4 with  $|P| = 2$ , and 20.3 with  $|P| = 3$ . With the same sample size, discovering  $\text{sum}(A+B)$  queries needs 4.3, 10.5, and 15.4 query executions, with  $|P| = 1$ ,  $|P| = 2$ , and  $|P| = 3$  respectively. Queries with larger predicates are more selective, thus it is less probable that tuples selected by the valid predicate will be sampled. Additionally, subpredicates of a larger valid predicate can select the same tuples as the larger predicate, but in turn the smaller predicates are less selective which reduces their probability of being a false positive. Hence, candidate queries with smaller predicates can have higher query suitability.

Note that with sampling, the number of candidate queries for  $\max(A)$  queries is significantly lower than that of  $\text{sum}(A+B)$  queries, as shown in Table 7. With single column queries identifying the ranking criteria is an easier task and we can limit the number of columns to consider as candidates. With  $\text{sum}(A+B)$  queries on the other hand, the task of finding the ranking criteria involves combinations of two columns, thus making it more complicated. Furthermore, it is difficult to limit the number of column combinations to consider since a certain column with very large numbers (e.g.,  $\text{total\_price}$  in TPC-H) can dominate the sum. Hence, we consider all possible column combinations as candidate ranking criteria and rank them according to their approximated L1 distance.

**Identifying Candidate Predicates.** We study the effect of sample size on the number of created candidate predicates. We observe that the number of candidate predicates decreases with larger sample size. Larger sample size increases the probability of sampling larger number of tuples with a valid predicate, which in turn allows for stricter criteria in qualifying a predicate as candidate. Following the sampling probability in Section 6, with larger sample size we increased the ratio of covered entities in order to denote a predicate as a candidate. Thus, for sample size of 5%, the ratio of covered entities was set to 0.5, for 10% to 0.6, for 20% to 0.7, and to 0.8 for a sample size of 30%. Lower ratio avoids false negatives, but comes at the cost of increasing the number of false positives, since more predicates will qualify as candidates.

It is important to note that the experiments with sampling introduced expected variability. Depending on which tuples are sampled, the probability of the candidate predicates varies. Furthermore identifying the ranking criteria with  $\text{sum}(A + B)$  queries is influenced by the sampled tuples. **Example:** We ran five executions of the input list from the second query in Table 6 with  $k = 10$  and sample of 5%. As a best case a valid query is found after only 2 query execution, while 125 executed candidate queries were needed

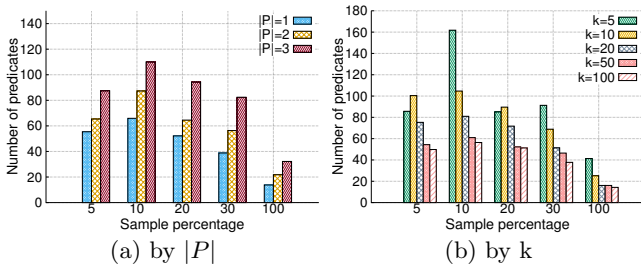


Figure 11: Number of candidate predicates for  $max(A)$  queries

in the worst case. In the first case, the sampled rows contain the correct predicate for each distinct entity, i.e., the predicate probability is 1.0. Additionally, the correct ranking criteria (column combination) has the second lowest L1 distance on the valid predicate. It seems that the sampled rows were good for approximating the ranking values for the correct columns. In the other case, the predicate probability is 0.84 (14th in the ranking), while the correct column combination has a very large L1 distance, since the sampled tuples were not a good approximation of the ranking values. The remaining executions resulted in 27, 16, and 39 query validations. With smaller sample it is more difficult to find the ranking criteria for  $sum(A + B)$  queries. This is a consequence of the non-uniform distribution of the values in  $A$  and  $B$ . Thus, the approximation depends on which tuples are sampled. Larger sample size mediates this problem. Having more tuples avoids the dependence on which tuples are sampled and leads to a more precise approximation.

## 8.2 Lessons Learned

With all tuples from  $R'$  available our system **always** discovers a valid query. Furthermore, for both datasets this is done efficiently and requires just a few query executions with only a single query validation for 76% and 68% of the top-k lists that stem from  $sum(A + B)$  and  $max(A)$  queries, respectively. On the other hand, sampling introduces the possibility of false negatives. However, we manage to discover a valid query for all top-k lists that result from a single column query. Finding valid  $sum(A+B)$  queries is more difficult and we manage to identify a valid query for 96.7% of the top-k lists with a sample size of 30%. Identifying the candidate predicates and ranking criteria is done in-memory and are very efficient. The smart result driven candidate query validation significantly reduces the number of query executions needed in finding a valid query. In addition, larger predicate size leads to more query validations. Larger sample size reduces both the number of false positives and false negatives in the candidate predicates. Furthermore, having more data improves the ranking of the candidate ranking criteria, since we have better approximation of the L1 distance.

## 9. CONCLUSION AND OUTLOOK

We proposed a framework to reverse engineer top-k OLAP queries. This has turned out a complex problem given the various dimensions of the search space, the potentially very large base relation, and the small input snippet in form of a top-k list. Our approach mainly operates on a subset of the base relation, held in memory, and further uses data samples, histograms, and simple descriptive statistics to identify potentially valid queries (that generate the input list). We proposed a probabilistic model that evaluates the suitability of a query discovered over a subset of  $R'$ , methodology that is directly applicable to the case of handling variations of  $R$  and considering partial match queries, i.e., queries that only

approximately match the input list. In any case, when trying to identify promising queries, the main difficulty is to limit the number of false positives—that cause unnecessary query validations—as well as to limit false negatives—that cause loss in recall. The ordering of potentially valid queries according to the probabilistic model in addition to an iterative refinement of the validation of candidate queries was proven to drastically decrease the amount of time to validate (or invalidate) queries in the final stage of the approach. This is specifically true for cases of low sampling rates—and expectedly likewise for partial-match scenarios.

As ongoing work, we investigate whether existing work on reverse engineering *join queries* is compatible with our approach and evaluate PALEO in partial-match scenarios.

## 10. REFERENCES

- [1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. *VLDB*, 1994.
- [2] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. *ICDE*, 2002.
- [3] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. *ICDE*, 2007.
- [4] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. Qagen: generating query-aware test databases. *SIGMOD*, 2007.
- [5] L. Blunski, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger. SODA: generating SQL for business users. *PVLDB*, 5(10), 2012.
- [6] N. Bruno, S. Chaudhuri, and D. Thomas. Generating queries with cardinality constraints for DBMS testing. *IEEE Trans. Knowl. Data Eng.*, 18(12), 2006.
- [7] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM J. Discrete Math.*, 17(1), 2003.
- [8] R. Kimball. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. John Wiley, 1996.
- [9] C. Mishra, N. Koudas, and C. Zuzarte. Generating targeted queries for database testing. *SIGMOD*, 2008.
- [10] F. Psallidas, B. Ding, K. Chakrabarti, and S. Chaudhuri. S4: top-k spreadsheet-style search for query discovery. *SIGMOD* 2015.
- [11] Y. Rubner, C. Tomasi, and L. J. Guibas. A metric for distributions with applications to image databases. *ICCV*, 1998.
- [12] A. D. Sarma, A. G. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. *ICDT*, 2010.
- [13] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. *SIGMOD*, 2014.
- [14] The Star Schema Benchmark. <http://www.odbms.org/2014/03/star-schema-benchmark/>
- [15] S. Tata and G. M. Lohman. SQAK: doing more with keywords. *SIGMOD*, 2008.
- [16] TPC. TPC benchmarks. <http://www.tpc.org/>
- [17] Q. T. Tran, C. Chan, and S. Parthasarathy. Query by output. *SIGMOD*, 2009.
- [18] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Norvag. Reverse top-k queries. *ICDE*, 2010.
- [19] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. *SIGMOD*, 2013.



# CrowdSky: Skyline Computation with Crowdsourcing

Jongwuk Lee  
Hankuk University of Foreign  
Studies, Republic of Korea  
julee@hufs.ac.kr

Dongwon Lee  
The Pennsylvania State  
University, PA, USA  
dongwon@psu.edu

Sang-Wook Kim  
Hanyang University, Republic  
of Korea  
wook@hanyang.ac.kr

## ABSTRACT

In this paper, we propose a crowdsourcing-based approach to solving skyline queries with incomplete data. Our main idea is to leverage crowds to infer the pair-wise preferences between tuples when the values of tuples in some attributes are unknown. Specifically, our proposed solution considers three key factors used in existing crowd-enabled algorithms: (1) minimizing a *monetary cost* in identifying a crowdsourced skyline by using a *dominating set*, (2) reducing the number of rounds for *latency* by parallelizing the questions asked to crowds, and (3) improving the *accuracy* of a crowdsourced skyline by dynamically assigning the number of crowd workers per question. We evaluate our solution over both simulated and real crowdsourcing using the Amazon Mechanical Turk. Compared to a sort-based baseline method, our solution significantly minimizes the monetary cost, and reduces the number of rounds up to two orders of magnitude. In addition, our dynamic majority voting method shows higher accuracy than both static majority voting method and the existing solution using unary questions.

## 1. INTRODUCTION

In recent years, crowdsourcing has become a new paradigm for implementing human computation. Many extensions to existing DBMS techniques have been proposed to employ crowdsourcing so that the problems difficult for machines but easier for humans can be solved better than ever, *e.g.*, CrowdDB [5], Qurk [13], Deco [19], AskIt! [1], and CyLog/Crowd4U [16]. Motivated by their success, ones attempt to leverage crowdsourcing into existing micro-level query operations. For instance, they are selection [6, 18, 21], join [14, 24, 25, 26], group-by [4], max [8, 22, 23], and sort [14].

In this paper, we study a crowdsourcing-based approach to solving *skyline queries* with incomplete data. The skyline queries have gained considerable attention for assisting multi-criteria decision making applications [2, 9, 17]. Given two tuples  $s$  and  $t$ , it is said that  $s$  *dominates*  $t$  if the values of  $s$  are no worse than those of  $t$  over all attributes and the values of  $s$  are better than those of  $t$  over any attribute. Given a set  $\mathcal{R}$  of tuples, the *skyline* is a set of tuples that are not dominated by any other tuples in  $\mathcal{R}$ . To illustrate this, we consider the following motivating example.

**EXAMPLE 1 (SKYLINE QUERY)** Suppose that Alice wants to find the skyline movies of her preference, *i.e.*, most popular and most romantic movies, released in 2010-2015.

```
SELECT * FROM movie_db
WHERE year >= 2010 and year <= 2015
SKYLINE OF box_office MAX, romantic MAX
```

The popularity can be estimated by the `box_office` (*i.e.*, the number of movie audiences) attribute. However, the `movie_db` table does not record how romantic a movie is. That is, in the `romantic` attribute, the values of tuples are all unknown (or missing). To address this problem, we utilize crowdsourcing that can effectively infer missing values on the subjective attribute. Specifically, we ask crowds which movie is more *romantic* with respect to two movies, and get a relative preference of movies by using a *pair-wise question*. By repeatedly asking such pair-wise questions and aggregating their answers, we can fill missing values of tuples, and then find a skyline. Note that our setting above is an extreme case (*i.e.*, all values of tuples are missing in the `romantic` attribute). When some values of tuples are missing, we can apply our proposed solution to only the tuples with missing values.  $\square$

A *crowd-enabled skyline query* is defined as a skyline query with incomplete data in which crowds are used to infer the missing preferences between tuples [12]. Although existing work [12] addresses crowd-enabled skyline queries, it assumes a fixed budget and computes a *probabilistic* skyline. In contrast, our goal is minimize the number of pair-wise questions to crowds in identifying a *complete* skyline. In addition, [12] is based on *unary* questions to assess missing values of tuples. Because it is difficult to obtain correct answers for unary questions, the skyline result in [12] can be inaccurate. In our empirical study, it is observed that the pair-wise questions achieve higher accuracy than the unary questions in existing work [12].

In order to address skyline queries with crowdsourcing, we deal with three key factors: (1) how to minimize rewards paid to crowds (*i.e.*, *monetary cost*), (2) how to reduce the delay of crowd-enabled computation (*i.e.*, *latency*), and (3) how to improve the quality of the skyline using the answers obtained from crowds (*i.e.*, *accuracy*). When asking questions to crowds, we have to pay certain monetary rewards. Assuming that a fixed amount of a reward per question is paid, the monetary cost is proportional to the number of questions asked to crowds. In order to measure the latency, we need to estimate the running time to obtain answers from crowds in a *round*. Assuming that each round has a fixed amount of time [25], the latency is proportional to the number of rounds needed for asking all questions.

We then propose a new solution that addresses crowd-enabled skyline queries for each factor. First, in order to minimize the monetary cost, it is essential to remove unnecessary questions while

computing a crowdsourced skyline. Toward this goal, we make use of two relationships between tuples, *dominance* and *incomparability*. In particular, we adopt a *dominating set* to remove unnecessary questions in identifying a skyline. That is, given a tuple  $t$ , the dominating set  $DS(t)$  is a set of tuples that dominate  $t$ . Using  $DS(t)$ , we can skip the questions for tuples with the *incomparability* relationship, and selectively ask the questions for tuples with the *dominance* relationships. We also prune unnecessary questions by using the *transitivity* of dominance relationships in  $DS(t)$ .

Second, we address how to reduce the number of rounds by asking multiple *independent* questions in a round. A set of questions are said to be independent if the answers of each question do not affect the other questions in the set. We show that independent questions can be asked in a round, yielding the *parallelization* of asking questions. Based on this observation, we develop two parallelization methods that significantly reduce the number of rounds without influencing other factors.

Lastly, we explain how to improve the accuracy of a crowdsourced skyline. When a single crowd worker is assigned per question, some answers can be erroneous as workers can make mistakes. To alleviate this problem, we assign multiple workers per question and decide the final answer by using *majority voting* [6, 11, 15, 18]. As the simplest method, we can *equally* assign the same number of workers for each question. Because it neglects the characteristics of skyline queries, however, we develop a *dynamic* assignment strategy in which the number of workers can be assigned differently depending on the *importance* of questions. The proposed dynamic assignment can improve the accuracy of the crowdsourced skyline without incurring additional monetary cost.

To summarize, our main contributions are as follows:

- We formulate the problem of crowd-enabled skyline queries with three key factors used in crowd-enabled algorithms.
- To minimize monetary cost, we propose a crowd-enabled skyline algorithm with three pruning methods on top of the notion of a dominating set.
- To reduce the number of rounds, we develop two parallelization methods based on the notion of dominating sets and skyline layers.
- To improve the accuracy of the crowdsourced skyline, we design a dynamic majority voting that assigns the number of workers depending on the importance of questions.
- We validate the effectiveness of our proposed algorithm in both simulated and real-life crowdsourcing using the Amazon Mechanical Turk. In terms of accuracy, we also compare our proposed solution against existing work [12].

The remainder of this paper is organized as follows. In Section 2, we explain the concept of crowd-enabled algorithms and formulate the skyline query with crowdsourcing. In Section 3, we first propose an algorithm to minimize monetary cost using several pruning methods. In Section 4, we develop two algorithms to minimize the number of rounds by parallelizing multiple questions in a round. In Section 5, we design a dynamic majority voting that improves a static majority voting by considering the importance of questions. In Section 6, we empirically compare our proposed algorithm with the baseline and existing crowdsourced skyline algorithms with unary questions. In Section 7, we review our related work. In Section 8, we summarize and conclude our work.

## 2. PRELIMINARIES

In this section, we first present the concept of crowd-enabled algorithms (Section 2.1), and then explain the basic notion of skyline queries (Section 2.2). We lastly formulate the problem of skyline queries with crowdsourcing, where questions are asked to crowds to acquire the relative preferences between tuples with missing values (Section 2.3).

### 2.1 Crowd-Enabled Algorithms

The crowd-enabled algorithms first require us to design the format of *micro-tasks* asked to crowds. Because the micro-tasks are typically represented as *questions*, we use both terms, micro-tasks and questions, interchangeably. Let  $\pi$  denote a *latent scoring function* with which crowds assess the missing value of a tuple. The argument of  $\pi$  can differ depending on formats of questions.

Specifically, the micro-tasks are classified into *quantitative* and *qualitative* formats [14]. First, the quantitative format asks crowds to determine an *absolute* (normalized) preference. This can be abstracted as a *unary function*  $\pi(t)$  for a tuple  $t$ , e.g., rating from 1 to 7 for the size of a given square. Let  $n$  denote the number of tuples with missing values. While the unary function is effective for minimizing the number of questions in determining the total order of tuples, i.e.,  $n$  questions, the accuracy of answers can be low. That is, because crowds usually have no *global* knowledge for missing values of tuples, it is difficult for them to return correct answers.

Second, the qualitative format allows crowds to judge the *relative* preference between two (or more) tuples. As the simplest format, it can be modeled as a *binary function*  $\pi(s, t)$  to compare two tuples  $s$  and  $t$ , e.g., selecting one with a larger size between two squares. (Possibly, it can be extended to an  $m$ -ary format.) Compared to the quantitative format, because it only requires relative preference between two tuples, more questions are usually asked. In the worst case,  $\binom{n}{2}$  questions are needed for obtaining a total order of  $n$  tuples. Meanwhile, because the crowd can answer binary questions correctly without global knowledge for missing values, the accuracy of answers in the qualitative format is higher than that in the quantitative format.

In this paper, we adopt binary function  $\pi(s, t)$  for questions in order to obtain more accurate answers. That is, we use a *pair-wise question*  $(s, t)$  with *ternary* answers, where it is *symmetric*, i.e.,  $(s, t) = (t, s)$ . Given  $(s, t)$ , the crowd chooses a *more* preferred one (i.e., either  $s$  or  $t$ ) or the third option, indicating that the two tuples are *equally* preferred. For example, the following questions are asked to crowds: “which square between the two is larger?” or “who is a more valuable baseball player?”

We next explain three key factors used in existing crowd-enabled algorithms, e.g., [4, 6, 18, 20, 22, 25]. Using the key factors, we formulate the problem of crowd-enabled skyline queries (Section 2.3).

(1) **Monetary cost:** Unlike existing machine-only algorithms, crowd-enabled algorithms compensate rewards to crowds. Assuming that a fixed amount of a reward per question is paid, the monetary cost is proportional to the number of questions asked to crowds. For monetary cost, there are two optimization directions: (1) minimizing the number of questions asked for obtaining a complete query result and (2) selecting the most important questions for a given budget. In this paper, we focus on minimizing the total number of questions during executing a skyline query with crowdsourcing.

(2) **Latency:** Since each question can take different time to finish, it is non-trivial to design an effective model for estimating latency. As an alternative way, we assume that a fixed amount of time is assigned per question. Because multiple questions can be performed in parallel, we use the number of *rounds* (or *iterations*.) to measure

the latency [25]. Specifically, there are two strategies in latency: (1) *one-shot strategy* that generates all questions at once, and (2) *adaptive strategy* that asks questions in an interactive manner. Because the one-shot strategy needs only one round, it is much faster than the adaptive strategy. Meanwhile, the adaptive strategy can identify unnecessary questions by using the answers of questions asked at the previous rounds, and thus reduces the total number of questions. In this paper, we leverage the adaptive strategy and discuss how to minimize the latency for a given question set.

(3) **Accuracy:** Because crowds can make mistakes in answering questions, the accuracy of a query result can be imperfect. There are two models to improve the accuracy: *query-independent* and *query-dependent* models. Existing work [6, 11, 15, 18] proposed various query-independent methods to improve the accuracy for a single question by considering the proficiency of workers and the difficulty of questions. Although improving the accuracy of a query result in a micro-level manner, they do not consider the *importance* of questions depending on an inherent property of a query type in a macro-level way, *e.g.*, [4, 20]. In this paper, we aim to develop a query-dependent method by distinguishing the importance of questions for a skyline query.

## 2.2 Skyline Queries with Missing Data

Let  $\mathcal{A}$  denote a finite set of  $d$  attributes,  $\mathcal{A} = \{A_1, \dots, A_d\}$ , in which the domain  $D_i$  of  $A_i$  is a set of positive numbers and a *missing value*, denoted by  $\square$ , *i.e.*,  $D_i := \mathbb{R}^+ \cup \{\square\}$ . A base dataset  $\mathcal{R}$  is an instance of database relations, *i.e.*,  $\mathcal{R} \subseteq D_1 \times \dots \times D_d$ . Each tuple  $t \in \mathcal{R}$  is represented by  $t = (t_1, \dots, t_d)$  such that  $t_i \in D_i$  for  $i = 1, \dots, d$ . In this paper, we use  $s, t, u$ , and  $v$  to point out arbitrary tuples.

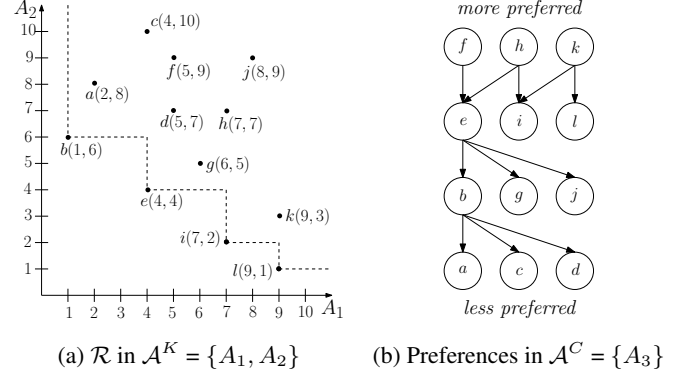
The attribute set  $\mathcal{A}$  is divided into two attribute subsets. First,  $\mathcal{A}^K$  is a set of attributes in which the values of tuples are *known*. The preference of values over  $\mathcal{A}^K$  can be represented by a total order. Second,  $\mathcal{A}^C$  is a subset of attributes in  $\mathcal{A}$ , where the values of tuples are *missing*. We call  $\mathcal{A}^C$  *crowd attributes*. Two attribute sets are *disjoint*, *i.e.*,  $\mathcal{A}^K \cap \mathcal{A}^C = \emptyset$ ,  $\mathcal{A}^K \cup \mathcal{A}^C = \mathcal{A}$ . We assume that all values of tuples in  $\mathcal{A}^C$  are missing, *i.e.*, *hand-off crowdsourcing* [7]. That is, for any tuple  $t \in \mathcal{R}$ ,  $\forall A_j \in \mathcal{A}^C : t_j = \square$  holds. This implies that all preferences between tuples in  $\mathcal{A}^C$  should be assessed by crowds. After the missing preferences between tuples in  $\mathcal{A}^C$  are judged by crowds, it can be represented by a *partial* order of tuples. When a subset of tuples in  $\mathcal{R}$  only has missing values in many real applications, some known values of tuples can be represented by a pre-defined partial order. Therefore, we can extend our proposed techniques for real-life scenarios.

We next define fundamental notions used in skyline literatures [2, 9, 17]. In this paper, we assume that smaller values over  $\mathcal{A}^K$  are more preferred. Given  $s, t \in \mathcal{R}$ , a *strict preference*  $s <_i t$  is defined if  $s$  is preferred over  $t$  in  $A_i$ . Let  $s \lesssim_i t$  define a *weak preference* if  $s <_i t$  or  $s =_i t$  in  $A_i$ . If no preference between  $s$  and  $t$  is inferred in  $A_i \in \mathcal{A}^C$ , an *indifferent preference*  $s \perp_i t$  is defined.

**DEFINITION 1 (DOMINANCE)** Given  $s, t \in \mathcal{R}$ ,  $s$  *dominates*  $t$  in  $\mathcal{A}$ , denoted by  $s \prec_{\mathcal{A}} t$ , if  $\forall A_i \in \mathcal{A} : s \lesssim_i t$  and  $\exists A_j \in \mathcal{A} : s <_j t$ . If  $s$  does not dominate  $t$  in  $\mathcal{A}$ , it is denoted as  $s \not\prec_{\mathcal{A}} t$ .

**DEFINITION 2 (INCOMPARABILITY)** Given  $s, t \in \mathcal{R}$ , they are *incomparable* in  $\mathcal{A}$ , denoted by  $s \not\prec_{\mathcal{A}} t$ , if (i)  $s \not\prec_{\mathcal{A}} t$  and  $t \not\prec_{\mathcal{A}} s$  or (ii)  $\exists A_j \in \mathcal{A}^C : s \perp_j t$ .

**DEFINITION 3 (SKYLINE)** Given a set  $\mathcal{R}$  of tuples, a *skyline* is a set of tuples that are not dominated by any other tuples in  $\mathcal{A}$ , *i.e.*,  $\text{SKY}_{\mathcal{A}}(\mathcal{R}) = \{t \in \mathcal{R} | \forall s \in \mathcal{R} : s \not\prec_{\mathcal{A}} t\}$ .



**Figure 1: A toy dataset for a crowdsourced skyline query with  $\mathcal{A} = \{A_1, A_2, A_3\}$**

These notions can be applied to a subset of  $\mathcal{A}$ . That is,  $\prec_{\mathcal{A}}$  in  $\prec_{\mathcal{A}}$ ,  $\not\prec_{\mathcal{A}}$ , and  $\prec \succ_{\mathcal{A}}$  can be replaced with  $\mathcal{A}^K$  or  $\mathcal{A}^C$ .

## 2.3 Problem Formulation

The earlier work [12] maximized the accuracy of the crowdsourced skyline at a fixed budget. When tuples have missing values, [12] adopts crowds to assess missing values of tuples with a unary question. In contrast to [12], in order to achieve more reliable skyline results, our goal is to minimize monetary cost while computing crowd-enabled skyline queries in which binary questions are used for obtaining the preferences between tuples. (In Section 7, we explain the differences between ours and [12].)

When the crowds evaluate missing values of tuples, a skyline result can be iteratively updated. Initially, since the preferences of tuples over  $\mathcal{A}^C$  are undefined, all tuples are incomparable to each other, and by default in the skyline. Let  $Q(t) = \{(s, t) | s \in \mathcal{R} \setminus \{t\}\}$  be a set of all possible pair-wise questions for  $t$ . After a question set  $Q'(t) \subseteq Q(t)$  is answered by crowds,  $s$  can exist such that  $s \prec_{\mathcal{A}} t$ . At this point,  $t$  becomes a non-skyline tuple, and is removed from the initial skyline. If the status of  $t$  is not changed regardless of remaining questions for  $t$ , it is called a *complete* tuple. That is, once  $t$  is determined as a complete tuple, additional questions for  $t$  are *unnecessary*.

**DEFINITION 4 (COMPLETE TUPLE)** After a set of questions  $Q'(t) \subseteq Q(t)$  is answered, a tuple  $t$  is said to be *complete* if (i)  $\exists s \in \mathcal{R} : s \prec_{\mathcal{A}} t$ , (*i.e.*, a *complete non-skyline tuple*) or (ii)  $\forall s \in \mathcal{R} : s \not\prec_{\mathcal{A}} t$  holds regardless of the answers of remaining questions  $Q(t) - Q'(t)$  for  $t$ , (*i.e.*, a *complete skyline tuple*).

**EXAMPLE 2 (CROWD-ENABLED SKYLINE QUERY)** Given a set  $\mathcal{R}$  of 12 tuples, Figure 1 depicts a toy dataset for a crowd-enabled skyline query such that  $\mathcal{A}^K = \{A_1, A_2\}$  and  $\mathcal{A}^C = \{A_3\}$ . Note that all values of  $\mathcal{R}$  in  $A_3$  are missing. Since  $\{b, e, i, l\}$  (a dashed line in Figure 1a) are not dominated by any other tuples in  $\mathcal{A}^K$ , they are always in the skyline in  $\mathcal{A}$  regardless of preferences of tuples in  $\mathcal{A}^C$ . Therefore, they are complete skyline tuples. Because the other tuples can be non-skyline tuples depending on the results of questions on  $\mathcal{A}^C$ , however, they are regarded to be incomplete tuples. Suppose that the preferences of tuples in  $\mathcal{A}^C$  are depicted in Figure 1(b), where an edge  $s \rightarrow t$  indicates that  $s$  is preferred over  $t$  in  $\mathcal{A}^C$  and transitivity between edges holds. (In Section 3.3, we discuss how to build a preference tree in  $\mathcal{A}^C$ .) Since  $b \prec_{\mathcal{A}^K} a$  and  $b \prec_{\mathcal{A}^C} a$ ,  $a$  becomes a non-skyline tuple. By using the preference relationships in Figure 1(b), all tuples become complete tuples, and the skyline is finally identified as  $\{b, e, i, l, k, f, h\}$ .  $\square$

In this paper, we consider three key factors, *i.e.*, monetary cost, latency, accuracy, in computing crowd-enabled skyline queries. We first aim to minimize the number of questions in identifying a complete skyline. Then, we further optimize the other factors for a given question set. Although this problem formulation does not identify a solution that optimizes three factors simultaneously, it can be a more practical setting as done in [24, 25]. Formally:

**PROBLEM 1 (CROWDSOURCED SKYLINE)** Let  $\mathcal{P}$  be a set of possible *execution plans* for computing a complete skyline. Our problem is to identify an execution plan  $P_{opt} \in \mathcal{P}$  that minimizes monetary cost  $C(P, \mathcal{R})$  for questions in  $\mathcal{A}^C$ .

$$P_{opt} = \operatorname{argmin}_{P \in \mathcal{P}} C(P, \mathcal{R})$$

Note that  $P_{opt}$  can also be used for optimizing the other factors such as latency and accuracy. ■

Assuming that a single question is asked at each round, execution plan  $P$  can be represented by a sequence set  $\mathcal{Q}$  of questions, *e.g.*,  $\mathcal{Q} = \langle (a, b), \dots, (k, l) \rangle$ . Let  $|\mathcal{Q}|$  denote the number of questions in  $\mathcal{Q}$ . When the monetary cost per question is equal,  $C(P, \mathcal{R})$  becomes proportional to  $|\mathcal{Q}|$ . Therefore, we minimize the number of questions  $|\mathcal{Q}|$  in identifying a complete skyline tuple.

In the following sections, we first assume that the answers of crowds are always correct. Based on this assumption, we propose a crowd-enabled skyline algorithm to minimize  $|\mathcal{Q}|$  (Section 3). When multiple questions are asked in parallel, we also develop how to reduce the number of rounds for  $\mathcal{Q}$  (Section 4). By relaxing such an assumption, we lastly discuss how to improve the accuracy of a complete skyline while both the number of questions and the latency are kept (Section 5).

### 3. MINIMIZING MONETARY COST

As a baseline method, we may ask all possible pair-wise questions between tuples, *i.e.*,  $\binom{n}{2}$  to obtain the total order of tuples. However, because it is too exhaustive, we modify existing sorting algorithms such as *tournament sort* and *bitonic sort* [3] with crowdsourcing. Specifically, the pair-wise comparisons in existing sorting-based algorithms [3] can be replaced by binary questions, and the total order of tuples in  $\mathcal{A}^C$  can be used to identify a crowd-sourced skyline. While the sorting-based method is effective for obtaining all missing preferences of tuples, it can incur unnecessary questions in computing the crowd-enabled skyline.

To address this problem, it is observed that the *dominance* and *incomparability* relationships of tuples in  $\mathcal{A}^K$  can be used to reduce unnecessary questions. Based on this observation, we adopt the notion of a *dominating set* to remove unnecessary questions. In the following sections, we also develop several pruning methods on top of the dominance set to remove additional questions.

For a simpler presentation, when  $\mathcal{A}^C$  has multiple attributes, *i.e.*,  $m = |\mathcal{A}^C| > 1$ , we suppose that  $m$  questions for  $(s, t)$  are asked at once, *i.e.*,  $\forall A_j \in \mathcal{A}^C : (s_j, t_j)$ . Because  $m$  questions can be asked simultaneously, we simply use  $(s, t)$  to denote  $m$  questions for  $(s, t)$ , when context is clear.

In addition, we suppose that the values of tuples in  $\mathcal{A}^K$  are distinct, *i.e.*, for any tuples  $s, t \in \mathcal{R}$ ,  $\exists A_i \in \mathcal{A}^K : s_i \neq t_i$  holds. As a degenerate case, we separately handle a tuple set with the same values in  $\mathcal{A}^K$ . Because we cannot exploit our pruning methods for the tuple set, it is performed as a pre-processing step. That is, given  $(s, t)$  such that  $\forall A_j \in \mathcal{A}^K : s_j = t_j$ , we remove a non-skyline tuple by identifying the dominance relationship between two tuples in  $\mathcal{A}^C$ . (This degenerate case is described in lines 1–3 in Algo-

**Table 1: Dominating sets and question sets for the toy dataset in Figure 1(a)**

$t$	$DS(t)$	$t$	$Q(t)$
$a$	$\{b\}$	$a$	$\{(a, b)\}$
$c$	$\{a, b, e\}$	$c$	$\{(c, a), (c, b), (c, e)\}$
$d$	$\{b, e\}$	$d$	$\{(d, b), (d, e)\}$
$f$	$\{a, b, d, e\}$	$f$	$\{(f, a), (f, b), (f, d), (f, e)\}$
$g$	$\{e\}$	$g$	$\{(g, e)\}$
$h$	$\{b, d, e, g, i\}$	$h$	$\{(h, b), (h, d), (h, e), (h, g), (h, i)\}$
$j$	$\{a, b, d, e, f, g, h, i\}$	$j$	$\{(j, a), (j, b), (j, d), (j, e), (j, f), (j, g), (j, h), (j, i)\}$
$k$	$\{i, l\}$	$k$	$\{(k, i), (k, l)\}$

(a) Dominating sets

(b) Question sets

gorithm 1.) After performing the pre-processing, we can safely make use of our pruning methods without loss of correctness.

#### 3.1 Using a Dominating Set

We first exploit the incomparability relationship of tuples to bypass unnecessary questions. For instance, when two tuples  $a = (2, 8, \square)$  and  $d = (5, 7, \square)$  in Figure 1(a) are incomparable in  $\mathcal{A}^K = \{A_1, A_2\}$ , they also become incomparable in  $\mathcal{A}$  regardless of the answer of  $(a, d)$  in  $\mathcal{A}^C = \{A_3\}$ . That is, we only need to compare a question  $(s, t)$  in  $\mathcal{A}^C$  by asking a question  $(s, t)$  if  $s$  and  $t$  are *not incomparable* in  $\mathcal{A}^K$  by using the property of sharing incomparability [10].

Based on this property, we adopt a *dominating set*  $DS(t)$  for a tuple  $t \in \mathcal{R}$  as a set of candidates that can affect the dominance relationship of  $t$  in  $\mathcal{A}$ . That is, the dominance set ensures that only the questions  $Q(t) = \{(s, t) | s \in DS(t)\}$  are enough to generate the dominance relationship between  $s$  and  $t$  in  $\mathcal{A}$ .

**DEFINITION 5 (DOMINATING SET)** A *dominating set*  $DS(t)$  of a tuple  $t \in \mathcal{R}$  is a set of tuples that dominate  $t$  in  $\mathcal{A}^K$ , *i.e.*,  $DS(t) = \{s \in \mathcal{R} | \forall s \in \mathcal{R} \setminus \{t\} : s \prec_{\mathcal{A}^K} t\}$ .

**LEMMA 1** Given  $t \in \mathcal{R}$  and  $s \notin DS(t)$ ,  $(s, t)$  is unnecessary in  $Q(t)$ .

**PROOF.** We prove this by contradiction. Assume that a question  $(s, t)$  exists that  $s \notin DS(t)$  dominates  $t$  in  $\mathcal{A}$ . By Definition 5,  $s \notin DS(t)$  does not dominate  $t$  in  $\mathcal{A}^K$ . Because  $\mathcal{A}^K \subset \mathcal{A}$ ,  $s$  cannot dominate  $t$  in  $\mathcal{A}$  regardless of asking  $(s, t)$ . That is, we do not have to ask a question  $(s, t)$  to determine whether  $t$  is complete. This contradicts that asking question  $(s, t)$  is necessary for  $t$  to be complete.

**EXAMPLE 3 (DOMINATING SET)** Continue to use the toy dataset in Figure 1(a). Table 1(a) illustrates dominating sets for each tuple. The questions generated by dominating sets are shown in Table 1(b). As a result, the total number of questions (*i.e.*, 26 questions) is calculated as  $\sum_{t \in \mathcal{R}} |DS(t)|$ , where  $|DS(t)|$  is the number of tuples in  $DS(t)$ . □

#### 3.2 Pruning Questions for Non-skylines in $\mathcal{A}$

While sequentially generating  $(s, t)$  such that  $s \in DS(t)$ ,  $t$  can be determined as a complete tuple (by Definition 4). For *all* of the questions, if  $t$  is preferred to  $s$ ,  $t$  becomes a *complete skyline tuple*. On the other hand, if  $t$  is not preferred to  $s$  for *any* question,  $t$  becomes a *complete non-skyline tuple* and remaining questions in  $Q(t)$  can be skipped. Once  $s \prec_{\mathcal{A}} t$  is determined, it can also be used for removing additional unnecessary questions for any

**Table 2: Sorted dominating sets and question sets after removing  $a$  ( $\circ$ ),  $g$  ( $\setminus$ ), and  $d$  ( $\diagdown$ ), respectively**

$t$	$DS(t)$	$t$	$Q(t)$
$a$	$\{b\}$	$a$	$\{(a, b)\}$
$g$	$\{e\}$	$g$	$\{(g, e)\}$
$d$	$\{b, e\}$	$d$	$\{(d, b), (d, e)\}$
$k$	$\{i, l\}$	$k$	$\{(k, i), (k, l)\}$
$c$	$\{a, b, e\}$	$c$	$\{(c, a), (c, b), (c, e)\}$
$f$	$\{a, b, d, e\}$	$f$	$\{(f, a), (f, b), (f, d), (f, e)\}$
$h$	$\{b, d, e, g, i\}$	$h$	$\{(h, b), (h, d), (h, e), (h, g), (h, i)\}$
$j$	$\{a, b, d, e, f, g, h, i\}$	$j$	$\{(j, a), (j, b), (j, d), (j, e), (j, f), (j, g), (j, h), (j, i)\}$

(a) Sorted dominating sets

(b) Question sets

other tuple. That is, once a tuple  $t$  is determined as a complete non-skyline tuple, we can safely skip to ask all questions for  $t$  as unnecessary ones.

**LEMMA 2** If  $s \prec_{\mathcal{A}} u$  and  $u \in DS(t)$  hold, then  $s \in DS(t)$  also holds.

**PROOF.** By definition of  $DS(t)$ ,  $u \in DS(t) \Leftrightarrow u \prec_{\mathcal{A}^K} t$ , and  $\mathcal{A}$  is divided to two subsets  $\mathcal{A}^K$  and  $\mathcal{A}^C$ . When  $s \prec_{\mathcal{A}} u$ , there are three cases:

1.  $(s \prec_{\mathcal{A}^K} u) \wedge (s \prec_{\mathcal{A}^C} u)$ : since  $s \prec_{\mathcal{A}^K} u$  and  $u \prec_{\mathcal{A}^K} t$  hold,  $s \prec_{\mathcal{A}^K} t$  also holds by transitivity.
2.  $(s \prec_{\mathcal{A}^K} u) \wedge (s =_{\mathcal{A}^C} u)$ : since  $s \prec_{\mathcal{A}^K} u$  and  $u \prec_{\mathcal{A}^K} t$  hold,  $s \prec_{\mathcal{A}^K} t$  also holds by transitivity.
3.  $(s =_{\mathcal{A}^K} u) \wedge (s \prec_{\mathcal{A}^C} u)$ : since  $s =_{\mathcal{A}^K} u$  and  $u \prec_{\mathcal{A}^K} t$  hold,  $s \prec_{\mathcal{A}^K} t$  also holds.

In all cases,  $s \prec_{\mathcal{A}^K} t$  always holds. By definition of  $DS(u)$ ,  $s \in DS(t)$  also holds.

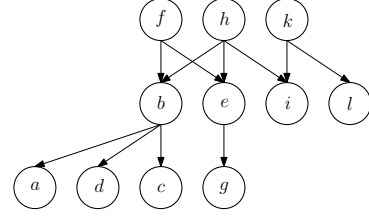
**COROLLARY 1** For a tuple  $t \in \mathcal{R}$ , if  $s \prec_{\mathcal{A}} u$  and  $u \in DS(t)$  hold, asking  $(t, u)$  is unnecessary in  $Q(t)$ .

**PROOF** When asking  $(s, t)$ , there exist two possible cases:

1.  $s \prec_{\mathcal{A}^C} t$  or  $s =_{\mathcal{A}^C} t$ : By Lemma 2,  $s \in DS(t)$  holds. Because  $s \prec_{\mathcal{A}^K} t$ ,  $s \prec_{\mathcal{A}} t$  holds. In that case,  $t$  becomes a complete non-skyline tuples regardless of asking  $(t, u)$ .
2.  $t \prec_{\mathcal{A}^C} s$ : Because  $s \prec_{\mathcal{A}} u$ ,  $u \not\prec_{\mathcal{A}^C} s$  holds. In that case, because  $u \not\prec_{\mathcal{A}^C} t$  always holds.

In both cases, asking  $(t, u)$  is unnecessary by asking  $(s, t)$ .  $\blacksquare$

In order to remove unnecessary questions for non-skyline tuples by Corollary 1, it is essential to identify complete non-skyline tuples as many as possible. That is, it is *optimal* if  $Q(t)$  is generated after all tuples in  $DS(t)$  become complete. Toward this goal, we decide the ordering of evaluating tuples in an *iterative* manner. We first identify  $\text{SKY}_{\mathcal{A}^K}(\mathcal{R})$  as complete tuples. Then, for a tuple  $t \in \mathcal{R}$ , we generate  $Q(t)$  with the following steps: (1) If any tuple in  $s \in DS(t)$  is a complete non-skyline tuple,  $s$  is removed from  $DS(t)$ ; (2) If all tuples in  $DS(t)$  are complete,  $Q(t)$  is generated from  $DS(t)$ ; (3)  $Q(t)$  is asked until  $t$  is determined as a complete tuple; (4) The complete tuple set is updated by appending  $t$ . We repeat this process until all tuples become complete.



**Figure 2: Preference tree  $\mathcal{T}$  in  $A_3$  until evaluating  $h$**

We now propose an efficient method that performs the iterative strategy. Given two tuples  $s$  and  $t$ ,  $Q(s)$  has to precede  $Q(t)$  if  $s \in DS(t)$ . It is found that this condition is always satisfied if tuples are evaluated by the ascending order of the size of dominating sets. That is, if  $s \in DS(t)$ , then  $|DS(s)| < |DS(t)|$  holds, implying that the size of the dominating set increases monotonically if the dominance relationship between tuples holds. That is, given  $s, t \in \mathcal{R}$ ,  $Q(s)$  such that  $s \in DS(t)$  is first asked before generating  $Q(t)$ . Formally:

**LEMMA 3** Given  $s, t \in \mathcal{R}$ , if  $|DS(t)| \geq |DS(s)|$ , then  $t \not\prec_{\mathcal{A}^K} s$  holds.

**PROOF** We prove this by contradiction. Assume that there are two tuples  $s$  and  $t$  such that  $t \prec_{\mathcal{A}^K} s$  and  $|DS(t)| \geq |DS(s)|$ . Because  $t \prec_{\mathcal{A}^K} s$ , two conditions hold: (1)  $t \in DS(s)$  and (2)  $DS(t) \subseteq DS(s)$ . By combining them,  $|DS(t)| < |DS(s)|$  holds, which contradicts the fact that  $|DS(t)| \geq |DS(s)|$ .  $\blacksquare$

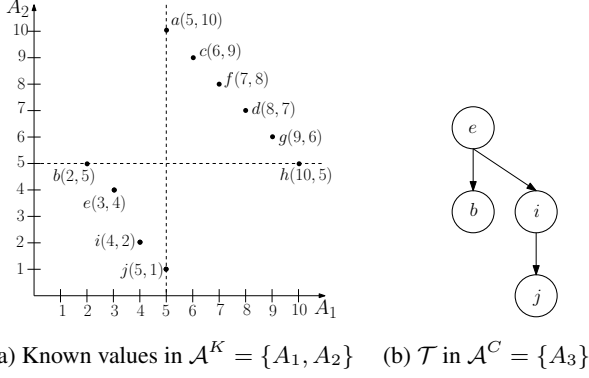
**EXAMPLE 4 (SORTING DOMINATING SETS)** We continue to use the toy dataset used in Example 3. Table 2(a) shows the dominating sets sorted by  $|DS(t)|$  in Table 1(a). According to the ordering of tuples in Table 2(a), the questions are sequentially generated from  $a$  to  $j$ . Assuming that  $\{a, g, d\}$  have been determined as non-skyline tuples, Table 2(b) illustrates questions for each tuple. When  $(a, b)$  is asked and  $a$  is identified as a non-skyline tuple (i.e.,  $b \prec_{\mathcal{A}} a$ ),  $a$  is removed from  $DS(c)$ ,  $DS(f)$ , and  $DS(j)$ . Similarly, after  $(g, e)$  is asked, if  $g$  is a non-skyline tuple,  $g$  is removed from  $DS(h)$  and  $DS(j)$ . As a result, it only generates 18 questions by pruning 8 questions, i.e.,  $\{(c, a), (f, a), (j, a)\}$ ,  $\{(h, g), (j, g)\}$ , and  $\{(f, d), (h, d), (j, d)\}$  for  $a, g$ , and  $d$ , respectively.  $\square$

### 3.3 Pruning Questions for Non-skylines in $\mathcal{A}^C$

When generating  $Q(t)$  from  $DS(t)$ , we can further reduce  $DS(t)$  to  $\text{SKY}_{\mathcal{A}^C}(DS(t))$ . For instance, after asking  $\{(f, b), (f, e)\}$  for  $f$ ,  $f$  has been determined as a complete skyline tuple, i.e.,  $f \prec_{\mathcal{A}^C} b$  and  $f \prec_{\mathcal{A}^C} e$ . In that case, the dominance relationship for  $f$  can be used for pruning questions for other tuples. For tuple  $j$ ,  $Q(j) = \{(j, b), (j, e), (j, f), (j, h), (j, i)\}$  for  $j$  is asked in Table 2(b). Because  $f \prec_{\mathcal{A}^C} \{b, e\}$ , it is better to ask  $(j, f)$  instead of asking  $(j, b)$  and  $(j, e)$ . If  $j \prec_{\mathcal{A}^C} f$  is obtained, we can infer  $j \prec_{\mathcal{A}^C} b$  and  $j \prec_{\mathcal{A}^C} e$  by transitivity. Based on this property, we can skip two questions  $(j, b)$  and  $(j, e)$  for  $Q(j)$ .

**LEMMA 4** For each tuple  $s \in \text{SKY}_{\mathcal{A}^C}(DS(t))$ , if  $t \prec_{\mathcal{A}^C} s$  holds, then  $t \prec_{\mathcal{A}^C} u$  such that  $u \in (DS(t) - \text{SKY}_{\mathcal{A}^C}(DS(t)))$  holds.

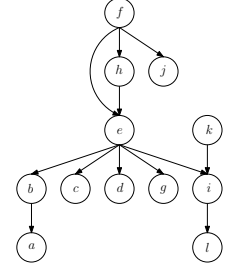
**PROOF.** By Definition 3, there exists that  $s \in \text{SKY}_{\mathcal{A}^C}(DS(t))$  dominates  $u \in (DS(t) - \text{SKY}_{\mathcal{A}^C}(DS(t)))$  in  $\mathcal{A}^C$ . As a result, if  $\forall s \in \text{SKY}_{\mathcal{A}^C}(DS(t)) : t \prec_{\mathcal{A}^C} s$  holds,  $t \prec_{\mathcal{A}^C} u$  holds by transitivity.



**Figure 3: An anti-correlated toy dataset for the crowd-enabled skyline query with  $\mathcal{A} = \{A_1, A_2, A_3\}$**

$t$	$P(t)$	$Q(t)$	$DS(t)$
$a$	-	$\{(a, b)\}$	$\{b\}$
$g$	-	$\{(g, e)\}$	$\{e\}$
$d$	$\{(b, e)\}$	$\{(d, e)\}$	$\{b, e\}$
$k$	$\{(i, l)\}$	$\{(k, i)\}$	$\{i, l\}$
$c$	-	$\{(c, e)\}$	$\{e\}$
$f$	-	$\{(f, e)\}$	$\{e\}$
$h$	$\{(e, i)\}$	$\{(h, e)\}$	$\{e, i\}$
$j$	$\{(f, h)\}$	$\{(j, f)\}$	$\{f, h\}$

(a) Questions asked per tuple



(b)  $\mathcal{T}$  in  $A_3$

**Figure 4: Overall procedure of probing dominating sets**

**COROLLARY 2** Given  $t \notin \text{SKY}_{\mathcal{A}^C}(DS(u))$ ,  $(t, u)$  is unnecessary in  $Q(u)$ .

**PROOF** For each  $s \in \text{SKY}_{\mathcal{A}^C}(DS(u))$ , if asking  $(s, u)$ , there are two cases:

1.  $s \prec_{\mathcal{A}^C} u$  or  $s =_{\mathcal{A}^C} u$ : Because  $s \prec_{\mathcal{A}^C} u$ ,  $s \prec_{\mathcal{A}} u$  holds, and  $(t, u)$  is not needed.
2.  $u \prec_{\mathcal{A}^C} s$ : Because  $s \prec_{\mathcal{A}^C} t$ ,  $u$  exists that  $u \prec_{\mathcal{A}^C} t$  by Lemma 4.

In both cases,  $(t, u)$  is not needed by asking  $(s, u)$ . ■

We now adopt a *preference tree*  $\mathcal{T}$  that visualizes the preferences of tuples in  $\mathcal{A}^C$  in order to compute  $\text{SKY}_{\mathcal{A}^C}(DS(t))$  efficiently. Each tuple  $t \in \mathcal{R}$  is represented by a node in  $\mathcal{T}$ . If  $s \prec_{\mathcal{A}^C} t$  holds, an edge  $s \rightarrow t$  exists. If there exists a path from  $s$  to  $t$  connected with multiple edges, then  $s \prec_{\mathcal{A}^C} t$  also holds by transitivity. If  $s \not\prec_{\mathcal{A}^C} t$ , there is no edge between  $s$  and  $t$ . After asking each question,  $\mathcal{T}$  is iteratively updated, and is used for identifying the dominance relationships by checking the path between tuples.

**EXAMPLE 5 (USING A PREFERENCE TREE)** Continuing from Example 4, Figure 2 depicts a snapshot of  $\mathcal{T}$  after evaluating  $h$  in  $A_3$ . After removing non-skyline tuples,  $DS(j)$  is  $\{b, e, f, h, i\}$ . By checking the dominance relationships between tuples in  $DS(j)$ ,  $f \prec_{\mathcal{A}^C} b$ ,  $f \prec_{\mathcal{A}^C} e$ , and  $h \prec_{\mathcal{A}^C} i$  can be found in  $\mathcal{T}$ . Therefore,  $\text{SKY}_{\mathcal{A}^C}(DS(j))$  is identified as  $\{f, h\}$ , and only  $Q(j) = \{(j, f), (j, h)\}$  is asked in Table 2(b). □

### 3.4 Probing Dominating Sets

In general, our pruning methods of Corollaries 1 and 2 are more effective if: (1) many tuples are determined as complete non-skyline tuples, and (2) many dominance relationships between tuples are inferred in  $\mathcal{A}^C$ . However, we observe that the pruning methods may not work well if many non-skyline tuples in  $\mathcal{A}^K$  become skyline tuples in  $\mathcal{A}$ , e.g., anti-correlated distribution. Figure 3(a) illustrates a dataset with 10 tuples in  $\mathcal{A}^K = \{A_1, A_2\}$ . This can be partitioned to two subsets  $\{b, e, i, j\}$  and  $\{a, c, f, d, g, h\}$ , i.e., the former is skyline tuples in  $\mathcal{A}^K$ , while the latter is non-skyline tuples in  $\mathcal{A}^K$ . When all non-skyline tuples become skyline tuples, our pruning methods cannot contribute to reduce the dominating sets of  $\{a, c, f, d, g, h\}$ . Thus, we have to ask a total of 24 ( $= 4 \times 6$ ) questions to crowds.

To overcome this problem, we progressively probe  $DS(t)$  for  $t$  to minimize  $\text{SKY}_{\mathcal{A}^C}(DS(t))$ . Specifically, in asking questions for  $\{b, e, i, j\}$ , the dominance relationship can be used for reducing the dominance sets of  $\{a, c, f, d, g, h\}$ . For instance, suppose that the dominance relationships for  $\{b, e, i, j\}$  in Figure 3(b) are updated after asking  $\{(b, e), (i, j), (e, i)\}$ . Since  $e$  dominates  $\{b, i, j\}$  in  $\mathcal{A}^C$ , each single question for each tuple in  $\{a, c, f, d, g, h\}$  are generated, i.e.,  $\{(e, a), (e, c), (e, f), (e, d), (e, g), (e, h)\}$ . As a result, our probing method for  $DS(t)$  enables us to only ask 9 ( $= 3 + 6$ ) questions.

We now discuss how to generate questions for probing  $DS(t)$ . Given  $DS(t)$ , the number of possible questions is  $\binom{|DS(t)|}{2}$ . Because all possible ordering in probing  $DS(t)$  is exponential and the dominance relationships of tuples in  $DS(t)$  are unpredictable, it is non-trivial to decide the right ordering of probing  $DS(t)$ . As one of feasible solutions, we propose a *greedy* method using the *frequency* of dominating tuples. Let  $P(t)$  denote a set of all possible questions  $\binom{|DS(t)|}{2}$  for probing  $DS(t)$ . For each question  $(u, v)$  in  $P(t)$ ,  $\text{freq}(u, v)$  is the number of tuples that are dominated by both  $u$  and  $v$ , i.e.,  $\text{freq}(u, v) = |\{x \in \mathcal{R} | u \prec_{\mathcal{A}^K} x \wedge v \prec_{\mathcal{A}^K} x\}|$ . As  $\text{freq}(u, v)$  is higher, we suppose that the pruning power of  $(u, v)$  gets stronger. We choose the question with the highest frequency in  $P(t)$ , and remove the questions for less preferred tuples from  $P(t)$ . This process repeats until no questions exist in  $P(t)$ .

**EXAMPLE 6 (PROBING DOMINATING SETS)** The tuples are sorted by the size of dominating sets as shown in Table 2(a). For each tuple  $t \in \mathcal{R}$ ,  $DS(t)$  is first pruned by using the two pruning methods in Corollaries 1 and 2. We then probe  $DS(t)$  before generating  $Q(t)$ . Figure 4(a) shows the questions asked per tuple. Before asking  $Q(d)$ ,  $P(d) = \{(b, e)\}$  is first probed. When  $e \prec_{\mathcal{A}^C} b$  has been decided,  $b$  no longer needs to be compared, and thus the questions for  $b$  such as  $(c, b)$ ,  $(f, b)$ ,  $(h, b)$ , and  $(j, b)$  can be removed for  $c, f, h$ , and  $j$ , respectively. Figure 4(b) depicts a preference tree  $\mathcal{T}$  in  $A_3$  after all tuples become complete. By probing dominating sets, we only ask 12 questions for identifying the final crowd-enabled skyline  $\{b, e, i, l, k, f, h\}$ . □

### 3.5 Algorithm Description

We present the pseudocode of our proposed algorithm, named **CrowdSky** (Algorithm 1). Overall, **CrowdSky** works as the combination of machine and crowds. That is, once the machine iteratively updates data structures and generates new questions, crowds return the answers. Specifically: (1) As a degenerate case, it first

---

**Algorithm 1: CrowdSky( $\mathcal{R}$ )**

---

```
1 foreach  $(s, t) \in \mathcal{R} \times \mathcal{R}$  do
2   if  $\forall A_j \in \mathcal{A}^K : s_j = t_j$  then
3     Ask  $(s, t)$  to crowds and remove a less preferred tuple from
      $\mathcal{R}$ 
4 Initialize a preference tree  $\mathcal{T}$  in  $\mathcal{A}^C$ 
5 For each tuple  $t \in \mathcal{R}$ , compute  $DS(t)$ 
6  $\text{SKY}_{\mathcal{A}}(\mathcal{R}) \leftarrow \{\}$  // Initialize a skyline in  $\mathcal{A}$ 
   // P1: Early pruning for non-skylines in  $\mathcal{A}$ 
7 Sort  $\mathcal{R}$  by ascending order of  $|DS(t)|$ 
8 for  $t \in \mathcal{R}$  do
   // P2: Pruning non-skylines in  $\mathcal{A}^C$ 
9    $DS(t) \leftarrow \text{SKY}_{\mathcal{A}^C}(DS(t))$ 
   // P3: Probing into  $DS(t)$ 
10   $P(t) \leftarrow \{(u, v) \mid u, v \in DS(t), u \neq v\}$ 
11  Sort  $P(t)$  by ascending order of  $freq(u, v)$ 
12  for  $(u, v) \in P(t)$  do
13    Ask  $(u, v)$  to crowds, and update  $\mathcal{T}$  with  $(u, v)$ 
14    if  $u \prec_{\mathcal{A}^C} v$  in  $\mathcal{T}$  then
15      For  $x \in DS(t)$ , remove  $(v, x)$  from  $P(t)$ 
16       $DS(t) \leftarrow DS(t) - \{v\}$ 
17    else if  $v \prec_{\mathcal{A}^C} u$  in  $\mathcal{T}$  then
18      For  $x \in DS(t)$ , remove  $(u, x)$  from  $P(t)$ 
19       $DS(t) \leftarrow DS(t) - \{u\}$ 
   // Generating  $Q(t)$  from  $DS(t)$ 
20   $Q(t) \leftarrow \{(s, t) \mid s \in DS(t)\}$ 
21  for  $(s, t) \in Q(t)$  do
22    Ask  $(s, t)$  to crowds, and update  $\mathcal{T}$  with  $(s, t)$ 
23    if  $s \prec_{\mathcal{A}^C} t$  in  $\mathcal{T}$  then
24      break //  $t$  is a non-skyline tuple
25  if  $\forall s \in DS(t) : s \not\prec_{\mathcal{A}^C} t$  then
26     $\text{SKY}_{\mathcal{A}}(\mathcal{R}) \leftarrow \text{SKY}_{\mathcal{A}}(\mathcal{R}) \cup \{t\}$ 
27 return  $\text{SKY}_{\mathcal{A}}(\mathcal{R})$ 
```

---

checks tuples with the same values in  $\mathcal{A}^K$ , and prunes less preferred tuples in  $\mathcal{A}^C$  (lines 1-3). (2) It then builds dominating sets, and sorts them by  $|DS(t)|$ , incurring  $O(n^2)$  in machine part (lines 6-7). (3) For each tuple  $t \in \mathcal{R}$ ,  $DS(t)$  is updated by  $\text{SKY}_{\mathcal{A}^C}(DS(t))$  to remove non-skyline tuples (line 9). (4) It then generates all possible questions  $P(t)$  to probe  $DS(t)$  by  $freq(u, v)$  (lines 10-11). By asking questions in  $P(t)$ ,  $\mathcal{T}$  is updated. In addition  $DS(t)$  and  $P(t)$  are updated (lines 12-19). (5) After that,  $Q(t)$  is generated from  $DS(t)$  (lines 20-24). (6) Finally, if  $t$  is not dominated by any other tuples in  $DS(t)$ ,  $t$  is appended to  $\text{SKY}_{\mathcal{A}}(\mathcal{R})$  (lines 25-26).

**THEOREM 1 (COMPLETENESS OF CROWDSKY)** CrowdSky guarantees that all tuples in  $\mathcal{R}$  are complete.

**PROOF** We prove this by contradiction. Assume that  $t$  exists that is not determined as a complete tuple. This means a question  $(t, u)$  exists for  $t$  to be complete. Because the questions in CrowdSky are pruned by Corollaries 1 and 2, if  $(t, u)$  is not asked, a tuple  $v$  exists such that  $v \prec_{\mathcal{A}^C} u$ . In that case,  $(t, u)$  is unnecessary to check if  $t$  is complete, which is a contradiction. ■

As a result, if crowds always return correct answers, CrowdSky can identify all complete skyline tuples and the result is correct.

## 4. REDUCING LATENCY

In this section, we discuss how to reduce the number of rounds. Although existing work [25] addresses a parallelization method for

asking multiple questions, it is based on a different problem for entity resolution, thereby being inapplicable to our problem. We develop two parallelization methods that are suitable for computing a crowdsourced skyline. Specifically, we first propose a *partitioning method* using dominating sets (Section 4.1), and then improve it using *skyline layers* (Section 4.2).

### 4.1 Parallelization with Dominating Sets

Given two questions  $(s, t)$  and  $(u, v)$ , if asking  $(s, t)$  is not related to prune  $(u, v)$  and vice versa, it is said that they are *independent*. Meanwhile, if  $(s, t)$  can be pruned by asking  $(u, v)$ ,  $(s, t)$  is said to be *dependent* on  $(u, v)$ . In order to remove unnecessary questions using our pruning methods in CrowdSky, the dependency of questions can happen as follows:

1. *Dominance relationship in  $\mathcal{A}^K$  (C1)*: Given two tuples  $s$  and  $t$  such that  $s \in DS(t)$ ,  $Q(s)$  needs to be asked before generating  $Q(t)$  by Corollary 1 (line 7 in CrowdSky). In other words, when  $s$  is determined as a non-skyline tuple,  $(s, t)$  is unnecessary in  $Q(t)$ .
2. *Overlap between  $DS(s)$  and  $DS(t)$  (C2)*: We suppose that  $DS(s)$  and  $DS(t)$  share a common tuple  $u$ . When probing  $DS(s)$  using the third pruning methods,  $u$  can be removed from  $\text{SKY}_{\mathcal{A}^C}(DS(t))$  (line 9 in CrowdSky), and  $(u, t)$  can be unnecessary in  $Q(t)$  by Corollary 2.
3. *Questions for  $DS(t)$  (C3)*: When  $Q(t)$  is sequentially generated from  $DS(t)$ ,  $t$  can be a non-skyline tuple (lines 20-24 in CrowdSky), and remaining questions in  $Q(t)$  are no longer needed.

Based on these observations, our idea of parallelizing questions in CrowdSky is to identify independent questions as many as possible at each round. Specifically, we first develop a *partitioning method* using dominating sets. First,  $\mathcal{R}$  is partitioned into several subsets of tuples with the same sizes of dominating sets. Given  $s, t \in \mathcal{R}$ , if  $|DS(s)| = |DS(t)|$ ,  $s$  and  $t$  cannot dominate each other (by Lemma 3). The questions for partitioned tuple sets can be asked together by avoiding dependent questions by (C1). We then check if dominating sets of tuples are disjoint. In that case, probing dominating sets can be parallelized without (C2). Lastly, because (C3) does not make parallelized questions,  $Q(t)$  is sequentially generated from  $DS(t)$ .

**EXAMPLE 7 (PARTITIONING METHOD)** After the dominating sets are first computed,  $\mathcal{R}$  is partitioned into  $\{\{a, g\}, \{d, k\}, \{c\}, \{f\}, \{h\}, \{j\}\}$  with the same sizes of dominating sets. For each partitioned set, it then checks if dominating sets are disjoint. Given  $\{a, g\}$  and  $\{d, k\}$ , because  $DS(a) \cap DS(g)$  and  $DS(d) \cap DS(k)$  are disjoint as illustrated in Figure 4(a), the questions for  $\{a, g\}$  and  $\{d, k\}$  can be asked together. For  $\{a, g\}$ ,  $\{(a, b), (g, e)\}$  is asked in a round. For  $\{d, k\}$ ,  $\{(b, e), (i, l)\}$  (in  $P(d)$  and  $P(k)$ ) and  $\{(d, e), (k, i)\}$  (in  $Q(d)$  and  $Q(k)$ ) are asked in 2 rounds. Because  $\{c, f, h, j\}$  is partitioned separately, 6 questions are asked in 6 rounds. As a result, our partitioning method generates 12 questions during 9 rounds by saving 3 rounds. □

### 4.2 Parallelization with Skyline Layers

Although our partitioning method reduces the number of rounds, the degree of parallelization is rather limited by keeping all dependencies of questions. To alleviate this problem, we adopt *skyline layers* that effectively visualize the dominance relationships between tuples in  $\mathcal{A}^K$ . Figure 5 depicts skyline layers for the toy dataset in Figure 1(a). To build skyline layers, skylines are

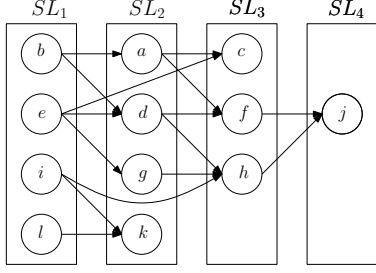


Figure 5: Skyline layers for the toy dataset in Figure 1(a)

computed in an iterative manner. Initially,  $SL_1$  is computed by  $\text{SKY}_{\mathcal{A}^K}(\mathcal{R})$ . Then, the  $i$ -th layer is computed by  $\text{SKY}_{\mathcal{A}^K}(\mathcal{R} - \bigcup_{j=1}^{i-1} SL_j)$  in an iterative manner.

**DEFINITION 6 (SKYLINE LAYER)** The  $i$ -th skyline layer  $SL_i$  is a set of skyline tuples in  $\mathcal{R} - \bigcup_{j=1}^{i-1} SL_j$ , i.e.,  $SL_i = \{t \in \mathcal{R} - \bigcup_{j=1}^{i-1} SL_j \mid \forall s \in \mathcal{R} - \bigcup_{j=1}^{i-1} SL_j : s \not\prec_{\mathcal{A}^K} t\}$ .

After building all layers, the dominance relationship of tuples in  $\mathcal{A}^K$  is constructed. Similar to a *dominating graph* [27], the dominance relationship can be represented by a directed edge between two tuples across different layers. In particular, our skyline layers permit the dominance relationship between tuples in any two layers, e.g.,  $e \rightarrow c$  and  $i \rightarrow h$  in Figure 5. Note that the dominance relationship via transitivity can be inferred from multiple edges.

We now explain how to make use of skyline layers to reduce the number of rounds. Let  $c(t)$  denote a set of tuples that directly point to  $t$ , i.e.,  $c(t) = \{s \in \mathcal{R} \mid s \rightarrow t\}$ . For each tuple  $t \in \mathcal{R}$ , we check if all tuples in  $c(t)$  are complete. If so, the questions for  $t$  are asked *independently* of those of other tuples. When all tuples in  $c(t)$  are determined as complete tuples, it implies that all tuples in  $DS(t)$  are also complete. Because this method can effectively check the dominance relationships of tuples in (C1), we can significantly improve the magnitude of parallelization. Meanwhile, it is observed that (C2) is a main bottleneck for parallelizing questions. We thus violate the dependency of questions in (C2), by generating additional questions for (C2). (Our empirical study shows that the number of additional questions is negligible.)

Algorithm 2 describes the procedure of parallelizing questions using skyline layers. Let  $\mathcal{C}$  be a set of complete tuples in  $\mathcal{R}$ . (1)  $\mathcal{C}$  is initialized as  $SL_1$  (line 4). (2) For each tuple  $t \in \mathcal{R}$ , if all tuples in  $c(t)$  are complete, the questions for  $t$  are asked in parallel. (lines 5-7). (3) After asking questions for  $t$ ,  $\text{SKY}_{\mathcal{A}}(\mathcal{R})$  and  $\mathcal{C}$  are updated (lines 8-10). This process continues until every tuple in  $\mathcal{R}$  has been determined as a complete tuple. Because this method is based on the same pruning methods used in *CrowdSky*, our proposed parallelization methods can assure the correctness of crowd-enabled skyline computation.

**EXAMPLE 8 (PARALLELIZATION WITH SKYLINE LAYERS)** Given the skyline layers in Figure 5, Table 3 depicts the procedure of parallelization using skyline layers. First,  $\mathcal{C}$  is initialized as  $SL_1$ , i.e.,  $\mathcal{C} = \{b, e, i, l\}$ . Because  $c(a) = \{b\}$ ,  $c(g) = \{e\}$ ,  $c(d) = \{b, e\}$  and  $c(k) = \{i, l\}$  are all complete tuples,  $\{(a, b), (g, e), (b, e), (i, l)\}$  is asked in parallel (round 1), and  $\mathcal{C} = \{b, e, i, l, a, g\}$  is updated (underlines in Table 3). After that, because  $c(c) = \{a, e\}$  is complete,  $\{(d, e), (k, i)\}$  is asked with  $\{(c, e)\}$ , and  $\mathcal{C} = \{b, e, i, l, a, g, d, k, c\}$  is updated (round 2). Because  $c(f) = \{a, d\}$  and  $c(h) = \{d, g, i\}$  are complete, the questions for  $f$  and  $h$  are asked (rounds 3-4), and  $\mathcal{C} = \{b, e, i, l, a, g, d, k, c, f, h\}$  is updated. Lastly, when  $c(j) = \{f, h\}$  is complete, the questions for

Algorithm 2: ParallelISL( $\mathcal{R}$ )

```

1 Execute lines 1-5 in Algorithm 1
2  $\text{SKY}_{\mathcal{A}}(\mathcal{R}) \leftarrow \{\}$  // Initialize a skyline in  $\mathcal{A}$ 
3 Compute  $SL_1, \dots, SL_k$  for  $\mathcal{R}$ 
4  $\mathcal{C} \leftarrow SL_1$  // Initialize a complete tuple set
5 for  $t \in \mathcal{R}$  do in parallel
6   if  $c(t) \subseteq \mathcal{C}$  then
7     For  $t$ , execute lines 9-24 in Algorithm 1
8     if  $\forall s \in DS(t) : s \notin \mathcal{C}$  then
9        $\text{SKY}_{\mathcal{A}}(\mathcal{R}) \leftarrow \text{SKY}_{\mathcal{A}}(\mathcal{R}) \cup \{t\}$ 
10       $\mathcal{C} \leftarrow \mathcal{C} \cup \{t\}$  // Update  $\mathcal{C}$ 
11 return  $\text{SKY}_{\mathcal{A}}(\mathcal{R})$ 

```

Table 3: Procedure of asking questions using skyline layers in Figure 4(a)

$t$	$c(t)$	1	2	3	4	5	6
$a$	$\{b\}$	<u><math>(a, b)</math></u>					
$g$	$\{e\}$	<u><math>(g, e)</math></u>					
$d$	$\{b, e\}$	<u><math>(b, e)</math></u>	$(d, e)$				
$k$	$\{i, l\}$	<u><math>(i, l)</math></u>	<u><math>(k, i)</math></u>				
$c$	$\{a, e\}$		<u><math>(c, e)</math></u>				
$f$	$\{a, d\}$			$(f, e)$			
$h$	$\{d, g, i\}$			<u><math>(e, i)</math></u>	<u><math>(h, e)</math></u>		
$j$	$\{f, h\}$					$(f, h)$	$(j, f)$

$j$  are asked (rounds 5-6). As a result, our parallelization method generates 12 questions during 6 rounds.  $\square$

## 5. IMPROVING ACCURACY OF ANSWERS

Because the answers of crowds are often erroneous, how to improve the accuracy is a central issue in crowdsourcing. As discussed in [24, 25], it was treated as a research problem orthogonal to the problem of minimizing the number of questions. Existing work [6, 11, 18] developed how to improve the accuracy for each question using *query-independent* methods.

As the simplest method, *majority voting* is used by assigning multiple workers per question. Let  $\omega$  denote the number of workers per question, and  $p$  denote the probability that each worker's answer is correct. The probability that question  $(u, v)$  is correct can be modeled as the binomial distribution.

$$P(u, v) = \sum_{i=\lceil \frac{\omega}{2} \rceil}^{\omega} \binom{\omega}{i} p^i (1-p)^{\omega-i},$$

where  $\omega$  is an odd integer. This method can improve the accuracy per question, but does not consider the *importance* of questions in computing a skyline. This method is called *static voting*. (In our experiments,  $\omega = 5$  by default.)

As the *query-dependent* method, we develop a heuristic method to reflect the importance of questions. When computing the skyline, it is observed that the questions with many dominance relationships in  $\mathcal{A}^K$  are more influential in identifying a more accurate preference tree  $\mathcal{T}$  in  $\mathcal{A}^C$ . Based on this observation, we propose to use the *frequency* of questions  $\text{freq}(u, v)$  for quantifying the importance of  $(u, v)$ , i.e.,  $\text{freq}(u, v) = |\{x \in \mathcal{R} \mid u \prec_{\mathcal{A}^K} x \wedge v \prec_{\mathcal{A}^K} x\}|$ . That is, as  $\text{freq}(u, v)$  gets larger,  $(u, v)$  becomes more important. Given question  $(u, v)$ , the different numbers of workers can be assigned, depending on  $\text{freq}(u, v)$ . We refer to this method as *dynamic voting*. Note that the dynamic voting can help prevent the propagation of false dominance relationships.



**Table 4: Parameter settings over synthetic datasets**

parameters	value	default
cardinality $n$	2K, 4K, 6K, 8K, 10K	4K
# of known attributes $ \mathcal{A}^K $	2, 3, 4, 5	4
# of crowd attributes $ \mathcal{A}^C $	1, 2, 3	1
data distribution	IND, ANT	

For instance, based on the idea of the dynamic voting, one may dynamically assign the number of workers using two parameters  $\alpha$  and  $\beta$  ( $\alpha < \beta$  and  $\alpha, \beta \geq 0$ ) as follows. Given  $\text{freq}(u, v)$ , we choose the number of workers  $\omega'$  with the following inequalities.

$$\omega' = \begin{cases} \omega - 2, & \text{if } \text{freq}(u, v) < \alpha \\ \omega, & \text{if } \alpha \leq \text{freq}(u, v) < \beta \\ \omega + 2, & \text{if } \text{freq}(u, v) \geq \beta \end{cases}$$

By considering the importance of questions, our dynamic voting can improve the accuracy of the skyline result compared to the static voting. Note that it can be easily extended for multiple categories with three or more cases.

## 6. EXPERIMENTS

In this section, we first evaluate our proposed algorithm over synthetic datasets with extensive parameter settings (Section 6.1). We show the performance of our proposed algorithm in terms of three key factors. We then evaluate our algorithm using the Amazon Mechanical Turk over real-life datasets (Section 6.2).

### 6.1 Evaluation in Synthetic Datasets

We first evaluate our proposed algorithm over synthetic datasets. Because real-life datasets are limited for evaluating extensive settings, we adopted benchmark datasets [2] that are widely used in skyline evaluation. In particular, we used two data distributions, *independent* (IND) and *anti-correlated* (ANT) in [2]. All attribute values were randomly generated from real values in [0, 1]. The values on crowd attributes were only used for obtaining the answers of crowds for simulated questions. Table 4 summarizes parameter settings over synthetic datasets.

We validate our proposed algorithm for three key factors: monetary cost, latency, and accuracy (as discussed in Section 2.1). Because the running time in crowdsourcing part is much slower than generating questions in machine part, we focus on evaluating the number of rounds for latency. All experiments were conducted in Windows 7 running on Intel Core i7 950 3.07 GHz CPU with 16GB RAM. All the algorithms were implemented in C++. The average values of 10 runs are reported for all experiments.

**Monetary cost.** When the fixed amount of a reward per question is paid, the monetary cost is proportional to the number of questions asked to crowds. We thus use the number of questions to measure the monetary cost. As one of the sorting algorithms, *tournament sort* is used as a baseline, denoted by **Baseline**. When the number of rounds is not limited, tournament sort can produce the *total order* of tuples with the minimum number of questions. An existing work [12] studied crowd-enabled skyline queries, but focused on selecting the most influential unary questions for a restricted budget. Because the optimization methods in [12] are not effective for reducing questions in our problem setting, the direct comparison between **CrowdSky** and [12] is not fair to [12]. As such, we simulate the unary questions in [12] for comparing the accuracy, and mainly focus on scrutinizing **CrowdSky** to quantify the advantage of optimization methods. Specifically, it is divided into four

phases: **DSet** (Section 3.1), **P1** (Section 3.2), **P2** (Section 3.3), and **P3** (Section 3.4).

Figures 6(a) and 7(a) depict the number of questions over varying cardinality. Note that **DSet** is basically used for all pruning methods. It is clear that **P1+P2+P3** minimizes the number of questions over all parameter settings, e.g., reducing the number of questions 10 times more than that of **Baseline** over independent distribution. In particular, several interesting observations are found in Figures 6(a) and 7(a): (1) While **DSet** produces fewer questions than **Baseline** in independent distribution, it is reversed in anti-correlated distribution. This is because the number of skyline tuples increases exponentially with the cardinality over anti-correlated distribution. (2) Both **P1** and **P2** can contribute to reducing unnecessary questions. Notably, **P2** (using the transitivity of tuples in  $\mathcal{A}^C$ ) is fairly effective over anti-correlated distribution. (3) As expected, **P3** (probing dominating sets) reduces a more number of unnecessary questions over anti-correlated distribution.

Figures 6(b) and 7(b) report the number of questions over varying  $|\mathcal{A}^K|$ . While **Baseline** shows a constant performance regardless of  $|\mathcal{A}^K|$ , our pruning methods reduce the number of questions as  $|\mathcal{A}^K|$  increases. This is because the size of dominating sets tends to decrease with  $|\mathcal{A}^K|$ . We also found two key observations: (1) **P1+P2+P3** minimizes the number of questions over all parameter settings. (2) When  $|\mathcal{A}^K|$  is low, our pruning methods significantly reduce unnecessary questions, and are much more effective over anti-correlated distribution than independent distribution. When  $|\mathcal{A}^K| = 2$ , **P1+P2+P3** reduces the number of questions in **DSet** by two orders of magnitude.

Figures 6(c) and 7(c) report the number of questions over varying  $|\mathcal{A}^C|$ . When  $|\mathcal{A}^C| > 1$ , suppose that all methods simply generate  $|\mathcal{A}^C|$  questions in  $\mathcal{A}^C$ . (It is possible to use a round-robin strategy for multiple crowd attributes to reduce unnecessary questions as they become incomparable in  $\mathcal{A}^C$ , but it is not applied to our evaluation.) (1) As  $|\mathcal{A}^C|$  increases, the number of questions increases for all methods. (2) Interestingly, when  $|\mathcal{A}^C| = 2$ , **P3** becomes marginal. As  $\mathcal{A}^C$  increases, **P3** becomes less effective for generating dominance relationships in  $\mathcal{A}^C$ . This implies that probing dominating sets mainly incurs the questions for tuples with incomparable relationships. In high dimensionality in  $\mathcal{A}^C$ , we have to consider to use **P3** in **CrowdSky**.

**Latency.** In order to measure latency, we used the number of rounds in performing an algorithm. We compared the following four algorithms: (1) **Baseline** is tournament sort; (2) **Serial** asks a single question in a round; (3) **ParallelDSet** is our parallelization algorithm using dominating sets (Section 4.1); (4) **ParallelSL** is our parallelization algorithm using skyline layers (Section 4.2).

Figure 8 reports the number of rounds over varying cardinality. Note that the y-axis is log-scaled. The gap between **Serial** and **ParallelDSet** widens by one order of magnitude as cardinality increases. In addition, **ParallelSL** outperforms **ParallelDSet** by two orders of magnitude, e.g., **ParallelSL** only needs about 20-30 rounds in both distributions. Although we did not report the number of questions for parallelization, it is found that **ParallelDSet** generates the same number of questions for **Serial**, and **ParallelSL** generates approximately 10% more questions than **ParallelDSet**, by violating the dependency of questions in (C2).

Figure 9 reports the number of rounds over varying dimensionality  $|\mathcal{A}^K|$ . Interestingly, while **Serial** incurs more rounds with higher  $|\mathcal{A}^K|$ , **ParallelDSet** and **ParallelSL** decrease the number of rounds with  $|\mathcal{A}^K|$ . This implies that the degree of parallelization becomes higher as  $|\mathcal{A}^K|$  increases. As consistently observed in Figure 8, **ParallelSL** significantly outperforms **ParallelDSet** by two orders of magnitude in both distributions.

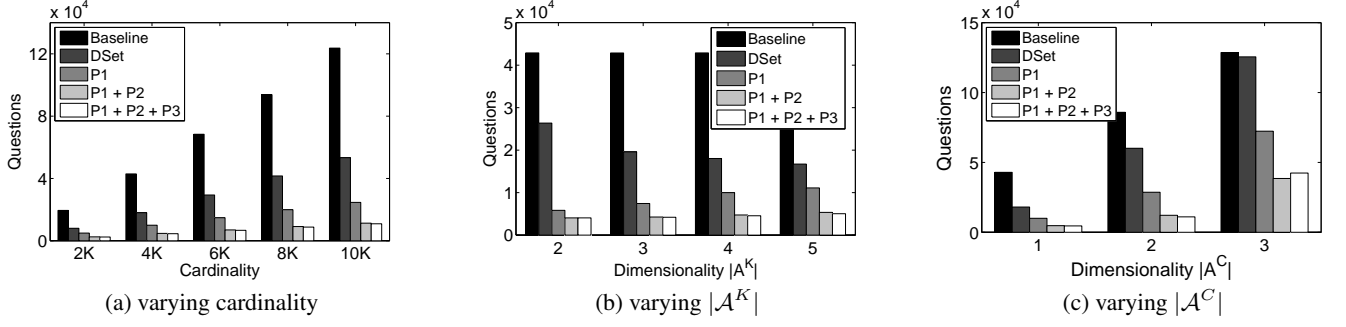


Figure 6: Comparisons on the number of questions over independent distribution

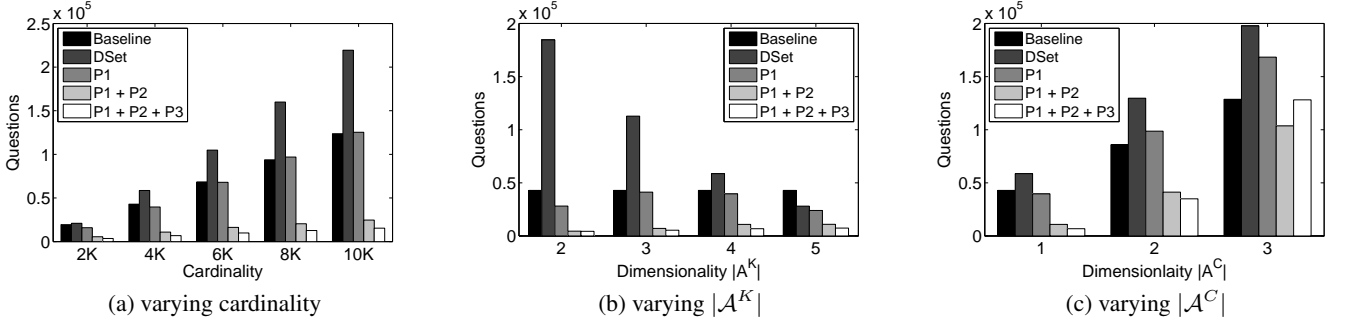


Figure 7: Comparisons on the number of questions over anti-correlated distribution

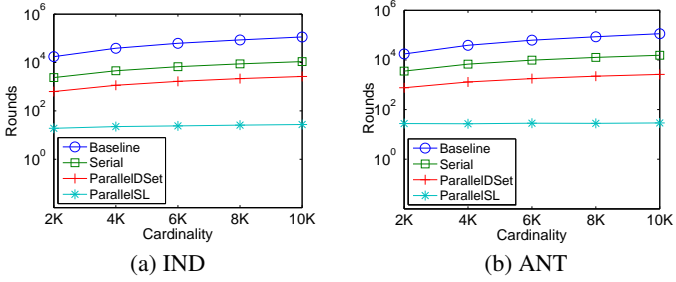


Figure 8: Comparisons on the number of rounds over varying cardinality

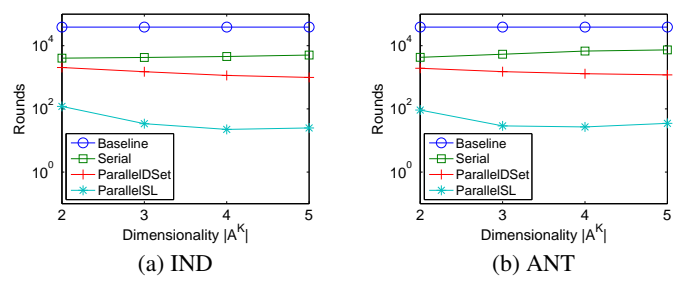


Figure 9: Comparisons on the number of rounds over varying dimensionality of known attributes

**Accuracy.** The crowdsourced skyline is defined as  $SKY_{\mathcal{A}}(\mathcal{R})$ . To measure the accuracy of skyline results, we only use a set of newly retrieved skyline tuples by crowdsourcing, *i.e.*,  $SKY_{\mathcal{A}}(\mathcal{R}) - SKY_{\mathcal{A}^K}(\mathcal{R})$ , with two metrics, *precision* and *recall*, which are widely used in Information Retrieval.

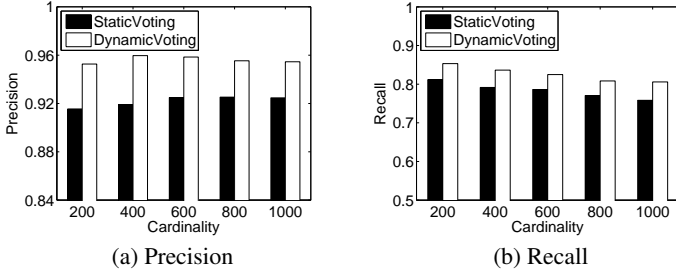
We first compared two assignment methods, *StaticVoting* and *DynamicVoting* in *CrowdSky* (as in Section 5). By default, we set  $\omega = 5$  and  $p = 0.8$ . While *StaticVoting* equally assigns  $\omega$  per question, *DynamicVoting* assigns  $\omega + 2$ ,  $\omega$ ,  $\omega - 2$  depending on the frequency of questions. For fair comparisons, we assigned the same total number of workers in both methods. The assignment of the number of workers in *DynamicVoting* is tuned as: the initial 30% questions are assigned to  $\omega + 2$ , and the last 30% questions are assigned to  $\omega - 2$ . This implies that the initial questions in *DynamicVoting* are more important than other questions.

Figure 10 reports the accuracy of two voting methods over varying cardinality. It is clear that *DynamicVoting* shows higher accuracy than *StaticVoting* for both metrics. Note that *DynamicVoting* improves the overall accuracy by assigning more workers to more

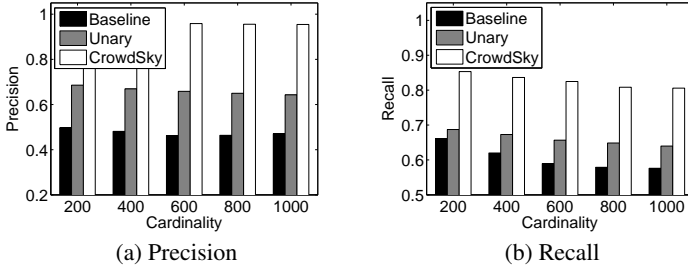
important questions and by reducing the propagation for false positives/negatives. In addition, the precision is higher than the recall in all parameter settings. While most of the skyline tuples are decided correctly, some correct skyline tuples are determined as non-skyline tuples. This is because our methods focus on asking questions for skyline candidates and do not perform an additional validation for non-skyline tuples.

We also compared the following three algorithms: (1) *Baseline* generates the total order of tuples in  $\mathcal{A}^K$  by performing the tournament sort, (2) *Unary* generates the total order of tuples by asking unary questions (as done in [12]), and (3) *CrowdSky* adopts *DynamicVoting*. To simulate the unary questions in [12], we randomly select a value from the normal distribution of actual value in  $\mathcal{A}^C$ . In this setting, it is found that *Unary* is more accurate for obtaining the total order of tuples than *Baseline*, *i.e.*, it is more favorable for *Unary*.

Figure 11 reports the accuracy of comparing *CrowdSky* and the two existing methods. Interestingly, even though *Baseline* generates more numbers of questions than *CrowdSky*, *Baseline* is



**Figure 10: Accuracy comparisons of two voting methods in CrowdSky over independent distribution**



**Figure 11: Accuracy comparisons of CrowdSky with existing methods (i.e., Unary as in [12]) over independent distribution**

worse than CrowdSky. Because more questions in Baseline incur wrong answers, the total order of tuples in Baseline is less effective for identifying a correct skyline. In contrast, CrowdSky generates questions more selectively for skyline candidates. While Unary is better than Baseline, it is worse than CrowdSky. Because the pruning methods in [12] do not work well in our problem setting, Unary is less effective for identifying correct skyline tuples.

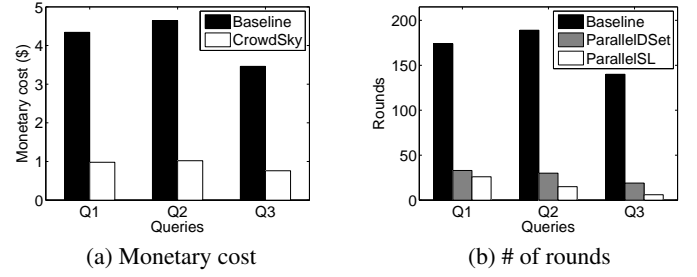
## 6.2 Evaluation in Real-life Datasets

We used three real-life datasets to validate our proposed algorithms: (1) **Rectangles**, adopted from [14], contains 50 images whose sizes are  $\{(30 + 3i) \times (40 + 5i) | i \in [0, 50)\}$  and are randomly rotated. (2) **IMDb Movies** includes 50 popular movies released in 2000-2012 (<http://www.imdb.com/>). (3) **MLB players** includes 40 baseball pitchers played in 2013 (<http://espn.go.com/mlb/stats/>). For these datasets, we used small-scale datasets in order to manage the monetary cost in real-life experiments, and executed the following crowd-enabled skyline queries:

**Q1:** Find the skyline using rectangle data with  $\mathcal{A}^K = \{\text{width, height}\}$  and  $\mathcal{A}^C = \{\text{area}\}$ . The larger values in  $\mathcal{A}^K$  and  $\mathcal{A}^C$  are more preferred. As the ground truth for crowd attribute *area* can be obtained using *width* and *height*, the accuracy of the crowdsourced skyline can be measured.

**Q2:** Find the skyline using movie data with  $\mathcal{A}^K = \{\text{box\_office, release\_year}\}$  and  $\mathcal{A}^C = \{\text{rating}\}$ . The larger values in  $\mathcal{A}^K$  and  $\mathcal{A}^C$  are more preferred. Since IMDb shows aggregated rating scores of movies, we compare the crowdsourced skyline (using preferences culled from crowds) against the IMDb rating-based skyline.

**Q3:** Find the skyline using MLB player data with  $\mathcal{A}^K = \{\text{wins, strikes\_outs, ERA}\}$  and  $\mathcal{A}^C = \{\text{valuable}\}$ . The larger values are more preferred, except for ERA. The crowd attribute



**Figure 12: Comparisons of three queries over real-life datasets**

*valuable* is the preference of crowds on how *valuable* each pitcher is. For this query, we indirectly compare the crowdsourced skyline against the candidates of the “Cy Young” award, given to the best pitchers annually.

The Amazon Mechanical Turk (AMT), a well-known crowdsourcing platform, was used for asking questions to crowds in real-life datasets. The budget per question was set to \$0.02, and 5 workers were assigned for each question, i.e.,  $\omega = 5$ . Let  $r$  denote the total number of rounds, and  $|Q_i|$  denote the number of questions at the  $i$ -th round. The total monetary cost is thus calculated as:  $0.02 \times 5 \times \sum_{i=1}^r \lceil \frac{|Q_i|}{5} \rceil$ , where 5 questions are issued at each task. To filter out spam workers, we only permitted *Masters* workers who are qualified as the most reliable workers in AMT.

**Monetary cost.** Figure 12(a) compares the monetary cost between Baseline and CrowdSky. Note that CrowdSky saves the cost of Baseline by 3-4 times. (Because the cardinality of real-life datasets is smaller than that of synthetic datasets, the performance gap between Baseline and CrowdSky is reduced.) For  $Q1$  and  $Q2$ , while Baseline needs more than 200 questions, CrowdSky generates about 50 questions, where most questions are used for validating non-skyline tuples.

**Latency.** Figure 12(b) compares the latency of three algorithms: Baseline, ParallelDSet, and ParallelSL. For  $Q1$ ,  $Q2$ , and  $Q3$ , the average working time per HIT was 22 secs, 49 secs, and 1 min 33 secs, respectively, implying that  $Q3$  is the most difficult task. While Baseline incurs more than 140 rounds over all queries, ParallelDSet and ParallelSL only generate less than 30 rounds. In addition, ParallelSL generates 50% less rounds than ParallelDSet, without increasing the cost for all queries. For  $Q3$ , while Baseline and ParallelDSet need 140 and 19 rounds, ParallelSL computes the skyline with only 6 rounds.

**Accuracy.** For  $Q1$ , CrowdSky identifies the same skyline as the ground truth, yielding *Precision* = 1.0 and *Recall* = 1.0. For  $Q2$ , the crowdsourced skyline includes 5 movies such as {Avatar, The Avengers, Inception, The Lord of Rings: The Fellowship of the Ring, The Dark Knight Rises}. Except for the existing skyline {Avatar, The Avengers} in  $\mathcal{A}^K$ , we found that the average rating of three skyline movies is very high (i.e., 8.7 out of 10.0) in IMDb, indicating that crowds were able to find decent skyline movies. For  $Q3$ , the skyline includes four players such as {Clayton Kershaw, Bartolo Colon, Yu Darvish, Max Scherzer} who were Cy Young award candidates, representing the best pitchers, in 2013. In particular, “Clayton Kershaw” and “Max Scherzer” were the winners of the Cy Young award in 2013. As such, we claim that the crowdsourced skyline be reasonably accurate. Based on the results, we argue that CrowdSky yields high accuracy while keeping the monetary cost and the latency low.

## 7. RELATED WORK

Skyline queries have been actively studied for assisting multi-criteria decision making applications. Pioneered by [2], skyline queries are used for various data settings such as distributed and stream environments. Tuples are represented by incomplete and probabilistic values, and data types vary from partially-ordered and categorical attributes. Existing work focused on developing efficient skyline computation with pre-defined preferences. In contrast, we aimed to collect missing preferences from crowds.

In recent years, in data management community, there have been active investigations toward crowd-enabled algorithms. Some of recent highlights include the following data operations with crowdsourcing embedded: selection [6, 18, 21], max [8, 22, 23], sorting [14], top- $k$  [4], top- $k$  set [20], join [14, 24, 25, 26], and group by [4]. Based on these advancements, our work combines the skyline queries with crowdsourcing.

In particular, our crowd-enabled skyline query is related to [12], which is the first work to address the problem of skyline queries with a crowdsourcing idea. However, our work has clear differences from [12] as follows:

- *Problem formulation:* While [12] used crowds to improve the accuracy of incomplete skyline queries, our work addressed a complete skyline by collecting all missing preferences in crowd attributes.
- *Optimization direction:* While [12] mainly aimed at maximizing the accuracy of skyline results, we considered three factors (monetary cost, latency, and accuracy) together.
- *Formats of questions:* Since [12] used the quantitative questions, it is inapplicable for crowd attributes with a large range. On the contrary, since our work is based on the qualitative questions, it is easier for crowds to evaluate tuples in crowd attributes without any constraints.

Because the optimization methods in [12] are not effective for reducing questions in our problem setting, the direct comparison between CrowdSky and [12] would have been unfair to [12]. As such, in Section 6, we have simulated the unary questions in [12] for comparing the accuracy in our setting, and indirectly demonstrated the superiority of CrowdSky with a strong evidence (*i.e.*, Figure 11).

## 8. CONCLUSIONS

In this paper, we have studied the problem of computing skyline queries with crowdsourcing. Specifically, we dealt with three key factors such as monetary cost, latency, and accuracy. Our proposed algorithm first aimed to minimize the number of questions with several pruning methods on top of the notion of a dominating set. We then developed an algorithm to minimize the number of rounds using skyline layers. We lastly improved the accuracy of a crowdsourced skyline using dynamic majority voting. Our experimental results showed that our proposed algorithm optimizes the three key factors effectively over synthetic and real-life datasets.

## Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2014R1A2A1A10054151 and No. 2015R1C1A1A01055442).

## 9. REFERENCES

- [1] R. Boim, O. Greenshpan, T. Milo, S. Novgorodov, N. Polyzotis, and W. C. Tan. Asking the right questions in crowd data sourcing. In *ICDE*, pages 1261–1264, 2012.

- [2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.
- [3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.
- [4] S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Using the crowd for top- $k$  and group-by queries. In *ICDT*, 2013.
- [5] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. In *SIGMOD*, pages 61–72, 2011.
- [6] J. Gao, X. Liu, B. C. Ooi, H. Wang, and G. Chen. An online cost sensitive decision-making method in crowdsourcing systems. In *SIGMOD*, pages 217–228, 2013.
- [7] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. W. Shavlik, and X. Zhu. Corleone: hands-off crowdsourcing for entity matching. In *SIGMOD Conference*, pages 601–612, 2014.
- [8] S. Guo, A. G. Parameswaran, and H. Garcia-Molina. So who won?: dynamic max discovery with the crowd. In *SIGMOD*, pages 385–396, 2012.
- [9] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.
- [10] J. Lee and S. won Hwang. BSKyTree: scalable skyline computation using a balanced pivot selection. In *EDBT*, pages 195–206, 2010.
- [11] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang. CDAS: A crowdsourcing data analytics system. *PVLDB*, 5(10):1040–1051, 2012.
- [12] C. Lofi, K. E. Maarry, and W.-T. Balke. Skyline queries in crowd-enabled databases. In *EDBT*, 2013.
- [13] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Demonstration of Qurk: a query processor for humanoperators. In *SIGMOD*, pages 1315–1318, 2011.
- [14] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Human-powered sorts and joins. *PVLDB*, 5(1):13–24, 2011.
- [15] L. Mo, R. Cheng, B. Kao, X. S. Yang, C. Ren, S. Lei, D. W. Cheung, and E. Lo. Optimizing plurality for human intelligence tasks. In *CIKM*, pages 1929–1938, 2013.
- [16] A. Morishima, N. Shinagawa, T. Mitsuishi, H. Aoki, and S. Fukusumi. CyLog/Crowd4U: A declarative platform for complex data-centric crowdsourcing. *PVLDB*, 5(12):1918–1921, 2012.
- [17] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.
- [18] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: algorithms for filtering data with humans. In *SIGMOD*, pages 361–372, 2012.
- [19] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: declarative crowdsourcing. In *CIKM*, pages 1203–1212, 2012.
- [20] V. Polychronopoulos, L. de Alfaro, J. Davis, H. Garcia-Molina, and N. Polyzotis. Human-powered top- $k$  lists. In *WebDB*, 2013.
- [21] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, pages 673–684, 2013.
- [22] P. Venetis, H. Garcia-Molina, K. Huang, and N. Polyzotis. Max algorithms in crowdsourcing environments. In *WWW*, pages 989–998, 2012.
- [23] V. Verroios, P. Lofgren, and H. Garcia-Molina. tdp: An optimal-latency budget allocation strategy for crowdsourced MAXIMUM operations. In *SIGMOD*, pages 1047–1062, 2015.
- [24] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. CrowdER: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.
- [25] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, pages 229–240, 2013.
- [26] S. Wang, X. Xiao, and C. Lee. Crowd-based deduplication: An adaptive approach. In *SIGMOD*, pages 1263–1277, 2015.
- [27] L. Zou and L. Chen. Dominant graph: An efficient indexing structure to answer top- $k$  queries. In *ICDE*, pages 536–545, 2008.

# Cohesive Keyword Search on Tree Data

Aggeliki Dimitriou  
School of Electrical and Computer Engineering  
National Technical University of Athens, Greece  
angela@dblab.ntua.gr

Dimitri Theodoratos  
Department of Computer Science  
New Jersey Institute of Technology, USA  
dth@njit.edu

Ananya Dass  
Department of Computer Science  
New Jersey Institute of Technology, USA  
ad292@njit.edu

Yannis Vassiliou  
School of Electrical and Computer Engineering  
National Technical University of Athens, Greece  
yv@cs.ntua.gr

## ABSTRACT

Keyword search is the most popular querying technique on semistructured data. Keyword queries are simple and convenient. However, as a consequence of their imprecision, there is usually a huge number of candidate results of which only very few match the user's intent. Unfortunately, the existing semantics for keyword queries are ad-hoc and they generally fail to "guess" the user intent. Therefore, the quality of their answers is poor and the existing algorithms do not scale satisfactorily.

In this paper, we introduce the novel concept of cohesive keyword queries for tree data. Intuitively, a cohesiveness relationship on keywords indicates that they should form a cohesive whole in a query result. Cohesive keyword queries allow term nesting and keyword repetition. Cohesive keyword queries bridge the gap between flat keyword queries and structured queries. Although more expressive, they are as simple as flat keyword queries and not require any schema knowledge. We provide formal semantics for cohesive keyword queries and rank query results on the proximity of the keyword instances. We design a stack based algorithm which efficiently evaluates cohesive keyword queries. Our experiments demonstrate that our approach outperforms in quality previous filtering semantics and our algorithm scales smoothly on queries of even 20 keywords on large datasets.

## 1. INTRODUCTION

Keyword search has been by far the most popular technique for retrieving data on the web. The success of keyword search relies on two facts: First, the user does not need to master a complex structured query language (e.g., SQL, XQuery, SPARQL). This is particularly important since the vast majority of people who retrieve information from the web do not have expertise in databases. Second, the user does not need to have knowledge of the schema of the data sources over which the query is evaluated. In practice, in the

web, the user might not even be aware of the data sources which contribute results to her query. The same keyword query can be used to extract data from multiple data sources which have different schemas and they possibly adopt different data models (e.g., relational, tree, graph, flat documents).

There is a price to pay for the simplicity, convenience and flexibility of keyword search. Keyword queries are imprecise in specifying the query answer. They lack expressive power compared to structured query languages. As a consequence, there is usually a very large number of candidate results from which very few are relevant to the user intent. This weakness incurs at least two major problems: (a) because of the large number of candidate results, previous algorithms for keyword search are of high complexity and they cannot scale satisfactorily when the number of keywords and the size of the input dataset increase, and (b) correctly identifying the relevant results among the plethora of candidate results, becomes a very difficult task. Indeed, it is practically impossible for a search system to "guess" the user intent from a keyword query and the structure of the data source.

The focus of this work is on keyword search over web data which are represented as trees. Currently, huge amounts of data are exported and exchanged in tree-structure form [31, 33]. Trees can conveniently represent data that are semistructured, incomplete and irregular as is usually the case with data on the web. Multiple approaches assign semantics to keyword queries on data trees by exploiting structural and semantic features of the data in order to automatically filter out irrelevant results. Examples include Smallest LCA [18, 40, 37, 10], Exclusive LCA [16, 41, 42], Valuable LCA [11, 20], Meaningful LCA [24, 38], MaxMatch [28], Compact Valuable LCA [20] and Schema level SLCA [15]. A survey of some of these approaches can be found in [29]. Although filtering approaches are intuitively reasonable, they are sufficiently ad-hoc and they are frequently violated in practice resulting in low precision and/or recall. A better technique is adopted by other approaches which rank the candidate results in descending order of their estimated relevance. Given that users are typically interested in a small number of query results, some of these approaches combine ranking with top-k algorithms for keyword search. The ranking is performed using a scoring function and is usually based on IR-style metrics for flat documents (e.g., tf\*idf or PageRank) adapted to the tree-structure form of the data [16, 11, 3, 5, 21, 10, 38, 23, 29, 32]. Nevertheless,

scoring functions, keyword occurrence statistics and probabilistic models alone cannot capture effectively the intent of the user. As a consequence, the produced rankings are, in general, of low quality [38].

**Our approach.** In this paper, we introduce a novel approach which allows the user to specify cohesiveness relationships among keywords in a query, an option not offered by the current keyword query languages. Cohesiveness relationships group together keywords in a query to define indivisible (cohesive) collections of keywords. They partially relieve the system from guessing without affecting the user who can naturally and effortlessly specify such relationships.

For example, consider the keyword query `{XML John Smith George Brown}` to be issued against a large bibliographic database. The user is looking for publications on XML related to the authors John Smith and George Brown. If the user can express the fact that the instances of John and Smith should form a unit where the instances of the other keywords of the query George, Brown and XML cannot slip into to separate them (that is, the instances of John and Smith form a *cohesive whole*), the system would be able to return more accurate results. For example, it will be able to filter out publications on XML by John Brown and George Smith. It will also filter out a publication which cites a paper authored by John Davis, a report authored by George Brown and a book on XML authored by Tom Smith. This information is irrelevant and no one of the previous filtering approaches (e.g., ELCA, VLCA, CVLCA, SLCA, MLCA, MaxMach etc.) is able to automatically exclude it from the answer of the query. Furthermore, specifying cohesiveness relationships prevents wasting time searching for these irrelevant results. We specify cohesiveness relationships among keywords by enclosing them between parentheses. For example, the previous keyword query would be expressed as `(XML (John Smith) (George Brown))`.

Note that the importance of bridging the gap between flat keyword queries and structured queries in order to improve the accuracy (and possibly the performance) of keyword search has been recognized before for flat text documents. Google allows a user to specify, between quotes, a phrase which has to be matched intact against the text document. However, specifying a cohesiveness relationship on a set of keywords is different than phrase matching over flat text documents in IR. Indeed, a cohesiveness relationship on a number of keywords over a data tree does not impose any syntactic restriction (e.g., the order of the keywords) on the instances of these keywords on the data tree. It only requires that the instances of these keywords form a cohesive unit. We provide formal semantics for queries with cohesiveness relationships on tree data in Section 2.

Cohesiveness relationships can be nested. For instance the query `(XML (John Smith) (citation (George Brown)))` looks for a paper on XML by John Smith which cites a paper by George Brown. The cohesive keyword query language conveniently allows also for keyword repetition. For instance, the query `(XML (John Smith) (citation (John Brown)))` looks for a paper on XML by John Smith which cites a paper by John Brown.

Cohesive queries better express the user intent while being as simple as flat keyword queries. However, despite their increased expressive power, they enjoy both advantages of traditional keyword search: they do not require any previous knowledge of a query language or of the schema of the data

sources. The users can naturally and effortlessly specify cohesiveness relationships when writing a query. The benefits, though, in query answer quality and performance compared to other flat keyword search approaches are impressive.

**Contribution.** The main contributions of our paper are as follows:

- We formally introduce a novel keyword query language which allows for cohesiveness relationships, cohesiveness relationship nesting and keyword repetition. Our semantics interpret the subtrees rooted at the LCA of the instances of cohesively related keywords in the data tree as impenetrable units where the instances of the other keywords cannot slip in.
- We rank the results of cohesive keyword queries on tree data based on the concept of LCA size. The LCA size reflects the proximity of keywords in the data tree and, similarly to keyword proximity in IR, it is used to determine the relevance of the query results.
- We design an efficient multi-stack-based algorithm which exploits a lattice of stacks—each stack corresponding to a different partition of the query keywords. Our algorithm does not rely on auxiliary index structures and, therefore, can be exploited on datasets which have not been preprocessed.
- We show how cohesive relationships can be leveraged to lower the dimensionality of the lattice and dramatically reduce its size and improve the performance of the algorithm.
- We analyze our algorithm and show that for a constant number of keywords it is linear on the size of the input keywords' inverted lists, i.e., to the dataset size. Our analysis further shows that the performance of our algorithm essentially depends on the maximum cardinality of the largest cohesive term in the keyword query.
- We run extensive experiments on different real and benchmark datasets to assess the effectiveness of our approach and the efficiency and scalability of our algorithm. Our results show that our approach largely outperforms previous filtering approaches, which do not benefit from cohesiveness relationships, achieving in most cases perfect precision and recall. They also show that our algorithm scales smoothly when the number of keywords and the size of the dataset increase achieving interactive response times even with queries of 20 keywords having in total several thousands of instances on large datasets.

## 2. DATA AND QUERY MODEL

We consider data modeled as an ordered labeled tree. Every node has an id, a label and possibly a value. For identifying tree nodes we adopt the Dewey encoding scheme [9], which encodes tree nodes according to a preorder traversal of the data tree. The Dewey encoding scheme naturally expresses ancestor-descendant and parent-child relationships among tree nodes and conveniently supports the processing of nodes in stacks [16].

A keyword  $k$  may appear in the label or in the value of a node  $n$  in the data tree one or multiple times, in which case we say that node  $n$  constitutes an *instance* of  $k$ . A node may contain multiple distinct keywords in its value and label, in which case it is an instance of multiple keywords.

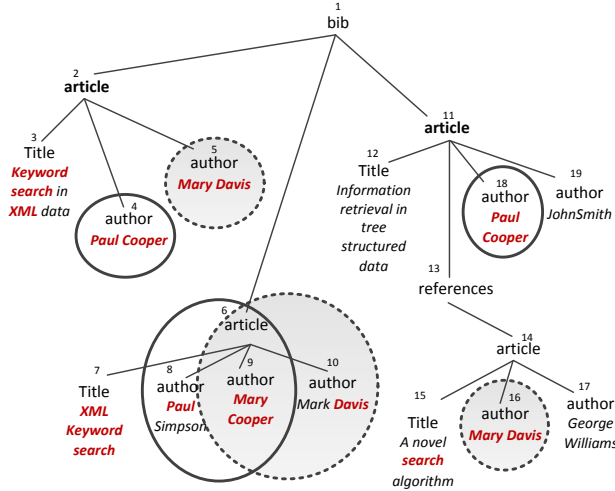


Figure 1: Example data tree  $D_1$

## 2.1 Cohesive semantics

A *cohesive keyword query* is a keyword query, which besides keywords may also contain groups of keywords called terms. Intuitively, a term expresses a cohesiveness relationship on keywords and/or terms. More formally, a cohesive keyword query is recursively defined as follows:

**DEFINITION 1 (COHESIVE KEYWORD QUERY).** A term is a multiset of at least two keywords and/or terms. A cohesive keyword query is: (a) a set of a single keyword, or (b) a term. Sets and multisets are delimited within a query using parentheses.

For instance, the expression  $((\text{title XML}) ((\text{John Smith}) \text{author}))$  is a keyword query. Some of its terms are  $T_1 = (\text{title XML})$ ,  $T_2 = ((\text{John Smith}) \text{author})$ ,  $T_3 = (\text{John Smith})$ , and  $T_3$  is nested into  $T_2$ .

A keyword may occur multiple times in a query. For example, in the keyword query  $((\text{journal (Information Systems)}) ((\text{Information Retrieval}) \text{Smith}))$  the keyword *Information* occurs twice, once in the term  $(\text{Information Systems})$  and once in the term  $(\text{Information Retrieval})$ .

In the rest of the paper, we may refer to a cohesive keyword query simply as query. The syntax of a query  $Q$  is defined by the following grammar where the non-terminal symbol  $T$  denotes a term, and the terminal symbol  $k$  denotes a keyword:

$$\begin{aligned} Q &\rightarrow (k) \mid T \\ T &\rightarrow (S S) \\ S &\rightarrow S S \mid T \mid k \end{aligned}$$

We now move to define the semantics of cohesive keyword queries. Keyword queries are embedded into data trees. In order to define query answers, we need to introduce the concept of query embedding. In cohesive keyword queries,  $m$  occurrences of the same keyword in a query are embedded to one or multiple instances of this keyword as long as these instances collectively contain at least  $m$  times this keyword. Cohesive keyword queries may also contain terms, which, as mentioned before, express a cohesiveness relationship on their keyword occurrences. In tree structured data, the keyword instances in the data tree (which are nodes) are represented by their LCA [36, 16, 29]. The instances of the

keywords of a term in the data tree should form a cohesive unit. That is, the subtree rooted at the LCA of the instances of the keywords which are internal to the term should be impenetrable by the instances of the keywords which are external to the term. Therefore, if  $l$  is the LCA of a set of instances of the keywords in a term  $T$ ,  $i$  is one of these instances and  $i'$  is an instance of a keyword not in  $T$ , then  $\text{lca}(i', i) = \text{lca}(i', l) \neq l$ .

As an example, consider query  $Q_1 = (\text{XML keyword search (Paul Cooper) (Mary Davis)})$  issued against the data tree  $D_1$  of Figure 1. In Figure 1, keyword instances are shown in bold and the instances of the keywords of a term below the same article are encircled. The mapping that assigns Paul to node 8, Mary and Cooper to node 9 and Davis to node 10 is not an embedding of  $Q_1$  since Mary slips into the encircled subtree of Paul and Cooper rooted at article node 6: the two circles of article node 6 overlap. These ideas are formalized below.

**DEFINITION 2 (QUERY EMBEDDING).** Let  $Q$  be a keyword query on a data tree  $D$ . An embedding of  $Q$  to  $D$  is a function  $e$  from every keyword occurrence in  $Q$  to an instance of this keyword in  $D$  such that:

- if  $k_1, \dots, k_m$  are distinct occurrences of the same keyword  $k$  in  $Q$  and  $e(k_1) = \dots = e(k_m) = n$ , then node  $n$  contains keyword  $k$  at least  $m$  times.
- if  $k_1, \dots, k_n$  are the keyword occurrences of a term  $T$ ,  $k$  is a keyword occurrence not in  $T$ , and  $l = \text{lca}(e(k_1), \dots, e(k_n))$  then: (i)  $e(k_1) = \dots = e(k_n)$ , or (ii)  $\text{lca}(e(k), l) \neq l$ .

Given an embedding  $e$  of a query  $Q$  involving the keyword occurrences  $k_1, \dots, k_m$  on a data tree  $D$ , the *minimum connecting tree* (MCT)  $M$  of  $e$  on  $D$  is the minimum subtree of  $D$  that contains the nodes  $e(k_1), \dots, e(k_m)$ . Tree  $M$  is also called an MCT of query  $Q$  on  $D$ . The root of  $M$  is the *lowest common ancestor* (LCA) of  $e(k_1), \dots, e(k_m)$  and defines one *result* of  $Q$  on  $D$ . For instance, one can see that the article nodes 2 and 11 are results of the example query  $Q_1$  on the example tree  $D_1$ . In contrast, the article node 6 is not a result of  $Q_1$ .

We use the concept of LCA size to rank the results in a query answer. Similarly to metrics for flat documents in information retrieval, LCA size reflects the proximity of keyword instances in the data tree. The *size* of an MCT is the number of its edges. Multiple MCTs of  $Q$  on  $D$  with different sizes might be rooted at the same LCA node  $l$ . The size of  $l$  (denoted  $\text{size}(l)$ ) is the minimum size of the MCTs rooted at  $l$ .

For instance, the size of the result article node 2 of query  $Q_1$  on the data tree  $D_1$  is 3 while that of the result article node 11 is 6 (note that there are multiple MCTs of different sizes rooted at node 11 in  $D_1$ ).

**DEFINITION 3.** The answer to a cohesive keyword query  $Q$  on a data tree  $D$  is a list  $[l_1, \dots, l_n]$  of the LCAs of  $Q$  on  $D$  such that  $\text{size}(l_i) \leq \text{size}(l_j), i < j$ .

For example, article node 2 is ranked above article node 11 in the answer of  $Q_1$  on  $D_1$ .

## 2.2 Result ranking using cohesive terms

The LCA size naturally reflects the overall proximity of the query keyword instances in the subtree of a result LCA. Every result LCA contains partial LCAs corresponding to

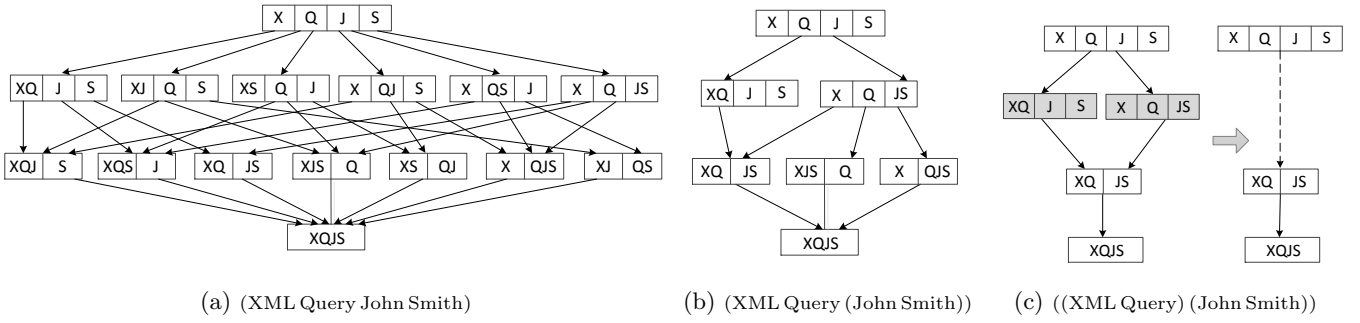


Figure 2: Lattices of keyword partitions for the query (XML Query John Smith) with different cohesiveness relationships

the nested cohesive terms in the query. These partial LCAs contribute with their size to the overall size of the LCA. However, we would like to take also into account how compactly the keyword instances are combined to form partial LCAs for each one of the nested cohesive terms. Consider, for instance, Figure 1 and the query  $Q_1 = (\text{XML keyword search (Paul Cooper) (Mary Davis)})$ . Article node 2 is an LCA for  $Q_1$  and author node 4 is a partial LCA for the cohesive term (Paul Cooper) contributing with LCA size 0 to the total size of the LCA node 2. The fact that the keyword instances of Paul and Cooper are compactly connected to form a partial LCA of size 0 is also important, as is that the total size of the result is small.

To reflect the importance of intra-cohesive-term proximity, we propose a new ranking scheme which takes also into account the cohesive terms and their sizes in a query result. Our ranking method does not require the preprocessing of the dataset and does not constrain the result representation to specific index terms [4].

We represent each query result as a vector in the cohesive term space for a given query  $Q$ . Let  $Q$  be a query with  $m$  cohesive terms, including the outermost term, i.e., the query itself. Each LCA  $l_j$  of a data tree  $D$  with respect to  $Q$  is represented by the following vector:

$$\vec{l}_j = (C_1 s_{1,j}, C_2 s_{2,j}, \dots, C_m s_{m,j})$$

where  $C_i$  is the weight of the term  $T_i$  in the query  $Q$  with respect to the dataset  $D$  and  $s_{i,j}$  is the size of the partial LCA for the term  $T_i$  within the LCA  $l_j$ . Intuitively, the parameter  $C_i$  reflects the compactness of the term  $T_i$  in the dataset  $D$ . That is, how closely the instances of the keywords in  $T_i$  appear in the partial LCAs for  $T_i$  in the dataset  $D$ . Let  $P_i$  be the set of the LCAs of a term  $T_i$  in  $D$ . Then,  $C_i$  is defined as follows:

$$C_i = \frac{|P_i|}{1 + \sum_{p \in P_i} \text{size}(p)}$$

The smaller the average size of the LCAs of a term in a dataset  $D$  the more compact the term is in  $D$ . The vector  $\vec{l}_j$  of an LCA  $l_j$  is used to define the score of  $l_j$ :

$$\text{score}(l_j) = |\vec{l}_j|$$

The query results are ranked in ascending order of their score. The weight  $C_i$  rewards results which demonstrate small sizes for non-compact terms and penalizes results that demonstrate large sizes for terms, which are expected to be compact.

### 3. THE ALGORITHM

We designed algorithm CohesiveLCA for keyword queries with cohesiveness relationships. Algorithm CohesiveLCA computes the results of a cohesive keyword query and ranks them in descending order of their LCA size. The idea behind CohesiveLCA is that LCAs of keyword instances in a data tree can result from combining LCAs of subsets of these instances (i.e., partial LCAs of the query) in a bottom-up way in the data tree. CohesiveLCA progressively combines partial LCAs to eventually return full LCAs of instances of all query keywords higher in the data tree. During this process, LCAs are grouped based on the keywords contained in their subtrees. The members of these groups are compared among each other in terms of their size. CohesiveLCA exploits a lattice of partitions of the query keywords.

**The lattice of keyword partitions.** During the execution of CohesiveLCA, multiple stacks are used. Every stack corresponds to a partition of the keyword set of the query. Each stack entry contains one element (partial LCA) for every keyword subset belonging to the corresponding partition. Stack based algorithms for processing tree structured data push and pop stack entries during the computation according to a preorder traversal of the data tree. Dewey codes are exploited to index stack entries which at any point during the execution of the algorithm correspond to a node in the data tree. Consecutive stack entries correspond to nodes related with parent-child relationships in the data tree.

The stacks used by algorithm CohesiveLCA are naturally organized into a lattice, since the partitions of the keyword set (which correspond to stacks) form a lattice. Coarser partitions can be produced from finer ones by combining two of their members. Partitions with the same number of members belong to the same coarseness level of the lattice. Figure 2a shows the lattice for the keyword set of the query (XML Query John Smith). CohesiveLCA combines partial LCAs following the source to sink paths in the lattice.

**Reducing the dimensionality of the lattice.** The lattice of keyword partitions for a given query consists of all possible partitions of query keywords. The partitions reflect all possible ways in which query keywords can be combined to form partial and full LCAs. Cohesiveness relationships restrict the ways keyword instances can be combined in a query embedding to form a query result. Keyword instances may be combined individually with other keyword instances to form partial or full LCAs only if they belong to the same term: if a keyword  $a$  is “hidden” from a keyword  $b$  inside a



term  $T_a$ , then an instance of  $b$  can only be combined with an LCA of all the keyword instances of  $T_a$  and not individually with an instance of  $a$ . These restrictions result in significantly reducing the size of the lattice of the keyword partitions as exemplified next.

Figures 2b and 2c show the lattices of the keyword partitions of two queries. The queries comprise the same set of keywords **XML**, **Query**, **John** and **Smith** but involve different cohesive relationships. The lattice of Figure 2a is the full lattice of 15 keyword partitions and allows every possible combination of instances of the keywords **XML**, **Query**, **John** and **Smith**. The query of Figure 2b imposes a cohesiveness relationship on **John** and **Smith**. This modification renders several partitions of the full lattice of Figure 2a meaningless. For instance, in Figure 2b, the partition **[XJ, Q, S]** is eliminated, since an instance of **XML** cannot be combined with an instance of **John** unless the instance of **John** is already combined with an instance of **Smith**, as is the case in the partition **[XJS, Q]**. The cohesiveness relationship on **John** and **Smith** reduces the size of the initial lattice from 15 to 7. A second cohesiveness relationship between **XML** and **Query** further reduces the lattice to the size of 3, as shown in Figure 2c. Note that in this case, besides partitions that are not permitted because of the additional cohesiveness relationship (e.g., **[XJS, Q]**), some partitions may not be productive, which makes them useless. **[XQ, J, S]** is one such partition. The only valid combination of keyword instances that can be produced from this partition is **[XQ, JS]**, which is a partition that can be produced directly from the source partition **[X, Q, J, S]** of the lattice. The same holds also for the partition **[X, Q, JS]**. Thus, these two partitions can be eliminated from the lattice.

**Algorithm description.** Algorithm *CohesiveLCA* accepts as input a cohesive keyword query and the inverted lists of the query keywords and returns all LCAs which satisfy the cohesiveness relationships of the query, ranked on their LCA size.

The algorithm begins by building the lattice of stacks needed for the cohesive keyword query processing (line 2). This process will be explained in detail in the next paragraph. After the lattice is constructed, an iteration over the inverted lists (line 3) pushes all keyword instances into the lattice in Dewey code order starting from the source stack of the lattice, which is the only stack of coarseness level 0. For every new instance, a round of sequential processing of all coarseness levels is initiated (lines 6-9). At each step, entries are pushed and popped from the stacks of the current coarseness level. Each stack has multiple columns corresponding to and named by the keyword subsets of the relevant keyword partition. Each stack entry comprises a number of elements one for every column of the stack. The constructor *PartialLCA* produces a partial LCA element taking as parameters the Dewey code of a node, the term corresponding to a keyword partition, the size of the partial LCA and its provenance. Popped entries contain partial LCAs that are propagated to the next coarseness levels. An entry popped from the sink stack (located in the last coarseness level) contains a full LCA and constitutes a query result. After finishing the processing of all inverted lists, an additional pass over all coarseness levels empties the stacks producing the last results (line 10).

Procedure *push()* pushes an entry into a stack after ensuring that the top stack entry corresponds to the parent of

the partial LCA to be pushed (lines 11-16). This process triggers pop actions of all entries that do not correspond to ancestors of the entry to be pushed. Procedure *pop()* is where partial and full LCAs are produced (lines 17-34). When an entry is popped, new LCAs are formed (lines 21-28) and the parent entry of the popped entry is updated to incorporate partial LCAs that come from the popped child entry (lines 29-34). The construction of new partial LCAs is performed by combining LCAs stored in the same entry.

**Construction of the lattice.** The key feature of *CohesiveLCA* is the dimensionality reduction of the lattice which is induced by the cohesiveness relationships of the input query. This reduction, as we also show in our experimental evaluation, has a significant impact on the efficiency of the algorithm. Algorithm *CohesiveLCA* does not naively prune the full lattice to produce a smaller one, but wisely constructs the lattice needed for the computation from smaller component sublattices. This is exemplified in Figure 3.

Consider the data tree depicted in Figure 1 and the query

---

### Algorithm 1: CohesiveLCA

---

```

1 CohesiveLCA( $Q$ : cohesive keyword query,  $invL$ : inverted
  lists)
2   buildLattice()
3   while  $currentNode \leftarrow getNextNodeFromInvertedLists()$ 
4     do
5        $curPLCA \leftarrow PartialLCA(currentNode.dewey,$ 
6          $currentNode.kw, 0, null)$ 
7        $push(initStack, curPLCA)$ 
8       for every coarsenessLevel  $cL$  do
9         while  $pl \leftarrow next\ partial\ LCA\ of\ cL$  do
10          for every stack  $S$  of  $cL$  containing  $pl.term$ 
11           do
12              $push(S, pl)$ 
13   emptyStacks()
14 push( $S$ : stack,  $pl$ : partial LCA)
15 while  $S.dewey$  not ancestor of  $pl.node$  do
16    $pop(S)$ 
17 while  $S.dewey \neq pl.node$  do
18    $addEmptyRow(S)$ 
19    $replaceIfSmallerWith(S.topRow, pl.term, pl.size)$ 
20 pop( $S$ : stack)
21  $p \leftarrow S.pop()$ 
22 if  $S.columns = 1$  then
23    $addResult(S.dewey, p[0].size)$ 
24 if  $S.columns > 1$  then
25   for  $i \leftarrow 0$  to  $S.columns$  do
26     for  $j \leftarrow i$  to  $S.columns$  do
27       if  $p[i]$  and  $p[j]$  contain sizes and
28          $p[i].provenance \cap p[j].provenance = \emptyset$  then
29          $t \leftarrow findTerm(p[i].term, p[j].term)$ 
30          $sz \leftarrow p[i].size + p[j].size$ 
31          $prv \leftarrow p[i].provenance \cup p[j].provenance$ 
32          $pLCA \leftarrow PartialLCA(S.dewey, t, sz, prv)$ 
33 if  $S$  is not empty and  $S.columns > 1$  then
34   for  $i \leftarrow 0$  to  $S.columns$  do
35     if  $p[i].size + 1 < S.topRow[i].size$  then
36        $S.topRow[i].size \leftarrow p[i].size + 1$ 
37        $S.topRow[i].provenance \leftarrow$ 
38          $\{lastStep(S.dewey)\}$ 
39    $removeLastDeweyStep(S.dewey)$ 

```

---

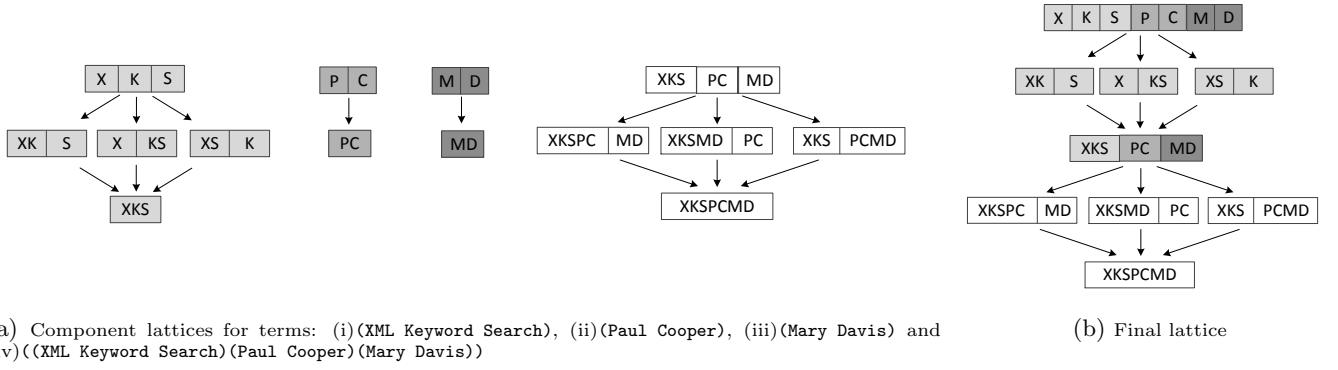


Figure 3: Component and final lattices for the query ((XML Keyword Search) (Paul Cooper) (Mary Davis))

((XML Keyword Search) (Paul Cooper) (Mary Davis)) issued on this data tree. If each term is treated as a unit, a lattice of the partitions of three items is needed for the evaluation of the query. This is lattice (iv) of Figure 3a. However, the input of this lattice consists of combinations of keywords and not of single keywords. These three combinations of keywords each defines its own lattice shown in the left side of Figure 3a (lattices (i), (ii) and (iii)). The lattice to be finally used by the algorithm CohesiveLCA is produced by composing lattices (i), (ii) and (iii) with lattice (iv) and is shown in Figure 3b. This is a lattice of only 9 nodes, whereas the full lattice for 7 keywords has 877 nodes.

Function `buildLattice()` constructs the lattice for evaluating a cohesive keyword query. This function calls another function `buildComponentLattice()` (line 8). Function `buildComponentLattice()` (lines 18-21) is a recursive and builds all lattices for all terms which may be arbitrarily nested. The whole process is controlled by the `controlSets` variable which stores the keyword subsets admissible by the input cohesiveness relationships. This variable is constructed by the procedure `constructControlSet()` (lines 9-17).

### 3.1 Algorithm analysis

Algorithm CohesiveLCA processes the inverted lists of the keywords of a query exploiting the cohesiveness relationships to limit the size of the lattice of stacks used. The size of a lattice of a set of keywords with  $k$  keywords is given by the Bell number of  $k$ ,  $B_k$ , which is defined by the recursive formula:

$$B_{n+1} = \sum_{i=0}^n \binom{n}{i} B_i, \quad B_0 = B_1 = 1$$

In a cohesive query containing  $t$  terms the number of sublattices is  $t+1$  counting also the sublattice of the query (outer term). The size of the sublattice of a term with cardinality  $c_i$  is  $B_{c_i}$ . A keyword instance will trigger in the worst case an update to all the stacks of all the sublattices of the terms in which the keyword participates. If the maximum nesting depth of terms in the query is  $n$  and the maximum cardinality of a term or of the query itself is  $c$ , then an instance will trigger  $O(nB_c)$  stack updates. For a data tree with depth  $d$ , every processing of a partial LCA by a stack entails in the worst case  $d$  pops and  $d$  pushes, i.e.,  $O(d)$ . Every pop from a stack with  $c$  columns implies in the worst case  $c(c-1)/2$

---

#### Function `buildLattice`

---

```

1 buildLattice( $Q$ : query)
2   singletonTerms  $\leftarrow$  {keywords( $Q$ )}
3   stacks.add(createSourceStack(singletonTerms))
4   constructControlSet( $Q$ ) for every control set cset in
   controlSets with not only singleton keywords do
5     | stacks.add(createSourceStack(cset))
6   for every s in stacks do
7     | buildComponentLattice(s)
8 constructControlSet( $qp$ : query subpattern)
9    $c \leftarrow$  new Set()
10  for every singleton keyword k in s do
11    | c.add(k)
12  for every subpattern sqp in s do
13    | subpatternTerm  $\leftarrow$  constructControlSet(sqp)
14    | c.add(subpatternTerm)
15  controlSets.add(c)
16  return newTerm(c)
17 buildComponentLattice( $s$ : stack)
18  for every pair t1, t2 of terms in s do
19    | newS  $\leftarrow$  newStack(s, t1, t2)
20    | buildComponentLattice(newS)

```

---

combinations to produce partial LCAs and  $c$  size updates to the parent node, i.e.,  $O(c^2)$ . Thus, the time complexity of CohesiveLCA is given by the formula:

$$O(dnc^2 B_c \sum_{i=1}^c |S_i|)$$

where  $S_i$  is the inverted list of the keyword  $i$ . The maximum term cardinality for a query with a given number of keywords depends on the number of query terms. It is achieved by the query when all the terms contain one keyword and one term with the exception of the innermost nested term which contains two keywords. Therefore, the maximum term cardinality is  $k-t-1$  and the maximum nesting depth is  $t$ . Thus, the complexity of CohesiveLCA is:

$$O(dt(k-t-1)^2 B_{k-t-1} \sum_{i=1}^k |S_i|)$$

This is a parameterized complexity which is linear to the size of the input (i.e.,  $\sum |S_i|$ ) for a constant number of keywords and terms.

	DBLP	XMark	NASA	PSD	Baseball
size	1.15 GB	116.5 MB	25.1 MB	683 MB	1.1 MB
maximum depth	5	11	7	6	5
# nodes	34,141,216	2,048,193	530,528	22,596,465	26,432
# keywords	3,403,570	140,425	69,481	2,886,921	1984
# distinct labels	44	77	68	70	46
# dist. label paths	196	548	110	97	46

Table 1: DBLP, XMark, NASA, PSD and Baseball dataset statistics

## 4. EXPERIMENTAL EVALUATION

We implemented our algorithm and we experimentally studied: (a) the effectiveness of the CohesiveLCA semantics and (b) the efficiency of the CohesiveLCA algorithm.

The experiments were conducted on a computer with a 1.6GHz dual core Intel Core i5 processor running Mac OS 10.8. The code was implemented in Java.

### 4.1 Datasets and queries

We used four real datasets: the bibliographic dataset DBLP<sup>1</sup>, the astronomical dataset NASA<sup>2</sup>, the Protein Sequence Database (PSD)<sup>3</sup> and the sports statistics dataset Baseball<sup>4</sup>. We also used a synthetic dataset, the benchmark auction dataset XMark<sup>5</sup>. These datasets cover different application areas and display various characteristics. Table 1 shows their statistics.

The DBLP is the largest and XMark the deepest dataset. For the effectiveness experiments, we used the real datasets DBLP, PSD, NASA and Baseball. For the efficiency evaluation, we used the DBLP, XMark and NASA datasets in order to test our algorithm on data with different structural and size characteristics. The keyword inverted lists of the parsed datasets were stored in a MySQL database.

We selected five cohesive keyword queries for each one of the four real datasets with an intuitive meaning. The queries display various cohesiveness patterns and involve 3-6 keywords. They are listed in Table 2. The binary relevance (correctness) and graded relevance assessments of all the LCAs were provided by five expert users. For the graded relevance, a 4-value scale was used with 0 denoting irrelevance. In order to avoid the manual assessment of each LCA in the XML tree, which is unfeasible because the LCAs are usually numerous, we used the tree patterns that the query instances of these LCAs define in the XML tree. These patterns show how the query keyword instances are combined under an LCA to form an MCT, and how the LCA is connected to the root of the data tree. Since they are bound by the schema of a data set they are in practice much less numerous than the LCAs. The relevance of an LCA is the maximum relevance of the patterns with which the query instances of the LCA comply.

### 4.2 Effectiveness of cohesive semantics

In this section we evaluate the effectiveness of cohesive semantics both as a filtering and as a ranking mechanism.

**Filtering cohesive semantics.** We compared the CohesiveLCA semantics with the *smallest* LCA (SLCA) [18, 40,

<sup>1</sup><http://www.informatik.uni-trier.de/ley/db/>

<sup>2</sup><http://www.cs.washington.edu/research/xmldatasets/www/repository.html>

<sup>3</sup><http://pir.georgetown.edu/>

<sup>4</sup><http://ibiblio.org/xml/books/biblegold/examples/baseball/>

<sup>5</sup><http://www.xml-benchmark.org>

DBLP	
$Q_1^D$	((proof (Scott theorem)))
$Q_2^D$	((IEEE transactions communications) (wireless networks))
$Q_3^D$	((Lei Chen) (Yi Guo))
$Q_4^D$	((Wei Wang) (Yi Chen))
$Q_5^D$	((VLDB journal) (spatial databases))
PSD	
$Q_1^P$	((african snail) mRNA)
$Q_2^P$	((alpha 1) (isoform 3))
$Q_3^P$	((penton protein) (human adenovirus 5))
$Q_4^P$	((B cell) stimulating factor) (house mouse))
$Q_5^P$	((spectrin gene) (alpha 1))
NASA	
$Q_1^N$	((ccd photometric system) magnitudes)
$Q_2^N$	((stars types) (spectral classification))
$Q_3^N$	((Astronomical (Data Center)) (Wilson luminosity codes))
$Q_4^N$	((year 1968) (Zwicky Abell clusters))
$Q_5^N$	((title Orion Nebula) (author Parenago))
Baseball	
$Q_1^B$	((Matt Williams (third base))
$Q_2^B$	((team (Johnson (first base)) (Wilson pitcher))
$Q_3^B$	((player surname (0 errors))
$Q_4^B$	((player (relief pitcher) (0 losses))
$Q_5^B$	((player (0 errors) (7 games))

Table 2: Queries for the effectiveness experiments on various datasets

37, 10], *exclusive* LCA (ELCA) [16, 41, 42], *valuable* LCA (VLCA) [11, 20] and *meaningful* LCA [24] (MLCAs) filtering semantics. These are the best known filtering semantics discussed in the literature. An LCA is an SLCA if it is not an ancestor of another LCA in the data tree. An ELCA is an LCA of a set of keyword instances which are not in the subtree of any descendant LCA. An LCA is a VLCA if it is the root of an MCT which does not contain any label twice except when it is the label of two leaf nodes of the MCT. The MLCA semantics requires that for any two nodes  $n_a$  and  $n_b$  labeled by  $a$  and  $b$ , respectively, in an MCT, no node  $n'_b$  labeled by  $b$  exists which is more closely related to  $n_a$  (i.e.,  $lca(n_a, n'_b)$  is descendant of  $lca(n_a, n_b)$ ). SLCA and ELCA semantics are based purely on structural characteristics, while VLCA and MLCA take also into account the labels of the nodes in the data tree.

Table 3 displays the number of results for each query and approach on the DBLP, PSD, NASA and Baseball datasets. Notice that, with the exception of SLCA and ELCA which satisfy a containment relationship (SLCA  $\subseteq$  ELCA), all other approaches are pairwise incomparable. That is, one might return results that the other excludes and vice versa. The CohesiveLCA approach returns all the results that satisfy the cohesiveness relationships in the query. Since these relationships are imposed by the user, any additional result returned by another approach is irrelevant. For instance, for query  $Q_5^P$ , only 3 results satisfy the cohesiveness relationships of the user, and therefore, SLCA, VLCA, MLCA return at least 37 and ELCA at least 40 irrelevant results.

The CohesiveLCA semantics is a ranking semantics and ranks the results in layers based on their size. In order to

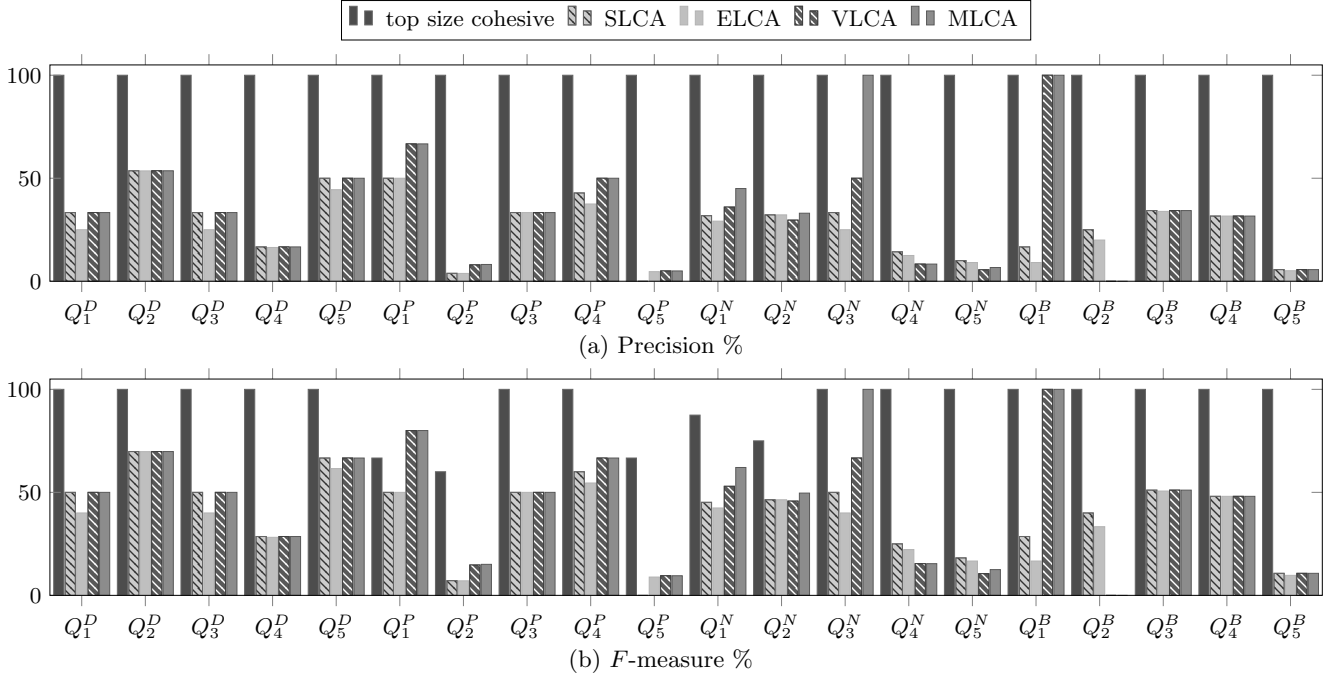


Figure 4: Precision and  $\mathcal{F}$ -measure of top size Cohesive LCA, SLCA and ELCA filtering semantics

compare its effectiveness with filtering semantics we restrict the results to the top layer (top-1-size results). Recall that these are the results with the minimum LCA size. The comparison is based on the widely used *precision* ( $P$ ), *recall* ( $R$ ) and  $\mathcal{F}$ -measure =  $\frac{2P \times R}{P+R}$  metrics [4]. Figure 4 depicts the results for the five semantics on the four datasets. Since all approaches demonstrate high recall, we only show the precision and  $\mathcal{F}$ -measure results in the interest of space.

The diagram of Figure 4a shows that CohesiveLCA largely

outperforms the other approaches in all cases. Top-1-size CohesiveLCA shows perfect precision for all queries on all datasets. This is not surprising since, CohesiveLCA can benefit from cohesiveness relationships specified by the user to exclude irrelevant results. CohesiveLCA also shows perfect  $\mathcal{F}$ -measure on the DBLP and Baseball datasets. Its  $\mathcal{F}$ -measure on the PSD and NASA datasets is lower. This is due to the following reason: contrary to the shallow DBLP and Baseball datasets, the PSD and NASA datasets are deep and complex with a large amount of text in their nodes. This complexity leads to results of various sizes for most of the queries. Some of the relevant results are not of minimum size and they are missed by top-1-size CohesiveLCA. Nevertheless, any relevant result missed by top-1-size CohesiveLCA is retrieved by CohesiveLCA which returns all relevant results, as one can see in Table 4.

Table 4 summarizes the precision, recall and  $\mathcal{F}$ -measure values of the queries on all four datasets. The table displays values for the five filtering semantics but also for the CohesiveLCA semantics (without restricting the size of the results). Both CohesiveLCA and top-1-size CohesiveLCA outperform the other approaches in all three metrics. Top-1-size CohesiveLCA demonstrates perfect precision while CohesiveLCA with a slightly lower precision guarantees perfect recall. These remarkable results on the effectiveness of our approach are obtained thanks to the cohesiveness relation-

dataset	query	# of results				
		CohesiveLCA	SLCA	ELCA	VLCA	MLCA
DBLP	$Q_1^D$	2	3	4	3	3
	$Q_2^D$	527	981	982	981	981
	$Q_3^D$	2	3	4	3	3
	$Q_4^D$	11	60	61	60	60
	$Q_5^D$	5	8	9	8	8
PSD	$Q_1^P$	3	2	3	3	3
	$Q_2^P$	14	78	79	88	85
	$Q_3^P$	2	4	4	3	3
	$Q_4^P$	4	7	8	6	6
	$Q_5^P$	3	40	43	40	40
NASA	$Q_1^N$	17	22	24	25	20
	$Q_2^N$	85	90	90	118	106
	$Q_3^N$	1	3	4	2	1
	$Q_4^N$	6	7	8	12	12
	$Q_5^N$	9	10	11	18	15
Baseball	$Q_1^B$	10	5	6	1	1
	$Q_2^B$	7	4	5	0	0
	$Q_3^B$	216	516	522	516	516
	$Q_4^B$	145	335	335	335	335
	$Q_5^B$	49	177	196	177	177

Table 3: Number of results of queries on various datasets

	CohesiveLCA	top-1-size CohesiveLCA	SLCA	ELCA	VLCA	MLCA
Precision %	67.4	100	25.1	27.6	32.6	35.7
Recall %	100	96.9	88.0	93.0	95.0	95.0
$\mathcal{F}$ -measure %	76.8	92.8	39.8	36.8	44.4	46.8

Table 4: Average precision, recall and  $\mathcal{F}$ -measure values over all queries and datasets for all semantics

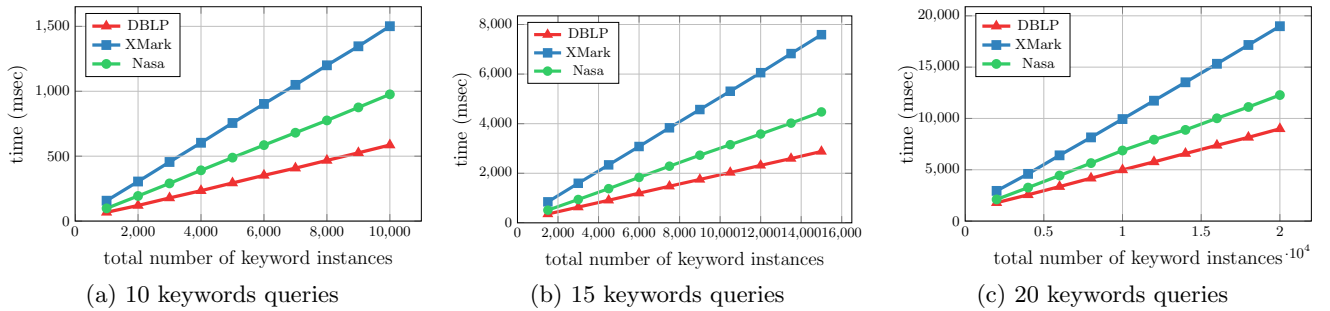


Figure 5: Performance of CohesiveLCA for queries with 10, 15 and 20 keywords varying the number of instances

ships which are provided effortlessly by the user.

**Ranking cohesive semantics** We evaluated our result ranking scheme by computing the Mean Average Precision (MAP) [4] and the Normalized Discounted Cumulative Gain (NDCG) [4] on the queries of Table 2. MAP is the average of the individual precision values that are obtained every time a correct result is observed in the ranking of the query. If a correct result is never retrieved, its contributing precision value is 0. MAP penalizes an algorithm when correct results are missed or incorrect ones are ranked highly. Given a specific position in the ranking of a query result set, the Discounted Cumulative Gain (DCG) is defined as the sum of the grades of the query results until this ranking position, divided (discounted) by the logarithm of that position. The DCG vector of a query is the vector of the DCG values at the different ranking positions of the query result set. Then, the NDCG vector is the result of normalizing this vector with the vector of the perfect ranking (i.e., the one obtained by the grading of the result set by the experts). NDCG penalizes an algorithm when the latter favors poorly graded results over good ones in the ranking.

MAP (%)			
DBLP	PSD	NASA	Baseball
94	99	94	97
NDCG (%)			
DBLP	PSD	NASA	Baseball
100	99	98	100

Table 5: MAP and NDCG measurements on the four datasets for the queries of Table 2

Table 5 shows the MAP and NDCG values of the cohesive ranking on the queries of Table 2. The excellent values of NDCG show that the ranking in ascending order of the scores of the result LCAs (which take into account the partial LCA sizes and the cohesive term weights) is very close to the correct ranking of the results provided by the expert users. Most MAP values are slightly inferior to 100. This means that a small number of non-relevant LCAs are ranked higher than some relevant. However, the high NDCG values, guarantee that these LCAs are not highly located in the total rank.

### 4.3 Efficiency of the CohesiveLCA algorithm

In order to study the efficiency of our algorithm we run

experiments to measure: (a) its performance scalability on the dataset size, (b) its performance scalability on the query maximum term cardinality and (c) the improvement in efficiency over previous approaches. We used collections of queries with 10, 15 and 20 keywords issued against the DBLP, XMark and NASA datasets. For each query size, we formed 10 cohesive query patterns. Each pattern involves a different number of terms of different cardinalities nested in various depths. For instance, a query pattern for a 10-keyword query is  $(xx((xxxx)(xxxx)))$ . We used these patterns to generate keyword queries on the three datasets. The keywords were chosen randomly. In order to stress our algorithm, they were selected among the most frequent ones. In particular, for each pattern, we generated 10 different keyword queries and we calculated their average evaluation time. We generated, in total, 100 queries for each dataset. For each query, we run experiments scaling the size of each keyword inverted list from 100 to 1000 instances with a step of 100 instances.

**Performance scalability on dataset size.** Figure 5 shows how the computation time of CohesiveLCA scales when the total size of the query keyword inverted lists grows. Each plot corresponds to a different query size (10, 15 or 20 keywords) and displays the performance of CohesiveLCA on the three datasets. Each curve corresponds to a different dataset and each point in a curve represents the average computation time of the 100 queries that conform to the 10 different patterns of the corresponding query size. Since the keywords are randomly selected among the most frequent ones this figure reflects the performance scalability with respect to the dataset size.

All plots clearly show that the computation time of CohesiveLCA is linear on the dataset size. This pattern is followed, in fact, by each one of the 100 contributing queries. In all cases, the evaluation times on the different datasets confirm the dependence of the algorithm’s complexity on the maximum depth of the dataset: the evaluation on DBLP (max depth 5) is always faster than on NASA (max depth 7) which in turn is faster than on XMark (max depth 11).

It is interesting to note that our algorithm achieves interactive computation times even with multiple keyword queries and on large and complex datasets. For instance, a query with 20 keywords and 20,000 instances needs only 20 sec to be computed on the XMark dataset. These results are achieved on a prototype without the optimizations of a commercial keyword search system. To the best of our knowledge, there is no other experimental study in the relevant literature that considers queries of such sizes.

**Performance scalability on max term cardinality.** As we showed in the analysis of algorithm CohesiveLCA (Section 3.1), the key factor which determines the algorithm’s performance is the maximum term cardinality in the input query. The maximum term cardinality determines the size of the largest sublattice used for the construction of the lattice ultimately used by the algorithm (see Figure 3). This dependency is confirmed in the diagram of Figure 6. We used queries of 10, 15 and 20 keywords with a total number of 6000 instances each on the DBLP dataset. The x axis shows the maximum term cardinality of the queries. The computation time shown by the bars (left y axis) is averaged over all the queries of a query size with the corresponding maximum cardinality. The curve displays the evolution of the size of the largest sublattice as the maximum term cardinality increases. The size of the sublattice is measured by the number of the stacks it contains (right y axis).

It is interesting to observe that the computation time depends primarily on the maximum term cardinality and to a much lesser extent on the total number of keywords. For instance, a query of 20 keywords with maximum term cardinality 6 is computed much faster than a query of 10 keywords with maximum term cardinality 7. This observation shows that as long as the terms involved are not huge, CohesiveLCA is able to efficiently compute queries with a very large number of keywords.

**Performance improvement with cohesive relationships.** In order to study the improvement in execution time brought by the cohesiveness relationships to previous algorithms, we compare CohesiveLCA with algorithms which compute a superset of LCAs and rank them with respect to their size. In these cases, since taking into account cohesiveness relationships filters out irrelevant results, the quality of the answer is improved. It is not meaningful to compare with other algorithms since their result sets are incomparable to that of CohesiveLCA (no result set is inclusive of the other) and they do not rank their results on their size. For the experiments we used the DBLP dataset—the results on the other datasets are similar.

There are two previous algorithms that can compute the full set of LCAs with their sizes: algorithm LCAsz [13, 14] and algorithm SA [17]. LCAsz is an algorithm that computes all the LCAs with their sizes which, similarly to CohesiveLCA, exploits a lattice of stacks. The SA algorithm [17]

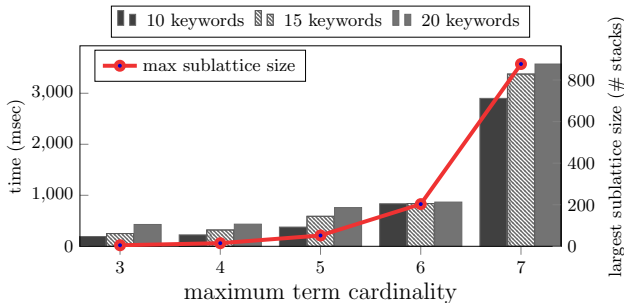


Figure 6: Performance of CohesiveLCA on queries with 6000 keyword instances for different maximum term cardinalities on the DBLP dataset

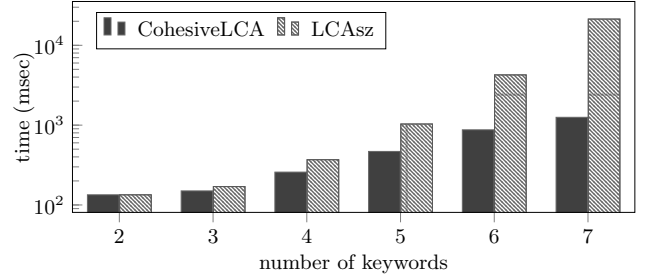


Figure 7: Improvement of CohesiveLCA over LCAsz varying the number of query keywords

computes all LCAs together with a compact form of their matching MCTs, called GDMCTs, which allows determining the size of an LCA. We implemented algorithm SAOne [17] which is a more efficient version of SA since it computes LCAs without explicitly enumerating all the GDMCTs.

Figure 7 compares the execution time of LCAsz and CohesiveLCA in answering keyword queries on the DBLP dataset varying the number of keywords. The execution time of LCAsz is averaged over 10 random queries with frequent keywords. Various cohesiveness relationships patterns were defined for CohesiveLCA (their number depends on the total number of keywords), and for each one of them 10 random queries of frequent keywords were generated. The execution time for CohesiveLCA is averaged over all the queries generated with the same number of keywords. In all cases, each keyword inverted list was restricted to 1000 instances.

As we can see in Figure 7, CohesiveLCA outperforms LCAsz. The improvement reaches an order of magnitude for 6 keywords and increases for 7 keywords or more. Further, CohesiveLCA scales smoothly compared to LCAsz since, as explained above, its performance is dependent on the maximum term cardinality, as opposed to the total number of keywords that determines the performance of LCAsz.

Figure 8 compares the execution times of CohesiveLCA, LCAsz and SAOne for queries of 6 keywords varying the total number of keyword instances. The measurements are average execution times over multiple random queries and cohesiveness relationships patterns (for CohesiveLCA) as in the previous experiment. As one can see, CohesiveLCA clearly outperforms all previous approaches. LCAsz, in turn, largely outperforms SAOne which also scales worse than the other two

## 5. RELATED WORK

Keyword queries facilitate the user with the ease of freely forming queries by using only keywords. Approaches that evaluate keyword queries are currently very popular especially in the web where numerous sources contribute data often with unknown structure and where end users with no specialized skills need to find useful information. However, the imprecision of keyword queries results often in low precision and/or recall of the search systems. Some approaches combine structural constraints with keyword search [11]. Other approaches try to infer useful structural information implied by keyword queries by exploiting statistical information of the query keywords on the underlying datasets [5, 22, 38, 25, 7]. These approaches require a minimum knowledge

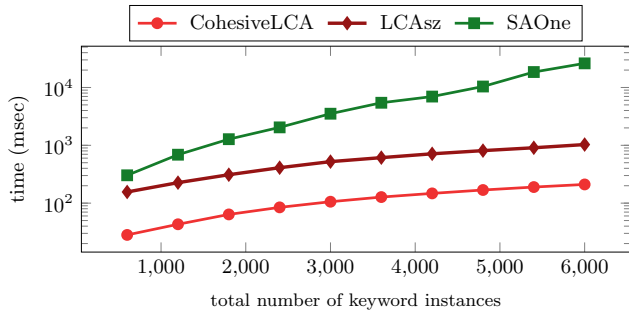


Figure 8: CohesiveLCA scaling comparison with other approaches for queries of 6 keywords

of the dataset or a heavy dataset preprocessing in order to be able to accurately assess candidate keyword query results. A great amount of previous work elaborates on keyword query evaluation on graph data (e.g., RDF databases or graphs extracted from Relational databases) [18, 12, 30, 7]. However, the focus of this work is on tree data.

The task of locating the nodes in a data tree which most likely match a keyword query has been extensively studied in [11, 18, 17, 20, 27, 41, 19, 22, 10, 13, 6, 23, 26, 38, 32, 2, 14, 1]. All these approaches use LCAs of keyword instances as a means to define query answers. The *smallest* LCA (SLCA) semantics [40, 28] validates LCAs that do not contain other descendant LCAs of the same keyword set. A relaxation of this restriction is introduced by *exclusive* LCA (ELCA) semantics [16, 41], which accepts also LCAs that are ancestors of other LCAs, provided that they refer to a different set of keyword instances.

In a slightly different direction, semantic approaches account also for node labels and node correlations in the data tree. *Valuable* LCAs (VLCAs) [11, 20] and *meaningful* LCAs [24] (MLCAs) aim at “guessing” the user intent by exploiting the labels that appear in the paths of the subtree rooted at an LCA. All these semantics are restrictive and depending on the case, they may demonstrate low recall rates as shown in [38].

The efficiency of algorithms that compute LCAs as answers to keyword queries depend on the query semantics adopted. By design they exploit the adopted filtering semantics to prune irrelevant LCAs early on in the computation. Stack based algorithms are naturally offered to process tree data. In [16] a stack-based algorithm that processes inverted lists of query keywords and returns ranked ELCAs was presented. This approach ranks also the query results based on precomputed tree node scores inspired by PageRank [8] and IR style keyword proximity in the subtrees of the ranked ELCAs. An algorithm that computes all the LCAs ranked on LCA size is presented in [13, 14]. In [40], two efficient algorithms for computing SLCAs are introduced, exploiting special structural properties of SLCAs. This approach also introduces an extension of the basic algorithm, so that it returns all LCAs by augmenting the set of already computed SLCAs. Another algorithm for efficiently computing SLCAs for both AND and OR keyword query semantics is developed in [37]. The Indexed Stack [41] and the Hash Count [42] algorithms improve the efficiency of [16] in computing ELCAs. Finally, [5, 6] elaborate on sophisticated ranking of candidate LCAs aiming primarily on effective keyword

query answering.

Filtering semantics are often combined with (i) structural and semantic correlations [16, 21, 10, 38, 11, 2], (ii) statistical measures [16, 11, 21, 10, 22, 38] and (iii) probabilistic models [38, 32, 23] to perform a ranking to the results set. Nevertheless, such approaches require expensive preprocessing of the dataset which makes them impractical in the cases of fast evolving data and streaming applications.

The most well known ranking models in the literature of IR assume term independency [4]. Extensions of the basic ranking models such as the generalized vector model [39] and the set based vector model [34] represent queries and documents using sets of terms. However, the ranking scheme in these models requires preprocessing of the data collection to compute  $tf * idf$  style metrics for a representative subset of the term vocabulary. The work in [35] enhances keyword queries with structure to extract information from knowledge bases. However, their approach targets graph databases with semantic information and their queries are schema dependent. In contrast, our cohesive queries only contain groupings of the keywords expressing cohesiveness relationships which are not related to any schema construct. As such, the same cohesive query can be issued against any type of dataset (flat text documents, trees, graphs, etc.).

## 6. CONCLUSION

Current approaches for assigning semantics to keyword queries on tree data cannot cope efficiently or effectively with the large number of candidate results and produce answers of low quality. The convenience and simplicity offered to the user by the keyword queries cannot offset this weakness. In this paper, we claim that the search systems cannot guess the user intent from the query and the characteristics of the data to produce high quality answers on any type of dataset and we introduce a cohesive keyword query language which allows the users to naturally and effortlessly express cohesiveness relationships on the query keywords. We design an algorithm which builds a lattice of stacks to efficiently compute cohesive keyword queries and rank the results leveraging cohesiveness relationships to reduce the lattice dimensionality. A theoretical analysis and experimental evaluation show that our approach outperforms previous approaches in producing answers of high quality and scales smoothly succeeding to evaluate efficiently queries with a very large number of frequent keywords on large and complex datasets where previous algorithms for flat keyword queries fail.

We are currently working on alternative ways for defining semantics for cohesive keyword queries on tree data and in particular in defining skyline semantics which considers all the cohesive terms of a query in order to rank the query results.

## 7. REFERENCES

- [1] C. Aksoy, A. Dimitriou, and D. Theodoratos. Reasoning with Patterns to Effectively Answer XML Keyword Queries. *VLDB Journal, Vol. 24, Issue 3, Springer*, pages 441–465, 2015.
- [2] C. Aksoy, A. Dimitriou, D. Theodoratos, and X. Wu. XReason: A Semantic Approach that Reasons with Patterns to Answer XML Keyword Queries. In *DASFAA*, pages 299–314, 2013.

- [3] S. Amer-Yahia and M. Lalmas. XML Search: Languages, INEX and Scoring. *SIGMOD Record*, 35(4):16–23, 2006.
- [4] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval - the concepts and technology behind search*. Pearson Education Ltd., England, 2011.
- [5] Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective XML Keyword Search with Relevance Oriented Ranking. In *ICDE*, pages 517–528, 2009.
- [6] Z. Bao, J. Lu, T. W. Ling, and B. Chen. Towards an Effective XML Keyword Search. *IEEE Trans. Knowl. Data Eng.*, 22(8):1077–1092, 2010.
- [7] S. Bergamaschi, F. Guerra, M. Interlandi, R. T. Lado, and Y. Velegrakis. Combining user and database perspective for solving keyword queries over relational databases. *Inf. Syst.*, 55:1–19, 2016.
- [8] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks*, 30(1-7):107–117, 1998.
- [9] O. C. L. Center. Dewey Decimal Classification, 2006.
- [10] L. J. Chen and Y. Papakonstantinou. Supporting top-K Keyword Search in XML Databases. In *ICDE*, pages 689–700, 2010.
- [11] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSearch: A Semantic Search Engine for XML. In *VLDB*, pages 45–56, 2003.
- [12] A. Dass, C. Aksoy, A. Dimitriou, and D. Theodoratos. Keyword pattern graph relaxation for selective result space expansion on linked data. In *ICWE*, pages 287–306, 2015.
- [13] A. Dimitriou and D. Theodoratos. Efficient keyword search on large tree structured datasets. In *ACM KEYS*, pages 63–74, 2012.
- [14] A. Dimitriou, D. Theodoratos, and T. Sellis. Top-k-size keyword search on tree structured data. *Inf. Syst.*, 47:178–193, 2015.
- [15] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, pages 436–445, 1997.
- [16] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD Conference*, pages 16–27, 2003.
- [17] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword Proximity Search in XML Trees. *IEEE TKDE*, 18(4):525–539, 2006.
- [18] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on xml graphs. In *ICDE*, pages 367–378, 2003.
- [19] L. Kong, R. Gilleron, and A. Lemay. Retrieving meaningful relaxed tightest fragments for XML keyword search. In *EDBT*, pages 815–826, 2009.
- [20] G. Li, J. Feng, J. Wang, and L. Zhou. Effective Keyword Search for Valuable LCAs over XML documents. In *CIKM*, pages 31–40, 2007.
- [21] G. Li, C. Li, J. Feng, and L. Zhou. SAIL: Structure-aware Indexing for Effective and Progressive top-k Keyword Search over XML Documents. *Inf. Sci.*, 179(21):3745–3762, 2009.
- [22] J. Li, C. Liu, R. Zhou, and W. Wang. Suggestion of Promising Result Types for XML Keyword Search. In *EDBT*, pages 561–572, 2010.
- [23] J. Li, C. Liu, R. Zhou, and W. Wang. Top-k Keyword Search over Probabilistic XML Data. In *ICDE*, pages 673–684, 2011.
- [24] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, pages 72–83, 2004.
- [25] X. Liu, L. Chen, C. Wan, D. Liu, and N. Xiong. Exploiting structures in keyword queries for effective XML search. *Inf. Sci.*, 240:56–71, 2013.
- [26] X. Liu, C. Wan, and L. Chen. Returning Clustered Results for Keyword Search on XML Documents. *IEEE TKDE*, 23(12):1811–1825, 2011.
- [27] Z. Liu and Y. Chen. Identifying meaningful return information for XML keyword search. In *SIGMOD Conference*, pages 329–340, 2007.
- [28] Z. Liu and Y. Chen. Reasoning and Identifying Relevant Matches for XML Keyword Search. *PVLDB*, 1(1):921–932, 2008.
- [29] Z. Liu and Y. Chen. Processing Keyword Search on XML: a Survey. *WWW*, 14(5-6):671–707, 2011.
- [30] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: Give me an example of what you need. *PVLDB*, 7(5):365–376, 2014.
- [31] B. Mozafari, K. Zeng, L. D’Antoni, and C. Zaniolo. High-performance complex event processing over hierarchical data. *ACM TDS*, 38(4):21, 2013.
- [32] K. Nguyen and J. Cao. Top-k Answers for XML Keyword Queries. *WWW*, 15(5-6):485–515, 2012.
- [33] P. Ogden, D. B. Thomas, and P. Pietzuch. Scalable XML query processing using parallel pushdown transducers. *PVLDB*, 6(14):1738–1749, 2013.
- [34] B. Póssas, N. Ziviani, W. M. Jr., and B. A. Ribeiro-Neto. Set-based vector model: An efficient approach for correlation-based ranking. *ACM Trans. Inf. Syst.*, 23(4):397–429, 2005.
- [35] J. Pound, I. F. Ilyas, and G. E. Weddell. Expressive and flexible access to web-extracted data: a keyword-based structured query language. In *ACM SIGMOD Conference*, pages 423–434, 2010.
- [36] A. Schmidt, M. L. Kersten, and M. Windhouwer. Querying XML Documents Made Easy: Nearest Concept Queries. In *ICDE*, pages 321–329, 2001.
- [37] C. Sun, C. Y. Chan, and A. K. Goenka. Multiway SLCA-based Keyword Search in XML Data. In *WWW*, pages 1043–1052, 2007.
- [38] A. Termehchy and M. Winslett. Using Structural Information in XML Keyword Search Effectively. *ACM Trans. Database Syst.*, 36(1):4, 2011.
- [39] S. K. M. Wong, W. Ziarko, and P. C. N. Wong. Generalized vector space model in information retrieval. In *Proceedings of the 8th annual international ACM SIGIR 1985*, pages 18–25, 1985.
- [40] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD Conference*, pages 527–538, 2005.
- [41] Y. Xu and Y. Papakonstantinou. Efficient LCA based keyword search in XML data. In *EDBT*, pages 535–546, 2008.
- [42] R. Zhou, C. Liu, and J. Li. Fast ELCA Computation for Keyword Queries on XML Data. In *EDBT*, pages 549–560, 2010.



# Generic Keyword Search over XML Data

Manoj K Agarwal  
Search Technology Center  
Microsoft India  
agarwalm@microsoft.com

Krithi Ramamritham  
Dept. of Computer Sc. and Eng.  
IIT Bombay – India  
krithi@cse.iitb.ac.in

Prashant Agarwal  
Dept. of Computer Sc. and Eng.  
NIT Allahabad - India  
agprashant.mnnit@gmail.com

## ABSTRACT

XML and JSON have become the default formats to exchange the information for web application or within enterprises. Keyword Search over XML data has been motivated by the need to relieve users from writing difficult XQueries since otherwise users are required to know the complex XML schema. In existing XML keyword search techniques the XML nodes returned for a keyword query are the Lowest Common Ancestor (LCA) nodes for the query keywords. In this paper, we argue that the LCA based techniques *still* require users to be well versed with the XML schema and also the data to be able to obtain meaningful query results.

To address these shortcomings, we present a novel system, Generic Keyword Search (GKS), - for a given keyword query  $Q$ , instead of identifying (and returning information) only from LCA nodes, GKS returns ‘meaningful’ information from *any* XML node, which contains a subset of keywords in the search query  $Q$ . GKS response includes LCA nodes, if any, that would have been returned by LCA based techniques.

GKS is also able to find highly relevant keywords and XML schema elements, deeper analytical insights - called *DI* - in the XML data *in the context of the user query*. *DI* enables users to navigate the XML data and to refine their queries even if they are not familiar with the data and the schema. Our experiments on real data sets show that GKS is able to return highly relevant responses to keyword queries efficiently.

## 1. INTRODUCTION

Semi-structured data, e.g. XML and JSON, are default formats to represent and exchange data within and across enterprises and web [18]. XML data is represented as a labeled, ordered tree  $T$  as shown in Figure 1(i). The nodes in  $T$  are either XML schema elements or text nodes. In response to a given keyword query, XML keyword search systems return one or more nodes in  $T$ , each of which is a Lowest Common Ancestor (LCA) node for *all* the query keywords in the XML data tree  $T$  [2][5][6][16][17][4]. For instance, in Figure 1, node  $x_2$  is the LCA node for query  $Q_1$ . We refer to XML keyword search technique that return LCA nodes in the XML tree, in response to a given keyword query, as *LCA based techniques*. LCA based techniques follow the AND-semantics, i.e., each LCA node contains at least one instance of each query keyword [4].

## 1.1 Motivation

For a given keyword query  $Q = \{k_1, \dots, k_n\}$  ( $|Q|=n$ ), instead of identifying LCA nodes and returning information only from these nodes, Generic Keyword Search (GKS) returns *any* node in the labeled tree  $T$ , if it contains  $s$  or more keywords in the search query  $Q$  ( $s \leq n$ ). More formally, the GKS problem is defined as follows: For a keyword query  $Q$ , an integer  $s \geq 1$ , search returns all the XML nodes which contain at least  $\min(s, |Q|)$ , keywords from  $Q$ . The set of XML nodes returned by GKS in response to query  $Q$  is denoted by  $R_Q(s)$ .  $|R_Q(s_1)| \leq |R_Q(s_2)|$  if  $s_1 > s_2$  (cf. Section 2.2).

There are many notions of LCA nodes in the literature but SLCA (Smallest LCA) [13] and ELCA (Exclusive LCA) [17], are most widely used. An SLCA node contains all the query keywords in its sub-tree and there is no node in its sub-tree which contains all the keywords. An ELCA set of nodes is a superset of the SLCA nodes. In Figure 1, for query  $Q_1$ , node  $x_1$  is an ELCA node but not an SLCA node due to the presence of  $x_2$  in its sub-tree. In the figure,  $k_i$  is an instance of keyword  $k$  (e.g.  $a_i$ s are instances of  $a$ ). For different notions of LCA nodes, progressively faster algorithms have been proposed to retrieve them [16]. The nodes in GKS response set follow the semantics of SLCA.

As pointed out by the authors of [19] “LCA based techniques work poorly for documents having *irregular schema* that have *missing elements*” because the schema allows certain XML nodes to be optional. Further noting that if a document is not complete, the resulting output could be different from the intended output. Authors of [19] develop an alternate approach whose basic premise is: for a given keyword search query, specific XML node types are targeted [15][19]. However, if the document has “missing XML elements”, nodes other than targeted nodes could also be returned due to the constraint on LCA based techniques (only LCA nodes are returned for the keyword query). Clearly, the motivation for [19] highlights that for LCA based techniques a) users need to be aware of the schema (i.e., users need to be aware which XML nodes to target); b) query keywords must be chosen by taking into account the semantic relationship between them (query must be formed such that the target nodes could be returned); and c) users need to be aware of the keywords in the XML document(s) and their distribution in XML tree  $T$  (otherwise nodes other than targeted nodes could become LCA nodes). In other words, in order to be able to effectively search the data using LCA based techniques, users have to be well acquainted with the data and the schema.

AND-semantics constraints underlying LCA based techniques are further highlighted by the following example:

**Example 1:** Consider keyword queries  $Q_1, Q_2, Q_3$ , on the XML document in Figure 1(i). Each leaf node in the XML document is a text node (text node is an XML element directly containing its value). We have represented the document as shown in Figure 1(i) for brevity. Response of SLCA and ELCA based algorithms are

shown in Table 1. For query  $Q_3$ , even though the user is able to select all the keywords present in the document, the response of LCA algorithms is root  $\{r\}$ . ‘ $r$ ’ is not a meaningful response as it is available to the user even in the absence of any query.

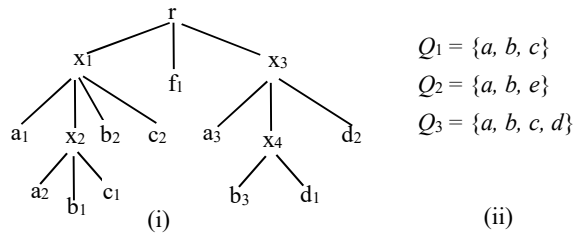


Figure 1. Labeled XML data tree and a set of queries.

Table 1. Nodes returned for different queries on labeled XML tree by different keyword search algorithms

Queries	GKS (ranked)	ELCA	SLCA
$Q_1, s= Q_1 $	$\{x_2\}$	$\{x_1, x_2\}$	$\{x_2\}$
$Q_2, s=2$	$\{x_2, \{x_3\}$	NULL	NULL
$Q_3, s=2$	$\{x_2, \{x_3\}, \{x_4\}$	$\{r\}$	$\{r\}$

Therefore, to construct meaningful queries, users need to know the keywords distribution in the document. To know the keyword distribution, it is imperative to know the semantic relationship between query keywords. In order to be aware of the semantic relationship between the query keywords, users must know the schema of the XML repository. Users must also know the schema to form the query such that targeted XML nodes are returned.

MESSIAH [19] addresses the issues arising due to the AND-semantics of LCA based techniques. Its authors propose an efficient FSLCA algorithm to identify intended nodes, in case of missing XML elements in the data. MESSIAH addresses the missing element problem, if the data is ‘imperfect’. However, the issues of ‘missing data elements’ is still not handled in [19] if the user query is ‘imperfect’. For instance, if a query keyword occurs in the wrong sub-tree, it is difficult to determine the intended return nodes. Hence, for a keyword query, possibly containing semantically uncorrelated keywords, nodes other than targeted nodes will be returned by [19] even if the missing XML elements are identified.

Consider the following scenario: User starts with query  $Q_2$  and  $Q_3$  (shown in Figure 1). GKS returns a set of XML nodes to the user, as shown in Table 1, which contain a significant fraction of the query keywords but not necessarily all the query keywords ( $s=2$ ). Besides returning these nodes, let us say, GKS system also suggests to the user that query  $Q_2$  can be morphed to  $\{a, b, c\}$  or  $\{a, b, d\}$  from  $\{a, b, e\}$ . The user may not be aware of the existence of the keywords ‘ $c$ ’, ‘ $d$ ’ or their relevance in the context of the query. Similarly, for  $Q_3$ , the system suggests that query be partitioned into  $\{a, b, c\}$  and  $\{a, b, d\}$ . Such refinements of the user queries are non-trivial. Overall, we are motivated by the following goals 1) to relax the need for users to know the XML data precisely; this enables them to browse the XML data in a manner similar to web search; 2) to relax the need for users to know the XML schema as user queries can be refined progressively (as for query  $Q_3$ ).

## 1.2 Generic Keyword Search

In this paper, we introduce a novel concept of **Generic Keyword Search (GKS)** over XML data to address the shortcomings listed above. It enables the users to navigate XML data with ease, as demonstrated with the help of an example on real data below run on the implemented GKS system [20]:

**Example 2:** We have a DBLP dataset with more than 2.5 million articles. A query  $Q_d = \{\text{"Peter Buneman" "Wenfei Fan" "Scott Weinstein" "Prithviraj Banerjee"}\}$  is run on this dataset. The user is most likely interested in articles jointly written by these authors. In its response, a total of 234 articles (for  $s=1$ ) are found by GKS, i.e., GKS return all the articles by any of the authors in the keyword query since  $s=1$ .

Since the response of GKS contains a large number of XML nodes (i.e.,  $\langle inproceedings \rangle$ ), with different XML nodes in the response containing different number of authors, the results are ranked such that the more relevant XML nodes are ranked higher (cf. Section 5). For query  $Q_d$ , the  $\langle inproceedings \rangle$  nodes with higher number of query keywords (i.e., author names) in their sub-tree are likely to be ranked higher. We use just  $\langle ip \rangle$  for  $\langle inproceedings \rangle$  later on.

In the DBLP dataset, there is no article jointly written by Prithviraj Banerjee with any of the remaining authors. Of the five articles jointly written by the remaining three authors in DBLP dataset, 4 were returned as top 4 results in the ranked list of XML nodes by GKS. The remaining article was also in top 10 (it was ranked lower due to many co-authors, details Section 5). In the context of this example, we now explain how GKS overcomes the shortcomings of the LCA based techniques:

### GKS relaxes the need for users’ familiarity with the contents:

For the given query, an LCA based technique would have returned  $\{\text{DBLP root}\}$ , containing millions of articles as the response due to the presence of one “wrong” keyword ‘Prithviraj Banerjee’ in the query. On the other hand, GKS produced a more “meaningful” response in the presence of “wrong” query keyword(s). This helps the users as follows;

a) With GKS, users can navigate the XML data without complete awareness with underlying data (for LCA based techniques, users need to know, which authors have published articles together). GKS returns a ranked list of most relevant XML nodes, in the context of the query, considerably enhancing the users’ ability to search the data with high precision and recall.

b) More importantly, even when users are able to formulate the query precisely, there is a lot of information which could be of their interest, which are not returned by LCA based techniques due to the constraint that *only* LCA nodes must be returned. For instance, in Example 2, the articles by a large enough subset of authors in the query  $Q_d$  could also be of interest to the user in the context of the query. Exposing such results in the data helps users navigate the data as well as to refine their queries (cf. Section 6.1).

### GKS relaxes the need for users’ familiarity with the schema:

GKS identifies the XML nodes, which are not necessarily LCA nodes but that could be of interest to the user in the context of the query. This ability of GKS can be exploited to discover most relevant keywords and their semantics in the underlying XML data, in the context of the user query. This information is called *deeper analytical insights* or *DI*. For the query in Example 2, GKS exposes  $\langle ip: journal: SIGMOD Record \rangle$ ,  $\langle ip: year: 2001 \rangle$ ,  $\langle ip: author: Alok N Choudhary \rangle$  and  $\langle ip: booktitle: ICPP \rangle$ , etc., as *DI* from the XML data in the context of the query (GKS returns a well-constructed XML chunk. Truncated representation is due to lack of space). *DI* exposes the most relevant journals, year and authors in the query response. The user may not be aware of these keywords or their relevance in the context of the query.

*DI* is defined formally in Section 2.3 (Def. 2.3.1). Discovery of *DI* (Section 6) enhances the users’ ability to navigate the data even if they are unaware of the schema details and the semantic relationship between the various data keywords. To discover *DI*,

we exploit the XML schema, embedded in the structure of the XML data, in the context of a user query. A novel node categorization model is proposed that identifies certain XML node types as Least Common Entity nodes or LCE nodes (cf. Section 2.2). LCE nodes are central to our methodology to discover *DI*. A subset of XML nodes  $E_Q \subseteq R_Q(s)$  in GKS response for query  $Q$  can be Least Common Entity (LCE) nodes;  $0 \leq |E_Q| \leq |R_Q(s)|$ . For an XML node  $u$  containing a sub-set of query keywords (of size  $\geq s$ ) in its sub-tree, its corresponding LCE node will be either  $u$  itself or its ancestor.

In Example 2,  $\langle ip \rangle$  node is an LCE node. GKS exposes the semantics of the *DI* keywords, i.e., 2001 is a  $\langle year \rangle$  with the aid of XML elements on the path from the root of LCE node  $\langle ip \rangle$  till the keyword “2001”. Semantics are important as in a different context, 2001 could be a street number. Query keywords, either XML element names or text keywords, may carry different meaning in different context [9]. Exposing the relevant keywords and their semantic meaning helps users refine their queries in the absence of knowledge about the schema and the data.

**GKS returns meaningful response:** The meaningfulness of the results of a search query is defined by their recall and precision. In LCA based search, a keyword query typically targets XML nodes belonging to specific schema elements  $\mathbf{E}$  in the associated XML schema [19].  $\langle E \rangle \in \mathbf{E}$  is an XML schema element. The target nodes are the LCA nodes of the query keywords. If the returned LCA nodes are of targeted schema element type(s), it constitutes a meaningful response. For the query  $Q_d$  in Example 2, the meaningful LCA nodes for this query are all of type  $\langle ip \rangle$  in the corresponding XML schema.

However, due to imperfect data with missing XML elements or due to imperfect query, LCA based techniques often return LCA nodes other than the target XML elements type [19]. For instance, for query  $Q_d$ , LCA based techniques will return the DBLP root. A more meaningful response is a ranked list of articles, jointly written by a sub-set of authors in the query, i.e., returning nodes of same type, which were targeted. For GKS system, all the XML nodes that contain any subset of keywords in a query (of size  $\geq s$ ) are returned. Therefore, recall of GKS is likely to be high since GKS query response is likely to have XML nodes which are instances of target XML schema element in  $\mathbf{E}$  for a user query  $Q$  (any XML node containing  $s \leq |Q|$  keywords in its sub-tree is returned).

In the context of a keyword query, the relevance of a XML node is high if it contains a large fraction of query keywords. The precision of the GKS system will be high if the most relevant XML nodes in the GKS query response are ranked higher. We present a novel ranking methodology (Section 4) to ensure high precision.

### 1.3 Research Challenges and Contributions

Similar to a web search engine, Generic Keyword Search has the twin objectives of: a) locating the most relevant XML nodes for the given keyword query efficiently; and b) ordering the search results to rank more meaningful results higher.

GKS has three primary challenges; 1) **Efficiency** – GKS has much larger search space as opposed to LCA based techniques (Lemma 3). Therefore, a major challenge for GKS is to be able to retrieve the relevant nodes efficiently (Section 4); 2) **Ranking** – Number of XML nodes retrieved by GKS could be large and the structure of the different XML nodes in the search results could be different. Therefore, it is imperative to rank the nodes such that more meaningful and relevant nodes are ranked higher (Section 5); 3) **Analysis** - GKS aims to enable the users to refine their queries without needing them to be familiar with schema and data. GKS

meets this challenge by exposing relevant keywords in the data and their semantics in the context of the user query (Section 6).

In this paper, we make the following contributions:

1. Existing XML Keyword Search techniques work within LCA framework. We introduce Generic Keyword Search (GKS) that enables XML search beyond LCA framework.
2. We propose a XML node categorization model. With the aid of this model, we expose most relevant XML elements and data keywords, called *DI*, in the context of a given keyword query. Users can refine their query with the aid of *DI*. *DI* is discovered *because* GKS does not impose the LCA constraint.
3. We introduce a ranking methodology to rank more meaningful XML nodes, retrieved by GKS, higher. Node ranking is further exploited for *DI* discovery.
4. We present an evaluation of GKS system on real data sets. Our results show that GKS is able to return highly relevant response for the given keyword queries efficiently. We further show that our system is able to find highly relevant *DI* that enables the users to navigate the XML data seamlessly.

The organization of the paper is as follows. Section 2 introduces the GKS node categorization model along with the definitions of LCE nodes, *DI* and the GKS indexing structure. Related work is presented in Section 3. Our methodology to identify the relevant XML nodes efficiently is the subject of Section 4. In Section 5, we present a novel XML node ranking methodology. In Section 6, we discuss our mechanism to discover *DI*. In Section 7, we present experimental results followed by conclusion in Section 8.

## 2. XML NODE CATEGORIZATION AND DEFINITIONS

In this section, we first present a novel XML node categorization model. The XML node categorization helps us exploit the XML schema, embedded in the XML data, to identify relevant data keywords and XML schema elements in the context of a user query. We also present the definitions of LCE nodes and *DI*, GKS system architecture and the indexes maintained by GKS.

### 2.1 Preliminaries

An XML document is a rooted tree  $T$  as shown in Figure 2(a). Nodes in the tree are labeled with Dewey id [1]. Dewey id is a unique id assigned to a node that describes its position in the tree  $T$ . A node with Dewey id 0.2.3 is the fourth child of its parent node 0.2.  $n_{id}$  represents an XML node with Dewey id  $id$ .  $v \prec_a u$  denotes that node  $v$  is an ancestor of node  $u$ .  $v \triangleleft_a u$  denote that  $v \prec_a u$  or  $v=u$ .  $U$  represent a set of XML nodes (or keywords) in XML tree  $T$ .  $v \prec_{lca} U$  denotes that  $v$  is the lowest common ancestor of nodes in set  $U$ . For a text keyword or XML node  $k$ ,  $k \in v$  denotes that  $k$  occurs in the sub-tree rooted at XML node  $v$  and  $k \notin v$  denotes that  $k$  does not occur in  $v$ 's sub-tree.  $u^*$  denote that one or more siblings of node  $u$  exist in tree  $T$  with same XML element label.  $v \prec_e u$  denotes that  $v$  is an entity node w.r.t.  $u$  (Def. 2.1.3) and  $u \in v$  or  $v=u$ .  $v \prec_{lce} u$  denotes that XML node  $v$  is the lowest common entity node (LCE) w.r.t. node  $u$  (Def. 2.2.1) and  $u \in v$  or  $v=u$ .

### 2.2 Node Definitions

We divide the XML nodes in the following categories, based on the structure of their sub-trees in  $T$ .

**2.1.1. Attribute Node (AN):** A node which contains only one child that is its value. For instance, in Figure 2(a) node  $\langle Name \rangle (n_{0.1.0})$  is an attribute node. Attribute nodes are also represented as ‘text nodes’ in XML data. The parent node of an attribute node is

considered the lowest ancestor for keyword(s) in its value (and not the attribute node itself). Thus, ancestor of ‘Databases’ is node  $n_{0.1}$ .

**2.1.2. Repeating Node (RN):** Let  $v \prec_{lca} u^*$ , i.e.,  $v$  is the lowest common ancestor of multiple instances of node  $u$ .  $u$  is called the repeating node w.r.t. node  $v$ . For instance, in Figure 2(a), nodes with label <Student> are repeating nodes w.r.t. <Students>. The repeating nodes most likely correspond to a physical world object which could be a concrete or an abstract object [3]. A node that directly contains its value *and* also has siblings with the same XML tag is considered a repeating node (and not an attribute node), i.e., <Student> nodes in Figure 2(a).

**2.1.3. Entity Node (EN):** Let  $v$  be an XML node in XML tree  $T$  such that  $v \prec_{lca} (u^*, A) \mid \forall a \in A, a \notin u^*$ .  $v$  is an entity node.  $A$  is a set of attribute nodes. An attribute node  $a \in A$  does not occur in any repeating node  $u$ , i.e.,  $a$  does not have  $u$  in its XPath from root.

An *entity node*  $v$  is a lowest common ancestor of repeating nodes  $u$  and one or more attribute nodes ( $|A| \geq 1$ ). In Figure 2(a) <Area> ( $n_{0.1}$ ) is an *entity node*; it is the lowest common ancestor of attribute node <Name> ( $n_{0.1.0}$ ) and repeating nodes <Course> ( $n_{0.1.1.x}$ ). <Course> nodes are not the direct children of  $n_{0.1}$  (Attribute nodes and Repeating nodes can be indirect children of *entity node*). Similarly, <Course> nodes ( $n_{0.1.1.0}, n_{0.1.1.1}, n_{0.1.1.2}, \dots$ ) are the entity nodes.

**2.1.4. Connecting Node (CN):** Nodes which are in none of the above categories. In Figure 2(a), <Courses> ( $n_{0.1.1}$ ) is a connecting node.

**Table 2: Notation**

$s$	Minimum number of keywords from a query that must appear in the sub-tree of a XML node.
$R_Q(s)$	Set of XML nodes for a given $s$ , returned by GKS in response of query $Q$
$R(e)$	For an LCE node $e \in R_Q(s)$ , $R(e)$ is a subset of text keywords, extracted from attribute nodes of $e$ .
$S_Q^w$	Weighted set of text keywords, identified from the LCE nodes in set $R_Q(s)$ .
$\prod_r R_Q$	Set of XML nodes, after recursively applying the GKS algorithm $r$ times over the query results $R_Q(s)$ .

XML documents follow pre-order arrival of nodes. Hence, different node types are identified in a single pass over the data. GKS does not need the XML schema in order to categorize nodes. XML nodes are categorized at the instance level. This information is stored in an index (Section 2.4). Hence, each node is categorized based on the structure of its sub-tree. For example, all the instances of <Course> node in Figure 2(a) are *entity nodes* (Def. 2.1.3). However, if a <Course> node had just one student in its sub-tree, that instance would have been stored as ‘Connecting node’ in the index. GKS can be easily extended to take into account the XML schema to categorize the nodes. This is part of our future work.

The node categories described above extend the node categorization model in [3]. It is argued in [3] that in the hierarchical structure of XML data, repeating nodes (Def. 2.1.2) capture the concept of physical world object. The physical object could be a concrete or an abstract object. In normalized XML data, attributes of an XML node that contains repeating nodes in its sub-tree, represent the information that is common to these repeating nodes [14]. The fundamental design principle underlying the normalized XML schema is, the attribute nodes of an XML node define the context of the repeating nodes in its sub-tree through their values. In GKS node categorization model, such XML nodes are termed entity nodes (Def. 2.1.3). As shown in the experiment in Section 7.2, we count the total number of XML nodes and XML

nodes that were labeled as entity nodes, attribute nodes and repeating nodes, respectively by GKS for many standard XML data repositories. The result shows that the real world data repositories are normalized. The node categories described above naturally capture the normalized XML data.

A node can be an *entity node* and at the same time a *repeating node* for another *entity node* higher up in the hierarchy. For instance, in Figure 2(a), <Course> nodes are both entity nodes as well as repeating node within the sub-tree of node <Area> ( $n_{0.1}$ ). Let  $Q$  be a keyword query,  $|Q| \geq s$ , and  $Q' \subseteq Q$ ;  $|Q'| \geq s$ . Let LCA node  $u$  for  $Q'$  is not an *entity node* and  $v$  is the lowest ancestor of node  $u$  such that  $v \prec_e u$ . Hence, node  $u$  can either be a connecting node or a repeating node w.r.t.  $v$ . Since  $u$  does not have the attribute nodes, as it is not an *entity node*, the context of the node  $u$  is most specifically defined by the attribute nodes of node  $v$ . In Figure 2(a), attribute <Course: Name: Data Mining> defines the context that <student> nodes in its sub-tree are registered in this course.

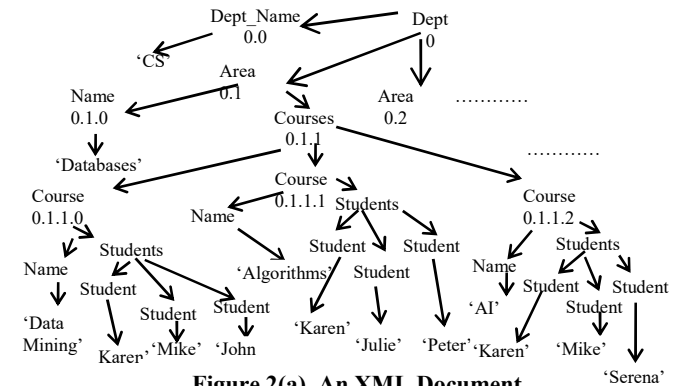
For a given keyword query, the closer the entity node is to the query keywords in its sub-tree, the more specific the context would be for those keywords. As we move up in the hierarchy, the context of the corresponding sub-tree becomes more general. In Figure 2(a), <Dept> node and <Course> node both are entity nodes and both contain the query keywords for a query  $Q = \{‘Karen’, ‘Mike’\}$ . However, the context of entity node <Dept> is much more general compared to more specific context of the node <Course>. Hence, to find the more meaningful response for a given query, we discover the entity node closest to the query keywords or Least Common Entity (LCE). LCE is formally defined below.

Let  $S_c$  be a set of all the *entity nodes* in the sub-tree rooted at an *entity node*  $e_c$ , i.e.,  $e_c \prec_a e$ ;  $\forall e \in S_c$ . Let  $Q$  be a keyword query,  $Q = \{k_1, \dots, k_n\}$ .

**Def 2.2.1 LCE Nodes:** An *entity node*  $e_c$  is an LCE node for query  $Q$  if  $\exists k \in Q \mid k \in e_c \wedge \forall e \in S_c, k \notin e$ .

Hence, for an *entity node*  $e_c$  to be LCE node for a given query  $Q$ , there exists at least one keyword  $k \in Q$  in the sub-tree of  $e_c$ , which is *not* contained in any other *entity node*  $e$  such that  $e_c \prec_a e$ .

Keyword  $k$  is called an independent witness for LCE node  $e_c$ . Similar to an SLCA node, an LCE node also needs at least one independent witness.



**Figure 2(a). An XML Document**

**Lemma 1:** Let  $v \triangleleft_a u$  denote a relationship that  $v \prec_a u$  or  $v = u$ . Let  $u$  be an XML node that is an LCA node for a set of keywords  $Q_s \subseteq Q$ ,  $|Q_s| \geq s$ . Let  $v$  be an LCE node for keywords in  $Q_s$ .  $v \triangleleft_a u$

**Proof:** Obvious. □

For a given user query  $Q$ , GKS returns a set of XML nodes  $R_Q(s)$  such that for each node  $u \in R_Q(s)$ ,  $u$  contains at least  $s$  keywords from query  $Q$ .

**Lemma 2:** For a keyword query  $Q$  and integers  $s_1$  and  $s_2$ ,  $|Q| \geq s_1 > s_2$ ,  $|R_Q(s_1)| \leq |R_Q(s_2)|$ .

**Proof:** Since  $s_1 > s_2$ ,  $\forall v \in R_Q(s_1), \exists u \in R_Q(s_2) | v \prec_a u$ . However,  $\forall u \in R_Q(s_2)$  there can be at most one  $v \in R_Q(s_1) | v \prec_{lce} u$ . Thus, for  $\forall v, v \in R_Q(s_1)$  there exist a corresponding node in  $R_Q(s_2)$  but vice versa is not true. Thus,  $|R_Q(s_1)| \leq |R_Q(s_2)|$ .  $\square$

**Example 3:** Let there be a user query  $Q_4 = \{\text{student, karen, mike, john, harry}\}$ ,  $s=2$ . The intent of the query is to find the information about these students. For the data shown in Figure 2(a), 3 courses contain the names of at least one of these students. The GKS response constitutes the XML nodes as shown in Figure 2 (b). The XML nodes are LCE nodes since they are the lowest entity nodes, w.r.t. query keywords. Attribute nodes of respective entity nodes exposes the context, i.e., name of the respective courses students are enrolled in. The XML nodes are ranked (cf. Section 5).

As one can see, the user query in Example 3 is ‘imperfect’. To construct a ‘perfect’ query, for a LCA based technique, user needs to be aware which students are enrolled in same courses. User still has to run multiple queries to get the complete response. GKS returns the relevant and meaningful information in the context of this ‘imperfect’ query. We further enhance a user’s capability to refine an ‘imperfect’ query by exposing the deeper analytical insights in the query response as explained in the next section.

### 2.3 Deeper Analytical Insights (DI)

For the query in Example 3, let’s say user runs a ‘perfect’ query  $Q_5 = \{\text{student, karen, mike, john}\}$ . The response of a LCA based technique [2][5] will be XML sub-tree rooted at node  $n_{0.1.1.0.1}$  <Students> node. Even though the query is perfect, the response still does not yield any meaningful information. On the other hand, GKS response is node  $n_{0.1.1.0}$  for  $s=|Q|$  ( $n_{0.1.1.0}$  is an LCE node for  $Q_5$ ) with the aid of its node categorization model. Thus, GKS response exposes the information that the students are registered in ‘Data Mining’ course. This information, <Course: Name: ‘Data Mining’>, is called *deeper analytical insights* or *DI*. *DI* enables users to navigate the XML data by exposing relevant schema and the data elements that help users not only understand the query response but also help refine their queries.

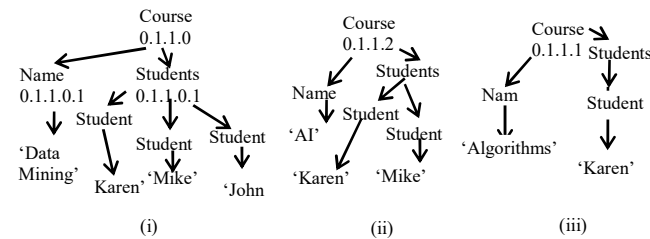


Figure 2(b). Response of the GKS System for  $Q_4$

To discover *DI*, for a user query  $Q$  and  $s$ , GKS prepares a set of keywords  $S^w_Q$  from nodes in set  $R_Q(s)$  as follows: For each node  $u \in R_Q(s)$ , if  $u$  is an LCE node, GKS extracts the text keywords from its attribute nodes and put them in set  $S^w_Q$ . For instance, for the query in Example 3, entity nodes  $n_{0.1.1.0}$ ,  $n_{0.1.1.1}$  and  $n_{0.1.1.2}$  are the LCE nodes in the set  $R_Q(s)$  (Figure 2(b)). Each of the entity nodes has an attribute node <Name: Data Mining>, <Name: AI> and <Name: Algorithms>. The set  $S^w_Q$  will contain keywords {“Data Mining”, “AI”, “Algorithm”}.  $R(e)$  represents the set of attribute

nodes in the sub-tree of entity node  $e$  (see Table 2). Given a query response  $R_Q(s)$ , we prepare a set of keywords  $S^w_Q = \{k_1 \dots k_n\}$  containing the text keywords embedded in the attribute nodes for each of the entity nodes in  $R_Q(s)$ .

**Def 2.3.1 DI:** Let  $E_Q \subseteq R_Q(s)$  be the set of all LCE nodes in GKS response for keyword query  $Q$  and let  $S^w_Q = \bigcup R(e) | e \in E_Q$ .

$$DI \subseteq S^w_Q | \forall k \in DI; k \notin Q.$$

For a keyword  $k$  in *DI*, let  $e$  be its corresponding LCE node. For the keyword  $k$ , we also associate the XML elements in the path from node  $e$  till keyword  $k$ . The keywords and the associated XML elements with each keyword together form the *DI*.

*DI* can also be discovered recursively for a user query as described below. We use only set  $R_Q(s)$  and not  $E_Q$  since context is clear.

- i) GKS parses the LCE nodes in set  $R_Q(s)$ , for a given keyword query  $Q$  and prepares a weighted set of keywords  $S^w_Q$  by identifying a subset of text keywords in each of the LCE nodes (Section 6.2).
- ii) *Top-m* most weighted keywords in the set  $S^w_Q$  are fed to GKS as a query. GKS identifies a set of XML nodes w.r.t. these keywords from set  $S^w_Q$ . This set of XML nodes is denoted as  $\prod_1 R_Q(s)$ . Set

$$\prod_0 R_Q (\prod_0 S^w_Q) \text{ is denoted by just } R_Q(s) (S^w_Q).$$

The above steps can be applied recursively --  $\prod_r R_Q(s)$  represents the set of LCE nodes after  $r^{th}$  recursion.

- iii) GKS prepares the set of keywords  $\prod_r S^w_Q$  from the nodes in  $\prod_r R_Q(s)$ . *DI<sub>r</sub>* is extracted from  $\prod_r S^w_Q$ ;  $r \geq 0$ .

*DI* can be discovered recursively for a user query  $Q$  by extracting a ranked list of most relevant keywords and their semantics from  $\prod_i S^w_Q$  at each step  $i$  of recursion. In summary, *DI* is discovered 1) with the aid of GKS node categorization model; and 2) *because* GKS does not impose the LCA constraints and thus retrieves all the relevant XML nodes in the query context. These XML nodes help discover meaningful *DI*.

### 2.4 GKS Architecture and Indexes

In Figure 3, we depict the architecture of GKS. The GKS takes as input XML data and prepares an index on it. The XML data could be spread over multiple files. For a user query  $Q$ , GKS produces a) ranked search results on the data; b) deeper analytical insights (*DI*) by analyzing search results. GKS contains three modules; i) Indexing Engine; ii) Search Engine; iii) Search Analysis Engine.

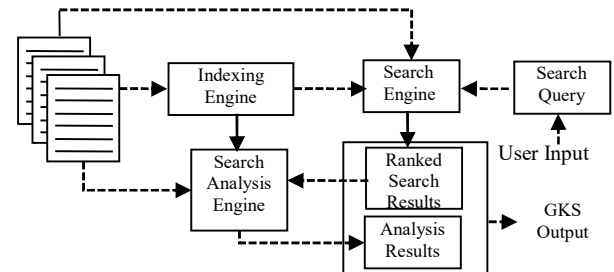


Figure 3. Architectural of the GKS System

For a given XML data repository, we first prepare an index on it. This is a onetime activity. We keep the following indexes:

**Inverted Index for text keywords:** For each unique text keyword that appears in the XML document repository, we keep an inverted index list. If text appearing under a ‘text node’ comprises multiple

keywords, a separate index entry is created for each of the keywords after stop words removal and stemming. A partial inverted index for document in Figure 2(a) is shown in Table 3. The inverted index list for a keyword  $k_i$  contains the Dewey id of all the nodes which contain that keyword. Dewey id for each node has been appended with the document id 'did'. Thus, GKS search is seamlessly expanded over multiple documents by prefixing Dewey ids with corresponding document id. For a keyword  $k_i$  present in the XML document repository,  $S_i$  denotes its inverted index list.

**Hash tables:** We keep two hash tables corresponding to XML elements. Hash table 1, called '*entityHash*', keeps the Dewey id of entity nodes. Hash table 2, called '*elementHash*', keeps the Dewey ids of repeating nodes and connecting nodes. Both hash tables also store the number of direct children each node has. This information is used while computing the rank of a node (Section 5). If a XML element is both a 'repeating node' and an 'entity node', its entry is present in both the hash tables.

Since XML nodes arrive *pre-order* (an ancestor of an XML node always appears before it), the hash tables and the inverted index are created in a single pass over XML data.

**Table 3. Partial inverted index for XML document in Fig 2(a).**

Karen	did.0.1.1.0.1.0	did.0.1.1.2.1.0	.....
Mike	did.0.1.1.0.2.0	did.0.1.1.2.2.0	.....

We provide two functions: i) *isEntity* (Deweyid); ii) *isElement* (DeweyId). Both the functions return the number of direct children the given node has if true, null otherwise.

### 3. EXISTING WORK in the GKS CONTEXT

A large body of work exists to understand the user's intent for a keyword query over XML data. The work related to GKS can be divided into 3 categories: 1) Identifying meaningful return nodes for a keyword query, 2) Result type deduction techniques and 3) Ranking the XML nodes retrieved in response to a user query.

**Identifying meaningful return nodes:** Users present their keyword query and the underlying algorithm interprets the user's intent and tries to identify the return nodes accordingly [2][3][5][6]. The existing approaches for identifying most relevant return nodes are based on first discovering SLCA nodes [13]. Different heuristics are applied on the set of SLCA nodes to identify meaningful return nodes. In XSeek [3], authors propose a technique that first finds the SLCA nodes for a given keyword query. The keywords in the query are understood as the 'where' clause whereas 'return' nodes are inferred based on the semantics of 'query keywords'. MaxMatch [11] and RTF [12] are SLCA based approaches to identify meaningful return nodes. In [11], irrelevant match results are filtered from each SLCA node. In [12], authors propose an improved algorithm to address *redundancy* and *false positive* problems of [11]. In all the approaches above, a set of SLCA nodes is identified for given keyword query. In [10] authors address the problem due to imprecise XPath queries.

**Deducing result types:** Deducing return node types is also an important goal for GKS since for most keyword queries, users target certain node types. However, due to lack of knowledge about the distribution of keywords in the document, different semantic meaning of same keywords or due to lack of familiarity with the document schema, the query may not be semantically 'perfect'. In [15][19], it is assumed that the keyword query is semantically correct and certain node types are the target nodes for a given query. XReal [9] and XBridge [4] address the problem of deducing the return nodes types. In [9] the authors count the confidence level to deduce the result node types. In [4] authors highlight the fact that keywords may exist in different context. XBridge automatically

predicts the intended result types for XML keyword queries by considering the value and structural distributions of the data. The more generic solution to this problem is to enable users to further refine their queries. GKS approach is a step in that direction.

**Ranking the XML nodes:** The XML ranking techniques are divided into IR [9][8] based methods and relevance score based [15][7] methods. XRank [7], XSearch [8] are techniques to rank the keyword query search results based on LCA nodes. XRank takes into account the keyword proximity in the XML nodes whereas XSearch computes the node rank based on TF-IDF based method. The basic differences between these methods and GKS technique is: In existing XML ranking methods, each of the XML nodes that is ranked contain a fixed set of all query keywords. XML nodes in GKS response contain varying number of query keywords. We have outlined the issue arising due to this difference in Section 5 when we present GKS ranking methodology.

## 4. SEARCHING GKS NODES

The basic difference between the LCA based search and GKS-Search is: GKS has exponential search space compared to LCA

based techniques. For query  $Q$  ( $|Q|=n$ ), a total of  $(2^n - \sum_{i=1}^s \binom{n}{i-1})$

sub-sets, of size at least  $s$ , can be formed;  $s \leq n$ . To identify GKS nodes, a naïve approach would be to create all the keyword subsets (of size  $\geq s$ ) for query  $Q$ , and for each of these keyword subsets, identify the LCA nodes. Together, all the LCA nodes thus discovered can be used to produce the GKS response. However, this approach results in an exponential number of sub-queries.

**Lemma 3:** For a given query  $Q$ ,  $|Q|=n$ ,  $s \leq \lfloor n/2 \rfloor$ ; GKS has exponential search space w.r.t. an LCA based techniques.

**Proof:** For a given query  $Q$ ,  $|Q|=n$ , a total of  $U = 2^n - \sum_{i=1}^s \binom{n}{i-1}$  sub-

sets can be formed such that each set is of size at least  $s$ . Now,

$$\sum_{i=1}^n \binom{n}{i-1} = 2^n - 1 \Rightarrow \sum_{i=1}^n \binom{n}{i-1} \leq 2^n \text{ since } \binom{n}{i} = \binom{n}{n-i}. \text{ Hence,}$$

$$\sum_{i=1}^{\lfloor n/2 \rfloor} \binom{n}{i-1} \leq 2^n / 2; \quad \text{Therefore } \sum_{i=1}^s \binom{n}{i-1} \leq 2^n / 2; s \leq \lfloor n/2 \rfloor$$

Since,  $U = 2^n - \sum_{i=1}^s \binom{n}{i-1} \Rightarrow U \geq 2^{n-1}$ . Therefore, an exponential

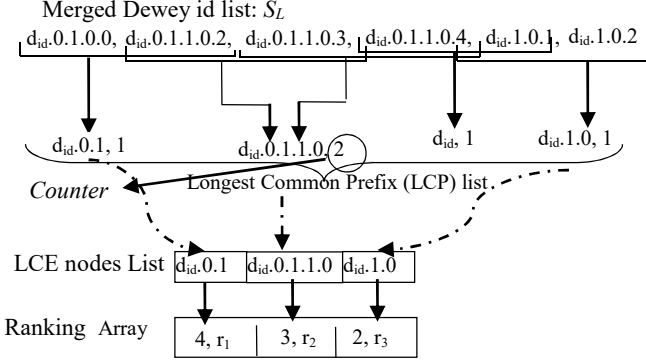
number of sub-sets are formed when  $s \leq \lfloor n/2 \rfloor$  with each sub-set leads to one keyword query for an LCA based techniques.  $\square$

Lemma 3 shows GKS has exponential search space w.r.t. LCA based techniques. Further, the naïve approach does not discover the LCE nodes, in absence of GKS node categorization model, which allow GKS to expose *DI* in the context of the user query. Hence, LCA techniques cannot be applied as is for GKS-Search. In this section, we present an efficient method to find relevant XML nodes for GKS-Search. We call them GKS nodes. A subset of GKS nodes can be LCE nodes. We also present the correctness analysis and time complexity analysis of our method.

### 4.1 Efficient Method to Search GKS nodes

For the query keywords  $k_i \in Q$ , we first merge their respective inverted index lists such that in the merged list, keywords follow their arrival order in the XML document. Since the Dewey ids of the XML nodes follow *pre-order* traversal, if the merged list is sorted on Dewey ids, we achieve such ordering. Let  $d$  be the depth of the XML tree  $T$  being queried. Depth of the tree  $T$  is defined as

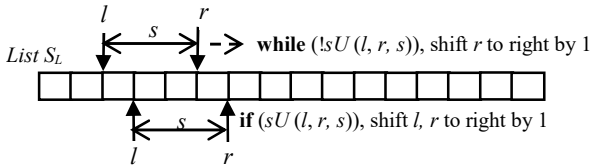
the number of edges from the root of the tree to its deepest leaf. Let  $|Q|=n$ , i.e.,  $n$  lists are merged. Let  $|S_i|$  be the inverted index length for keyword  $k_i \in Q$ . Let  $S_L$  be the merged and sorted list.  $|S_L| \leq \sum_{i=1}^n |S_i|$ . The time complexity to merge  $k$  sorted lists, of total length  $l$ , into a single sorted list in  $O(l \log k)$ . Since the inverted index list for each keyword is sorted on its Dewey id, therefore the  $n$  lists are merged in a single sorted list  $S_L$  in  $O(d|S_L| \log n)$ .



**Figure 4. List of longest common prefixes for Dewey id blocks of size  $s=2$**

**Generating list of candidate GKS Nodes:** Our next objective is to generate the list of candidate GKS nodes that have  $Q_s \subseteq Q$  keywords appearing in their subtree such that  $|Q_s| \geq s$ . Towards that end, in the merged list  $S_L$ , longest common prefix is identified for a continuous block of  $s$  entries, as shown in Figure 4 (in Figure 4,  $s=2$ ). We traverse the list  $S_L$  from left to right. Since the list  $S_L$  is sorted, the Dewey ids of the nodes in a common sub-tree occur next to each other, with an ancestor node preceding its descendent. Therefore, in  $S_L$ , the longest common prefix of a block of  $s$  nodes will be the Dewey id of the common ancestor for the nodes in this block. There will be at most  $(|S_L| - s)$  such prefixes.

The prefixes are put in Longest Common Prefix (LCP) list as shown in Figure 4. With each prefix entry in LCP list, we associate a *counter* which is initialized to 1. If a prefix exists in the LCP list (i.e., more than  $s$  query keywords exist in its sub-tree), its counter is increased by 1. Since the block of  $s$  entries in the list  $S_L$  slides to the right by 1 at a time, the *counter* can increase by only 1 at a time.



**Figure 5. Traversal of list  $S_L$**

The objective of GKS is to collect  $s$  unique keywords from query  $Q$  in the sub-tree of a GKS node. However, it is possible that not all  $s$  keywords in the continuous block of length  $s$  are unique. Therefore, we first collect a block such that there exist  $s$  unique keywords in it, as shown in Figure 5. For a block of length  $s$ , let  $l$  and  $r$  represent the left and right end of the block respectively. Function  $sU(l, r, s)$  returns true if there are  $s$  unique keywords in the range of  $l$  to  $r$  (with the aid of hash tables, Section 2.4). Until  $sU(\cdot)$  is true, we just move  $r$  to the right, keeping  $l$  fixed. When  $sU(\cdot)$  is true, range  $l$  and  $r$  represent a block containing  $s$  unique keywords. Once the correct block is found, the longest common prefix of the block is added to the LCP list.

We generate the list of LCE nodes from LCP list. For each of the entries in LCP list, we check the *entityHash*, prepared at the time of parsing XML document repository. For each entry in the LCP list, we check if it is an entity node or any of its ancestors is an entity node (using *Dewey id* we can get the Dewey ids of all if its ancestors). If the node (or any of its ancestors) is found to be entity node, we add the corresponding Dewey id into a LCE node list. We also maintain a ‘Ranking array’ which has an entry corresponding to each GKS node. Each entry in the ranking array maintains two scores as shown in Figure 4. One is the number of keywords  $k_i \in Q$  appearing in GKS node sub-tree and the other is its ranking score (computation of the ranking score is described in Section 5). The number of query keywords in the GKS sub-tree is  $(s+counter-1)$ .

```

Algorithm GKSNodes (Set  $Q$ ) //  $Q$  contains query keywords
Merge the sorted inverted index list  $S_i$  for  $\forall k_i \in Q$  into list  $S_L$ 
//Find Longest Common Prefix (LCP) list
 $l=0$ ;  $r=s-1$ ;
Traverse  $S_L$  from left to right
while ( $!sU(l, r, s)$ )  $r++$ ; //Identify block of  $s$  unique entries
Find longest common prefix (LCP) of  $s$  unique entries;
Add LCP to LCP list;
if ( $sU(l, r, s)$ )  $r++$ ;  $l++$ ;
for each entry  $e_n$  in LCP list //Find LCE node list from LCP list
 $e_c = \text{null}$ ;
if ( $\text{isEntity}(e_n) > 0$ )
Add  $e_n$  to LCE node list;  $e_c = e_n$ ; Remove  $e_n$  from LCP list;
else if (any ancestor  $e \prec_a e_n$  &  $\text{isEntity}(e)$  &  $e \notin$  LCE node list)
Add  $e$  to LCE node list;  $e_c = e$ ; Remove  $e_n$  from LCP list;
if ( $e_c \neq \text{null}$ )
for ( $\forall e \prec_a e_c$ )
if ( $\text{isEntity}(e) > 0$  &  $e \in$  LCE node list)
Update LCE node ( $e$ );
Rank nodes in LCE/LCP node lists;
return ranked LCE/LCP lists;

```

**Figure 6: GKS algorithm for finding XML nodes**

**Example 4:** In Figure 4,  $d_{id}.0.1$  is the longest common prefix (LCP) of block of first  $s$  nodes. Its entry is created in the LCP list with *counter* set to 1. In Figure 4, node  $d_{id}.0.1$  is found to be entity node. An entry for it is created in LCE node list with keyword counter set to  $(s+counter-1)=2$ . Similarly, the next entry in LCP list  $d_{id}.0.1.1.0$  is initiated with counter set to 1. While checking its ancestors, node  $d_{id}.0.1$  is found to be an entity node. Since  $d_{id}.0.1$  already exist in the LCE node list, its entry (i.e., number of keywords in its sub-tree) is updated to 3 (since node  $d_{id}.0.1.1.0$  appears in its sub-tree). Finally, the keyword count of node  $d_{id}.0.1$  is incremented to 4 and for node  $d_{id}.0.1.1.0$  to 3 (due to next keyword with Dewey id  $d_{id}.0.1.1.0.4$ ). Once the LCE nodes list is computed along with the number of keywords in its sub-tree, we compute a ranking score  $r_i$  for each LCE node, as explained in Section 5. It is also possible that for some node in LCP list, no corresponding LCE node is found.

## 4.2 Correctness and Time Complexity

In this section, we present the analysis of our method and prove the correctness of our methodology to discover the LCE nodes.

For LCE node  $e$ , there must exist at least one keyword  $k \in Q$  that is not contained in any other entity node within its sub-tree (Def. 2.2.1).  $k$  is called the independent witness of node  $e$ . Correctness is defined as discovering LCE nodes according to Def. 2.2.1. We now prove the correctness of our methodology to discover LCE nodes. To discover LCE nodes, LCP list is traversed from left to right. Let left and right pointers  $l$  and  $r$  of the current block under consideration are at position  $p_1$  and  $p_2$  respectively in list  $S_L$  when entity node  $e$  is first time being added to the LCE list.

**Lemma 4:** For entity node  $e$ , just being added to the LCE list, *only* Dewey id of the keyword at position  $p_1$  or at position  $p_2$  can be the smallest Dewey id which is independent witness for node  $e$ .

**Proof:** Omitted.  $\square$

Let  $k$  be the independent witness for node  $e$  with smallest Dewey id. We associate the Deweyid of keyword,  $k$ , with node  $e$ . Let  $e_n$  be the LCE node added immediately after node  $e$  in the LCE list. If  $e \prec_a e_n$  and  $e_n \prec_a k$ , the entity node  $e$  is removed from the LCE list. The reason is:  $k$  is the earliest keyword in document order which was an independent witness for node  $e$  at the time of its addition to LCE list. Since  $k$  itself appears in the sub-tree of its descendent entity node  $e_n$ ,  $e$  is left with no independent witness and hence removed from the LCE list. Note,  $e$  can come back in LCE list if any other keyword is found to be an independent witness for it later in list  $S_L$ . Any entity node  $e$  that survives has at least one independent witness. In Figure 4, entry  $d_{id}.0.1$  survives in LCE node list at the time of addition of  $d_{id}.0.1.1.0$  since it has an independent witness. If any entity node  $e$ ,  $e \prec_a e_n$ , of a newly added entity node  $e_n$  remained in the LCE list, we update its ranking array.

**Lemma 5:** For each LCE node  $e$  that survives in LCE node list, there exists a keyword  $k$  that is an independent witness of  $e$ .

**Claim 1:** Let  $e$  be a lowest common entity node for a block of  $s$  keywords. We claim that at least one of the keywords in the block is an independent witness for node  $e$ .

**Proof:** Proof is by contradiction. Suppose no keyword in the block of  $s$  keywords is an independent witness for LCE node  $e$ . Hence there must exist another LCE node in the sub-tree of  $e$ , which contains all the keywords from the block. Hence node  $e$  is not the *lowest* common entity node, contradicting the initial assumption.  $\square$

**Claim 2:** Any ancestor entity node  $e$ , of entity node  $e_n$ , which is *not* already present in the LCE node list at the time of addition of node  $e_n$  in the LCE list, is *not* the LCE node.

**Proof:** As entity node  $e$ ,  $e \prec_a e_n$ , is not in LCE list, therefore it has no independent witness keyword at the time of addition of  $e_n$ . Since  $e_n$  is the lowest entity node for the current block of keywords,  $e$  cannot be an LCE node.  $\square$

Thus, LCE node  $e$  that survives in LCE node list, there exists a keyword that is an independent witness. When the traversal of LCP list is complete, LCE list contains only true LCE nodes (Def. 2.2.1).

Time complexity to generate the longest common prefix list is  $O(d \cdot |S_L|)$  due to Lemma 6 below (the worst case time complexity could be  $s \cdot d \cdot |S_L|$  where  $s$  is a small constant). Since the Dewey ids are sorted, we just need to find the longest common prefix of first and last Dewey id in the block of  $s$  Dewey ids. There are  $O(|S_L|)$  entries in longest common prefix list and depth of the document is  $d$ . Hence, time complexity to generate LCE nodes list is  $O(d \cdot |S_L|)$ .

**Lemma 6:** For lexically sorted block of  $s$  strings, the common prefix of first and last string is the longest common prefix for the strings in the block.  $\square$

As the time complexity to generate merged Dewey id list is  $O(d \cdot |S_L| \cdot \log n)$ , total time complexity to generate LCE node list, along with its ranking score list is  $O(d \cdot |S_L| \cdot (\log n))$ . Therefore, we efficiently identify LCE nodes in a single pass.

For the LCA based search, the time complexity of the state of the art algorithm to find LCA nodes for query  $Q$ ,  $|Q|=n$ , is  $O(d \cdot n \cdot |S_{min}| \cdot \log |S_{max}|)$  where  $|S_{min}|$  ( $|S_{max}|$ ) is the length of the shortest (longest) inverted index list consisting the Deweyid of the keyword in query  $Q$  [6][16]. We see that the time complexity of our algorithm to find the GKS nodes is only marginally worse than the

time complexity to find LCA nodes, even though the search space for GKS is exponential compared to LCA nodes.

Nodes in Longest Common Prefix (LCP) list contain at least  $s$  keywords in their sub-tree. For each node  $u$  in LCP list we keep a mapping with its associated LCE node  $v$  in LCE list,  $v \prec_a u$ . There may exist some nodes in LCP list such that no corresponding entity node is found for them due to the structure of the XML data.

The XML nodes in LCE list along with those nodes in LCP list for which *no* corresponding LCE node exist together constitute the GKS response  $R_Q(s)$ . These nodes are ranked with the aid of ranking function presented in the next section.

## 5. RANKING

Node ranks help GKS construct a more meaningful response. Number of GKS nodes can be large and the response may comprise a variety of XML node types. The relevance of these nodes varies in the context of a given query. For LCA based techniques, each LCA node is the common ancestor of *all* the keywords in query  $Q$ .

Due to the basic differences between GKS and LCA based search, existing ranking algorithms [8][15] are insufficient for GKS. Existing ranking methods work by using aggregated statistical information for entire XML repository. For a fixed set of keywords, nodes are ranked based on statistical relevance of a query keyword in the context of a given XML node. For GKS, any node containing a subset of keywords belonging to query (of size  $\geq s$ ) is the node of our interest. Further, GKS response may contain a variety of differently structured XML nodes. Therefore, any statistical method is insufficient to compare the relevance of one XML node w.r.t. other due to the structural difference in their sub-trees.

Therefore, we introduce a novel ranking function that computes the rank of each XML node in  $R_Q(s)$  for query  $Q$  based on i) the number of keywords from  $Q$  appearing in its sub-tree; and ii) the structure of the sub-tree rooted at that node.

### 5.1 Ranking Methodology

We use a potential flow model to compute the rank of the XML nodes in  $R_Q(s)$ . Potential of a node is like the amount of water present in a reservoir which flows in a network of pipes coming out from it. The potential flow model automatically incorporates the structure of the sub-tree rooted at an XML node.

We assign an initial potential,  $P|e$  to each node  $e \in R_Q(s)$ .  $P|e$ , for node  $e$  is equal to the number of unique query keywords  $k \in Q_s$ ,  $Q_s \subseteq Q$ , present in its sub-tree.

$$P|e = |Q_s|; Q_s \subseteq Q; Q = \{k_1, \dots, k_n\}$$

$P|e$  just accounts for the presence of a keyword  $k \in Q$  in the sub-tree of node  $e$ . If the keyword  $k$  is present multiple times in node  $e$ , only its highest occurrence in its sub-tree is considered. This highest occurrence of a query keyword in the sub-tree is termed *terminal point*. If a keyword  $k$  is present multiple times at the highest level, *each* of its occurrences is considered a terminal point. For example, if a keyword  $k \in Q$  is a repeating XML element name in the sub-tree of an LCE node, each of its occurrence will be considered as a terminal point (assuming that is the highest level at which keyword  $k$  occurs). Hence, for a user query  $Q = \{k_1, \dots, k_n\}$ , each candidate XML node has a starting potential. As shown in Figure 7, for node  $e_1$ , the highest occurrence of keywords  $k_1, k_2, k_3$  are *terminal points*.

The rank of a node  $e \in R_Q$  is computed as follows: The potential of a node  $e$  is equally divided into each of its child nodes. For a node



$e$  with potential  $(P|e)$ , with  $m$  children, each of its child nodes will receive  $(P|e)/m$  potential, where  $m$  is the number of direct child the node  $e$  has. The rank of the node is sum of the total potential received by each of the terminal points.

Let  $i \rightarrow k$  denotes the relationship that node  $i$  is parent of node  $k$ . The rank of an entity node  $e$  is computed as follows:

$$Rank_e = \sum_{k \in Q} \frac{P_i}{m_i} | i \rightarrow k$$

where  $k$  is a terminal point in the sub-tree of node  $e$ ,  $p_i$  is the potential received by its parent node  $i$  and  $m_i$  is the total number of direct children node  $i$  has. The potential received at terminal points depends on the structure of the sub-tree rooted at the XML nodes.

Intuitively, it implies that the number of distinct query keywords in its sub-tree and the structure of its sub-tree determine the rank of an XML node. Each LCE node is ranked independently, irrespective of its relative depth w.r.t. document root.

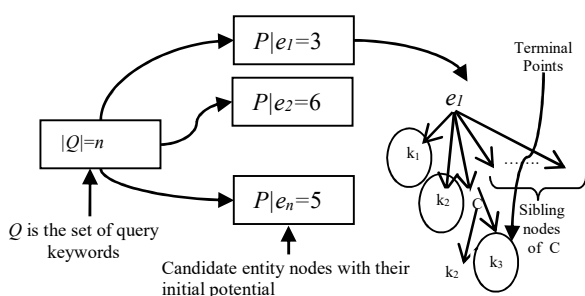


Figure 7. LCE nodes containing keywords in set  $R_Q(s)$

**Example 5:** We illustrate the computation of XML nodes rank with the example XML document shown in Figure 1. For query  $Q_3 = \{a, b, c, d\}$ , GKS returned 3 XML nodes  $x_2, x_3$  and  $x_4$ . The initial potential of node  $x_2$  is  $P|x_2|=3$ . The rank of node  $x_2$  is the potential received by the terminal nodes in its sub-tree, i.e., nodes  $a_2, b_1, c_1$ .

The rank of node  $x_2 = \sum_{k \in Q} P|x_2|/n = 3 \times 3/3 = 3$ . Similarly, for

node  $x_3$ , the initial potential  $P|x_3|$  is 3. The three terminal nodes are nodes  $a_3, b_3, d_3$ . Each of the three children of node  $x_3$  received  $1/3^{\text{rd}}$  of the initial potential. The potential received by  $x_4$  is further divided equally into its two children. Therefore, the total potential received by the terminal nodes, i.e., the rank of node  $x_3$  is,  $2 \times 3/3 + 3/3 \times 1/2 = 2.5$ . Similarly, the rank of node  $x_4$  is 2. Hence, GKS ranking methodology ranks the nodes as  $x_2 > x_3 > x_4$ .

## 6. SEARCH RESULTS ANALYSIS

GKS enables the users to refine their queries. The user query  $Q$  can be refined by either removing or adding the most relevant keywords to  $Q$ , in the context of the query. We now describe how GKS aids the users to refine their query by analyzing the search response.

### 6.1 Query Refinement

Let us consider the Example 1. For query  $Q_3 = \{a, b, c, d\}$ , the response of GKS comprised nodes,  $x_2, x_3$  and  $x_4$ . GKS ranks the nodes such that most relevant nodes are ranked higher. The two top ranked nodes are  $x_2$  containing keywords  $\{a, b, c\}$  followed by  $x_3$  containing keywords  $\{a, b, d\}$ . With this information, the user is exposed to the fact that there is no XML node that contains all the query keywords and that the distribution of the query keywords in the document is as shown in the query results. With this insight, users can refine their queries. In the example above, user can refine the query  $Q_3$  to  $\{a, b, c\}$  or  $\{a, b, d\}$  given the GKS response.

Therefore, for a user query  $Q$ , the query can be refined seamlessly to one or more sub-queries  $Q_r$ s with the aid of the GKS results. As one can see, for LCA based techniques [2][5][17], such refinement of the query  $Q$  is non-trivial as multiple sub-queries of  $Q$  needs to be run to collect the complete response.

## 6.2 DI-Discovery from the LCE Nodes

GKS enables the discovery of  $DI$  from the XML data in the context of the user query which can be used to refine the user query. For a given query, the attribute nodes of a LCE node expose the context for the keywords appearing in its sub-tree and are regarded as the relevant  $DI$  (Def 2.3.1).

A natural way to discover  $DI$  is by identifying  $top-m$  most popular attribute keywords in the LCE nodes present in the query response, i.e., identifying keywords that appear in maximum number of attribute nodes ( $m$  is tunable). At the same time, the  $DI$  must be relevant for most of the query keywords. However, these two goals may translate into two different set of top  $DI$  keywords. In response of the query in Example 2 (Section 1.2), the most popular keyword is found to be `<booktitle: ICPP>`. However, the keyword became most popular due to presence of keyword 'Prithviraj Banerjee'. He is the only author who had published articles in this journal but total number of articles by him alone in this journal made it the most popular keyword in the query response. However, this keyword is not relevant for majority of the other query keywords. The keyword `<journal: SIGMOD Record>` is relevant for the largest sub-set of the query keywords (for remaining three authors) but it is not the most popular. Therefore, to identify most relevant keywords in the context of a query, we adopt the following approach.

Rank of a LCE node is the function of number of query keywords present in its sub-tree. Each attribute node is assigned a weight equal to the rank of its LCE node. Therefore, each keyword in set  $S^w_Q$  is assigned its attribute weight and we prepare a weighted set  $S^w_Q$ . Let  $E_Q \subseteq R_Q(s)$  be a set of all the entity nodes in  $R_Q(s)$ .

$$S^w_Q = \{k : w | w = \sum_{e \in E_Q} Rank_e ; e \in E_Q \wedge k \in attr(e) \wedge k \notin Q\}$$

Each element of set  $S^w_Q$  is a tuple  $k : w$ , where  $k$  is the attribute keyword.  $k$  is assigned a weight that is sum of the rank of all the LCE nodes in set  $E_Q$  that contain  $k$ . The  $top-m$  most weighted keywords constitute  $DI$ . If a keyword in the attribute node is part of the user query  $Q$ , it is not included in the set  $S^w_Q$ . We identify  $top-m$  elements in set  $S^w_Q$ , total time complexity to identify  $DI$  is  $O(|S^w_Q| + m \cdot \log |S^w_Q|) = O(|S^w_Q|)$  as  $|S^w_Q| \gg m$ . Since  $|S^w_Q| = O(|R_Q(s)|)$  and  $|R_Q(s)| \leq Sl$ , the time complexity to identify  $DI$  is better by a factor of  $O(\log |Q|)$  compared to the time complexity to identify LCE nodes and  $DI$  discovery does not constraint the system. In Example 2,  $DI$  contained `<year: 2001>`, `<booktitle: ICPP>`, `<author: Alok N Choudhary>`, etc., as top  $DI$  keywords. Recursive  $DI$  can be discovered by preparing a keyword query using the text keywords identified from  $S^w_Q$ . The recursive  $DI$  may reveal deeper insights.

Therefore, a user query  $Q$  can be refined seamlessly to  $Q_r$  with the aid of  $DI$ . We see that with the aid of response produced by GKS and with the aid of  $DI$ , user queries can be refined by adding or removing the keywords from the initial keyword query.

## 7. EXPERIMENTS

We have built a prototype of GKS [20]. Observations in this section are based on experiments using this prototype over the XML data sets shown in Table 4 [21]. Shakespeare's plays are distributed over multiple files. The experiments were carried out on a Core2 Duo 2.1GHz, 4GB RAM machine running Windows 7 and Java. These data sets are used in many prior works [3][11][13][19]. The size of

Protein Sequence dataset is comparable to the biggest dataset used in a recent work [19]. Our DBLP dataset size is 100% bigger.

Our experiments are designed to assess 1) Performance of GKS; 2) Appropriateness of the node categorization model given the real world XML repositories; 3) Effectiveness of GKS in finding the relevant results for keyword queries and to rank them; 4) Ability of GKS in finding the relevant  $DI$ ; 5) User feedback.

## 7.1 Performance of GKS

### 7.1.1 Size of Index

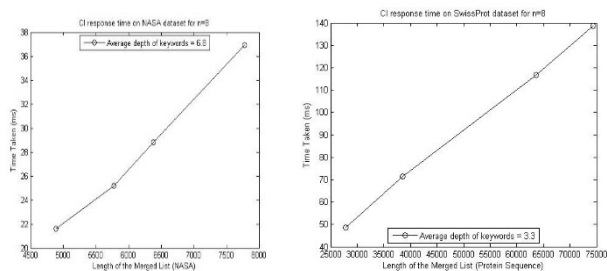
Creating the index is a onetime activity. The size of index and the time taken to prepare them are presented in Table 4. *Our technique is highly scalable as index preparation time increases linearly with the data size.* The number of entity nodes for different datasets varied from 535 for Mondial to 2.62M for DBLP.

**Table 4. Index Size and Index Preparation Time**

Data Set	Data Set Size	Index Size	XML Depth	Index Preparation Time
SIGMOD Records	483KB	416KB	6	0.15s
Mondial	1.7MB	1.45MB	5	0.28s
Plays	1.8MB	1.6MB	5	0.29s
TreeBank	82MB	79MB	36	19.3s
SwissProt	112MB	101MB	8	21.3s
Protein Sequence	683MB	612MB	7	108s
DBLP	1.45GB	1.13GB	6	238s

### 7.1.2 Response Time

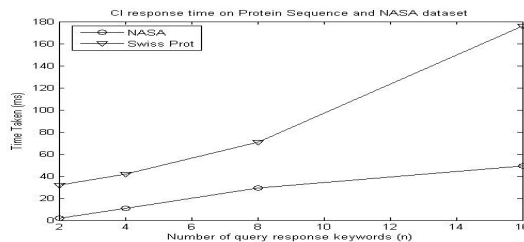
In this experiment, we assess the response time (RT) of GKS for given user queries. We also validate GKS time complexity analysis to discover the XML nodes for the given queries. We give RT results for two datasets: i) NASA dataset containing astronomy data (24MB) and ii) SwissProt dataset containing protein sequence data (112MB). In our first experiment, the number of keywords for each query ( $n$ ) was fixed at 8. However, the size of the merged Dewey id list ( $S_L$ ) varied for each query. The average keyword depth  $d$  for the NASA dataset varied from 6.7-6.9 and from 3.1-3.5 for SwissProt dataset. Results are presented in Figure 8. As shown in Section 4.2, for given  $d$  and  $n$ , the RT increases linearly with  $S_L$ . Response time varies from 21.5ms to 139ms for different queries. Hence, the RT of GKS is only a few tens of ms, similar to LCA based algorithms on similar data.



**Figure 8. Response Time Vs. Merged List Size**

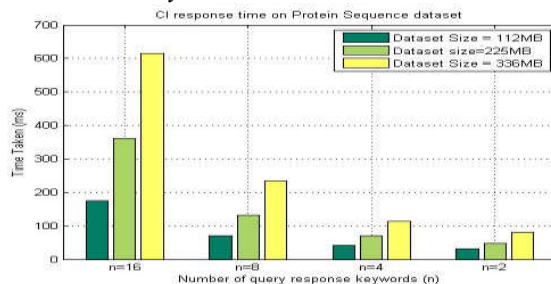
In Figure 9, we plot the RT for queries by varying the number of query response keywords  $n$  from 2 to 16. The query response time validates our analysis (cf. Section 4.2). The list size  $|S_L|$  for query on SwissProt dataset with  $n=16$  was 102,233. In Figure 9, for the NASA dataset, when  $n$  is increased from 8 to 16 for a query, the increase in RT was less than twice, as the length of the list  $|S_L|$  increased only marginally and the change in RT is logarithmic in  $n$ .

For a query run on the DBLP dataset, the RT was found to be 2ms for  $|S_L|=213$ . Hence, *RT depends on the query, i.e., depth  $d$ ,  $n$  and  $S_L$  ( $O(d \cdot |S_L| \cdot \log n)$ ), and not on the size of the data being queried.*



**Figure 9. Response time vs. keywords in query response ( $n$ )**

### 7.1.3 Scalability



**Figure 10. Response time for different dataset sizes**

To assess the scalability of GKS, we replicated the Swiss Prot dataset to create three datasets of size 112MB, 225 MB and 336 MB. For same query, the number of LCE nodes scales linearly. In Figure 10, we plot our results. We can see that *query processing time is scaling linearly with data size*, as expected.

## 7.2 Validation of Node Categorization Model

In this experiment, we analyze the structure of the real world data repositories. XML nodes are placed in different categories as described in Section 2.2. In Table 5, we show the number of different XML element types belonging to different node categories for various XML repositories. As we see, the fraction of nodes labeled as Connecting Nodes (CN) varies from around 15% for InterPro to less than 3% for DBLP dataset. In DBLP/Sigmod Records some nodes with similar schema as that of *entity nodes* (EN) are marked as CN because of the presence of just a single author. We compared the results of our analysis with the ground truth, i.e., with XML schema. For Sigmod Records, two XML elements, `<articles>` and `<authors>`, were the connecting nodes as per the XML schema. The count of `<authors>` node, was 1504, and count of `<articles>` node was 67 (remaining 447 XML nodes were marked CN due to presence of `<article>` nodes with a single author). The results show *our node categorization model captures the structure of the real world data repositories very well.*

**Table 5. Distribution of XML element**

Data Sets	Count of AN	Count of EN	Count of RN	Count of CN	Total Nodes
Sigmod Record	10574	1022	3766	2018	15263
DBLP	27.58M	2.62M	10.56M	972367	39.52M
Mondial-3.0	7467	535	15074	663	22423
InterPro	515316	32614	1472021	303079	2088766
SwissProt	4044884	176128	1776676	187300	5166890

## 7.3 Finding and ranking the XML nodes

The purpose of this experiment is to assess the quality of GKS results. Therefore, some queries are designed for which SLCA

response is obviously inadequate (the response of an SLCA technique is either null or document root for these queries). For a few queries, SLCA returns meaningful response (QM1, QI1, QI2). Queries for different datasets are shown in Table 6. The number of keywords for the queries was varied from 2 to 8. We present the number of XML nodes retrieved by GKS and SLCA. We vary  $s$ , the minimum number of query keywords in the XML node subtree.  $s$  is set to be 1 and  $|Q|/2$  respectively.

**Table 6. Keyword queries run on different datasets**

$Q$	SIGMOD Records
QS1	2 "Anthony I. Wasserman" "Lawrence A. Rowe"
QS2	4 "S. Jerrold Kaplan" "Robert P. Trueblood" "David J. DeWitt" "Randy H. Katz"
QS3	6 "Sakti P. Ghosh" "C. C. Lin" "Timos K. Sellis" "David A. Patterson" "Garth A. Gibson" "Randy H. Katz"
QS4	8 "Barbara T. Blaustein" "Umeshwar Dayal" "Alejandro P. Buchmann" "Upen S. Chakravarthy" "M. Hsu" "R. Ledin" "Dennis R. McCarthy" "Arnon Rosenthal"
$Q$	DBLP
QD1	2 "Dimitrios Georgakopoulos" "Joe D. Morrison"
QD2	4 "Peter Buneman" "Wenfei Fan" "Scott Weinstein" "Prithviraj Banerjee"
QD3	6 "E. F. Codd" "Mark F. Hornick" "Frank Manola" "Alejandro P. Buchmann" "Dimitrios Georgakopoulos" "Joe D. Morrison"
QD4	8 "E. F. Codd" "Kenneth L. Deckert" "Irving L. Traiger" "Vera Watson" "Jim Gray" "Chin-Liang Chang" "Nick Roussopoulos" "Jean-Marc Cadiou"
$Q$	Mondial
QM1	2 country Muslim
QM2	3 Laos country name
QM3	6 Polish Spanish German Luxembourg Bruges Catholic
QM4	8 Chinese Thai Muslim Buddhism Christianity Hinduism Orthodox Catholic
$Q$	InterPro
QI1	2 Kringle Domain
QI2	3 Publication 2002 Science

Results for different queries are shown in Table 7. We see a large number of XML nodes returned for the queries by GKS ( $s=1$ ) compared to SLCA response. Thus, a lot of information that could be of the interest to the user for the given keyword query is not returned by LCA based techniques. Further, the number of XML nodes for ( $s=|Q|/2$ ), is non-zero for all the queries. When we compared GKS with FSLCA [19], the top XML node for both QI1 and QI2 for GKS was present in FSLCA result set. For QM1, many XML nodes of FSLCA were among the top 10 nodes of GKS results. For QM2, no FSLCA node exists but GKS was able to find the XML nodes having subset of query keywords. There are no entity nodes which were relevant, i.e., contained at least  $s$  query keywords, but not identified by GKS. Therefore, *GKS is able to find valid response for the given user queries, without binding them to LCA framework, enhancing the users' ability to search the data.*

We next assess the ability of GKS to rank the discovered XML nodes. Since, the schema for DBLP and Sigmod Records is not very deep, the structure of all the XML nodes is similar for these two datasets. Hence, we adopt the measure that the higher the number of query keywords in an XML node sub-tree, the more relevant it is. Given this measure, we determine where GKS places the XML nodes with the highest number of query keywords in its ranking list.

Let  $L$  be a ranked list of XML nodes, in set  $R_Q(s)$ , returned by GKS for a query  $Q$  and for a given  $s$ ,  $|L| = p = |R_Q(s)|$ . The nodes are numbered 1 to  $p$  according to their ranks. The XML nodes that contain the highest number of keywords from query  $Q$  in their subtree are called the true XML nodes. Let  $L'$  ( $L' \subseteq L$ ) be a set of true

XML nodes. Let  $w$  be the lowest rank of a true XML node in the list of XML nodes  $L$ . To each true XML node, we assign a weight of  $(w+1-i)$  where  $i$  is the rank of the true XML node in the list  $L$ . We compute the aggregated weight of true XML nodes as  $w_a = \sum_{i \leq w} (w+1-i)$  for all the true XML nodes in the list  $L$ . The

total score is computed as  $w_t = w(w+1)/2$ . Finally, we compute a rank score as  $w_a/w_t$ . We penalize the rank score if a true XML node occurs lower in the list  $L$ . Score of 1 means that no true XML node is ranked lower than a XML node which is not in set  $L'$ . In Table 7, we show our results. The ranking score is computed for GKS response when  $s=1$ . We see that *the aggregate weight, i.e., rank score is very high for all the queries.* For every query, except QM3, the top-most result is always a true XML node. For QM3, it appeared at 3<sup>rd</sup> position.

**Table 7. Comparison with SLCA and Rank Score**

Query	#GKS, $s=1$	#GKS, $s= Q /2$	SLCA	Max keywords in a GKS node	Rank Score
QS1	8	NA	0	1	1
QS2	43	13	0	2	1
QS3	28	4	0	3	1
QS4	36	2	1	8	1
QD1	30	NA	1	2	1
QD2	234	10	0	3	0.72
QD3	190	7	0	5	1
QD4	267	4	0	6	1
QM1	230	NA	98	2	1
QM2	234	NA	1	2	1
QM3	37	4	0	3	0.17
QM4	116	3	0	6	1
QI1	8170	NA	8	2	0.893
QI2	2517	2517	281	3	1

In summary, we see that *GKS is able to retrieve and appropriately rank the relevant XML nodes, in the context of the user queries.*

## 7.4 Quality of DI Discovered by GKS

One of the most important attractions of GKS is its ability to discover *DI* in the data. In Table 8, we show the *DI* discovered for the queries in Table 6 for different values of  $s$ . This experiment highlights that the *DI* discovered by GKS is highly relevant for the given queries. For instance, for QD3, *DI* exposes the most relevant year (1999) and the most relevant 'booktitle' (ICCD). The keywords exposed as *DI* also help users understand GKS response since the *DI* keywords also represent the summary of the query response. For some queries, *DI* varies for different values of  $s$ .

**Table 8. DI discovered for different queries**

Query	DI, $s=1$	DI, $s= Q /2$
QS1	<title: Third-Generation Database System Manifesto >	<title: Cache Consistency and Concurrency Control>
QS2	<title: Chair's Message>	<title: Database Research Activities at the University of Wisconsin >
QS3	<title: article title>	<title: Implementation of a Prolog-INGRES Interface>
QS4	<title: article title>	NA
QD1	NA	<year:2000>, <journal:TCS>
QD2	<year: 2001>, <journal: SigmodRecords>	<year:1998>, <volume:2>
QD3	<year: 1999>, <booktitle: ICCD>	<journal: TCS>, <year: 2001>, <number: 1>
QD4	<year: 2001>, <journal: JACM>	<journal:IBM Research Report>, <year: 2001>

QM1	<country:f0_475>, <Year : 90>	NA
QM2	<Name : Zimbabwe>, <population_growth : 1.41>	<Name:Zimbabwe>, <percentage : 100>
QM3	<country:f0_337>,<year: 90>	<country:f0_337>,<year:90>
QM4	<country:f0_663>, <percentage:100>	<country:f0_663>,<name:Brunei>
QI1	<author_list:Patthy L>, <taxon_data name:Eukryot>	NA
QI2	<taxon_data_name:"Bacteria">, <proteins_count : "1">	NA

We now show, with the aid of query QD1, how *DI* helps refine queries. For QD1, GKS returned a total of 30 XML nodes ( $s=1$ ). The *DI* was <author: Marek Rusinkiewicz>. After analyzing the query response, QD1 is refined to ("Dimitrios Georgakopoulos", "Marek Rusinkiewicz"). Interestingly, for the refined query we found that there were 10 articles jointly written by these two authors as opposed to just 1 joint article by authors in original query. This is an example of how *GKS* helps users refine their queries and guides them to navigate the data by recursive application of *GKS*.

### 7.5 Crowd-Sourced Feedback: GKS & SLCA

We asked 40 users to compare the GKS response with SLCA response on a scale of 1-4; 1 being 'GKS Very Useful' and 4 being 'SLCA Very Useful'. Results are shown in the table below. For almost all the queries, *GKS* response is found to be either very useful (1) or better than SLCA (2). If we categorize the response as 'GKS-better' (rating 1 or 2) and 'SLCA-better' (rating 3 or 4), 430 out of 480 responses found the *GKS* response better (89.6%).

Query	1	2	3	4
QS1	24	16	0	0
QS2	17	22	1	0
QS3	17	14	5	4
QS4	12	22	3	4
QD1	24	15	1	0
QD2	18	17	3	2
QD3	20	18	1	1
QD4	24	13	3	0
QM1	16	20	3	1
QM2	15	18	5	2
QM3	14	18	5	3
QM4	16	21	3	0

### 7.6 GKS Performance for Hybrid Queries

We further studied how well GKS behaves in the presence of clearly separable keywords in the query, i.e., subsets of the keywords in a query indeed refer to different entity type nodes. We call such queries 'hybrid queries'. To study GKS performance for hybrid queries, we merged DBLP and Sigmod Record datasets into a single dataset (with a 'common root'). We also increased the depth of Sigmod Record elements by introducing two connecting nodes between the 'common root' and the root of Sigmod Record data. We ran the query "Jean-Marc Meynadier" "Patrick Behm" "Lawrence A. Rowe" "Michael Stonebraker",  $s=2$ . First two authors appeared together only in <inproceedings> entity node type in DBLP dataset and last two in <article> entity node type in Sigmod Record. Clearly, the keywords in the query target two different XML node types. GKS was able to return all 8 corresponding XML nodes present in our dataset, 3 <inproceedings> nodes in DBLP data by first 2 authors and 5 <article> nodes in Sigmod Record data by last 2 authors. Thus, *GKS* returned correct response even when multiple XML node types were targeted by a single query. Note, only these 8 nodes were returned by GKS.

Further, two <article> nodes, by last two authors, were ranked higher (as they were the only authors in all the three articles) despite higher relative depth w.r.t. root (articles by first 2 authors had multiple other authors). Hence, *the entity nodes are ranked based on only the number of query keywords present in their sub-tree and the distribution of these keywords*, and not according to their absolute depth in the XML tree, as analyzed in Section 5.

**Summary:** In summary, experiments show that GKS is scalable, imposes low overhead and retrieves the XML nodes efficiently. The experiments validate our node categorization model and show that XML nodes and *DI* discovered by GKS are highly relevant

## 8. CONCLUSION

We presented a novel system GKS that enables generic keyword search over XML data and yields highly meaningful response without imposing the AND-Semantics of LCA based techniques. We show that our system exposes deeper analytical insights (*DI*) in the data in the context of user queries. GKS exploits the XML schema, embedded in the XML data, in the context of the query to find the most relevant data keywords and schema elements with the aid of a novel node categorization model. In conjunction with a novel XML node ranking method, GKS is able to expose the *DI* elegantly. One of our future research direction is to extend GKS to enable analytics over raw XML data.

## 9. REFERENCES

- [1] I. Tatarinov, et al., "Storing and querying ordered XML using a relational database system", in SIGMOD, 2002.
- [2] Y. Xu, Y. Papakonstantinou, "Efficient Keyword Search for Smallest LCAs in XML Databases", in EDBT 2008.
- [3] Z. Liu, Y. Chen, "Identifying Meaningful Return Information for XML Keyword Search", in SIGMOD 2007.
- [4] J. Li, C. Liu, R. Zhou, W. Wang, "Suggestion of promising result types for xml keyword search", in EDBT, 2010.
- [5] R. Zhou, C. Liu, J. Li, "Fast ELCA Computation for Keyword Queries on XML Data", in EDBT 2010.
- [6] L. Chen, Y. Papakonstantinou, "Supporting Top-K Keyword Search in XML Databases", in ICDE 2010.
- [7] L. Guo, et al., "XRANK: Ranked Keyword Search over XML Documents", in SIGMOD 2003.
- [8] S. Cohen, J. Mamou, Y. Kanza, Y. Sagiv, "XSEarch: A Semantic Search Engine for XML", in VLDB 2003.
- [9] Z. Bao, J. Lu, T. W. Ling, B. Chen, "Towards an effective xml keyword search", in IEEE TKDE, 22(8):1077-1092, 2010.
- [10] H. Cao, et al., "Feedback-driven Result Ranking and Query Refinement for Exploring Semi-structured Data Collections", in EDBT 2010.
- [11] Z. Liu and Y. Chen. "Reasoning and identifying relevant matches for xml keyword search", in PVLDB, 1(1), 2008.
- [12] L. Kong, R. Gilleron, A. Lemay. "Retrieving meaningful relaxed tightest fragments for xml keyword search", in EDBT, 2009.
- [13] Y. Xu, Y. Papakonstantinou. "Efficient keyword search for smallest lcas in xml databases", in SIGMOD, 2005, pp 537-38.
- [14] M. Arenas, "Normalization Theory for XML", in SIGMOD Record, Vol. 35, No. 4, December 2006.
- [15] Z. Bao, T. Ling, B. Chen, J. Lu, "Effective XML Keyword Search with Relevance Oriented Ranking", in ICDE 2009.
- [16] J. Zhou et al., "Efficient query processing for XML keyword queries based on the IDList index", The VLDB Journal, February 2014, Volume 23, Issue 1, pp 25-50.
- [17] J. Zhou et al., "Fast SLCA and ELCA Computation for XML Keyword Queries based on Set Intersection", in ICDE 2012
- [18] Manish Bhide, Manoj K. Agarwal, et. al., "XPEDIA: XML Processing for Data Integration", in VLDB 2009. .
- [19] B. Truong, et al., "MESSIAH: Missing Element Conscious SLCA Nodes Search in XML Data", in SIGMOD 2013.
- [20] Manoj K Agarwal, Krithi Ramamritham, "Enabling Generic Keyword Search over Raw XML Data", ICDE, 2015, pp 1496-99.
- [21] <http://www.cs.washington.edu/research/xmldatasets/www/repository.htm>

# Answering Keyword Queries involving Aggregates and GROUPBY on Relational Databases

Zhong Zeng  
National University of  
Singapore  
zengzh@comp.nus.edu.sg

Mong Li Lee  
National University of  
Singapore  
leeml@comp.nus.edu.sg

Tok Wang Ling  
National University of  
Singapore  
lingtw@comp.nus.edu.sg

## ABSTRACT

Keyword search over relational databases has gained popularity as it provides a user-friendly way to explore structured data. Current research in keyword search has largely ignored queries to retrieve statistical information from the database. The work in [13] extends keywords by supporting aggregate functions in their SQAK system. However, SQAK does not consider the semantics of objects and relationships in the database, and thus suffers from the problems of returning incorrect answers. In this work, we propose a semantic approach to answer keyword queries involving aggregates and GROUPBY. Our approach utilizes the ORM schema graph to capture the Object-Relationship-Attribute (ORA) semantics in the database, and determines the various interpretations of a query before generating the corresponding SQL statements. These semantics enable us to distinguish objects with the same attribute value and detect duplications of objects in relationships to compute the answers correctly. Our approach can also handle unnormalized relations in the database and GROUPBY in keyword queries which SQAK cannot. Experiments on the TPC-H and ACM Digital Library publication datasets demonstrate the advantages of the proposed semantic approach in retrieving correct statistical information for users.

## 1. INTRODUCTION

As databases increase in size and complexity, the ability for users to issue structured queries in SQLs has become a challenge. Keyword search over relational databases has gained traction as it enables users to query the database without knowing the database schema or having to write complicated SQL queries [7, 10, 8, 1, 2]. Research on relational keyword search has focused on the efficient computation of the minimal set of tuples that contain all the query keywords [9, 6, 5, 4], and strategies to retrieve relevant answers to the query [6, 10, 14, 3]. However, these works do not handle keyword queries involving aggregate functions and GROUPBY. We call these aggregate queries.

Aggregate queries is a powerful mechanism that provides users with a summary of the data. The work in [13] developed a prototype system called SQAK that allows aggregate queries to be expressed using simple keywords. An aggregate query in SQAK comprises of a set of terms and one of the terms is an aggregate function such as *COUNT*, *SUM*, *AVG*, *MIN*, or *MAX*. SQAK models the database schema as a schema graph where each node represents a relation and each edge represents a foreign key-reference. Then SQAK identifies the matches of each term in a query. A relation is matched if a term matches its name, or the name of one of its attributes, or the value of some of its tuples. A set of minimal connected subgraphs of the schema graph that contain the matched relations are generated. These subgraphs are translated into SQL statements to retrieve answers from the database. Note that an aggregate function(s) is applied to the attribute that follows the aggregate term in the query.

Figure 1 shows a sample university database. Suppose we want to know the total credits obtained by the student Green. We can issue a keyword query  $Q_1 = \{\text{Green SUM Credit}\}$ , where the term *SUM* indicates the aggregate function *SUM* on the course credits, and SQAK will generate the following SQL statement for the query:

```
SELECT S.Sname, SUM(C.Credit)
FROM Student S, Enrol E, Course C
WHERE E.Sid=S.Sid AND E.Code=C.Code
AND S.Sname='Green' GROUP BY Sname
```

Student			Course			Enrol			Teach		
Sid	Sname	Age	Code	Title	Credit	Sid	Code	Grade	Code	Lid	Bid
s1	George	22	c1	Java	5.0	s1	c1	A	c1	l1	b1
s2	Green	24	c2	Database	4.0	s1	c2	B	c1	l1	b2
s3	Green	21	c3	Multimedia	3.0	s1	c3	B	c1	l2	b1
						s2	c1	A	c2	l1	b2
						s3	c1	A	c2	l1	b3
						s3	c3	B	c3	l2	b4

Textbook			Lecturer			Department			Faculty	
Bid	Tname	Price	Lid	Lname	Did	Did	Dname	Fid	Fid	Fname
b1	Programming Language	10	l1	Steven	d1	d1	CS	f1	f1	Engineering
b2	Discrete Mathematics	15								
b3	Database Management	12								
b4	Multimedia Technologies	20	l2	George	d1					

Figure 1: Example university database

We observe that SQAK may compute incorrect answers when a query term matches multiple tuples. We see that the term *Green* in  $Q_1$  matches the names of two students  $s_2$  and  $s_3$  in Figure 1. This naturally implies that we should find the sum of the credits obtained by each of these students,

i.e., the total credits for  $s_2$  is 5 while the total credits for  $s_3$  is 8. However, SQAK does not distinguish between these two “different” name matches, and outputs a total credits of 13 for students called **Green**, which is incorrect.

Similarly, SQAK may return incorrect answers when a query matches a relation that has more than 2 foreign keys. For instance, the *Teach* relation in Figure 1 contains 3 foreign keys that reference the relations *Course*, *Lecturer* and *Textbook* respectively, and depicts that a course can be taught by more than one lecturer using different textbooks. Suppose we have a query  $Q_2 = \{\text{Java SUM Price}\}$ , where the term **Java** matches a course title while the term **Price** matches an attribute of the *Textbook* relation. This implies that we should return the total price of the textbooks that are used in the **Java** course. Based on the *Teach* relation, there are 2 such textbooks  $b_1$  and  $b_2$  whose total price is 25. But SQAK will generate the following SQL statement:

```
SELECT C.Title, SUM(B.Price)
FROM Course C, Teach T, Textbook B
WHERE T.Bid=B.Bid AND T.Code=C.Code
AND C.Title='Java' GROUP BY C.Title
```

which returns 35 for total price because textbook  $b_1$  appears 2 times for the **Java** course (i.e.,  $c_1$ ) in the *Teach* relation. This answer is incorrect as a student does not need 2 copies of the same textbook for a course.

In addition, many applications often denormalize their databases to improve runtime performance. This denormalization leads to data duplication which affects the database schema graph. As SQAK does not consider unnormalized relations in the database, it will return incorrect answers for aggregate queries.

Figure 2 shows an unnormalized university database where the *Lecturer* relation now has a foreign key that references the *Faculty* relation. Consider the query  $Q_3 = \{\text{Engineering COUNT Department}\}$ , where the term **Engineering** matches a faculty name while the term **Department** matches the name of the *Department* relation. SQAK will find the number of departments in **Engineering** faculty by joining the relations *Department*, *Lecturer* and *Faculty*, and output an incorrect answer 2. This is because the values of attributes *Did* and *Fid* in the *Lecturer* relation are duplicated.

Lecturer				Department		Faculty	
Lid	Lname	Did	Fid	Did	Dname	Fid	Fname
l1	Steven	d1	f1	d1	CS	f1	Engineering
l2	George	d1	f1				

Figure 2: An unnormalized university database

We advocate that a relational database is essentially a repository of objects that interact with each other via relationships that are embedded in foreign key-key references. Since SQAK does not consider the semantics of objects and relationships in the database, it will not be able to distinguish objects with the same attribute value (as in  $Q_1$ ), and it will fail to detect the duplications of objects in relationships (as in  $Q_2$ ). This leads to the incorrect computations of aggregate queries. Further, if relations are unnormalized with duplicate information of objects and relationships, SQAK may compute the same information repeatedly and return incorrect answers to aggregate queries (as in  $Q_3$ ).

In this paper, we propose a semantic approach to answer keyword queries involving aggregates and GROUPBY on re-

lational databases. Our approach utilizes the ORM schema graph introduced in [15] to capture the Object-Relationship-Attribute (ORA) semantics in the database. Given an aggregate query, we analyze the context of query keywords, interpret the various interpretations of the query and then apply aggregate functions and GROUPBY on the appropriate attributes of objects/relationships based on the ORM schema graph. Each query interpretation is denoted as a minimal connected graph called annotated query pattern. The top-k ranked annotated query patterns are translated into SQL statements to compute the answers to the aggregate query. By using the ORA semantics, we can distinguish the objects with the same attribute value as well as detect the duplications of objects in relationships in order to avoid incorrect computations of aggregate functions. Otherwise, it is impossible to answer aggregate queries correctly. We also develop a mechanism to detect duplicate information of objects/relationships arising from unnormalized relations so that the aggregate functions will not repeatedly compute statistics for the same information.

The contributions of our work are summarized as follows:

1. We examine how SQAK answers aggregate queries in relational keyword search, and identify its problems of returning incorrect answers due to its unawareness of the ORA semantics in the database.
2. We extend the keyword query language to incorporate aggregates and GROUPBY, and propose a semantic approach to process aggregate queries. We show that without the ORA semantics, it is impossible to process the aggregate functions correctly.
3. By using the ORA semantics, we detect the duplications of objects and relationships arising from unnormalized relations, and extend our approach to handle aggregate queries on unnormalized databases correctly.
4. We conduct extensive experiments to demonstrate the correctness of our approach in retrieving statistical information for users.

## 2. PRELIMINARIES

The work in [15] extends the keyword query language to include keywords that match meta-data, i.e., the names of relations and attributes. These keywords reduce query ambiguity by providing the context of subsequent keywords in the query.

Consider the query  $\{\text{Lecturer George}\}$  on the database in Figure 1. The keyword **George** can refer to a student name or a lecturer name. However, since the keyword **Lecturer** matches the name of the relation *Lecturer* and provides the context of the keyword **George**, we deduce that the user is more likely to be interested in a lecturer named **George** rather than a student. Here, we further extend the query language to incorporate aggregates and GROUPBY.

**DEFINITION 1.** A keyword query  $Q$  is a sequence of terms  $\{t_1 t_2 \dots t_n\}$  where each term  $t_i$  either matches a relation name, an attribute name, a tuple value, GROUPBY or an aggregate function MIN, MAX, AVG, SUM or COUNT.

In order to properly interpret a keyword query involving aggregate functions and GROUPBY, we impose the following constraints on the terms in the query:

1. The last term  $t_n$  cannot match an aggregate function or GROUPBY.
2. For each term  $t_i$ ,  $i < n$  that matches the aggregate function  $MIN$ ,  $MAX$ ,  $AVG$  or  $SUM$ , the next term  $t_{i+1}$  should match an attribute name.
3. For each term  $t_i$ ,  $i < n$  that matches  $COUNT$  or GROUPBY, the next term  $t_{i+1}$  should match either a relation name or an attribute name.

A query that satisfies the last constraint is {COUNT Student GROUPBY Course}. An SQL statement to find the number of students in each course is generated as follows:

```
SELECT C.Code, COUNT(S.Sid) As numSid
FROM Student S, Enrol E, Course C
WHERE E.Sid=S.Sid AND E.Code=C.Code
GROUPBY C.Code
```

Note that the terms Student and Course match the names of the *Student* and *Course* relations, and are mapped to *Sid* and *Code* respectively.

## 2.1 Query Patterns

The work in [16] classifies the relations in a database into object relations, relationship relations, mixed relations and component relations. An object (relationship resp.) relation captures the information of objects (relationships resp.), i.e., the single-valued attributes of an object class (relationship type). Multivalued attributes of an object class (relationship type) are captured in object/relationship component relations. A mixed relation contains information of both objects and relationships, which occurs when we have a many-to-one relationship. We call these semantics the Object-Relationship-Attribute (ORA) semantics.

A keyword query is inherently ambiguous as each keyword can have multiple matches. [15] introduces the notion of query patterns to depict the interpretations of a keyword query. These query patterns are generated from the Object-Relationship-Mixed (ORM) schema graph of the database. The ORM schema graph is an undirected graph that captures the ORA semantics in the database. Each node in the ORM schema graph comprises of an object/relationship/mixed relation and its component relations, and is associated with a type (object, relationship and mixed). Two nodes are connected if there exists a foreign key - key reference between the relations in these two nodes.

In Figure 1, the relations *Student*, *Course*, *Faculty* and *Textbook* are object relations while *Enrol* and *Teach* are relationship relations. Relations *Lecturer* and *Departement* are mixed relations because of the many-to-one relationships between lecturers and departments, and the many-to-one relationships between departments and faculties respectively. Figure 3 shows the ORM schema graph of the database.

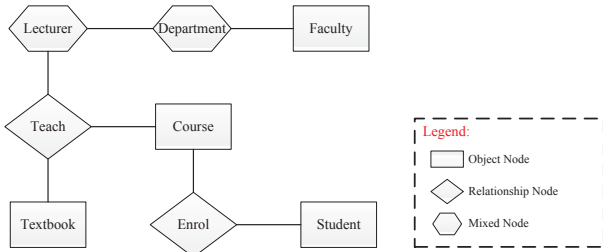


Figure 3: ORM schema graph of Figure 1

A query pattern for the keyword query {Green George Code} is shown in Figure 4. This pattern depicts the query interpretation to find the common courses taken by students Green and George. We generate this pattern by first identifying the matches of each term in the query. The term Code matches the name of an attribute in the *Course* relation, while both terms Green and George match some tuple value in the *Student* relation, specifically, the value of *Sname* attribute. Based on these matches, we know that Green and George refer to two student objects and Code refers to a course object. Thus, we create 2 Student nodes and 1 Course node to represent these objects. From the ORM schema graph in Figure 3, we know that the Student node and the Course node are connected via an Enrol node. Hence, we create 2 Enrol nodes and obtain the query pattern in Figure 4.



Figure 4: Query pattern of {Green George Code}

Note that the term George can refer to a lecturer object since it also matches some tuple value in the *Lecturer* relation. Hence, this keyword query has more than one query pattern. We rank these query patterns and translate the top-k ranked patterns into SQL statements.

In this work, we want to utilize these query patterns to capture the interpretations of an aggregate query. However, since we extend the keyword query to include GROUPBY and aggregate functions, we need to annotate the nodes that the GROUPBY and aggregate functions are applicable to. Annotating the appropriate nodes is important as it will facilitate the translation of the query pattern into SQL statements to retrieve the correct answers for the aggregate query. We will discuss how we achieve this in the next section.

## 3. AGGREGATE QUERIES ON NORMALIZED DATABASE

### 3.1 Simple Aggregate Queries

Given a keyword query  $Q = \{t_1 t_2 \dots t_n\}$ , we first classify the terms  $t_i$  into *basic terms* and *operators*. A basic term matches a relation name, or an attribute name, or a tuple value, while an operator matches GROUPBY or an aggregate function. Then we process  $Q$  as follows:

1. **Pattern generation and annotation.** We utilize the ORM schema graph of the database and the basic terms in the query to generate a set of query patterns, and annotate these patterns with the operators.
2. **Pattern disambiguation.** We disambiguate the query patterns by annotating the object/mixed nodes with GROUPBY. This is to distinguish objects with the same attribute value in the database.
3. **Pattern translation.** We translate the top-k ranked query patterns into SQL statements to compute the aggregate functions in the query.

We explain the details of these steps next.

#### 3.1.1 Pattern Generation and Annotation

We use the basic terms in a query to generate a set of initial query patterns. Each pattern  $P$  contains a set of

nodes that represent the objects or relationships referred to by the basic terms. The nodes are connected based on the ORM schema graph as described in [15]. A node is annotated with the condition  $a = t$  if the basic term  $t$  refers to the value of the attribute  $a$  of the object/relationship.

For each operator  $t_i \in Q$ , we examine the matches of its subsequent term  $t_{i+1}$  in  $Q$  to annotate query pattern  $P$ . We have two cases:

- a.  $t_{i+1}$  matches the name of some object/ mixed/ relationship relation.

This indicates that  $t_{i+1}$  refers to some object or relationship, and the operator  $t_i$  is applied on the identifier  $id$  of this object/relationship. We annotate the node that represents this object/relationship in  $P$  with  $t_i(id)$ ,  $id$  is given by the primary key of the relation.

- b.  $t_{i+1}$  matches the name of a component relation or an attribute name.

This indicates that  $t_{i+1}$  refers to some attribute  $a$  of an object or relationship, and  $t_i$  is applied on this object/relationship attribute. We annotate the node that represents this object/relationship in  $P$  with  $t_i(a)$ .

**EXAMPLE 1.** Consider query  $Q_4 = \{\text{Green George COUNT Code}\}$ . Figure 4 shows a query pattern obtained using the basic terms *Green*, *George* and *Code*. For the operator *COUNT*, its subsequent term *Code* matches an attribute name in the *Course* relation. Hence, we annotate the *Course* node with *COUNT(Code)*. Figure 5(a) shows the annotated query pattern  $P_1$  that depicts the query interpretation to find the total number of courses taken by students *Green* and *George*.  $\square$

**EXAMPLE 2.** Query  $Q_5 = \{\text{COUNT Lecturer GROUPBY Course}\}$  has two basic terms *Lecturer* and *Course*. We generate a query pattern that contains a *Teach* relationship node between the objects *Lecturer* and *Course*. For the operator *GROUPBY*, since its subsequent term *Course* matches the name of the *Course* relation, and refers to a course object, we obtain the identifier of the course object and annotate the corresponding *Course* node in the query pattern with *GROUPBY(Code)*. Similarly, operator *COUNT* has a subsequent term *Lecturer* that matches the name of the *Lecturer* relation. We annotate the *Lecturer* node with *COUNT(Lid)* and obtain the annotated query pattern  $P_2$  in Figure 5(b). This query pattern indicates that the user is interested in the number of lecturers for each course.  $\square$

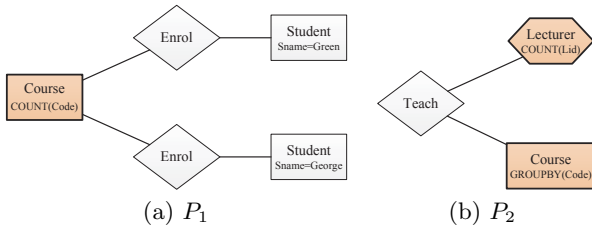


Figure 5: Annotated query patterns of  $Q_4$  and  $Q_5$

### 3.1.2 Pattern Disambiguation

After annotating the query pattern with operators, we examine the object and mixed nodes in the pattern. An object/mixed node with the condition  $a = t$  refers to an

object such that its value of attribute  $a$  matches the basic term  $t$ . However, since this condition could be satisfied by more than one object in the database, there are two different interpretations of  $t$  in the context of the aggregate query:

1. Apply the aggregate function(s) for every *distinct* object satisfying the condition  $a = t$ .
2. Apply the aggregate function(s) for *all* the objects satisfying the condition  $a = t$ .

These two interpretations will lead to different results of the aggregate function(s) and we need to distinguish them in the annotated query pattern. Note that SQAk does not distinguish objects satisfying the same condition, and thus returns incorrect answers to the query.

Let  $P$  be an annotated query pattern for aggregate query  $Q$  and  $U$  be a set of object/mixed nodes in  $P$ . We generate a set of patterns  $S$  to indicate if objects with the same value will be distinguished for aggregates. Initially,  $S$  only contains the pattern  $P$ . For each node  $u \in U$  that is annotated with the condition  $a = t$ , we check if more than one object satisfies this condition in the database. If so, we create a copy of each pattern in  $S$  to indicate if objects that satisfy the condition  $a = t$  will be distinguished in these patterns. Let  $P_1$  be a pattern in  $S$  and  $P_2$  be a copy of  $P_1$ . We annotate  $u$  in  $P_2$  with *GROUPBY(id)*, where  $id$  is the identifier of the object referred to by  $u$ . In particular,  $P_1$  indicates that aggregate function(s) is applied for all the objects that satisfy  $a = t$ , while  $P_2$  indicates that aggregate function(s) is applied for every distinct object with  $a = t$ .

**EXAMPLE 3.** Consider the query pattern  $P_1$  for the query  $Q_4 = \{\text{Green George COUNT Code}\}$  in Figure 5(a). This pattern contains three object/mixed nodes: one *Course* node and two *Student* nodes that are annotated with the conditions *Sname = Green* and *Sname = George* respectively. For the *Student* node imposed by the condition *Sname = George*, we do not need to create new copies of query patterns as there is only one student called *George* in Figure 1. However, for the *Student* node imposed by the condition *Sname = Green*, we know that there are two students called *Green* in Figure 1. Hence, we create a copy  $P_3$  of the pattern  $P_1$  and annotate this node with *GROUPBY(Sid)* in  $P_3$ . Figure 6 shows the query pattern  $P_3$ . It indicates that the aggregate function counts the number of courses enrolled by each student called *Green*. In contrast, pattern  $P_1$  indicates that the count aggregate is applied for both these two students.  $\square$

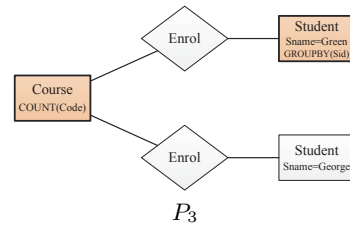


Figure 6: A query pattern for  $Q_4$

Next, we rank the query patterns. [15] classifies nodes in a query pattern into target nodes and condition nodes. A target node specifies the search target of the query, and a condition node indicates the search conditions of the query. A query pattern is ranked based on its number of object/mixed



nodes and the average distance between the target and condition nodes. Patterns with fewer object/mixed nodes, and a shorter average distance are ranked higher.

Here, we extend the definitions of target nodes and condition nodes to rank the query patterns of an aggregate query. Let  $P$  be an annotated query pattern and  $u$  be a node in  $P$ . We say  $u$  is a target node if  $u$  is annotated with an aggregate function. Otherwise,  $u$  is a condition node if  $u$  is annotated with a condition or GROUPBY. We can now use the same method as [15] to rank the annotated query patterns.

### 3.1.3 Pattern Translation

Finally, we translate the top-k ranked query patterns into SQL statements. A straightforward way to translate an annotated query pattern is to join the relations of all the nodes in the pattern, select the tuples that satisfy the conditions imposed by basic terms from the join result, and then apply GROUPBY and aggregate function(s) on the selected tuples. However, this may generate an SQL statement that gives an incorrect answer to the query.

**EXAMPLE 4.** Consider the query pattern  $P_2$  for the query  $Q_5 = \{COUNT\ Lecturer\ GROUPBY\ Course\}$  in Figure 5(b). If we simply translate  $P_2$  into an SQL statement that joins the relations *Teach*, *Lecturer* and *Course*, and applies the count aggregate on the lecturer id *Lid* after grouping the tuples by the course code, we may obtain wrong answers as the same lecturer may be counted multiple times. This is because the *Teach* node in  $P_2$  is in fact a ternary relationship involving the objects course, lecturer and textbook (see the ORM schema graph in Figure 3). Since different *Bid* may have the same *Lid* and *Code*, we should project the *Teach* relation on the foreign keys  $\langle Lid, Code \rangle$  to remove duplicates before joining with the relations *Lecturer* and *Course*.  $\square$

The above example demonstrates the need to examine the type of nodes in a query pattern if we want to generate the SQL statement correctly. In particular, if the query pattern contains a relationship node  $u$ , we should look at its corresponding node  $v$  in the ORM schema graph to determine if a projection is needed to remove duplicates. Note that SQAK does not detect the duplications of objects in relationships, and suffers from the problem of returning incorrect answers.

Given a query pattern  $P$ , we generate the clauses in an SQL statement as follows:

**SELECT clause.** If a node  $u \in P$  is annotated with  $t(a)$  and  $t$  matches an aggregate function, we include  $t$  in the SELECT clause.  $t$  is applied on attribute  $a$ . If  $u$  is annotated with  $GROUPBY(a)$ , we include  $a$  in the SELECT clause to facilitate user understanding of the aggregate function(s).

**FROM clause.** This clause includes the relations of all the nodes in  $P$ . For each relationship node  $u \in P$ , we check its corresponding node  $v$  in the ORM schema graph. Let  $N_u = \{u_1, u_2, \dots, u_x\}$  be a set of object/mixed nodes that are directly connected to  $u$  in  $P$ , and  $N_v = \{v_1, v_2, \dots, v_y\}$  be the set of object/mixed nodes that are directly connected to  $v$  in the ORM schema graph. If  $x < y$ , then this indicates that  $P$  contains a subset of the participating objects of the relationship  $v$ , and we project the foreign keys  $k_1, k_2, \dots, k_x$  in the relation of  $u$  such that  $k_i$  references the relation of  $u_i$  in  $N_u$ ,  $i \in [1, x]$ . This projection eliminates duplicates and we replace the relation of  $u$  in the FROM clause with the relation obtained by this projection.

**WHERE clause.** The WHERE clause joins all the relations in the FROM clause based on foreign key - key references. For each node  $u \in P$  that is annotated with a condition  $a = t$ , we include the condition “ $R_u.a$  contains  $t$ ” where  $R_u$  is the relation corresponding to  $u$ .

**GROUPBY clause.** If a node  $u$  is annotated with  $t(a)$  and  $t$  matches GROUPBY, then we include the attribute  $a$  in the GROUPBY clause.

**EXAMPLE 5.** The query pattern  $P_3$  in Figure 6 for query  $Q_4 = \{Green\ George\ COUNT\ Code\}$  depicts the total number of courses enrolled by the student *George* and each of the students called *Green*. The *Course* node is annotated with  $COUNT(Code)$ , thus we include  $COUNT(Code)$  in the SELECT clause. The FROM clause contains the relations corresponding to each of the nodes in  $P_3$ . Next, we add the conditions to join these relations in the WHERE clause, as well as the conditions in the two annotated *Student* object nodes. Since the *Student* node imposed by the condition  $Sname = Green$  is also annotated with GROUPBY to distinguish different students called *Green*, we include the id of its relation in the GROUPBY clause, and obtain the SQL statement:

```
SELECT S1.Sid, COUNT(C.Code) AS numCode
FROM Course C, Enrol E1, Student S1, Enrol E2, Student S2
WHERE C.Code=E1.Code AND C.Code=E2.Code
      AND S1.Sid=E1.Sid AND S1.Sname contains 'Green'
      AND S2.Sid=E2.Sid AND S2.Sname contains 'George'
GROUP BY S1.Sid
```

By applying GROUPBY on student ids, we distinguish students  $s_2$  and  $s_3$  who have the same name *Green*, and the aggregate function  $COUNT$  is computed for the courses of each student.  $\square$

**EXAMPLE 6.** The query pattern  $P_2$  in Figure 5(b) for query  $Q_5 = \{COUNT\ Lecturer\ GROUPBY\ Course\}$  depicts the number of lecturers for each course. The *Lecturer* node is annotated with  $COUNT(Lid)$  while the *Course* node is annotated with  $GROUPBY(Code)$ . Hence, we include the aggregate function  $COUNT(Lid)$  in the SELECT clause, and the attribute  $Code$  in the GROUPBY clause. The *Teach* node in  $P_2$  is connected to two object/mixed nodes, while the corresponding *Teach* node in the ORM schema graph in Figure 3 is connected to three object/mixed nodes. We generate a subquery “ $SELECT\ DISTINCT\ Lid, Code\ FROM\ Teach$ ” to project the attributes *Lid* and *Code* in the *Teach* relation. The subquery has a *DISTINCT* keyword and thus eliminates any duplicates of  $\langle Lid, Code \rangle$  for different *Bid*. We use the result of this subquery to join the other relations in the FROM clause. The SQL statement generated is:

```
SELECT C.Code, COUNT(L.Lid) AS numLid
FROM Lecturer L, Course C,
      (SELECT DISTINCT Lid, Code FROM Teach) T
WHERE T.Lid=L.Lid AND T.Code=C.Code
GROUP BY C.Code
```

## 3.2 Nested Aggregate Queries

So far, we have described how to handle keyword queries involving simple aggregate functions and GROUPBY. In order to maximize the power of aggregate queries, we also want to support queries with nested aggregate functions.

Given a keyword query  $Q = \{t_1\ t_2\ \dots\ t_n\}$ , we relax the constraints on the terms so that if the term  $t_i$ ,  $i < n$  matches

an aggregate function, the next term  $t_{i+1}$  can also match an aggregate function. In this case, the aggregate function  $t_i$  is applied on the result of the aggregate function  $t_{i+1}$ .

Let  $P$  be a query pattern obtained from basic terms in the query. We annotate  $P$  with  $t_i(f)$ , where  $f$  is the attribute name assigned to the result of aggregate function  $t_{i+1}$ . Then we generate a nested SQL statement for  $P$ . The inner query computes the aggregate function  $t_{i+1}$ , while the outer query includes the inner query in the FROM clause and computes the aggregate function  $t_i$ .

**EXAMPLE 7.** Suppose the user issues a query  $\{\text{AVG COUNT Lecturer GROUPBY Course}\}$  to find the average number of lecturers that teach a course. Both the terms *AVG* and *COUNT* match some aggregate function. We obtain the query pattern and annotate the operators *COUNT* and *GROUPBY*. For the *AVG* operator, we annotate the pattern with  $\text{AVG}(\text{numLid})$ , where *numLid* is the attribute name given to the result of the aggregate function *COUNT*. Figure 7 shows the annotated query pattern. We translate the query pattern by first generating an inner SQL query similar to that in Example 6. Then we put it in the FROM clause of the outer SQL query to compute the aggregate function *AVG* as follows:

```
SELECT AVG(R.numLid) AS avgnumLid
FROM ( SELECT C.Code, COUNT(L.Lid) AS numLid
      FROM Lecturer L, Course C,
           (SELECT DISTINCT Lid, Code FROM Teach) T
      WHERE T.Lid=L.Lid AND T.Code=C.Code
      GROUP BY C.Code) R
```

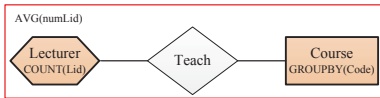


Figure 7: Query pattern in Example 7

#### 4. AGGREGATE QUERIES ON UNNORMALIZED DATABASE

Relations in a relational database are often unnormalized to reduce the number of joins and improve query processing performance. A relational database that contains unnormalized relations is called an unnormalized database. The denormalization process will duplicate information in the database and SQAk may obtain incorrect results for keyword queries involving aggregates.

Recall that in Figure 2, the *Lecturer* relation is denormalized by adding a foreign key *Fid* that references the *Faculty* relation. This allows queries that are frequently issued on lecturers and their faculties to be answered quickly without the need to join the *Department* relation. Given a query  $Q_3 = \{\text{Engineering COUNT Department}\}$ , SQAk will join the relations *Lecturer*, *Department* and *Faculty* and return incorrect number of departments in the Engineering faculty as it does not handle unnormalized relations.

In order to generate SQL statements correctly for keyword queries involving aggregates, we need to determine if relations are unnormalized. This can be done by examining the functional dependencies that hold on the relations.

Consider the unnormalized relation *Enrolment* in Figure 8 that is obtained by joining the *Student*, *Enrol* and *Course* relations in Figure 1. The following functional dependencies hold on the *Enrolment* relation:

- $Sid \rightarrow Sname, Age$
- $Code \rightarrow Title, Credit$
- $Sid, Code \rightarrow Grade$

We deduce that  $\{Sid, Code\}$  is the key of the *Enrolment* relation, and it violates the second normal form (2NF) definition as *Sname* and *Age* only depend on *Sid*.

Enrolment						
Sid	Sname	Age	Code	Title	Credit	Grade
s1	George	22	c1	Java	5.0	A
s1	George	22	c2	Database	4.0	B
s1	George	22	c3	Multimedia	3.0	B
s2	Green	24	c1	Java	5.0	A
s3	Green	21	c1	Java	5.0	A
s3	Green	21	c3	Multimedia	3.0	B

Figure 8: An unnormalized relation

A naive approach to handle a keyword query involving aggregate functions on the unnormalized database is to generate a copy of the database where every relation is normalized and then process the query as described in Section 3. However, this approach is expensive and not feasible in practice.

We observe that although the relations are unnormalized, the information of objects and relationships in the database remain the same. Hence, if we can keep track of the objects and relationships information in an unnormalized database, then we can continue to process keyword queries involving aggregates and *GROUPBY* correctly.

Recall that the ORM schema graph captures the information of objects and relationships in the database by classifying the relations into different types. These relations are assumed to be in 3NF. Thus, we generate a *normalized view* of the unnormalized database comprising of a minimal set of relations in 3NF. Then we classify the relations in this normalized view and construct the ORM schema graph to represent the ORA semantics in the unnormalized database.

Let  $D = \{R_1, R_2, \dots, R_j\}$  be the set of relations in the original unnormalized database schema, and  $D'$  be the set of relations in the normalized view. For each  $R_i \in D$ ,  $1 \leq i \leq j$ , if  $R_i$  is in 3NF, then we add it to  $D'$ . Otherwise, we normalize  $R_i$  into a set of relations in 3NF and add them to  $D'$ . Finally, relations in  $D'$  with the same key are merged. We use relational algebra operators to express the mappings of the relations from  $D$  to  $D'$ , and vice versa.

**EXAMPLE 8.** Let us generate the normalized view of the unnormalized database in Figure 8. The database consists of a single relation *Enrolment* and has the schema  $D$  below:

$Enrolment(Sid, Code, Sname, Age, Title, Credit, Grade)$

Since the *Enrolment* relation is not in 3NF, we decompose it into 3NF relations *Student'*, *Enrol'* and *Course'*. Thus, the normalized view  $D'$  will have the relations:

$Student'(Sid, Sname, Age)$

$Enrol'(Sid, Code, Grade)$

$Course'(Code, Title, Credit)$

Based on  $D'$ , we construct the ORM schema graph of the unnormalized database as shown in Figure 9. Table 1 shows the mappings of the relations in  $D$  and  $D'$ .  $\square$

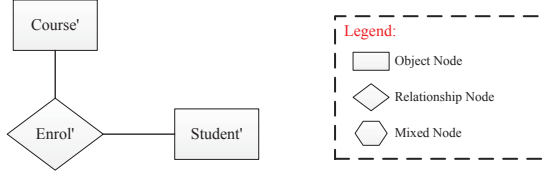


Figure 9: ORM schema graph of Figure 8

Table 1: Mappings of relations in Example 8

$Student' = \Pi_{Sid, Sname, Age}(Enrolment)$
$Enrol' = \Pi_{Sid, Code, Grade}(Enrolment)$
$Course' = \Pi_{Code, Title, Credit}(Enrolment)$

(a) From original schema  $D$  to normalized view  $D'$

$$Enrolment = Student' \bowtie Enrol' \bowtie Course'$$

(b) From normalized view  $D'$  to original schema  $D$

After obtaining the normalized view  $D'$  and the ORM schema graph  $G$  of the unnormalized database with schema  $D$ , we can proceed to evaluate an aggregate query  $Q$  on  $D$  correctly as follows:

First, we identify the matches of each basic term in the unnormalized database. Let  $R$  be the relation in  $D$  such that a basic term  $t$  matches the relation name of  $R$ , or the name of an attribute in  $R$ , or the value of some tuples in  $R$ . We obtain the corresponding relations of  $R$  in  $D'$  based on the mappings from  $D$  to  $D'$ .

Next, we utilize the relations in  $D'$  to generate the query patterns based on  $G$ , and annotate these patterns with the operators in the query as described in Section 3. Note that the generated query patterns are based on the normalized view  $D'$ , since  $G$  is constructed from  $D'$ .

Finally, we translate the annotated query patterns into SQL statements to be executed over the original unnormalized database. This requires us to map the relations in  $D'$  back to their corresponding relations in  $D$ . Depending on the mappings, a relation  $R'$  that corresponds to a node in the query pattern may become a subquery in SQL.

**EXAMPLE 9.** Consider query  $Q_4 = \{Green\ George\ COUNT\ Code\}$  on the unnormalized database in Figure 8. The terms *Green* and *George* match the *Sname* attribute values of some tuples in the *Enrolment* relation, while the term *Code* matches the name of an attribute in *Enrolment*.

Based on Table 1(a), these matches correspond to 2 *Student'* relations and 1 *Course'* relation in the normalized view of the database respectively, indicating that *Green* and *George* refer to two student objects, while *Code* refers to a course object. Based on the ORM schema graph in Figure 9, we generate a query pattern that connects 2 *Student'* nodes and 1 *Course'* node via 2 *Enrol'* nodes, and annotate it with operators *COUNT* and *GROUPBY*.

Figure 10 shows the query pattern obtained. It depicts the query interpretation to find the total number of courses taken by the student *George* and each student called *Green*. We use the mappings in Table 1(b) to translate the query pattern and obtain an SQL statement with 5 subqueries, namely, 2 subqueries for *Student'* relation, 2 subqueries for *Enrol'* relation, and 1 subquery for *Course'* relation in the normalized view of the database.

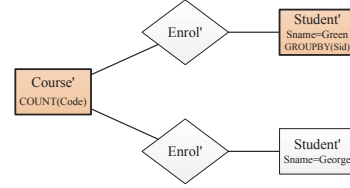


Figure 10: Query pattern in Example 9

```
SELECT S1'.Sid, COUNT(C'.Code) AS numCode FROM
(SELECT DISTINCT Code, Title, Credit FROM Enrolment) C',
(SELECT Sid, Code, Grade FROM Enrolment) E1',
(SELECT DISTINCT Sid, Sname, Age FROM Enrolment) S1',
(SELECT Sid, Code, Grade FROM Enrolment) E2',
(SELECT DISTINCT Sid, Sname, Age FROM Enrolment) S2'
WHERE C'.Code=E1'.Code AND C'.Code=E2'.Code
AND S1'.Sid=E1'.Sid AND S1'.Sname contains 'Green'
AND S2'.Sid=E2'.Sid AND S2'.Sname contains 'George'
GROUP BY S1'.Sid
```

## 4.1 Query Rewriting

The generated SQL statement may contain a lot of subqueries since the mapping from a relation  $R' \in D'$  to a relation  $R \in D$  for a node in  $P$  often involves a subset of the attributes of  $R$ . Joining relations obtained from subqueries is time consuming due to the lack of indexes. Hence, it is crucial to rewrite the SQL to improve query performance.

We observe that some attributes in the SELECT clause of subqueries are never used, and can be removed. In Example 9, we can rewrite the subquery “SELECT DISTINCT Code, Title, Credit FROM Enrolment” to “SELECT DISTINCT Code FROM Enrolment”, since Title and Credit are not used.

Further, some select conditions in the SQL statement can be moved to the WHERE clause of subqueries so that tuples can be filtered out before the join, e.g., we can rewrite the subquery “SELECT DISTINCT Sid, Sname, Age FROM Enrolment” to “SELECT DISTINCT Sid, Sname, Age FROM Enrolment WHERE Sname contains 'Green'” to filter out the students whose names are not Green.

Finally, relations are unnormalized to reduce the number of joins. We can try to use the unnormalized relation to replace the joining of relations obtained from subqueries. For example, the *Enrolment* relation is equivalent to the joins of relations obtained from the subqueries “SELECT DISTINCT Code, Title, Credit FROM Enrolment”, “SELECT Sid, Code, Grade FROM Enrolment”, and “SELECT DISTINCT Sid, Sname, Age FROM Enrolment”. Hence, we can use the *Enrolment* relation to replace these subqueries.

Based on the above observations, We derive the following heuristics to rewrite an SQL statement  $sql$  for the unnormalized database:

**Rule 1:** If a subquery projects an attribute that does not appear in the SELECT and WHERE clause of  $sql$ , then remove this attribute.

**Rule 2:** If a subquery projects an attribute  $a$  that appears in the condition “ $a$  contains  $t$ ” of  $sql$ , then put this condition in the WHERE clause of the subquery.

**Rule 3:** Let  $s_1, s_2, \dots, s_m$  be a set of subqueries in  $sql$ . If there exists a relation  $R$  such that  $s_1 \bowtie s_2 \bowtie \dots \bowtie s_m = \Pi_L(R)$ , where  $L$  is a superkey of  $R$ , then replace  $s_1 \bowtie s_2 \bowtie \dots \bowtie s_m$  with  $R$ .

EXAMPLE 10. Consider the SQL statement in Example 9. Since the joins of the subqueries “SELECT DISTINCT Code, Title, Credit FROM Enrolment”, “SELECT Sid, Code, Grade FROM Enrolment”, and “SELECT DISTINCT Sid, Sname, Age FROM Enrolment” is equivalent to the Enrolment relation, we replace  $C' \bowtie E1' \bowtie S1'$  by the Enrolment relation. Further, we see that the joins of the subqueries “SELECT Sid, Code, Grade FROM Enrolment” and “SELECT DISTINCT Sid, Sname, Age FROM Enrolment” is equivalent to a relation obtained by projecting a super key (Sid, Code, Title, Credit, and Grade) of the Enrolment relation. Hence, we can also replace  $E2' \bowtie S2'$  by the Enrolment relation, and obtain:

```
SELECT R1.Sid, COUNT(R1.Code) AS numCode
FROM Enrolment R1, Enrolment R2
WHERE R1.Code=R2.Code AND R1.Sname contains 'Green'
AND R2.Sname contains 'George'
GROUP BY R1.Sid
```

## 5. ALGORITHMS

Algorithm 1 generates a normalized view  $D'$  of the database schema  $D$ , if  $D$  is not normalized. For each relation  $R$  in  $D$ , if  $R$  is in 3NF, we add it into  $D'$  directly, otherwise we normalize  $R$  into a set of 3NF relations and add them into  $D'$ . After checking the relations in  $D$ , we enumerate each pair of relations  $R_1'$  and  $R_2'$  in  $D'$ . If  $R_1'$  and  $R_2'$  have the same key, then we merge them into a single relation  $R'$ .

After generating the database schema  $D$  and the normalized view  $D'$ , we obtain the mappings between  $D$  and  $D'$ . We call Algorithm 2 to process a keyword query  $Q$  and generate SQL statements. If the schema  $D$  is normalized, we construct the ORM schema graph  $G$  based on  $D$ . For each basic term  $t$  in  $Q$ , we find its matches in the database and create a tag for each of the matches to capture the interpretation of  $t$ . We insert these tags into a tag set  $taglist$  (Lines 4-7). Based on  $taglist$  and  $G$ , we generate a list of query patterns  $ptnlist$  as described in [15], and annotate these patterns with the operators in  $Q$  (Lines 8-9). Then we translate each pattern  $P$  in  $ptnlist$  into an SQL statement  $sql$  according to  $D$ , and insert it into  $sqllist$  (Lines 10-12).

If the schema  $D$  is unnormalized, we construct the ORM schema graph  $G$  based on  $D'$ . For each basic term  $t$ , we create tags for  $t$  based on the matches in  $D$  and the mappings in  $D'$ . We generate a list of query patterns  $ptnlist$  based on the tags and the ORM schema graph, and annotate each pattern  $P$  in  $ptnlist$  with the operators. Then we translate each pattern  $P$  into an SQL statement  $sql$  based on  $D'$ , and map the relations of  $D'$  back to the relations of  $D$  in  $sql$ . Finally, we rewrite  $sql$  to  $sql'$  to reduce the number of subqueries and insert  $sql'$  into  $sqllist$  (Lines 14-26).

Algorithm 3 annotates the query patterns. For each pattern  $P$  in  $ptnlist$ , we annotate  $P$  with operators. For each operator  $t$  in  $Q$ , let  $t'$  be the next term of  $t$  in  $Q$ . If  $t'$  is a basic term, we check its matches in  $D$ . Let  $u$  be a node in  $P$  and  $R$  be the relation of  $u$ . If  $t'$  matches the name of  $R$ , then we annotate  $u$  with  $t(R.key)$ ; otherwise if  $t'$  matches an attribute  $a$  of  $R$ , we annotate  $u$  with  $t(R.a)$ . If  $t'$  is also an operator, then we annotate pattern  $P$  with  $t(t')$  to indicate that  $t$  is a nested aggregate function (Lines 3-12). Next, we check the annotated nodes in the patterns. For each pattern  $P$  in  $ptnlist$ , we create a set  $S$  and add  $P$  into  $S$ . For each object/mixed node  $u$  in  $P$ , if  $u$  is annotated with the condition  $a = t$ , we find a set of tuples  $T$  that satisfy this

---

### Algorithm 1: NormalizeDB

---

```
Input: database schema  $D$ 
Output: normalized view  $D'$ 
1  $D' \leftarrow \emptyset$ ;
2 foreach relation  $R$  in  $D$  do
3   if  $R$  is in 3NF then
4     Add  $R$  into  $D'$ ;
5   else
6     Normalize  $R$  into a set of 3NF relations  $F$ ;
7     foreach relation  $R'$  in  $F$  do
8       Add  $R'$  into  $D'$ ;
9   foreach pair of relations  $R_1'$  and  $R_2'$  in  $D'$  do
10    if  $R_1'.key = R_2'.key$  then
11      Merge  $R_1'$  and  $R_2'$  into  $R'$ ;
12 return  $D'$ ;
```

---



---

### Algorithm 2: Keyword Search

---

```
Input: Query  $Q$ , database schema  $D$ , normalized view  $D'$ 
Output: a list of SQL statements  $sqllist$ 
1  $sqllist \leftarrow \emptyset$ ;  $ptnlist \leftarrow \emptyset$ ;  $taglist \leftarrow \emptyset$ ;
2 if  $D$  is normalized then
3    $G = createORMGraph(D)$ ;
4   foreach basic term  $t$  in  $Q$  do
5     matches = findMatch( $t, D$ );
6     tagset = createTag(matches,  $G$ );
7     Insert tagset into taglist;
8    $ptnlist = createPattern(taglist, G)$ ;
9    $ptnlist = annotatePattern(Q, ptnlist)$ ;
10  foreach Pattern  $P$  in  $ptnlist$  do
11     $sql = translate(P, D)$ ;
12    Insert  $sql$  into  $sqllist$ ;
13 else
14   $G = createORMGraph(D')$ ;
15  foreach basic term  $t$  in  $Q$  do
16    matches = findMatch( $t, D$ );
17    Map matches of  $D$  into matches' of  $D'$ ;
18    tagset = createTag(matches',  $G$ );
19    Insert tagset into taglist;
20   $ptnlist = createPattern(taglist, G)$ ;
21   $ptnlist = annotatePattern(Q, ptnlist)$ ;
22  foreach Pattern  $P$  in  $ptnlist$  do
23     $sql = translate(P, D')$ ;
24    Map the relations of  $D'$  to the relations of  $D$  in  $sql$ ;
25     $sql' = rewrite(sql)$ ;
26    Insert  $sql'$  into  $sqllist$ ;
27 return  $sqllist$ ;
```

---

condition in the relation of  $u$ . If  $T$  contains more than one tuple, we generate new copies of patterns in  $S$  to distinguish the objects that satisfy the same condition. For each pattern  $P$  in  $S$ , we create a copy  $P'$  of  $P$ , annotate node  $u$  in  $P'$  with GROUPBY( $R.key$ ), and add  $P'$  into  $S$ . Finally, we add the patterns in  $S$  into the list  $aptnlist$  (Lines 13-24).

## 6. PERFORMANCE STUDY

In this section, we evaluate the performance of our approach to process keyword queries involving aggregates and GROUPBY. We implement the algorithms in Java and carry out experiments on a 3.40 GHz CPU with 8 GB RAM. We use the relational databases TPC-H (TPCH) and ACM Digital Library publication (ACMDL). Table 2 shows the schemas of these databases. Tables 3 and 4 show the queries we constructed for each database and the corresponding descriptions (or search intentions).

---

**Algorithm 3:** Annotate Pattern

---

**Input:** query  $Q$  and a list of patterns  $ptnlist$   
**Output:** a list of annotated patterns  $aptnlist$

```
1  $aptnlist \leftarrow \emptyset$ ;  
2 foreach Pattern  $P$  in  $ptnlist$  do  
3   foreach operator  $t$  in  $Q$  do  
4     Let  $t'$  be the next term of  $t$  in  $Q$ ;  
5     if  $t'$  is a basic term then  
6       Let  $u$  be a node in  $P$  and  $R$  be the relation of  $u$ ;  
7       if  $t'$  matches the name of  $R$  then  
8         Annotate  $u$  with  $t(R.key)$ ;  
9       else if  $t'$  matches an attribute  $a$  in  $R$  then  
10        Annotate  $u$  with  $t(R.a)$ ;  
11       else if  $t'$  is an operator then  
12        Annotate  $P$  with  $t(t')$ ;  
13 foreach Pattern  $P$  in  $ptnlist$  do  
14    $S = \{P\}$ ;  
15   foreach Object/Mixed node  $u$  in  $P$  do  
16     if  $u$  is annotated with condition  $a = t$  then  
17       Let  $R$  be the relation of  $u$  and  $T$  be the tuples  
18       satisfying  $a = t$  in  $R$ ;  
19       if  $|T| > 1$  then  
20         foreach Pattern  $P$  in  $S$  do  
21           Create a copy  $P'$  of  $P$ ;  
22           Annotate  $u$  in  $P'$  with  
23           GROUPBY( $R.key$ );  
24           Add  $P'$  into  $S$ ;  
25   Add the patterns in  $S$  into  $aptnlist$ ;  
26 return  $aptnlist$ ;
```

---

## 6.1 Effectiveness Experiments

Our approach utilizes the ORM schema graph to capture the ORA semantics in the database, and generates a list of annotated query patterns from the ORM schema graph to represent the various interpretations of a keyword query. Based on these patterns, we distinguish the objects with the same value and detect duplicate objects in relationships in order to compute the answers correctly.

We compare our approach with SQAK [13], the state-of-the-art relational keyword search engine that processes aggregate queries without considering the ORA semantics.

SQAK takes an aggregate query and finds a set of relations that are matched by query terms. A relation is matched if a term matches the name of the relation, or the name of one of its attributes, or the relation tuples. Based on these relations, it generates a set of minimal connected graphs called simple query networks (SQN). The SQNs are used to generate the SQL statements to return the answers.

### 6.1.1 Results for TPCB Database

We use the generated SQL statements that best match the query descriptions in Table 3 to compute the query answers. Table 5 shows the results returned by SQAK and our approach, as well as explanations for these answers. Although both SQAK and our approach give the same answer for queries  $T1$  and  $T2$ , they differ greatly for the rest.

Queries  $T3$  and  $T4$  show that our approach is able to distinguish the various interpretations of query terms that match objects with the same value. For query  $T3$ , our approach returns the number of orders for each “royal olive” part, while SQAK returns the number of orders for all the “royal olive” parts. This is because we differentiate parts with the same name by their object identifiers `partkey`. Similarly, for  $T4$ , our approach returns the maximum account

**Table 2: Database schemas**

TPCH
Part( <u>partkey</u> , pname, type, size, retailprice)
Supplier( <u>suppkey</u> , sname, nationkey, acctbal)
Lineitem( <u>partkey</u> , <u>suppkey</u> , <u>orderkey</u> , quantity)
Order( <u>orderkey</u> , <u>custkey</u> , amount, date, priority)
Customer( <u>custkey</u> , cname, nationkey, mktsegment)
Nation( <u>nationkey</u> , nname, regionkey)
Region( <u>regionkey</u> , rname)

ACMDL
Paper( <u>paperid</u> , procid, date, ptitle)
Author( <u>authorid</u> , fname, lname)
Editor( <u>editorid</u> , fname, lname)
Proceeding( <u>procid</u> , acronym, title, date, pages, publisherid)
Publisher( <u>publisherid</u> , code, name)
Write( <u>paperid</u> , <u>authorid</u> )
Edit( <u>editorid</u> , <u>procid</u> )

balance of suppliers for each “yellow tomato” part, whereas SQAK returns the maximum account balance among all the suppliers that supply a “yellow tomato”. Note that our approach can also generate query patterns to compute aggregates without distinguishing objects with the same value.

Queries  $T5$  and  $T6$  show that by examining the relationships and their participating objects, our approach is able to detect duplicate objects in relationships and generate SQL statements that compute the aggregates correctly. For query  $T5$ , our approach returns 4 for the number of suppliers that supply “Indian black chocolate”. SQAK counts the same suppliers multiple times for different orders and returns 22, a value that is way above the actual number. Similarly for  $T6$ , our approach detects the duplicates of suppliers for different orders, and returns the correct number of parts supplied by each supplier, while SQAK returns incorrect answers.

Queries  $T7$  and  $T8$  demonstrate that our approach can answer aggregate queries that SQAK does not handle. Query  $T7$  requires an SQL statement that contains 2 aggregate functions in the SELECT clause. However, SQAK restricts that the SELECT clause of a generated SQL statement specifies exactly one aggregate function. Query  $T8$  requires an SQL statement to join 2 *Part* relations, but SQAK does not generate SQL statements that contain self joins of relations.

### 6.1.2 Results for ACMDL Database

Table 6 shows the answers and explanations for the queries on the ACMDL database. Query  $A1$  is relatively straightforward, and both our approach and SQAK return the correct answer. For  $A2$ , SQAK also gives the correct answer because the term SIGMOD matches a proceeding acronym and there is no proceedings with the same acronym.

However, for queries  $A3$  and  $A4$ , there are 61 editors with name Smith and 36 authors with name Gill in the database. Since SQAK does not distinguish the editors and authors with the same name, it returns incorrect number of proceedings and the most recent date of papers respectively.

Similarly, for query  $A5$ , our approach returns 6 answers while SQAK only returns 4 answers, as it mixes some papers with the same title.

Query  $A6$  involves 2 aggregate functions. Queries  $A6$  and  $A7$  require self joins of two *Author* relations and two *Editor* relations respectively. SQAK is unable to process these queries, while our approach returns the correct answers.

**Table 3: Queries for TPCB database**

#	Query	Description
T1	order AVG amount	Find the average amount of orders
T2	MAX COUNT order GROUPBY nation	Find the maximum number of orders among nations
T3	COUNT order “royal olive”	Find the number of orders that contains the “royal olive”
T4	supplier MAX acctbal “yellow tomato”	Find the maximum balance of suppliers that supply the “yellow tomato”
T5	COUNT supplier “Indian black chocolate”	Find the number of suppliers for “Indian black chocolate”
T6	COUNT part GROUPBY supplier	Find the number of parts supplied by each supplier
T7	COUNT order SUM amount GROUPBY mktsegment	Find the number of orders and their total amount for each market segment
T8	COUNT supplier “pink rose” “white rose”	Find the number of suppliers for “pink rose” and “white rose”

**Table 4: Queries for ACMDL database**

#	Query	Description
A1	proceeding AVG pages	Find the average pages of proceedings
A2	COUNT paper GROUPBY proceeding SIGMOD	Find the number of papers in each ‘SIGMOD’ proceeding
A3	COUNT proceeding editor Smith	Find the number of proceedings edited by ‘Smith’
A4	paper MAX date Gill	Find the date of the latest papers written by ‘Gill’
A5	COUNT author “database tuning”	Find the number of authors for each “database tuning” paper
A6	COUNT paper MAX date IEEE	Find the number of papers published by ‘IEEE’ and most recent date
A7	COUNT paper author John Mary	Find the number of papers co-authored by ‘John’ and ‘Mary’
A8	COUNT editor SIGIR CIKM	Find the number of editors that edit proceedings ‘SIGIR’ and ‘CIKM’

### 6.1.3 Queries on Unnormalized Databases

Next, we denormalize the ACMDL and TPCB databases, and obtain the unnormalized database schemas in Table 7. We use the queries in Tables 3 and 4 on the unnormalized databases and compare the results returned by SQAK and our approach.

Tables 8 and 9 show that our approach continues to return correct answers to the queries. In contrast, SQAK either returns incorrect answers or does not handle the queries. For queries T1 and T2, SQAK returns the values  $1.78 \times 10^5$  and 26485 respectively because the information of orders are duplicated in the unnormalized relation *Ordering*. Similarly, SQAK returns the answer 637 for A1, and 2000, 408, 14858, etc. (totally 36 answers) for A2, both of which are incorrect as the information of proceedings and papers are duplicated in the unnormalized relations *EditorProceeding* and *PaperAuthor*. Note that these queries are answered correctly by SQAK when the database is normalized.

For queries T3 to T6 and queries A3 to A5, SQAK returns the incorrect answers for the same reason as discussed in Section 6.1.1 and Section 6.1.2.

This set of experiments clearly demonstrate that the ORA semantics are important to distinguish the various interpretations of keyword queries so that the generated SQL statements will compute statistical information correctly.

## 6.2 Efficiency Experiments

Finally, we compare the time taken by our approach and SQAK to generate SQL statements. Figure 11 shows the results for TPCB and ACMDL queries in Tables 3 and 4.

We observe that our approach is slightly slower than SQAK for most of the queries. This is because SQAK does not analyze the interpretations of keyword queries but only finds SQNs containing all the query terms. It also does not distinguish objects with the same attribute value or detect the duplicate objects in relationships. Besides, it does not consider the duplications arising from unnormalized relations.

Take query A7 for example. We first parse this query into basic terms (paper, author, John, Mary) and operators (COUNT). Then, we generate a query pattern with one Paper

**Table 7: Unnormalized database schemas**

TPCB'
Ordering(partkey, suppkkey, orderkey, pname, type, size, retailprice, sname, nationkey, regionkey, acctbal, custkey, amount, date, priority, quantity)
Customer(custkey, cname, nationkey, regionkey, mktsegment)
Nation(nationkey, nname)
Region(regionkey, rname)
ACMDL'
PaperAuthor(paperid, authorid, procid, date, title, fname, lname)
EditorProceeding(editorid, procid, fname, lname, acronym, title, date, pages, publisherid)
Publisher(publisherid, code, name)

node, two Write nodes and two Author nodes. We annotate the Paper node with the Count operator, and distinguish the authors called John and the authors called Mary respectively. Finally, we detect if information of paper and author objects are duplicated in write relationships, and translate the patterns into SQL statements. In contrast, SQAK does not handle the query because both the terms John and Mary match the values of some tuples in the *Author* relation.

As the SQL execution time dominates the overall processing time (in seconds), we see that the extra time (in ms) required by our approach to interpret the keyword queries and detect the duplicates is a good tradeoff and important to retrieve correct answers from the databases.

## 7. RELATED WORK

Existing works on keyword search in relational databases can be classified into data graph approach and schema graph approach. In data graph approach, the relational database is modeled as a graph where each node represents a tuple and each edge represents a foreign key-key reference. BANKS [8] defines an answer to a keyword query as a Steiner tree that contains all the keywords, and proposes a backward expansion search to find the Steiner trees. [9] uses bidirectional

**Table 5: Answers of queries for normalized TPCB database**

#	SQAk		Our Proposed Approach	
	Answer	Explanation	Answer	Explanation
T1	AVG amount: $1.42 \times 10^5$	average amount of orders	AVG amount: $1.42 \times 10^5$	average amount of orders
T2	MAX COUNT order: 6568	maximum number of orders among nations	MAX COUNT order: 6568	maximum number of orders among nations
T3	1 answer: 229	incorrect answer: mix all "royal olive" parts	8 answers: 23, 22, 29, 27, 33, 35, 33, 27	number of orders for each "royal olive" part
T4	1 answer: 9844.00	incorrect answer: mix all "yellow tomato" parts	13 answers: 6361.20, 9538.15, ..., 7916.56	maximum account balance of suppliers for each "yellow tomato" part
T5	COUNT supplier: 22	incorrect answer: same suppliers are counted multiple times for various orders	COUNT supplier: 4	number of suppliers that supply "Indian black chocolate"
T6	1000 answers: 593, 571, 595, 606, ...	incorrect answers: same parts are counted multiple times for various orders	1000 answers: 80, 80, 79, 80, ...	number of parts supplied for each supplier
T7	N.A.	do not handle more than one aggregate	5 answers: $(2.99 \times 10^4, 4.26 \times 10^9), \dots$ $(3.03 \times 10^4, 4.33 \times 10^9)$	one answer for each market segment
T8	N.A.	do not handle self joins of relations	3 answers: 1, 1, 1	number of suppliers that supply a particular "pink rose" and a particular "white rose"

**Table 6: Answers of queries for normalized ACMDL database**

#	SQAk		Our Proposed Approach	
	Answer	Explanation	Answer	Explanation
A1	AVG ages: 297	average pages of proceedings	AVG ages: 297	average pages of proceedings
A2	36 answers: 84, 84, 82, ...	number of papers for each 'SIG-MOD' proceeding	36 answers: 84, 84, 82, ...	number of papers for each 'SIG-MOD' proceeding
A3	1 answer: 62	incorrect answer: mix all editors named 'Smith'	61 answers: 1, 1, 2, ...	number of proceedings edited by each editor named 'Smith'
A4	1 answer: 2011-06-13	incorrect answer: mix all authors named 'Gill'	36 answers: 1994-05-01, 1998-08-01, ...	most recent date of papers written by each author named 'Gill'
A5	4 answers: 2, 4, 6, 4	incorrect answers: mix papers with the same title	6 answers: 2, 2, 2, 6, 2, 2	number of authors for each "database tuning" paper
A6	N.A.	do not handle more than one aggregate	4 answers: (4011, 2011-01-25), ...	number of papers published by 'IEEE' and their most recent date
A7	N.A.	do not handle self joins of relations	46 answers: 1, 32, 8, 1, ...	number of papers co-authored by a particular author 'John' and a particular author 'Mary'
A8	N.A.	do not handle self joins of relations	2 answers: 1, 1	number of editors that edit a 'SIGIR' and a 'CIKM' proceeding

**Table 8: Query answers on unnormalized TPCB (Our approach returns the same answer as Table 5)**

#	SQAk	Explanation
T1	AVG amount: $1.78 \times 10^5$	incorrect answer: count duplicate orders
T2	MAX COUNT order: 26485	incorrect answer: count duplicate orders
T3	1 answer: 229	incorrect answers: The same reason as Table 5
T4	1 answer: 9844.00	
T5	COUNT supplier: 22	
T6	1000 answers: 593, 571, ...	
T7	N.A.	
T8	N.A.	

**Table 9: Query answers on unnormalized ACMDL (Our approach returns the same answer as Table 6)**

#	SQAk	Explanation
A1	AVG ages: 637	incorrect answers: count duplicate proceedings
A2	36 answers: 2000, 408, 14858, ...	incorrect answers: count duplicate papers
A3	1 answer: 62	incorrect answers: The same reason as Table 6
A4	1 answer: 2011-06-13	
A5	4 answers: 2, 4, 6, 4	
A6	N.A.	
A7	N.A.	
A8	N.A.	

expansion to reduce the search space. [4] employs dynamic programming to identify the top-k minimal group Steiner trees. [12] finds a subgraph that contains all the keywords within a given distance to be a query answer, and captures more information than a Steiner tree.

In schema graph approach, the database schema is modeled as a graph where each node represents a relation and each edge represents a foreign key - key constraint. DISCOVER [7] proposes a breadth-first traverse on the schema graph to generate a set of SQL statements. Each SQL joins

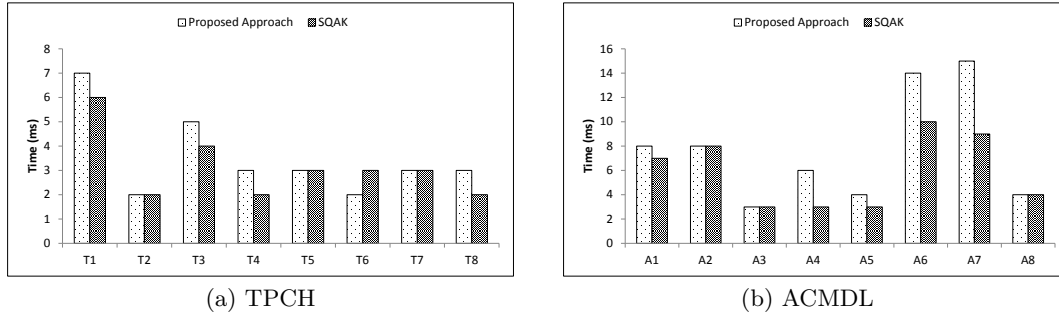


Figure 11: Comparison of the time taken by our approach and SQAk to generate SQL statements

a minimal number of relations and outputs tuples that contain all the keywords. [6] and [10] relax the requirement that output tuples should contain all the query keywords, and develop top-k keyword query techniques to improve efficiency of [7]. [1] exploits the relative positions of keywords in a query and auxiliary external knowledge to generate SQL statements that satisfy users' search intention.

The above works only examine tuples that contain query keywords and try to link them by foreign key-key references. [17] studies the problem of aggregate keyword search on a universal relation. Given a keyword query, it finds a set of tuples that are grouped by a minimal number of attributes and contain all the keywords. [11] classifies query keywords into dimensional and general keywords, and computes subgraphs that contain all dimensional keywords and some general keywords. These subgraphs are grouped based on dimensional keywords to compute the statistical information of the subgraphs. However, none of these works can answer queries involving aggregate functions and GROUPBY.

SQAk [13] generates a set of SQL statements from a keyword query containing reserved keywords to indicate the aggregate functions in SQL statements. But, it does not consider the ORA semantics, and thus returns incorrect answers as we have highlighted. Moreover, SQAk cannot handle queries when relations in the database are unnormalized.

## 8. CONCLUSION

In this paper, we have studied the problem of answering keyword queries involving aggregates and GROUPBY on relational databases. Existing work does not consider the ORA semantics, and thus fails to distinguish objects with the same attribute value and detect duplications of objects in relationships. This leads to incorrect computation of aggregate queries. To avoid these problems, we utilize the ORM schema graph to capture the ORA semantics, and propose a semantic approach to answer aggregate queries. Given an aggregate query, we generate a set of annotated query patterns to represent various interpretations of the query. Based on these patterns, we distinguish objects with the same attribute value and detect duplications of objects in relationships. The top-k ranked patterns are translated into SQL statements which apply aggregate functions to compute the statistical information correctly. Further, we develop a mechanism to detect duplications arising from unnormalized relations, and extend our approach to handle aggregate queries on unnormalized databases. Experimental results demonstrate that our approach returns correct answers to aggregate queries both on normalized and unnormalized databases.

## 9. REFERENCES

- [1] S. Bergamaschi, E. Domnori, F. Guerra, R. Trillo Lado, and Y. Velegrakis. Keyword search over relational databases: a metadata approach. In *SIGMOD*, 2011.
- [2] J. Coffman and A. C. Weaver. A framework for evaluating database keyword search strategies. In *CIKM*, 2010.
- [3] J. Coffman and A. C. Weaver. Learning to rank results in relational keyword search. In *CIKM*, 2011.
- [4] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, 2007.
- [5] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, 2007.
- [6] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, 2003.
- [7] V. Hristidis and Y. Papakonstantinou. DISCOVER: keyword search in relational databases. In *VLDB*, 2002.
- [8] A. Hulgeri and C. Nakhe. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.
- [9] V. Kacholia, S. Pandit, and S. Chakrabarti. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.
- [10] Y. Luo, X. Lin, W. Wang, and X. Zhou. SPARK: top-k keyword query in relational databases. In *SIGMOD*, 2007.
- [11] L. Qin, J. X. Yu, and L. Chang. Computing structural statistics by keywords in databases. In *ICDE*, 2011.
- [12] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Querying communities in relational databases. In *ICDE*, 2009.
- [13] S. Tata and G. M. Lohman. SQAk: Doing more with keywords. In *SIGMOD*, 2008.
- [14] X. Yu and H. Shi. CI-Rank: Ranking keyword search results based on collective importance. In *ICDE*, 2012.
- [15] Z. Zeng, Z. Bao, T. N. Le, M. L. Lee, and W. T. Ling. Expressq: Identifying keyword context and search target in relational keyword queries. In *CIKM*, 2014.
- [16] Z. Zeng, Z. Bao, M. L. Lee, and T. W. Ling. A semantic approach to keyword search over relational databases. In *ER*, 2013.
- [17] B. Zhou and J. Pei. Answering aggregate keyword queries on relational databases using minimal group-bys. In *EDBT*, 2009.



# Finding All Maximal Cliques in Very Large Social Networks

Alessio Conte<sup>1</sup>, Roberto De Virgilio<sup>2</sup>, Antonio Maccioni<sup>2</sup>,  
Maurizio Patrignani<sup>2</sup>, Riccardo Torlone<sup>2</sup>

<sup>1</sup>Università di Pisa, Pisa, Italy  
conte@di.unipi.it

<sup>2</sup>Università Roma Tre, Rome, Italy  
{dvr, maccioni, patrignani, torlone}@dia.uniroma3.it

## ABSTRACT

The detection of communities in social networks is a challenging task. A rigorous way to model communities considers maximal cliques, that is, maximal subgraphs in which each pair of nodes is connected by an edge. State-of-the-art strategies for finding maximal cliques in very large networks decompose the network in blocks and then perform a distributed computation. These approaches exhibit a trade-off between efficiency and completeness: decreasing the size of the blocks has been shown to improve efficiency but some cliques may remain undetected since high-degree nodes, also called hubs, may not fit with all their neighborhood into a small block. In this paper, we present a distributed approach that, by suitably handling hub nodes, is able to detect maximal cliques in large networks meeting both completeness and efficiency. The approach relies on a two-level decomposition process. The first level aims at recursively identifying and isolating tractable portions of the network. The second level further decomposes the tractable portions into small blocks. We demonstrate that this process is able to correctly detect all maximal cliques, provided that the sparsity of the network is bounded, as it is the case of real-world social networks. An extensive campaign of experiments confirms the effectiveness, efficiency, and scalability of our solution and shows that, if hub nodes were neglected, significant cliques would be undetected.

## Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous; G.2.3 [Discrete Mathematics]: Applications—*Maximal clique enumeration*

## Keywords

Community detection, maximal clique enumeration, scale-free networks

## 1. INTRODUCTION

The detection of groups of densely connected nodes, usually called communities, is used to reveal fundamental properties of networks in a variety of domains such as sociology, bibliography, and biology [13, 18, 29]. A rigorous way to model communities considers maximal cliques, that is, maximal subgraphs in which any pair of nodes is connected by an edge. Maximal clique enumeration (MCE) is a paradigmatic problem in computer science and, due to its known complexity, several solutions have been proposed to deal with real-world scenarios [6, 14, 16, 33, 34].

When very large networks are involved, state-of-the-art strategies consist of decomposing the network into blocks that are independently processed in a distributed and parallel environment [8, 10, 14, 20, 31, 36, 38]. A crucial aspect of this approach is the choice of the size  $m$  of the blocks. Clearly,  $m$  is bounded by the dimension of the memory, but it has been shown that artificially reducing  $m$  to values as low as 1/100 or 1/1000 of the available memory results in a more efficient computation [8, 9, 10]. On the other hand, if the size of the blocks is too small, the effectiveness of the approach is compromised. In fact, consider a node  $n$  such that the graph induced by its neighborhood does not fit into a block. We call such a node *hub*. In any block of the decomposition a portion of the neighborhood of  $n$  will be necessarily omitted and, consequently, some maximal cliques involving  $n$  may remain undetected and some non-maximal cliques could be erroneously found.

Hence, while fixing the size of the blocks, state-of-the-art decomposition approaches also need to find a trade-off between efficiency and effectiveness. Even if efficiency is not an issue, effectiveness can be jeopardized since real-world social networks often contain nodes whose degree (i.e., the number of incident edges) is so high that their neighborhood does not fit into main memory altogether.

Actually, high degree nodes are connatural in *scale-free networks*, where the degree distribution of the nodes follows a power law. This property implies that the number of nodes with  $h$  connections to other nodes decreases exponentially as  $h$  increases and that the set of nodes with arbitrary high degree is not empty [2]. Several works in literature show that social networks, such as Facebook and Twitter, are scale-free [12, 35]. It has also been shown that scale-freeness is exhibited whenever the network has a growth mechanism based on preferential attachment [3, 11], that is, when new connections are distributed among nodes according to how many connections they already have. Hence, as social networks grow, this property is expected to be exacerbated.

In this paper, we address these limitations by proposing an approach to the problem of maximal clique enumeration in very large social networks that meets both the requirements of completeness and efficiency. The approach leverages on sparsity, another property of real-world networks, which basically means that the network can have very dense areas but, overall, nodes and edges are of the same order of magnitude.

Our solution is based on a two-level decomposition of the network. The first-level decomposition aims at recursively identifying and isolating tractable portions involving non-hub nodes only. Intuitively, this operation allows us to “brake” hub nodes by progressively decreasing their degree. The second-level decomposition suitably splits tractable portions of the network into small blocks that can be handled separately. Within a block, we then select the most promising state-of-the-art algorithm for enumerating its maximal cliques. A suitable procedure allows us to recognize and filter out those that are not maximal for the overall network. We formally show that this process is able to correctly detect *all* maximal cliques, provided that the sparsity of the network is bounded, as it is the case of real-world social networks.

We have performed a large number of experiments over data from real-world social networks showing that our approach is effective, efficient, and scalable. The experimentation confirms that in order to have an efficient computation it is convenient to choose a relatively small size of the blocks, which further increases the number of hub nodes. The experiments also confirm that, if hub nodes were neglected, significant cliques would remain undetected.

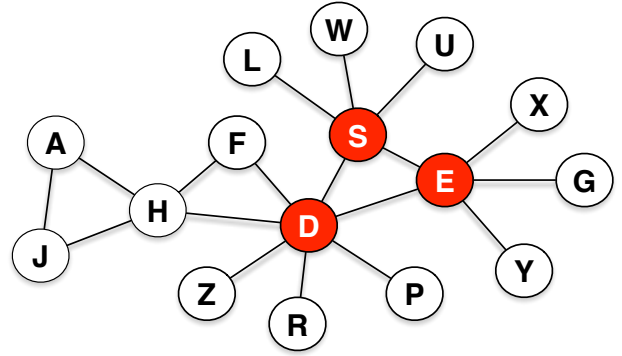
Summarizing, the contributions of this paper are the following.

- We propose a distributed approach to maximal clique enumeration in large social networks based on a novel decomposition strategy that, by suitably handling high-degree nodes, is able to progressively identify and isolate tractable portions of the network;
- We formally prove the correctness and the completeness of the approach;
- We provide experimental evidence of the efficiency and scalability of our solution and show that, if high-degree nodes were neglected, significant cliques would be undetected.

The rest of the paper is organized as follows. In Section 2 we provide a general overview of our technique. Section 3 describes in depth the two-level decomposition algorithm, Section 4 describes the computation of the maximal clique on a single block of the decomposition, and Section 5 provides the theoretical basis for the whole approach. In Section 6 we illustrate our campaign of experiments. Section 7 surveys the related work and Section 8 contains our conclusions.

## 2. OVERVIEW AND INTUITION

Our approach is based on a decomposition of the input network in smaller subgraphs called blocks that can partially overlap with each other. As we have discussed in the Introduction, this requires a careful choice of the size of the blocks, depending on hardware limitations and performance



**Figure 1: Feasible nodes (white) and hub nodes (red) when  $m = 5$ .**

issues. Whichever the choice, let  $m$  be the maximum number of nodes that can fit in a block. The value of  $m$  identifies two types of nodes in the network: (i) the set  $N_h$  of *hub nodes* having degree greater or equal than  $m$  (i.e., those nodes that would not fit into a block with all their neighbors) and (ii) the set  $N_f$  of *feasible nodes* having degree less than  $m$ .

Consider, for example, the network in Figure 1 and suppose  $m = 5$ . The set  $N_h$  consists of the red-coloured nodes D, S, and E of degree 7, 5, and 5 respectively, whereas  $N_f$  consists of the remaining white nodes.

Now, let  $C_f$  be the set of all maximal cliques of  $G$  involving at least one node in  $N_f$  and let  $C_h$  be the set of all maximal cliques in the network  $G_h$  induced<sup>1</sup> by the nodes in  $N_h$ . For example, in the network in Figure 1 we have that  $C_f$  includes the cliques  $\{A, J, H\}$  and  $\{H, F, D\}$ , as they both involve feasible nodes, while  $C_h$  includes the clique  $\{D, S, E\}$ , since  $G_h$  consists only of the nodes D, S, E and of the edges between them.

Our approach is based on the intuition that the set of all maximal cliques of the network  $G$  can be obtained from  $C_f$  and  $C_h$  alone. This is confirmed by Lemma 1 in Section 5, which establishes that the set of the maximal cliques of  $G$  is the union of  $C_f$  and the set  $C'_h$  obtained by filtering out from  $C_h$  any clique that is contained into a clique of  $C_f$ .

This result suggests that if we process separately the nodes in  $N_f$  and the nodes in  $N_h$ , no clique is left out. We then obtain an effective decomposition strategy which is also efficient since the neighbors of a feasible node fit into a block of size  $m$  by definition, while the degree of the nodes in the induced graph  $G_h$  is strongly reduced since  $G_h$  only involves a limited number of nodes in scale free networks. For instance, in the network of Figure 1,  $G_h$  is the cycle  $\{D, S, E\}$  and its maximum degree is two.

Regarding the computation of the cliques in  $C_f$  and  $C_h$  we proceed as follows.

$C_f$ : As in [10], we compute a suitable partition of  $N_f$  and add to each set  $S$  of the partition the neighborhood in  $G$  of the nodes in  $S$ . The obtained sets of nodes, together with the edges between them, form the blocks of the decomposition. Observe that a node (including

<sup>1</sup>We recall that the subgraph of  $G = (N, E)$  induced by a set of nodes  $N' \subseteq N$  is the restriction of  $G$  to the nodes in  $N'$  and the edges between them.

the hub ones) may be included into several blocks as a neighbor node, together with a subset of its edges. Differently from [10], we allow for blocks of heterogeneous size and high connectivity that can be processed independently in an efficient way. Then, taking advantage of a decision tree, we apply on each block the most promising MCE algorithm based on the block characteristics. For instance, if the block is sparse, we find the maximal cliques with the algorithm in [17], while if the block is dense we adopt the algorithm described in [34].

$C_h$ : We apply the whole approach recursively to  $G_h$  by partitioning its nodes  $N_h$  into two sets  $N'_f$  and  $N'_h$  of feasible and hub nodes, respectively. This is possible since the degree of the nodes in  $N_h$  is strongly reduced. The recursion produces a sequence of sets  $N'_f, N''_f, N'''_f, \dots$  of decreasing size until there are no more hub nodes remaining.

In Section 5 we prove that, under the hypothesis that the input graph is sparse enough, this recursive process converges, in the sense that it ends with a bipartition involving only tractable nodes. In addition, in Section 6, we report that in all our experiments on real-world data sets the process needed at most a few recursive steps.

Summarizing, our approach consists of the following.

1. **First level decomposition:** we identify the set  $N_f$  of *feasible* nodes of  $G$ , whose degree is less than  $m$ , and the set  $N_h$  of *hub* nodes of  $G$ , whose degree is greater or equal than  $m$ .
2. **Recursive call:** if  $N_h$  is not empty, we build the subgraph  $G_h$  of  $G$  induced by the nodes in  $N_h$  and apply recursively the whole process to  $G_h$ .
3. **Second level decomposition:** given a set of feasible nodes  $N_f$  we compute a set of blocks by partitioning  $N_f$  and by adding to each node of a block its neighbors.
4. **Block analysis:** we apply a suitable MCE algorithm to each block generated by the second level decomposition to compute all its maximal cliques. The MCE algorithm is chosen from a collection of alternatives, taken from the literature, based on the properties of the block, as described in Section 4.
5. **Filtering:** the output is obtained by taking the union of the maximal cliques computed in step 4 and those computed in step 2, filtering out redundant cliques.

In the following sections we will describe in more detail the various steps of this strategy.

### 3. NETWORK DECOMPOSITION

Algorithm 1 (FIND-MAX-CLIQUE) describes our recursive procedure for computing maximal cliques. The CUT procedure (line 1) performs the first-level decomposition while the BLOCKS procedure (line 2) performs the second-level decomposition. In this section we describe in detail both of them.

Algorithm BLOCK-ANALYSIS (line 5) is discussed in Section 4. Procedure induced (line 6) accepts as input a graph  $G$  and a subset  $N_h$  of its nodes and computes the subgraph of  $G$  induced by  $N_h$ . Procedure filter (line 7) accepts as input two sets  $C_h$  and  $C_f$  of cliques and outputs all cliques in  $C_h$  that are not contained into some clique of  $C_f$ .

---

#### Algorithm 1: FIND-MAX-CLIQUE: Overall algorithm

---

**Input** : A graph  $G = \langle N, E \rangle$  and a block size  $m$ .  
**Output**: The set  $C$  of the maximal cliques of  $G$ .

```

1  $\langle N_f, N_h \rangle \leftarrow \text{CUT}(G, m);$  /* 1st level decomp. */
2  $\mathcal{B} \leftarrow \text{BLOCKS}(G, N_f, m);$  /* 2nd level decomp. */
3  $C_f \leftarrow \emptyset;$ 
4 foreach  $b \in \mathcal{B}$  do
5    $C_f \leftarrow C_f \cup \text{BLOCK-ANALYSIS}(b);$ 
6  $C_h \leftarrow \text{FIND-MAX-CLIQUE}(\text{induced}(G, N_h), m);$ 
7  $C'_h \leftarrow \text{filter}(C_h, C_f);$ 
8 return  $C_f \cup C'_h;$ 

```

---

#### 3.1 First level decomposition

Algorithm 2 describes the CUT procedure that is responsible of identifying the set  $N_f$  of feasible nodes and the set  $N_h$  of hub nodes. This is done by means of the procedure *isfeasible* (also called by procedure BLOCKS) that takes as input a set of nodes, the graph  $G$  and the maximum block size  $m$  and checks whether the union of the given nodes and all their neighborhoods in  $G$  has less than  $m$  elements. The set  $N_h$  of hub nodes is simply obtained, at line 5, as the difference between the nodes of  $G$  and  $N_f$ .

---

#### Algorithm 2: CUT: First-level decomposition

---

**Input** : A graph  $G = \langle N, E \rangle$  and a block size  $m$ .  
**Output**: The sets  $N_f$  and  $N_h$  of feasible and hub nodes of  $G$ , respectively.

```

1  $N_f \leftarrow \emptyset;$ 
2 foreach  $n \in N$  do
3   if isfeasible( $\{n\}, G, m$ ) then
4      $N_f \leftarrow N_f \cup \{n\};$ 
5  $N_h \leftarrow N - N_f;$ 
6 return  $\langle N_f, N_h \rangle;$ 

```

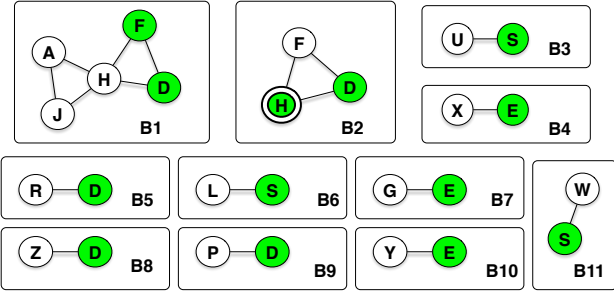
---

#### 3.2 Second level decomposition

Algorithm 3 describes the BLOCKS procedure, responsible of decomposing the input graph  $G$  into tractable blocks of maximum size  $m$ . The input graph  $G$  is assumed to have maximum degree  $m - 1$ . Here, we model blocks similarly to [10] but allow for blocks of heterogeneous sizes and leverage the adjacency of the nodes to put dense subgraphs into the same block. Hence, this step, in addition to distributing the computational load into tasks that could be accomplished separately in a distributed environment, also pre-processes the input producing internally homogeneous and compact chunks.

Blocks are defined sequentially in a greedy way. Each block will have *kernel* nodes, *border* nodes, and *visited* nodes. Each node of  $N_f$  is kernel node in exactly one block (i.e., kernel nodes form a partition of  $N_f$ ). All the nodes of  $G$  that are adjacent to at least one kernel node of a block  $B$  and that are not kernel nodes of  $B$  are divided into border nodes and visited nodes of  $B$ , where visited nodes are those nodes that have been already used as kernel nodes for some previously defined block. The block is completed with all the edges among its nodes, irrespectively of the type.

For instance, consider again the network in Figure 1.



**Figure 2: An example of graph decomposition obtained by focusing on the feasible nodes of the network of Figure 1.**

Nodes D, E, and S are identified as hub nodes by procedure FIND-MAX-CLIQUEs and will be processed in a subsequent recursive call of the same procedure. Figure 2 shows a possible decomposition of the network in eleven blocks obtained by focusing on the remaining non-hub nodes. In the figure kernel nodes are white, border nodes are green and visited nodes are double-marked. Note that all feasible nodes (white-filled in Figure 1) occur in exactly one block as kernel nodes (white-filled in Figure 2). Also, observe that each block of Figure 2 includes all the neighborhood of the kernel nodes. However, the hub nodes (D, E, and S) never occur as kernel nodes in any block. Instead, their neighborhood has been distributed among the various blocks. Finally, note that every maximal clique occurs in at least one block: this is an important property that allows us to independently process each block. If a maximal clique occurs in more than one block, only the occurrence without visited nodes will be considered. This is the case, for instance, for the maximal clique  $\{H, F, D\}$  that is detected both when processing  $B1$  and when processing  $B2$ , but is discarded in the latter case since it contains a visited node.

We start to build a block  $B$  by picking a node  $n$  from  $N_f$  (line 4 of Algorithm 3) and adding it to the set  $K$  of kernel nodes of  $B$  (line 8). We then build: (i) the set  $V$  of visited nodes (line 9), composed of neighbors of nodes in  $K$  that are already used as kernel nodes in a previously defined block (we maintain these latter nodes in  $\bar{K}$ , which is updated at line 7), and (ii) the set  $H$  of border nodes (line 10), composed of neighbors of  $K$  that are not yet visited. Then we proceed by selecting one node of  $N_f$  that is a border node of  $B$  and transforming it into a kernel node of  $B$  (line 10). In order to produce blocks that correspond to dense graphs, we order the candidate border nodes based on the number of their adjacency with kernel nodes, and we stop either if we exceed the limit  $m$  by adding further nodes (line 5) or if all candidate border nodes have a number of adjacency with kernel nodes below a specified threshold.

#### 4. MAXIMAL CLIQUES COMPUTATION

In order to find all maximal cliques in a block of the decomposition, we rely on a framework that leverages on a collection of algorithms taken from the literature with the goal of improving the overall performance of the computation.

The MCE problem has been subject of extensive study since the early 70's [6, 8, 10, 17, 21, 23, 34]. None of the

---

#### Algorithm 3: BLOCKS: Second-level decomposition

---

**Input** : A graph  $G = \langle N, E \rangle$ , a set  $N_f$  of feasible nodes, and a block size  $m$ .

**Output**: A set of blocks  $\mathcal{B}$ .

```

1  $\bar{K} \leftarrow \emptyset; \mathcal{B} \leftarrow \emptyset;$ 
2 while  $N_f \neq \emptyset$  do
3    $K, H, V \leftarrow \emptyset;$ 
4    $n \leftarrow \text{select}(N_f);$ 
5   while  $\text{isfeasible}(K \cup \{n\}, G, m)$  do
6      $N_f \leftarrow N_f - \{n\};$ 
7      $\bar{K} \leftarrow \bar{K} \cup \{n\};$ 
8      $K \leftarrow K \cup \{n\};$ 
9      $V \leftarrow N(n) \cap \bar{K};$ 
10     $H \leftarrow N(n) - V;$ 
11     $n \leftarrow \text{select}(N_f \cap H);$ 
12   $\mathcal{B} \leftarrow \mathcal{B} \cup \text{induced}(G, K \cup H \cup V);$ 
13 return  $\mathcal{B};$ 

```

---

available algorithms outperforms the others in every possible instance of the problem. However, some approaches tend to excel on graphs having specific properties. For example Eppstein *et al.* [17] propose an algorithm that runs in near-optimal time on graphs having small degeneracy<sup>2</sup>. On the contrary, this algorithm does not perform well on dense graphs where the degeneracy tends to be higher. On these graphs, the algorithm proposed by Tomita *et al.* [34] tends to be more efficient.

Our approach attempts at predicting, for each block, the *best-fit* among the available MCE algorithms, that is the one that achieves the best performance on it. The intuition behind this approach is that large heterogeneous networks yield blocks with very different characteristics, so that any algorithm would be suboptimal in a non-negligible portion of the blocks.

In order to efficiently predict the best-fit algorithm for a block, we first identified a set of easy-to-compute parameters to describe the block properties. Second, we selected a set of supporting data-structure and state-of-the-art MCE algorithms. Third, we measured the performance of each combination of data-structure/algorithm on a collection of heterogeneous graphs. Finally, we used the results as a training set to produce a decision tree aimed at selecting the best combination for a given block.

The parameters we used to classify blocks are the following: (a) number of nodes; (b) number of edges; (c) density; (d) degeneracy; and (e) the maximum value  $d^*$  for which the graph has at least  $d^*$  nodes with degree greater or equal than  $d^*$ . Parameter  $d^*$  can be computed in linear time and, intuitively, provides an estimate of the size of the densest portion of the graph, which we expect to dominate the performance of a search algorithm.

We considered three different data structures to represent the graph: adjacency matrices, bitsets, and adjacency lists (the latter including the inverted-table structure described in [17]).

As for the MCE algorithms, we implemented the following:

- **BK Pivot**: one of the original algorithms proposed by

<sup>2</sup>See Section 5 for a formal definition of degeneracy.

Algorithm	Matrix	Lists	BitSets
BKPivot [6]	7	0	2
Tomita [34]	5	3	12
Eppstein [17]	0	2	0
XPivot	7	12	0

**Table 1: Performance of the MCE algorithms.**

Bron and Kerbosch [6]. It uses a pivot to avoid redundant recursive calls. The node of highest degree in the candidate set  $P$  is chosen as the pivot.

- **Tomita**: a variation of BKPivot by Tomita *et al.* [34]. It uses as pivot the node  $u$  that maximizes the size of  $N(u) \cap P$ , where  $N(u)$  denotes the neighborhood of  $u$ .
- **Eppstein**: the algorithm by Eppstein and Strash [17]. It is based on a degeneracy ordering of the nodes to achieve a better complexity on sparse graphs.
- **XPivot**: a variation of BKPivot proposed by us. Like Tomita, it chooses the node that maximizes the size of  $N(u) \cup P$ , but the node  $u$  is chosen from the set of already visited nodes.

In Table 1 we show a performance comparison of the data-structure/algorithm combinations described above on a collection of 50 graphs, both synthetic (generated according to the models of Erdős-Renyi, Barabási-Albert and Watts-Strogatz models [2]) and real-world (taken from the SNAP project [22]). In particular, the table shows how many times a specific combination resulted the best performing among all the alternatives. It is apparent that no algorithm outperforms all the others in all cases.

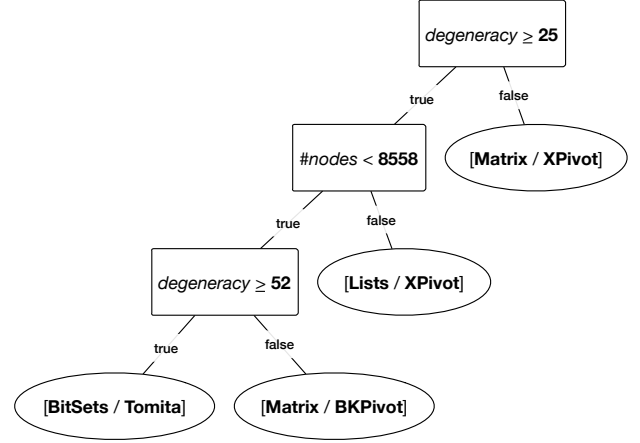
Table 2 shows the maximum and minimum values of the adopted parameters in the collection and confirms that the graphs have heterogeneous properties.

Metric	Min value	Max value
nodes	50	685230
edges	199	6649470
density	0.00027	0.89
degeneracy	10	266
$d^*$	15	713

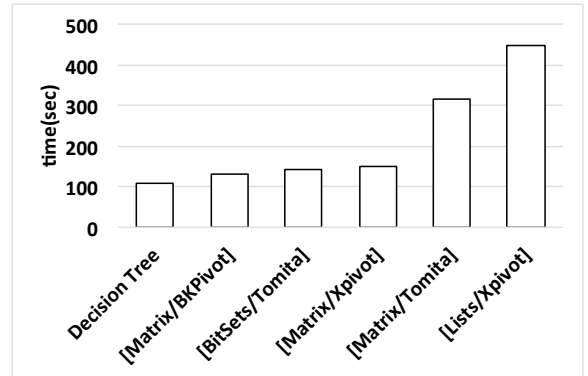
**Table 2: Ranges of adopted parameters for the chosen graphs.**

We divided the graph collection in training and testing set with an 80/20 ratio. We then used the training set and the above parameters to generate the decision tree in Figure 3, launching the *recursive partitioning* algorithm in [32]. Each internal node of the tree contains a predicate on the parameters and has two children, associated with the predicate being **true** or **false** on the current block. Each leaf of the decision tree contains a data-structure/algorithm combination. Traversing the tree from the root to a leaf according to the values of the predicates yields the data-structure/algorithm combination that is the best-fit for the block.

The testing set was used to evaluate the effectiveness of this approach. Figure 4 shows the total time taken by our approach to process the testing set and the five best performing combinations. Note that the use of the decision



**Figure 3: The decision tree for selecting the most suitable MCE algorithm.**



**Figure 4: Times to compute cliques with or without a decision tree.**

tree achieves better performance than any other algorithm taken singularly.

Algorithm 4 describes in detail the BLOCK-ANALYSIS procedure that computes all maximal cliques of the block given as input.

First, a suitable MCE procedure is identified by using the decision tree described above (line 1).

As described in Section 3.2, the purpose of Algorithm 4 is to find all maximal cliques that have at least one node in  $K$ , but no node in the set  $V$  of visited nodes of the input block. In line 3 we initialize  $\bar{V}$  with  $V$ . For each node  $k$  in the set  $K$  of kernel nodes of the input block, Algorithm  $MCE(k, P, \bar{V})$  enumerates all maximal cliques that contain  $k$  and no node in  $\bar{V}$  as long as all the neighbors of  $k$  are in  $P \cup \bar{V}$ . One can observe that all neighbors of  $k$  are either in the set  $H$  of border nodes of the input block, or in  $K$  or in  $\bar{V}$ . Therefore, procedure BLOCK-ANALYSIS detects all maximal cliques containing a node of  $K$  and no node in  $V$ . Finally, after  $k$  is visited, it is added to  $\bar{V}$  since all cliques containing  $k$  have been found.

## 5. THEORETICAL BASIS

In this section we prove under what conditions our approach is correct and complete. Namely, Lemma 1 proves

---

**Algorithm 4: BLOCK-ANALYSIS: Clique detection**


---

**Input** : A block  $\langle N = K \cup H \cup V, E \rangle$ .

**Output**: The maximal cliques  $\mathcal{C}$  of  $B$  that have at least one node in  $K$ , but no node in  $V$ .

```

1 MCE  $\leftarrow$  bestfit( $B$ );
2  $P \leftarrow K \cup H$ ;
3  $\bar{V} \leftarrow V$ ;
4 foreach  $k \in K$  do
5    $N_k \leftarrow N(k) \cap P$ ;
6    $\mathcal{C} \leftarrow \mathcal{C} \cup \text{MCE}(k, P \cap N_k, \bar{V} \cap N_k)$ ;
7    $P \leftarrow P - \{k\}$ ;
8    $\bar{V} \leftarrow \bar{V} \cup \{k\}$ ;
9 return  $\mathcal{C}$ ;

```

---

that FIND-MAX-CLIQUEs (Algorithm 1 in Section 3) actually computes all maximal cliques of the input network. Theorem 1, instead, shows that FIND-MAX-CLIQUEs terminates its recursive calls whenever the input network is sparse.

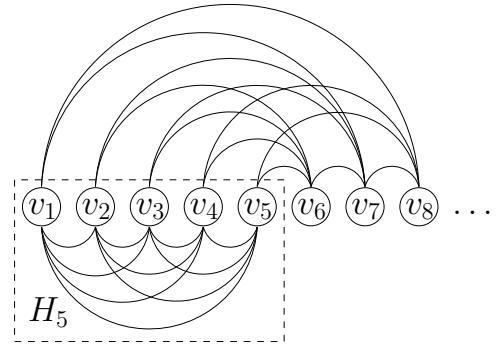
Sparsity is a well known property of social networks and can be formally measured in terms of their low *degeneracy* [35]. The degeneracy of a network, also called *coreness*, is the highest value  $d$  for which the network contains a  $d$ -core<sup>3</sup>. Hence, a network with a low degeneracy is inherently sparse. The degeneracy of a network can be easily computed, even in a distributed environment (see, e.g., [4]), and it is usually much lower than the maximum degree of the network. Indeed, real-world social networks have low degeneracy [35].

**LEMMA 1.** *Let  $N_1$  and  $N_2$  be any bipartition of the nodes of a graph  $G$ . Let  $C_1$  be the set of the maximal cliques of  $G$  containing at least one node of  $N_1$  and let  $C_2$  be the set of maximal cliques of the subgraph of  $G$  induced by the nodes in  $N_2$ . The set of the maximal cliques of  $G$  is the union of  $C_1$  and the set  $C'_2$  obtained by filtering out from  $C_2$  any clique that is contained into a clique of  $C_1$ .*

**PROOF.** Let  $K$  be a maximal clique of the network. We show that  $K$  is in  $C_1 \cup C'_2$ . We have two cases: (i) at least one node of  $K$  is in  $N_1$  or (ii) all nodes of  $K$  are in  $N_2$ . In the first case  $K$  belongs to  $C_1$  and, hence, it is also in the union of  $C_1$  and  $C'_2$ . In the second case  $K$  is in  $C_2$  and, since by hypothesis  $K$  is maximal, it is also in  $C'_2$ , and hence in the union of  $C_1$  and  $C'_2$ .

Conversely, let  $K$  be a clique in the union of  $C_1$  and  $C'_2$ . We show that  $K$  is a maximal clique. Suppose, for a contradiction, that  $K'$  is a clique containing  $K$  and having a vertex  $v$  in addition to the vertices of  $K$ . One of the following three cases applies: (a) at least one node of  $K$  belongs to  $N_1$ ; (b) all nodes of  $K$  belong to  $N_2$  and  $v$  also belongs to  $N_2$ ; or (c) all nodes of  $K$  belong to  $N_2$  and  $v$  belongs to  $N_1$ . In Case (a), both  $K$  and  $K'$  belong to  $C_1$ , contradicting the hypothesis that  $C_1$  is composed of maximal cliques. In Case (b), both  $K$  and  $K'$  belong to  $C_2$ , contradicting the hypothesis that  $C_2$  is composed of maximal cliques. Finally, in Case (c),  $K$  belongs to  $C_2$  while  $K'$  belongs to  $C_1$ . However, since  $K$  is contained into  $K'$ ,  $K$  does not belong to  $C'_2$ , contradicting the hypothesis that  $K$  belongs to the union of  $C_1$  and  $C'_2$ .  $\square$

<sup>3</sup>The  $d$ -core of a graph is obtained by recursively removing nodes with degree less than  $d$ .



**Figure 5: The construction for  $m = 4$  of graph  $H_n$  used to prove Statement 2 of Theorem 1.**

The following theorem shows that, if the network is sparse, the recursive algorithm FIND-MAX-CLIQUEs converges, in the sense that it ends with a bipartition involving only tractable nodes.

**THEOREM 1.** *Let  $G$  be a graph and let  $G_i$ , with  $i = 1, 2, 3, \dots$  be a sequence of subgraphs of  $G$  such that  $G_1 = G$  and  $G_i$ , for  $i > 1$  is the graph induced by the nodes of  $G_{i-1}$  of degree greater or equal than  $m$ . Let the degeneracy  $d$  of  $G$  be strictly less than  $m + 1$ .*

1. *There is a value  $q$  such that all  $G_j$ , with  $j \geq q$ , are empty graphs.*
2. *There exists a graph with  $n$  nodes for which  $q$  is  $\Omega(n)$ .*

**PROOF.** Statement 1 is proved by observing that graphs  $G_i$ , with  $i > 1$ , are obtained from  $G$  by iteratively removing nodes of degree less or equal than  $m$ . For  $i$  large enough, such iterative removal coincides with a recursive removal and, hence, leads by definition to the  $(m + 1)$ -core of  $G$ , which is the empty graph since  $d < m + 1$ .

Statement 2 is proved by producing a graph  $H_n$  with  $n$  nodes and whose degeneracy is  $d < m + 1$  such that  $q \in \Omega(n)$ , as follows. Start from  $H_1$  composed of the isolated node  $v_1$  and, for  $j = 2, 3, \dots, n$ , obtain  $H_j$  by adding a node  $v_j$  to  $H_{j-1}$ . For  $j \leq m + 1$  connect  $v_j$  to all previously inserted nodes, so that,  $H_j$ , with  $j \leq m + 1$ , is a complete graph on the first  $j$  nodes (see Figure 5 where  $m = 4$ ). For  $j > m + 1$  connect  $v_j$  to the previous  $m$  nodes that have lower degree. It is easy to check that:

- (a)  $v_j$  has degree  $m$  in  $H_j$ , for any  $j > m + 1$ . For example, in Figure 5 node  $v_6$  has degree 4 in  $H_6$ .
- (b)  $v_{j-1}$  has degree  $m + 1$  in  $H_j$ , for any  $j > m + 2$ . For example, in Figure 5 node  $v_6$  has degree 5 in  $H_7$ .
- (c)  $v_1, v_2, \dots, v_{j-2}$  have degree greater than  $m$  in  $H_j$ , for any  $j > m + 3$ . For example, in Figure 5 nodes  $v_1, v_2, \dots, v_6$  have degree greater than 4 in  $H_8$ .

Therefore, for  $j > m + 3$ , the three conditions (a), (b), and (c) hold and the removal of all nodes of degree less or equal than  $m$  from  $H_j$  only removes  $v_j$ , yielding  $H_{j-1}$ . This implies that: (i) recursively removing all nodes of degree less or equal than  $m$  from  $H_n$  yields the empty graph, i.e., the degeneracy of  $H_n$  is less than  $m + 1$  and (ii)  $\Omega(n)$  removals are needed to obtain the empty graph from  $H_n$ .  $\square$

Network	# of nodes	# of edges	Maximum degree
twitter1	2,919,613	12,887,063	39,753
twitter2	6,072,441	117,185,083	338,313
twitter3	17,069,982	476,553,560	2,081,112
facebook	4,601,952	87,610,993	2,621,960
google+	6,308,731	81,700,035	1,098,000

Table 3: The data sets used in the experimentation.

Theorem 1 proves that, in order to guarantee that all maximal cliques are detected, `FIND-MAX-CLIQUEs` only requires that  $m$  is chosen to be greater than  $d - 1$ , where  $d$  is the degeneracy of the network (Section 6 shows how to pick a good value for  $m$ ). We remark that, although the very special graph described in the proof of Theorem 1 requires  $\Omega(n)$  recursive steps, in all our experiments with real-world data sets the process needed at most a few of them (see Section 6).

## 6. EXPERIMENTAL RESULTS

We implemented our approach for maximal clique enumeration into a C++ system using OpenMPI v1.8 library. This section reports the results of the experimentation of the system.

### 6.1 Benchmark Environment

We deployed our system on a 10-nodes time-shared cluster, where each machine is equipped with 8 GB DDR3 RAM, 4 CPUs 2.67 GHz Intel Xeon with 4 cores and 8 threads, running Scientific Linux 5.7, with the TORQUE Resource Manager process scheduler. The system is provided with the Lustre file system v2.1. The performance of our system has been measured with respect to data loading (i.e., decomposition), and all maximal cliques computation time (i.e., block analysis).

For our experiments we used some of the largest available social networks (see Table 3) taken from SNAP [22] and from the Koblenz Konect repositories<sup>4</sup>. In particular we considered three portions of the “follower network” of Twitter (labeled `twitter1`, `twitter2` and `twitter3` in Table 3), the friendship network of Facebook enriched with posts to user’s wall (labeled `facebook`), and “circles” data from Google+ (labeled `google+`). All these data sets are scale-free networks and provide a significant number of hub nodes. Figure 6 shows a truncated degree distribution of all considered data sets: as discussed in the Introduction, all networks follow a power law for which most of the nodes (i.e. 91% of the total, on average) provide a degree included in the range  $[1, 20]$ . Nevertheless, on average, in each data set the amount of possible hub nodes (i.e. they provide the maximum degree) represents the 3% of the total set of nodes.

### 6.2 Network Decomposition

We distributed the input data set among the ten machines of our cluster: each data set is locally split into files whose records contain triples in the format  $\langle n_1, e, n_2 \rangle$ , where  $n_1$  and  $n_2$  are the labels of the nodes and  $e$  is the label of the edge between them. To speed-up the process we encoded node and edge labels with hashes.

<sup>4</sup>Available at <http://konect.uni-koblenz.de/downloads/#rdf>

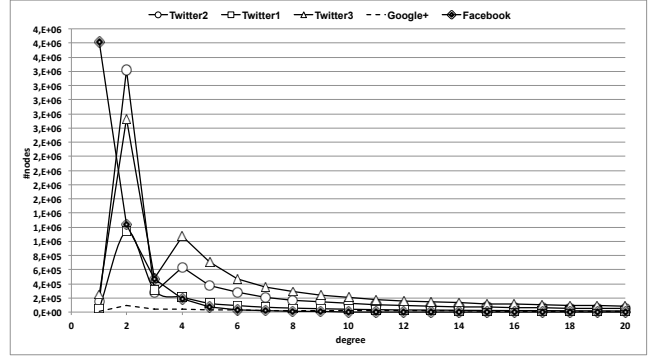


Figure 6: Truncated degree distribution of data sets.

On each data set we ran Algorithm `FIND-MAX-CLIQUEs` three times on each machine and measured the average time used to produce the blocks (including the I/O time). Figure 7 shows for each data set the average time to perform the two-level decomposition with respect to the ratio  $m/d$ , where  $m$  is the maximum number of nodes in a block and  $d$  is the maximum node degree. In the experiment we considered five ratios (i.e. 0.9, 0.7, 0.5, 0.3, and 0.1) obtained by decreasing  $m$ . As the block size limit decreases, the number of blocks increases and consequently it increases also the time to perform the decomposition. It also causes the increase of the number of hub nodes as well as the increase of the number of maximal cliques involving hubs (see Section 6.3).

We remark that for  $m/d \in \{0.5, 0.9\}$  all data sets required two iterations of the first-level decomposition, while for  $m/d \in \{0.1, 0.3\}$  all data sets were decomposed after three iterations. This confirms what formally enunciated in Theorem 1. The results in Figure 7 confirm the feasibility of the approach.

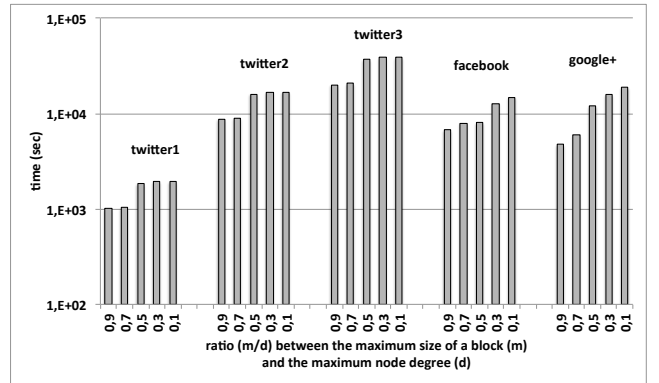


Figure 7: Times to compute the decomposition.

### 6.3 Clique computation

For evaluating the computation times of our approach we ran Algorithm `BLOCK-ANALYSIS` three times on all blocks and measured the average overall time (including the I/O time). Figure 8 shows the average response time in seconds to compute all maximal cliques with respect to the values 0.9, 0.7, 0.5, 0.3, and 0.1 for  $m/d$ . All times refer to a serial pro-

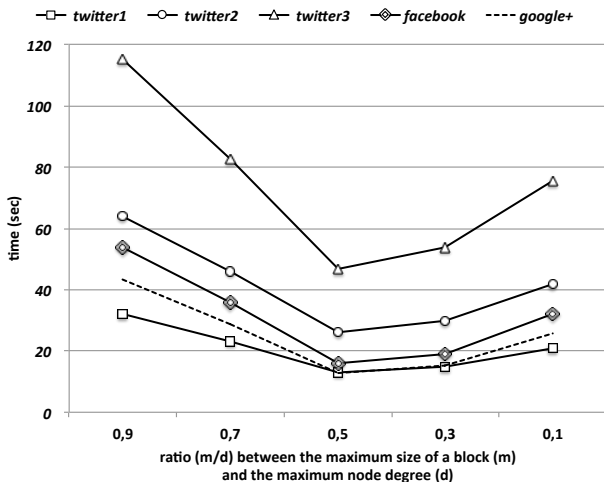


Figure 8: Times to compute all maximal cliques.

cessing (i.e., they do not account for the speed-up due to simultaneous computations on distributed platforms).

**Efficiency.** Our experiments confirm that the running times benefit from relatively small values of  $m$ . The fact that the overall performance is improved when smaller blocks are involved is likely due to the efficiency of the clique detection algorithms on small instances. Hence, it can be argued that the decomposition phase is playing the role of a pre-processing step for the MCE problem, producing blocks that can be regarded as approximate solutions to be refined by an exact MCE algorithm.

For small values of  $m/d$  (i.e., 0.3 and 0.1) we have many blocks and the performance of the entire process are affected by an increasing overlap among the neighborhood of each block and an increasing communication overhead among the machines of the cluster. As shown in Figure 8, the value  $m/d = 0.5$  is a common “saddle point” for all data sets.

**Effectiveness.** Figure 9.(a) and Figure 10.(a) show the number of cliques computed with respect to the same five ratios used above and Figure 9.(b) and Figure 10.(b) show the average size of the cliques. In all the figures, white bars denote maximal cliques computed from the blocks built from the feasible nodes, while gray bars refer to maximal cliques computed from the blocks built from the hub nodes. Figure 9.(a) and Figure 10.(a) clearly show the contribution of our approach: in all the experiments we had a non-negligible number of maximal cliques involving hub nodes only, that could be omitted or could induce the erroneous detection of non-maximal cliques if the techniques described in this paper were not adopted. In particular, as the ratio between the block size and the maximum node degree decreases, the portion of maximal cliques involving only hub nodes is significantly increased (i.e. reducing  $m$  artificially increases the number of hub nodes).

Figure 9.(b) and Figure 10.(b) focus on the size of the produced cliques. It turns out that the sizes of the cliques involving only hub nodes are comparable with (and, in average, greater than) the sizes of the cliques involving feasible nodes. This is more apparent when the ratio  $m/d$  is smaller

(i.e., 0.3 and 0.1). Furthermore, observe that the cliques involving only hub nodes are comparable in size with the biggest cliques contained in the network. Hence, even when the cliques computed on the hub nodes are a small percentage, they are among the most significant when their size is considered.

In order to better estimate how much significant are the maximal cliques composed exclusively of hub nodes, we focused on the 200 largest maximal cliques. Figure 11 shows the percentage of maximal cliques computed on the feasible nodes and the percentage of maximal cliques computed on the hub nodes (with respect to the same five values of  $m/d$  used for Figures 9 and 10). The percentage of maximal cliques computed on the hub nodes grows significantly around the value  $0.5m/d$ . In particular, for values of  $m/d \in [0.1, 0.5]$ , the percentage of maximal cliques computed on hub nodes is between 20% and 80% for all data sets. This confirms that decreasing the block size for boosting efficiency has a dramatic impact on the number of significant maximal cliques that would be lost if the techniques described in this paper were not adopted.

## 7. RELATED WORK

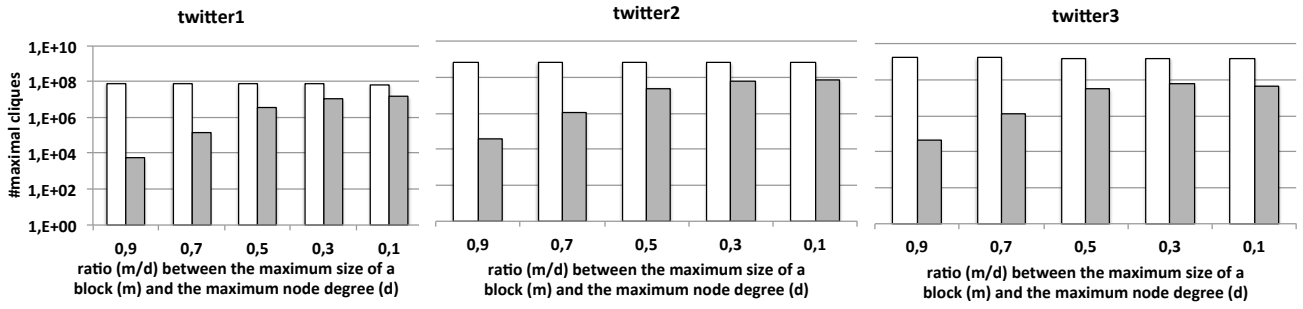
Despite a long research history, the MCE problem has recently re-emerged as one of the key research topics of graph mining. Due to the NP-completeness of the problem, traditional algorithms for enumerating maximal cliques rely on pruning techniques in order to reduce the search space and speed up the execution [27, 33]. With the increasing dimension of nowadays social networks such algorithms are not satisfactory anymore because the size of the input network often exceeds the available memory.

To address this issue, new approaches have been introduced [8, 10, 30, 36, 38, 7]. They usually rely on a decomposition phase that splits the graph into (partially overlapping) blocks and on distributed computation on the independent blocks to detect all the maximal cliques therein.

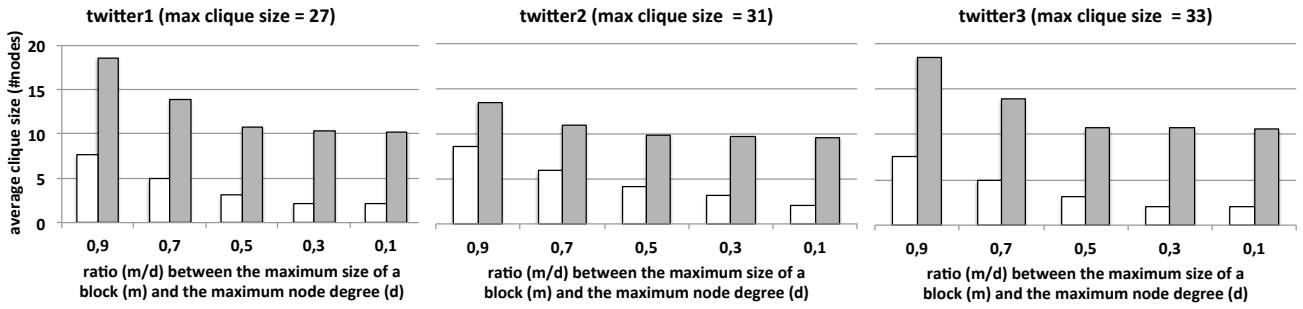
ExtMCE [8, 38] is the first algorithm that handles graphs that do not fit into main memory. It starts the search for maximal cliques from a sub-portion of the whole graph, called  $H^*$ -graph. However, ExtMCE works under the assumption that the  $H^*$ -graph fits into main memory, which may be again too restrictive with real-world networks.

The same authors improved their approach introducing the EmMCE algorithm [10] that takes advantage of parallelization to reduce I/O overhead and to distribute computation loads. As confirmed also by our experimentation, in [10] it is shown that producing blocks of much smaller size than the available memory yield better time performance. At the same time, though, algorithm EmMCE assumes that the neighborhood of each node fits within a block. This clearly poses a trade-off between efficiency and correctness. In fact when the neighborhood of a node does not fit into a single block some of its maximal cliques may be discarded and some non-maximal cliques could be erroneously detected. Furthermore, even if efficiency was not an issue, correctness and completeness are lost whenever the graph has nodes of degree so high that their neighborhood does not fit into main memory. Trying to address this problem in [10] it is suggested to decompose the graph considering nodes in increasing degree order. This results into artificially augmenting the size of a graph fitting into a block, since, when a hub node is chosen as a kernel node, its neighborhood would be



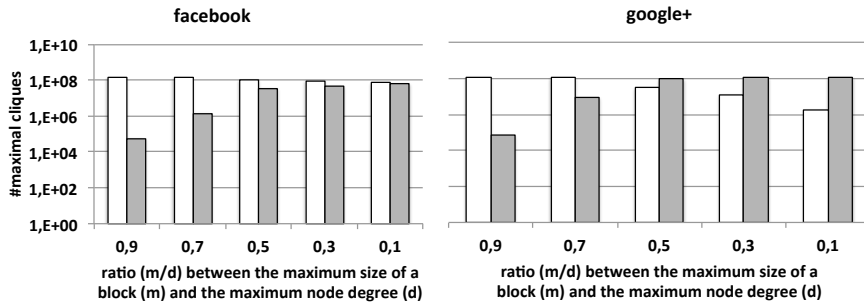


(a) Number of computed cliques

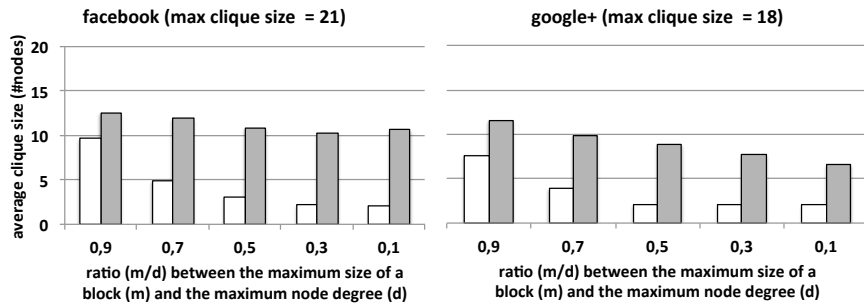


(b) Average number of nodes per clique

Figure 9: The results of the experimentation on twitter1, twitter2, and twitter3 data sets. White bars refer to cliques computed from the feasible nodes while gray bars refer to cliques containing only hub nodes.



(a) Number of computed cliques



(b) Average number of nodes per clique

Figure 10: The results of the experimentation on facebook and google+ data sets. White bars refer to cliques computed from the feasible nodes while gray bars refer to cliques containing only hub nodes.

largely composed by visited nodes, and edges among visited nodes could be omitted from the block. Nevertheless, all neighbors of a hub node need to be stored into the block as before, and the block size limit still hinder correctness.

Chang *et al.* [7] find all maximal cliques in polynomial delay: cliques are found one after the other and the time-complexity of finding the next clique in the sequence is polynomial. They improve over the previously polynomial-delay fastest algorithm for MCE [24] by using a strategy that partitions the graph into low and high degree nodes.

The authors in [38] focus on the skews of the parallel computation of cliques, since the analysis of few blocks takes far more time than the rest. They also propose algorithms that can incrementally update the maximal cliques when the graph is updated.

King *et al.* [36] use a recursive algorithm, called **BMC**, for partitioning the network into blocks. Then, they use an algorithm based on MapReduce to compute the cliques present in each block. Since **BMC** generates blocks having similar size, inter-block cliques are skipped and the approach is not complete. Gregori *et al.* [20] and Rossi *et al.* [30] find the maximal  $k$ -cliques and the largest cliques in a parallel way, respectively. These approaches can not be adapted to find all maximal cliques.

Computations over massive networks often take advantage of distributed graph processing systems such as GraphLab/PowerGraph [19] and Pregel/Giraph [25]. They provide: (a) a fault-tolerant infrastructure for processing distributed data; (b) a graph partitioning technique; and (c) an abstract computational model for implementing algorithms. While we could benefit from the infrastructures and abstract models, the partitioning techniques of such systems (point (b) above) are not suitable in the MCE context. They usually use random partitioning (i.e., hash partitioning) which is proven to be the worst possible partitioning for scale-free networks [15]. Instead, as shown in Section 6, we benefit from a decomposition that produces dense chunks of different size.

Maximal Clique Enumeration is especially used for detecting communities in social networks. Several approaches for community detection, rely on a relaxed concept with respect to the enumeration of maximal cliques and consider each subgraph that approximates a clique as a community [29, 39, 28, 1]. In the remaining part of this section we briefly review some of them.

**WalkTrap** [28] computes random traversals to individuate communities. The heuristic idea of **WalkTrap** is that a random path would likely stay “trapped” inside a subgraph of highly connected nodes. The random path cliques do not give any warranty on the quality of the solutions as, choosing randomly, they might not retrieve a tight community.

There are several approaches that find communities as the subgraphs resulting from the clustering of the edges in the network (see, for example, [1]). They uniquely assign each individual to a cluster. Clearly, this assumption is not suitable for social networks where an individual may belong to multiple communities. To face this aspect, a series of works have been proposed in order to allow overlapping communities (see the survey in [37]).

Differently from all approaches mentioned above, **SCD** [29] employs a parallel strategy to detect the subgraphs that maximize the number of contained triangles, since this measure is indicative of how tight is a community. In [39] it is

introduced the concept of *k-mutual-friend* to find communities and, additionally, it is described a system to browse the communities in a visual manner.

Finally, there are approaches that retrieve communities in terms of *k-plexes*, which are relaxations of cliques in which a node can miss at most  $k$  neighbours [5, 26].

## 8. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a novel technique for computing all the maximal cliques of an arbitrarily large network in a distributed environment. The approach relies on a two-level decomposition strategy that allows us to achieve efficiency by suitably lowering the size of the blocks without jeopardizing completeness. This is confirmed by a number of theoretical results showing the correctness and completeness of the technique over sparse graphs, a natural property of real-world social networks.

An extensive campaign of experiments conducted over real-world scenarios has shown the efficiency and scalability of our proposal. We have also demonstrated experimentally that, if our technique was not adopted, a significant portion of the most relevant cliques would have been lost.

In the future, we plan to explore the possibility of extending our approach to relaxed definitions of communities, such as  $k$ -cliques,  $k$ -clubs,  $k$ -clans, and  $k$ -plexes. We are also interested in studying an incremental version of our approach that takes into account the evolution of the social network.

## Acknowledgements

The authors are grateful to Lorenzo Dolfi and Gabriele De Capoa for their contribution in the development of the tools described in this paper. Research supported in part by the MIUR project AMANDA “Algorithmics for MASSive and Networked DATA”, prot. 2012C4E3KT\_001.

## 9. REFERENCES

- [1] Y.-Y. Ahn, J. P. Bagrow, and S. Lehmann. Link communities reveal multiscale complexity in networks. *Nature*, 466(7307):761–764, 2010.
- [2] R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74:47–97, 2002.
- [3] A.-L. Barabási and E. Bonabeau. Scale-free networks. *Scientific American*, 288(5):50–59, 2003.
- [4] V. Batagelj and M. Zaversnik. An  $o(m)$  algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.
- [5] D. Berlowitz, S. Cohen, and B. Kimelfeld. Efficient enumeration of maximal  $k$ -plexes. In *SIGMOD*, pages 431–444, 2015.
- [6] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.
- [7] L. Chang, J. X. Yu, and L. Qin. Fast maximal cliques enumeration in sparse graphs. *Algorithmica*, 66(1):173–186, 2013.
- [8] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by  $h^*$ -graph. In *SIGMOD*, pages 447–458, 2010.
- [9] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks. *ACM Trans. Database Syst.*, 36(4):21, 2011.

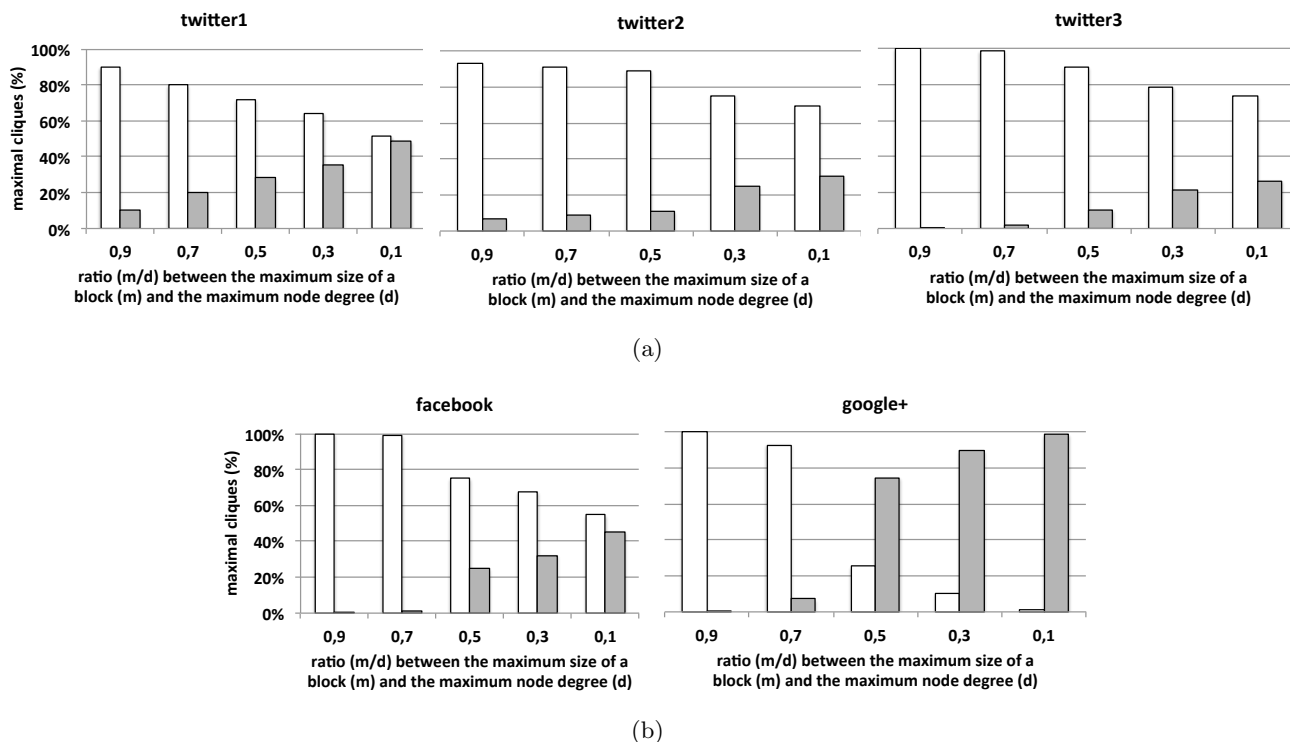


Figure 11: An analysis of the 200 largest maximal cliques of each data set. White bars represent the percentage of these cliques computed from the feasible nodes while gray bars represents the percentage of these cliques containing only hub nodes.

- [10] J. Cheng, L. Zhu, Y. Ke, and S. Chu. Fast algorithms for maximal clique enumeration with limited memory. In *KDD*, pages 1240–1248, 2012.
- [11] K. Choromański, M. Matuszak, and J. Miekisz. Scale-free graph with preferential attachment and evolving internal vertex structure. *Journal of Statistical Physics*, 151(6):1175–1183, 2013.
- [12] A. Cui, Z. Zhang, M. Tang, and Y. Fu. Emergence of scale-free close-knit friendship structure in online social networks. *CoRR*, abs/1205.2583, 2012.
- [13] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *SIGMOD*, pages 991–1002, 2014.
- [14] N. Du, B. Wu, L. Xu, B. Wang, and X. Pei. A parallel algorithm for enumerating all maximal cliques in complex network. In *ICDM Workshops*, pages 320–324, 2006.
- [15] Q. Duong, S. Goel, J. M. Hofman, and S. Vassilvitskii. Sharding social networks. In *WSDM*, pages 223–232, 2013.
- [16] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *ISAAC*, pages 403–414, 2010.
- [17] D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. In *SEA*, pages 364–375, 2011.
- [18] S. Fortunato. Community detection in graphs. *CoRR*, abs/0906.0612, 2009.
- [19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.
- [20] E. Gregori, L. Lenzini, and S. Mainardi. Parallel k-clique community detection on large-scale networks. *Trans. Parallel Distrib. Syst.*, 24(8):1651–1660, 2013.
- [21] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theor. Comput. Sci.*, 250(1-2):1–30, 2001.
- [22] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, 2015.
- [23] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. In T. Hagerup and J. Katajainen, editors, *Algorithm Theory - SWAT 2004*, volume 3111 of *Lecture Notes in Computer Science*, pages 260–272. Springer Berlin Heidelberg, 2004.
- [24] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. In *SWAT*, pages 260–272, 2004.
- [25] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.
- [26] B. McClosky and I. V. Hicks. Combinatorial algorithms for the maximum k-plex problem. *J. Comb. Optim.*, 23(1):29–49, 2012.
- [27] P. R. J. Östergård. A fast algorithm for the maximum

- clique problem. *Discrete Applied Mathematics*, 120(1-3):197–207, 2002.
- [28] P. Pons and M. Latapy. Computing communities in large networks using random walks. *J. Graph Algorithms Appl.*, 10(2):191–218, 2006.
- [29] A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey. High quality, scalable and parallel community detection for large real graphs. In *WWW*, pages 225–236, 2014.
- [30] R. A. Rossi, D. F. Gleich, A. H. Gebremedhin, and M. M. A. Patwary. Fast maximum clique algorithms for large graphs. In *WWW (Companion Volume)*, pages 365–366, 2014.
- [31] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *SIGKDD*, pages 1222–1230, 2012.
- [32] T. M. Therneau and E. J. Atkinson. An introduction to recursive partitioning using the RPART routines. Technical report, Division of Biostatistics 61, Mayo Clinic, 1997.
- [33] E. Tomita and T. Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *J. Global Optimization*, 44(2):311, 2009.
- [34] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, 2006.
- [35] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. *CoRR*, abs/1111.4503, 2011.
- [36] J. Xiang, C. Guo, and A. Abounaga. Scalable maximum clique computation using mapreduce. In *ICDE*, pages 74–85, 2013.
- [37] J. Xie, S. Kelley, and B. K. Szymanski. Overlapping community detection in networks: The state-of-the-art and comparative study. *ACM Comput. Surv.*, 45(4):43, 2013.
- [38] Y. Xu, J. Cheng, A. W. Fu, and Y. Bu. Distributed maximal clique computation. In *International Congress on Big Data*, pages 160–167, 2014.
- [39] F. Zhao and A. K. H. Tung. Large scale cohesive subgraphs discovery for social network visual analysis. *PVLDB*, 6(2):85–96, 2012.

# RPM: Representative Pattern Mining for Efficient Time Series Classification

Xing Wang, Jessica Lin  
George Mason University  
Dept. of Computer Science  
xwang24@gmu.edu,  
jessica@gmu.edu

Pavel Senin  
Bioscience Division, Los  
Alamos National Laboratory,  
Los Alamos, NM, 87544  
psenin@lanl.gov

Tim Oates, Sunil Gandhi  
University of Maryland,  
Baltimore County  
Dept. of Computer Science  
oates@cs.umbc.edu,  
sunilga1@umbc.edu

Arnold P. Boedihardjo

Crystal Chen

Susan Frankenstein

U.S. Army Corps of Engineers, Engineer Research and Development Center  
{arnold.p.boedihardjo, crystal.chen, susan.frankenstein}@usace.army.mil

## ABSTRACT

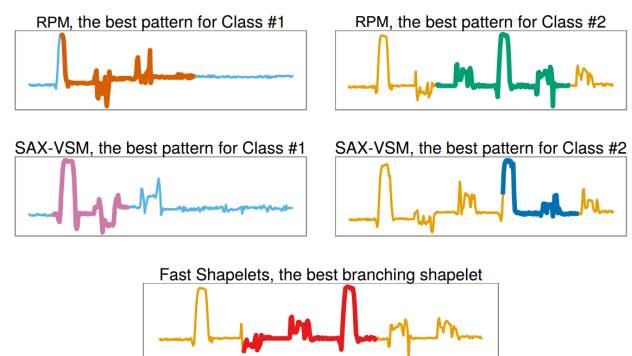
Time series classification is an important problem that has received a great amount of attention by researchers and practitioners in the past two decades. In this work, we propose a novel algorithm for time series classification based on the discovery of class-specific representative patterns. We define representative patterns of a class as a set of subsequences that has the greatest discriminative power to distinguish one class of time series from another. Our approach rests upon two techniques with linear complexity: symbolic discretization of time series, which generalizes the structural patterns, and grammatical inference, which automatically finds recurrent correlated patterns of variable length, producing an initial pool of common patterns shared by many instances in a class. From this pool of candidate patterns, our algorithm selects the most representative patterns that capture the class specificities, and that can be used to effectively discriminate between time series classes. Through an exhaustive experimental evaluation we show that our algorithm is competitive in accuracy and speed with the state-of-the-art classification techniques on the UCR time series repository, robust on shifted data, and demonstrates excellent performance on real-world noisy medical time series.

## 1. INTRODUCTION

Massive amount of time series data are generated daily in areas as diverse as medicine, astronomy, industry, sciences, and finance, to name just a few. Even with the explosion of interest in time series data mining during the past two decades, and increasing popularity of new emerging topics such as motif discovery, classification of time series still remains one of the most important problems with many real-world applications in diverse disciplines.

While many classification algorithms have been proposed for time series, it has been shown that the nearest neighbor classifier, albeit simple in design, is competitive with the more sophisticated algorithms like SVM [32]. As a result, many existing techniques on time series classification focus on improving the similarity measure, an essential part of the nearest neighbor classifier [4]. Recently, the notion of time series shapelets—time series subsequences that are “maximally representative” of a class—has been proposed. Shapelets generalize the lazy nearest neighbor classifier to an eager, decision-tree-like classifier [36][10], which typically improves the classification speed and interpretability of the results.

In this work, we focus on a similar problem of finding the most representative patterns for the classification task. We call our algorithm RPM (Representative Pattern Mining). The key motivation is that the identification of a small set of distinctive and interpretable patterns of each class allows us to exploit their key characteristics for discriminating against other classes. In addition, we hypothesize that the classification procedure based on a set of highly class-characteristic short patterns will provide high generalization performance under noise and/or translation/rotation, i.e. it shall be robust and shift/rotation invariant.



**Figure 1:** An illustration of the best patterns discovered by rival subsequence-based techniques on Cricket data [20].

Our work is significantly different from existing subsequence-based techniques such as *K*-shapelet discovery and

classification algorithms. Specifically, one major difference lies in our definition of *representative patterns*. We define representative patterns to be class-specific prototypes, i.e. each class has its own set of representative patterns, whereas in shapelets some classes may share a shapelet. Figure 1 shows the patterns/shapelets identified by different algorithms on the Cricket dataset [20]. Note that SAX-VSM [31] captures visually similar short patterns of the same length in both classes. Fast Shapelets [27] selects a single subsequence to build a classifier. Our algorithm, RPM, selects different patterns that capture the data specificity (characteristic left and right hand movements) for each class.

The methodology employed by our approach is also very different from existing shapelet-based techniques. Contrasting with a decision-tree-based shapelet classification which finds the best splitting shapelet(s) via the exhaustive candidate elimination and explicit distance computation, we rely on the grammar induction (GI) procedure that *automatically* (i.e. by the algorithm’s design) and *without computing any distance explicitly* [17][30] discovers frequent subsequences (motifs) of *variable length*, which we consider as representative pattern candidates. The number of candidates for the exhaustive shapelet search approach is  $O(nm^2)$  ( $n$  is the number of time series,  $m$  is their length) [27], since the algorithm examines all possible subsequences. For our method, the number of candidates considered is much smaller:  $O(K)$ , where  $K$  is the number of motifs since only patterns that frequently occur in a class can be *representative*.

In addition to speeding up the algorithm, grammar induction, by design, grows the lengths of the initial patterns when constructing grammar rules, thus eliminating the need for searching an optimal pattern length exhaustively – a procedure that is common to most of sliding window-based shapelet techniques [36][10][27].

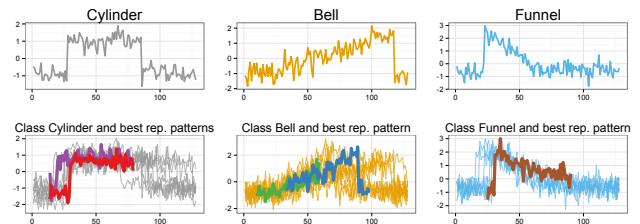
Since GI requires discrete input, our algorithm transforms real-valued time series into discrete values using Symbolic Aggregate approXimation (SAX) [18]. The algorithm operates in the approximate symbolic space inferring a grammar and generating a set of candidate patterns through the analysis of the grammar hierarchy in linear time. Next, the algorithm maps the discovered patterns back into real values and continues with pattern refinement using clustering. The cluster centroids (or medoids) are then reported as the best class-specific motifs, among which we select the most representative ones by verifying their classification power at the final step.

Figures 2 and 3 show examples of representative patterns discovered by our technique in two datasets: CBF, a synthetic dataset [22], and Coffee, a real dataset [2]. Representative patterns discovered in CBF highlight the most distinctive features in each of the three classes: a plateau followed by the sudden rise then followed by a plateau in Cylinder, the increasing ramp followed by a sudden drop in Bell, and a sudden rise by a decreasing ramp in Funnel. Representative patterns discovered in the Coffee dataset also correspond to the most distinctive natural features which not only include the discriminative caffeine and chlorogenic acid bands, but the spectra corresponding to other constituents such as carbohydrates, lipids, etc. [2].

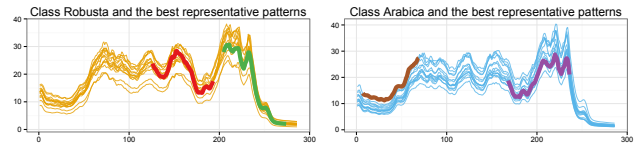
Since we use a different selection criterion, the representative patterns discovered by our technique are different from the shapelets or patterns found by other subsequence-based techniques, thus providing a complementary functionality

that can be used for exploratory studies. For example, the representative patterns discovered in CBF and Coffee datasets by our approach are different from the shapelets discovered by Fast Shapelets [27], a well-known shapelet discovery algorithm which we use for experimental comparison. For the CBF dataset, Fast Shapelets reports two branching shapelets that correspond to sudden rises in Cylinder and Funnel classes. For the Coffee dataset, Fast Shapelets reports a single branching shapelet corresponding to the caffeine spectra band (Arabica).

Note that the discovery of class-specific motifs, which is an integral part of our algorithm, also offers a unique advantage that extends beyond the classification task. Differing from the traditional notion of time series motifs [17][3], which can either be repeated subsequences of a fixed length within a long time series, or repeated time series instances within a group of data (e.g. shape motifs [16]), our class-specific motifs are variable-length sub-patterns that occur frequently in many time series of a data group. They are, in a sense, related to time series subspace clusters [15]. Therefore, our approach provides an efficient mechanism to discover these subspace patterns without exhaustively searching through all subsequences. Throughout the paper, we will use the terms “class-specific subspace motifs” and “class-specific motifs” interchangeably.



**Figure 2:** The Cylinder-Bell-Funnel (CBF) dataset and the best representative patterns for its classes discovered with the proposed technique.



**Figure 3:** Two classes from the Coffee dataset and the best representative patterns.

As we shall demonstrate, in addition to the excellent exploratory characteristics, our approach achieves competitive classification accuracy compared to the state-of-the-art techniques: nearest neighbor classifiers, characteristic subsequence-based classifier (SAX-VSM), and shapelet-based classifiers, while maintaining great efficiency.

The rest of the paper is organized as follows: Section 2 discusses related work and background materials. We describe our approach for finding representative patterns in Section 3, and discuss parameter optimization in Section 4. Section 5 presents experimental results. We demonstrate the utilities of our approach with two case studies in Section 6, and conclude in Section 7.

## 2. RELATED WORK AND BACKGROUND

## 2.1 Notation and definition

To precisely state the problem at hand, and to relate our work to previous research, we will define the key terms used throughout this paper. We begin by defining our data type, time series:

**Time series**  $T = t_1, \dots, t_m$  is a set of scalar observations ordered by time.

Since we focus on finding local patterns that are representative of a class, we consider time series subsequences:

**Subsequence**  $S$  of time series  $T$  is a contiguous sampling  $t_p, \dots, t_{p+n-1}$  of points of length  $n \ll m$  where  $p$  is an arbitrary position, such that  $1 \leq p \leq m - n + 1$ .

Typically subsequences are extracted from a time series with the use of a sliding window:

**Sliding window** subsequence extraction: for a time series  $T$  of length  $m$ , and a user-defined subsequence length  $n$ , all possible subsequences of  $T$  can be found by sliding a window of size  $n$  across  $T$ .

Given two time series subsequences  $S_1$  and  $S_2$ , both of length  $n$ , the distance between them is a real number that accounts for how much these subsequences are different, and the function which outputs this number when given  $S_1$  and  $S_2$  is called the **distance function** and denoted  $Dist(S_1, S_2)$ . One of the most commonly used distance functions is the **Euclidean distance**, which is the square root of the sum of the squared differences between each pair of the corresponding data points in  $S_1$  and  $S_2$ .

**Closest (i.e. best) match**: Given a subsequence  $S$  and a time series  $T$ , the time series subsequence  $T_p$  of length  $|S|$  starting at position  $p : 0 < p < |T| - |S|$  is the closest match of  $S$  if  $Dist(S, T_p) \leq Dist(S, T_k)$ , where  $T_k$  is a subsequence of  $T$  starting at any position  $k : 0 \leq k < |T| - |S|$ , and  $k \neq p$ . The  $Dist(S, T_p)$  is the **closest match distance**.

**Time series pattern** is a subsequence that possesses certain interesting characteristics. For example it can be a subsequence that occurs frequently, i.e. whose observance frequency is above some arbitrary threshold  $t$ . A frequently occurring subsequence is also called **time series motif**.

**Class-specific motif**: Given a class  $C$  and a set of training instances  $X_C$ , a class-specific motif  $M$  for  $C$  is a subsequence pattern  $S$  in  $C$  consisting of a set of similar subsequences from different training instances such that  $count(S) \geq (\gamma \cdot |X_C|)$ , where  $0 < \gamma \leq 1$ . This states that a pattern is considered frequent if it appears in at least  $\gamma$  of the training instances in the class. We will describe what we mean by “similar” in a later section.

**Representative patterns**: The most discriminative subsequences among the class motifs for class  $C$  are selected as the representative patterns for the class. The number of the representative patterns for each class is dynamically determined by the algorithm.

We will describe how to measure the “representativeness” and discriminative power of the candidate patterns in a later section.

**Time Series Transformation**: The set of the closest match distances between a time series  $T$  and the (candidate) representative patterns can be viewed as a transformation of  $T \in \mathbb{R}^{n \cdot m}$  into  $T' \in \mathbb{R}^{n \cdot K}$ , where  $K$  is the total number of the representative patterns from all classes.

## 2.2 Related work

Classification of time series has attracted much interest from the data mining community in the past two decades

[4][36][5][13][21][28][25][33][20][35][34]. Nevertheless, to date, the simple nearest neighbor classification is the most popular choice due to its simplicity and effectiveness [32]. Therefore, a large body of work on time series classification has focused on the nearest neighbor classification improvement by developing new data representations or distance measures [32]. To date, a nearest neighbor classification with an effective distance measure like Dynamic Time Warping (DTW) outperforms many existing techniques [32].

Among the proposed alternatives, many methods focus on finding local patterns in time series as predictive features of the class [5]. Recently, Ye and Keogh introduced a novel concept called time series “shapelet”. A shapelet is an exact time series subsequence that is “maximally representative” of a class [36]. Once found, a shapelet-based technique classifies an unlabeled time series by computing its similarity to the shapelet. The original shapelet technique proposed by the authors constructs a decision tree-based classifier which uses the shapelet similarity as the splitting criterion. While effective and interpretable, the original shapelet discovery technique is computationally intensive.

Numerous improvements were proposed. The Logical Shapelets [20] extends the original work by improving the efficiency and introducing an augmented, more expressive shapelet representation based on conjunctions or disjunctions of shapelets. Fast Shapelets [27] improves the efficiency of the original shapelets algorithm by exploiting the projections into a symbolic representation. Learning Shapelets [7] proposes a new mathematical formalization that iteratively reduces the shapelet search space by computing a classification precision-based objective function.

The “Shapelet Transform” [10] technique finds the best  $K$ -shapelets and transforms the original time series into a vector of  $K$  features, each of which represents the distance between a time series and a shapelet. This technique can thus be used with virtually any classification algorithm.

SAX-VSM [31] is another approximate algorithm that enables the discovery of local class-characteristic (representative) patterns based on the similar pattern-discrimination principle via **tf\*idf**-based patterns ranking [29]. While similar to our notion of representative patterns, the length of SAX-VSM-generated patterns equals to the sliding window length. In addition, the algorithm makes no additional effort to prune the discovered patterns, yielding a large sparse matrix of pattern weights.

Our algorithm can be related to the concept of mining interesting frequent patterns reviewed in [9], as it is essentially the selection of “interesting” pattern subset from a frequent pattern set. In our case, we regard the representative power of a pattern as its interestingness measure.

## 3. RPM: REPRESENTATIVE PATTERN MINING FOR CLASSIFICATION

The classification algorithm we propose consists of two stages: (a) Learning the representative patterns. (b) Classification using the representative patterns.

In the training stage, the algorithm identifies the most representative patterns for each class from the training data by 3 steps: (i) pre-processing training data; (ii) generating representative pattern candidates from processed data; (iii) selecting the most representative patterns from candidates.

Once the representative patterns are learned, we trans-

form the training data into a new feature space, where each representative pattern is a feature and each training time series is represented as a vector of distances to these patterns. We can then build a classifier from the transformed training data.

We will describe the algorithm in detail below, starting with the classification stage.

### 3.1 Time series classification using representative patterns

To classify a time series using the representative patterns learned from the training stage, the first step is to transform the test data  $T \in \mathbb{R}^{n \times m}$  into a feature space representation  $T' \in \mathbb{R}^{n \times K}$  by computing distances from  $T$  to each of the  $K$  representative patterns from all classes. The transformed time series is thus represented as a fixed-length vector – a universal data type which can be used with many of the traditional classification techniques. In this paper, we use SVM [24] for its popularity, but note that our algorithm can work with any classifier.

### 3.2 Training stage: Finding representative patterns

As mentioned previously, our goal is to find the most representative and distinctive patterns for each of the time series classes. These class-specific patterns should satisfy two requirements: (i) they should be class-specific motifs, i.e. shared by at least  $\gamma$  (a fraction) of the time series in the class; and (ii) they should enable the discrimination between the current and other classes – the capacity measured with a scoring function discussed below.

At the high level, the algorithm can be viewed as a succession of three steps. First, the algorithm performs data pre-processing for each class in the training data, by concatenating all instances from the same class into a long time series, and discretizing the concatenated time series. Second, the algorithm finds frequent patterns via grammar inference and *forms the candidate pool* for each class. Third, it refines the candidates pool by *eliminating redundant and non-discriminating patterns*, and outputs patterns that represent the class the best, i.e., the representative patterns.

In our previous work, we proposed GrammarViz (v2.0), which uses time series discretization and grammar inference for time series motif discovery and exploration [17][30]. We observe that the same approach can be leveraged to find class-specific subspace motifs, thus enabling the classification as well.

#### 3.2.1 Step 1: Pre-processing

We prepare the training data for subspace pattern discovery by concatenating all training time series from the same class into a single long time series. We note that the concatenation step is not required for our learning algorithm and can in fact be skipped. The reason for concatenating the training instances is for visualization purpose only, as will be shown in Figure 4.

Next, we discretize the concatenated time series into a sequence of tokens for grammar induction using Symbolic Aggregate approxImation (SAX) [18]. More specifically, we apply SAX to *subsequences* extracted from the concatenated time series of a training class via a sliding window. SAX performs subsequence discretization by first reducing the dimensionality of the subsequence with Piecewise Aggregate

Approximation (PAA) [12]. Towards that end, it divides  $z$ -normalized subsequence into  $w$  equal-sized segments and computes a mean value for each segment. It then maps these values to symbols according to a pre-defined set of breakpoints dividing the distribution space into  $\alpha$  equiprobable regions, where  $\alpha$  is the alphabet size specified by the user. This *subsequence discretization* process outputs an ordered list of SAX words, where each word corresponds to the left-most point of the sliding window. Two parameters affect the SAX transform granularity – the number of PAA segments (PAA size) and the SAX alphabet size.

Since neighboring subsequences extracted via a sliding window share all points except one, they are similar to each other and often have identical SAX representations. To prevent over-counting a pattern, we apply numerosity reduction [17]: if in the course of discretization, the same SAX word occurs more than once consecutively, instead of placing every instance into the resulting string, we record only its first occurrence.

As an example, consider the sequence  $S_0$  where each word (e.g.  $aba$ ) represents a subsequence extracted from the concatenated time series via a sliding window and then discretized with SAX. The subscript following each word denotes the starting position of the corresponding subsequence in the time series.

$S_0 = aba_1 bac_2 bac_3 bac_4 cab_5 acc_6 bac_7 bac_8 cab_9 \dots$

With numerosity reduction,  $S_0$  becomes:

$S_1 = aba_1 bac_2 cab_5 acc_6 bac_7 cab_9 \dots$

Numerosity reduction not only reduces the length of the input for the next step of the algorithm (hence making it more efficient and reducing its space requirements) and simplifies the identification of non-overlapping time series motifs by removing “noise” from overlapping subsequences. Most importantly, numerosity reduction enables the discovery of representative patterns of varying lengths as we show next.

#### 3.2.2 Step 2: Generating representative pattern candidates

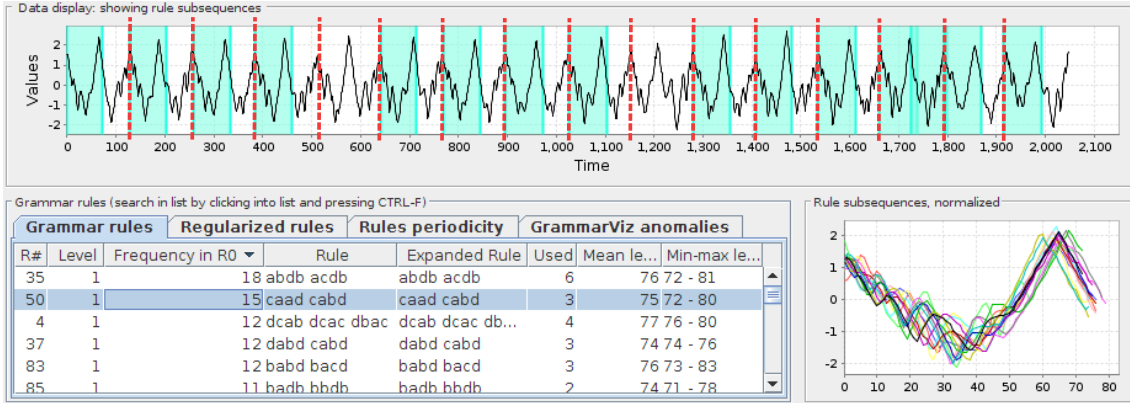
In this step, we use grammar induction to identify repeated patterns in the concatenated time series, and generate a *representative pattern candidates pool* from these patterns. The algorithm is outlined in Algorithm 1. The sequence of SAX words obtained from the pre-processing step is fed into a context-free grammar induction algorithm (Algorithm 1, Line 7). We use Sequitur, a string compression algorithm that infers a context-free grammar in linear time and space [23]. We choose Sequitur due to its efficiency and reasonably good compression capability, but note that our technique also works with other (context-free) GI algorithms. When applied to a sequence of SAX words, Sequitur treats each word as a token and recursively reduces all *digrams*, i.e. consecutive pairs of tokens (terminals or non-terminals), occurring more than once in the input string to a single new non-terminal symbol representing a grammar rule.

Consider the grammar induced by Sequitur from the input string  $S_1$ :

Grammar Rule	Expanded Grammar Rule
$R_0 \rightarrow aba \ R_1 \ acc \ R_1$	$aba_1 \ bac_2 \ cab_5 \ acc_6 \ bac_7 \ cab_9$
$R_1 \rightarrow bac \ cab$	$bac \ cab$

In this grammar,  $R_1$  describes a simplified version of the





**Figure 4:** RPM pre-processing visualization with GrammarViz 2.0 [30]. The time series from Class 4 of SwedishLeaf dataset were concatenated, discretized, Sequitur grammar was inferred, and one of the frequent motifs selected. Note that the grammar rule-corresponding subsequences vary in length from 72 to 80. The length of the original time series before concatenation is 128 as indicated by dotted lines. Note that one of time series does not contain the pattern and another contains it twice.

repeated pattern  $[bac\ cab]$ , which concurrently maps to two substrings of different lengths from  $S_0$ :  $[bac_2\ bac_3\ bac_4\ cab_5]$  (length of 4) and  $[bac_7\ bac_8\ cab_9]$  (length of 3), respectively.

By keeping SAX words’ offsets throughout the procedures of discretization and grammar induction, we are able to map rules and SAX words back to their original time series subsequences. Since each SAX word corresponds to a *single point* of the input time series (a subsequence starting point),  $R1$  maps to subsequences of variable lengths ([2-5] and [7-9]). Figure 4 shows an example of the recurrent subsequences found in the concatenated time series from Class 4 of the Swedish Leaf dataset. Note, that when processing a concatenated time series, the algorithm does not consider the subsequences that span time series junction points in order to avoid concatenation process artifacts.

---

#### Algorithm 1 Finding repeated patterns

---

```

1: function FINDCANDIDATES(Train, SAXParams,  $\gamma$ )
2:   candidates  $\leftarrow \emptyset$ 
3:   allCandidates  $\leftarrow \emptyset$ 
4:   for each TrainClassI in Train do
5:     cTS  $\leftarrow$  CONCATENATETS(TrainClassI)
6:     // {build grammar avoiding junctions (see Fig. 4)}
7:     allRepeats  $\leftarrow$  MODIFIEDGI(cTS, SAXParams)
8:     refinedRepeats  $\leftarrow \emptyset$ 
9:     // {r is repeated subsequences from a row in Fig. 4}
10:    for each r in allRepeats do
11:      clusters  $\leftarrow$  CLUSTERING(r)
12:      refinedRepeats.addAll(clusters)
13:    for each cluster in refinedRepeats do
14:      if cluster.size  $> \gamma \cdot |I|$  then
15:        centroid  $\leftarrow$  GETCENTROID(cluster)
16:        // {candidates for class I}
17:        candidates.add(centroid)
18:      // {candidates for all classes}
19:      allCandidates.add(candidates)
20:  return (allCandidates)

```

---

**Refining repeated subsequences:** Every grammar rule induced from Sequitur describes an *approximate* repeated pattern (our candidate motif) observed in time series; however, corresponding exact subsequences may differ significantly depending on the SAX granularity. To select the most representative of the frequent symbolic patterns (and essen-

tially of the training class), we use hierarchical clustering algorithm (complete-linkage) to cluster all rule-corresponding subsequences based on their similarity (Algorithm 1, Line 11). Note that the purpose of applying clustering here is to handle the situation where a candidate motif found by grammar induction algorithm may contain more than one group of similar subsequences. In this case, we should partition the subsequences into sets of clusters, each of which corresponds to one motif. To determine the appropriate number of clusters, we first set the number of clusters as *two*. If the cluster sizes are drastically different, e.g. one of the clusters contains less than 30% of subsequences from the original group, we do not split the original group. If both clusters contain sufficient numbers of subsequences, we will continue to split them into smaller groups. The partitioning stops when no group can be further split (Algorithm 1, Line 12).

Consistent with Section 3.2 requirement (i), if the size of a cluster is smaller than the specified threshold  $\gamma$ , the cluster is discarded. If a cluster satisfies the minimum size requirement, its centroid is added to the representative patterns candidates pool for a class (Algorithm 1, Line 14-15). Note an alternative is to use the medoid instead of centroid. As outlined in Algorithm 1, this refinement procedure outputs a list of representative pattern *candidates* for all classes.

#### 3.2.3 Step 3: Selecting the most representative patterns

The candidate patterns obtained so far are the frequent patterns that occur in a class. However, some of them may not be class-discriminative if they also occur in other classes. Addressing this issue, we prune the candidate patterns pool with Algorithm 2 whose input is the pool of candidate patterns identified from the previous step, and the entire training dataset. The algorithm outputs the set of class-specific patterns.

**Remove Similar Patterns:** The representative pattern candidates are repeated patterns found by the grammar induction algorithm applied to the discretized time series. Due to the aggregation, some structurally similar subsequences may be mapped to slightly different SAX strings, e.g. differ by one letter. Feeding such patterns to the subsequent step (selecting representative patterns) will slow down the search

---

**Algorithm 2** Find distinctive patterns
 

---

```

1: function FINDDISTINCT(Train, allCandidates)
2:   candidates  $\leftarrow$   $\emptyset$ 
3:    $\tau \leftarrow$  COMPUTETHRESHOLD
4:   // {Remove similarities in allCandidates}
5:   for each c in allCandidates do
6:     isNonSimilar  $\leftarrow$  true
7:     for each cns in candidates do
8:       // {c and cns may have different length}
9:       dist  $\leftarrow$  COMPUTECLOSESTMATCHDIST(c, cns)
10:      if dist <  $\tau$  then
11:        if cns.frequent < c.frequent then
12:          // {The frequency in concatenated TS}
13:          candidates.remove(cns)
14:          candidates.add(c)
15:          isNonSimilar  $\leftarrow$  false
16:          break();
17:      if isNonSimilar then
18:        candidates.add(c)
19:      // {Transform TS into new feature space where each feature
      // is the distance between time series and a candidate}
20:      TransformedTrain  $\leftarrow$  TRANSFORM(Train, candidates)
21:      // {Perform feature selection algorithm on new data}
22:      selectedIndices  $\leftarrow$  FSALG(TransformedTrain)
23:      // {Select patterns according to indices}
24:      patterns  $\leftarrow$  SELECT(candidates, selectedIndices)
25:      return (patterns)
  
```

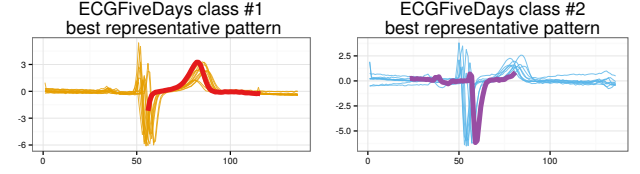
---

when removing correlated patterns in feature selection step.

In order to resolve this issue, our algorithm removes similar candidates from the candidate set, as shown in Algorithm 2, Lines 5 – 18. Before the removal, it computes a threshold used to determine if two patterns are similar. The threshold (Alg. 2, Line 3) is determined as follows: (i) Compute pairwise distances of subsequences within each refined grammar rule (i.e. the final clusters from Algorithm 1). (ii) Sort the distances in ascending order. (iii) Take the distance at the 30-th percentile as the threshold  $\tau$  for similarity. We will show the effect on accuracy and running time with different values of  $\tau$  in the experimental section.

**Select Representative Patterns:** After refining the representative pattern candidates pool from the previous steps, we transform the original time series into a distance feature vector by computing the closest match distance between each time series and all the candidate patterns. As an example, two patterns in dataset ECGFiveDays are shown in Figure 5, and the transformed training data is shown in Figure 6. The original time series from the two classes look visually similar. However, once we transform the raw time series into the two-dimensional feature vector (one feature from each class in this case), it is easy to separate the two classes. As can be seen in Figure 6, the transformed data is linearly separable.

Since the transformation uses all the candidates as new features, a feature selection step is needed to select the most distinctive features. Each feature represents the distance from the original time series to one of the candidate patterns. Thus, the features selected represent the most representative patterns. Any feature selection algorithms can be applied here. In this work, we use the correlation-based feature selection from [8], since it is capable of identifying features that are highly correlated with the class. After feature selection, the selected patterns will be used to classify future time series. Note that the number of selected patterns for each class is dynamically determined by the feature se-



**Figure 5:** Two classes from the ECGFiveDays dataset and the best representative patterns.



**Figure 6:** Transformed data of train data from ECGFiveDays

lection algorithm.

## 4. PARAMETER SELECTION AND CLASSIFICATION WITH DIFFERENT SAX PARAMETERS

As shown in Algorithm 1, there is a parameter vector *SAXParams* in the input. The vector consists of three SAX discretization parameters, namely the sliding window size, PAA size, and the SAX alphabet size. In this section, we shall describe our algorithm for the optimal SAX parameters selection. Since time series data in different classes may have varying characteristics, a parameter set that is optimal for one class may not be optimal for another. Therefore, the parameter optimization is performed for each class.

### 4.1 Search for the best SAX parameters exhaustively

One way to find the optimal parameter set is by brute force grid search – as shown in Algorithm 3. The algorithm tests all parameter combinations within a specified range and selects the optimal set (the one that results in the best F1 measure score from five-fold cross validation on the validation data set). For each parameter combination candidate, the grid search algorithm first divides the original training data into training and validation data 5 times for 5 different splits. For each split, the algorithm invokes Algorithms 1 and 2 to obtain the representative patterns

(Lines 8-9). The validation is performed on the validation data set (Line 12) with a five-fold cross validation. The F1 measure is computed using the classification result for each class. The parameter combination with the best F1 measure for each class is selected as the optimal parameters for the class.

---

**Algorithm 3** SAX Parameter selection (Brute-Force)

---

```

1: function PARAMSELECT(ParamsRange, OriginalTrain)
2:   for each Class I do
3:     bestSoFarForI  $\leftarrow$  0
4:   for each SAXPs in ParamsRange do
5:     // {Repeat 5 times for different splits}
6:     while iteration < 5 do
7:       {train, validate}  $\leftarrow$  SPLIT(OriginalTrain)
8:       candidates  $\leftarrow$  FINDCANDIDATES(train, SAXPs,  $\gamma$ )
9:       patterns  $\leftarrow$  FINDERDISTINCT(train, candidates)
10:      validatet  $\leftarrow$  TRANSFORM(validate, patterns)
11:      // {fMeasure is the f1 measure of each class}
12:      fMeasure  $\leftarrow$  5FOLDSCV(validatet)
13:      for each Class I do
14:        if fMeasure.i > bestSoFarForI then
15:          bestParamForClassI  $\leftarrow$  SAXPs
16:          bestSoFarForI  $\leftarrow$  fMeasure.i
17:      iteration ++
18:   for each Class I do
19:     bestParams.add(bestParamForClassI)
20:   return (bestParams)

```

---

Instead of running Algorithm 3 completely to find the best parameter set, pruning can be performed by the observed number of repeated patterns. In Line 8 of Algorithm 3, if no candidate for a class is returned because all repeated patterns for that class have frequency below the specified threshold ( $\gamma$ ), the algorithm abandons the current iteration and proceeds with the next parameter combination. The intuition rests upon the fact that given the number of time series in the training data, we can estimate the required minimal frequency for the repeated patterns.

## 4.2 Searching for the best SAX parameters using DIRECT

We optimize the search for the best discretization parameters set by using the Dividing RECTangles (DIRECT) algorithm [11] which is a derivative-free optimization scheme that possesses local and global optimization properties, converges quickly, and yields a deterministic optimal solution. DIRECT is designed to deal with optimization problems of the form:

$$\min_x f(x), f \in \mathbf{R}, x, X_L, X_U \in \mathbf{R}, \text{ where } X_L \leq x \leq X_U$$

where  $f(x)$  is the objective (error) function, and  $x$  is a parameters vector. The algorithm begins by scaling the search domain to a unit hypercube. Next, it iteratively performs a sampling procedure consisting of two steps: (i) partitioning the hypercube into smaller hyper-rectangles and (ii) identifying a set of potentially-optimal hyper-rectangles by sampling their centers. Iterations continue until the error function converges. DIRECT is guaranteed to converge to the global optimal function value as the number of iterations approaches infinity and the function is continuous in the neighborhood of a global optimum [11]. If interrupted at any iteration, for example after exhausting a time limit, the algorithm reports the best-so-far parameters set.

Since SAX parameters are integer values, we round the values reported by DIRECT to the closest integers when optimizing their selection for our cross validation-based error function (one minus  $fMeasure$ , as described in Section 4.1). While rounding affects the DIRECT convergence speed, this approach is not only much more efficient than the exhaustive search, but is also able to perform a time-constrained parameter optimization by limiting the number of iterations.

## 4.3 Classification with class-specific trained parameters

With the best SAX parameters learned from Section 4.2, the representative patterns can be obtained by calling Algorithms 1 and 2. To classify future instances, we follow the classification procedure described in Section 3.1. However, with the class-specific parameter optimization, we need to add more steps to the classification procedure to account for the adaptive parameter sets for different classes. Different classes may have different best SAX parameter combinations (SPCs). We learn the best SPCs for each class respectively as described previously. Then we apply Algorithms 1 and 2 to obtain the representative patterns for each SPC. We combine all these representative patterns together and remove the correlated patterns by applying feature selection again. We then obtain the final set of representative patterns, which will be used as the input for the algorithm described in Section 3.1 to classify test data.

## 5. EXPERIMENTAL EVALUATION

### 5.1 Setup and Baseline

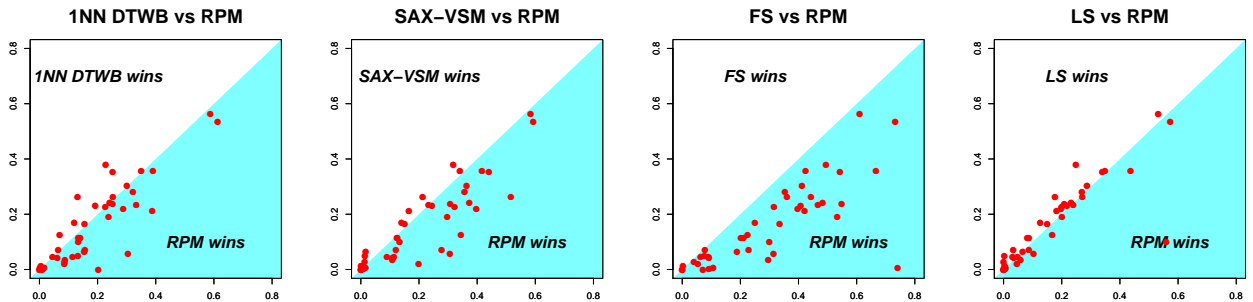
We evaluate the classification performance of our technique on the standard time series archive from the UCR repository [14]. Information on the datasets is shown in Table 1. The code and datasets used for the case study (discussed next section) are available on [1]. We compare our method Representative Pattern Mining (RPM) with five other classification techniques, among which are two nearest-neighbor classifiers based on the global distance measures: Euclidean distance (1NN-ED) and DTW with the best warping window (1NN-DTWB), and three classifiers based on the use of class-characteristic local patterns: Fast Shapelets (FS) [27], SAX-VSM [31] and Learning Shapelets (LS) [7]. These three subsequence-based techniques rely on different numbers of patterns for classification – while Fast Shapelets uses a minimal number of patterns to build a classification tree, SAX-VSM accounts for all patterns extracted via sliding window in each of the class-representing weight vectors. Learning Shapelets has the best accuracy so far.

### 5.2 Classification Accuracy

Table 1 shows the classification error rates for all six methods on the UCR datasets. The best error rate for each dataset is denoted with boldface. In addition, Figure 7 shows the summary comparison of our proposed technique with other methods. The results shown are with parameter optimization, and the  $\gamma$  (minimum cluster size) is set to be 20% of the training size for the class. From the results, our method is the second best on classification accuracy among these six methods. We slightly lose to the Learning Shapelets method, which has the most “wins” in classification. However, the p-value of wilcoxon test is 0.834 > 0.05, so the difference is not significant with a confidence

**Table 1:** Datasets description and the classification error rates.

Dataset	Classes	Train	Test	Length	1NN-ED	1NN-DTWB	SAX-VSM	FS	LS	RPM
50words	50	450	455	270	0.369	0.242	0.374	0.483	<b>0.232</b>	0.242
Adiac	37	390	391	176	0.389	0.391	0.417	0.425	0.437	<b>0.355</b>
Beef	5	30	30	470	0.333	0.333	<b>0.233</b>	0.467	0.240	<b>0.233</b>
CBF	3	30	900	128	0.148	0.004	0.010	0.092	0.006	<b>0.001</b>
ChlorineConcentration	3	467	3840	166	0.352	0.350	<b>0.341</b>	0.667	0.349	0.355
CinC_ECG_torso	4	40	1380	1639	0.103	<b>0.070</b>	0.344	0.225	0.167	0.125
Coffee	2	28	28	286	<b>0.000</b>	<b>0.000</b>	<b>0.000</b>	0.071	<b>0.000</b>	<b>0.000</b>
Cricket_X	12	390	390	300	0.423	0.252	0.308	0.549	<b>0.209</b>	0.236
Cricket_Y	12	390	390	300	0.433	<b>0.228</b>	0.318	0.495	0.249	0.379
Cricket_Z	12	390	390	300	0.413	0.238	0.297	0.533	0.201	<b>0.190</b>
DiatomSizeReduction	4	16	306	345	0.065	0.065	0.121	0.078	<b>0.033</b>	0.069
ECG200	2	100	100	96	<b>0.120</b>	<b>0.120</b>	0.140	0.250	0.126	0.170
ECGFiveDays	2	23	861	136	0.203	0.203	0.001	<b>0.000</b>	<b>0.000</b>	<b>0.000</b>
FaceAll	14	560	1690	131	0.286	<b>0.192</b>	0.245	0.408	0.218	0.231
FaceFour	4	24	88	350	0.216	0.114	0.114	0.091	0.048	<b>0.045</b>
FacesUCR	14	200	2050	131	0.231	0.088	0.109	0.296	0.059	<b>0.034</b>
Fish	7	175	175	463	0.217	0.154	<b>0.017</b>	0.189	0.066	0.065
Gun_Point	2	50	150	150	0.087	0.087	0.013	0.040	<b>0.000</b>	0.027
Haptics	5	155	308	1092	0.630	0.588	0.584	0.610	<b>0.532</b>	0.562
InlineSkate	7	100	550	1882	0.658	0.613	0.593	0.733	0.573	<b>0.535</b>
ItalyPowerDemand	2	67	1029	24	0.045	0.045	0.089	0.063	<b>0.031</b>	0.044
Lightning2	2	60	61	637	0.246	<b>0.131</b>	0.213	0.361	0.177	0.262
Lightning7	7	70	73	319	0.425	0.288	0.397	0.397	<b>0.197</b>	0.219
MALLAT	8	55	2345	1024	0.086	0.086	0.199	0.054	0.046	<b>0.020</b>
MedicalImages	10	381	760	99	0.316	<b>0.253</b>	0.516	0.443	0.271	0.262
MoteStrain	2	20	1252	84	0.121	0.134	0.125	0.202	<b>0.087</b>	0.113
OliveOil	4	30	30	570	0.133	0.133	0.133	0.300	0.560	<b>0.100</b>
OSULeaf	6	200	242	427	0.479	0.388	<b>0.165</b>	0.421	0.182	0.211
SonyAIBORobotSurface	2	20	601	70	0.304	0.305	0.306	0.315	0.103	<b>0.058</b>
SonyAIBORobotSurfaceII	2	27	953	65	0.141	0.141	0.126	0.211	<b>0.082</b>	0.114
SwedishLeaf	15	500	625	129	0.211	0.157	0.278	0.229	0.087	<b>0.070</b>
Symbols	6	25	995	398	0.101	0.062	0.109	0.091	<b>0.036</b>	0.042
synthetic_control	6	300	300	60	0.120	0.017	0.017	0.107	<b>0.007</b>	<b>0.007</b>
Trace	4	100	100	275	0.240	0.010	<b>0.000</b>	<b>0.000</b>	<b>0.000</b>	<b>0.000</b>
Two_Patterns	4	1000	4000	128	0.093	<b>0.002</b>	0.004	0.741	0.003	0.005
TwoLeadECG	2	23	1139	82	0.253	0.132	0.014	0.075	<b>0.003</b>	0.048
uWaveGestureLibrary_X	8	896	3582	315	0.261	0.227	0.323	0.316	<b>0.200</b>	0.226
uWaveGestureLibrary_Y	8	896	3582	315	0.338	0.301	0.364	0.412	<b>0.287</b>	0.303
uWaveGestureLibrary_Z	8	896	3582	315	0.350	0.322	0.356	0.353	<b>0.269</b>	0.279
Wafer	2	300	3000	426	0.005	0.005	<b>0.001</b>	0.003	0.004	0.013
WordsSynonyms	25	267	638	270	0.382	<b>0.252</b>	0.440	0.542	0.340	0.353
Yoga	2	300	3000	426	0.170	0.155	0.151	0.335	<b>0.150</b>	0.165
# of best (including ties)					2	9	7	2	19	15
Wilcoxon Test p-values (RPM vs Other)					0.006	0.287	0.217	0.002	0.834	-



**Figure 7:** Our technique and current state of the art classifiers performance comparison.

of 95%. Moreover, as we can visually inspect from Figure 7, the error rate difference between Learning Shapelets and RPM is very small — most of the points are located around the diagonal. In the next section, we will show that our method is much faster than the Learning Shapelets method.

### 5.3 Efficiency

The pre-processing, discretization and Sequitur grammar induction all have linear time complexity in the size of training data and, in practice, can be done simultaneously, since they process data sequentially. To select the best representative patterns for each class, we need to first cluster the candidate motifs identified by the grammar rules. The time required for clustering depends on the number of motif instances identified. Suppose there are (on average)  $u$  motif instances in each grammar rule, then the complexity is  $O(u^3 \cdot |rules|)$  for hierarchical clustering. The centroids of the qualifying clusters are the pattern candidates. For each pattern candidate, we perform subsequence matching to identify the best matches ( $O(|Candidates| \cdot |Train|)$ ). Even though the number of pattern candidates for each class is relatively small compared to the training size, this step seems to be the bottleneck of the training stage due to the repeated distance call. We use early abandoning strategy [32] to speed up the subsequence matching, but other options are possible such as approximate matching. The training needs to be repeated for each class. Thus, the training complexity is  $O(|Train| + c \cdot (u^3 \cdot |rules| + |Candidates| \cdot |Train|))$ , where  $c$  is the number of classes, and  $u$  is the average number of motif instances in each grammar rule.

If the best SAX parameters are known, our algorithm is fast — in this case, simply using the classification method described in Section 3.1 completes the classification task. To get the best SAX parameters, we evaluated cross-validation based parameter selection techniques, exhaustive search and DIRECT described in section 4.1 and 4.2. Exhaustive search was found time-consuming even with early abandoning, therefore we use DIRECT. The overall running time using DIRECT in the worst case is  $O((|Train| + c \cdot (u^3 \cdot |rules| + |Candidates| \cdot |Train|)) \cdot R)$ , where  $R$  is the number of SAX parameters combinations tested by DIRECT algorithm. From the experiments on 42 UCR time series datasets, the average value for  $R$  is less than 200, which is smaller than the average time series length 363. In most of the  $R$  evaluations, the program terminated search early because of the minimum motif frequency requirement (Sec. 3.2). The classification time, compared to training, is negligible.

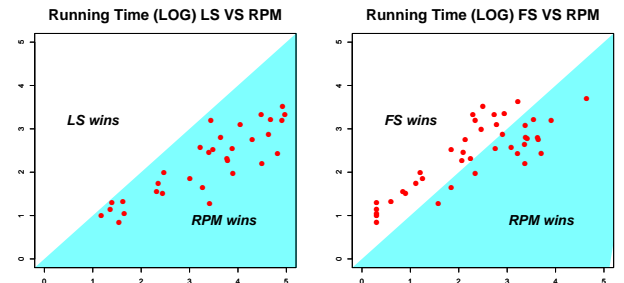
We compare the total running time of our algorithm using DIRECT with that of Fast Shapelets (FS) and Learning Shapelets (LS), and the results are shown in Table 2. Even when accounting for parameter selection, our algorithm is comparable to Fast Shapelets in running time, and as shown in Table 1, our method is significantly more accurate than Fast Shapelets (p-value equals 0.002). Compared to Learning Shapelets, our method is a lot faster. The greatest speedup we achieve through these 42 datasets is 1587X on dataset Adiac, and the average speedup is 178X. The experiment results show that our method is comparable to the fastest algorithm in speed and to the most accurate algorithm in accuracy.

In Section 3.2.3, we use  $\tau$  as the threshold to remove similar patterns. We choose the value at the 30<sup>th</sup> percentile of the pair-wise distances as the threshold. We also compare

the results using the 10<sup>th</sup>, 50<sup>th</sup>, 70<sup>th</sup>, and 90<sup>th</sup> percentiles. The running time and classification error changes are shown in Figure 9. The average running time and classification error changes on the 42 UCR data set are shown in Table 3. The average standard deviation for running time is 268.71 seconds, and 0.014 for classification error .

**Table 2:** Running time and classification accuracy comparison between Fast Shapelets, Learning Shapelets and Representative Pattern Mining

Dataset	Running Time (Seconds)		
	LS	FS	RPM
50words	3396298	<b>1666</b>	4221
Adiac	1551130	<b>290</b>	977
Beef	5971	<b>175</b>	202
CBF	275	<b>7</b>	32
ChlorineConcentration	7668	572	<b>347</b>
CinC_ECG_torso	46979	3521	<b>1636</b>
Coffee	293	<b>15</b>	98
Cricket_X	252834	2286	<b>438</b>
Cricket_Y	249889	2378	<b>1208</b>
Cricket_Z	260107	2611	<b>594</b>
DiatomSizeReduction	1013	<b>17</b>	69
ECG200	224	<b>12</b>	55
ECGFiveDays	41	<b>3</b>	20
FaceAll	93442	<b>538</b>	2139
FaceFour	1853	69	<b>43</b>
FacesUCR	30516	<b>195</b>	2141
Fish	42766	802	<b>755</b>
Gun_Point	209	<b>6</b>	34
Haptics	81751	8100	<b>1575</b>
InlineSkate	314244	43930	<b>4970</b>
ItalyPowerDemand	14	<b>1</b>	9
Lightning2	1657	1212	<b>373</b>
Lightning7	7923	219	<b>93</b>
MALLAT	65920	1645	<b>267</b>
MedicalImages	19864	<b>136</b>	555
MoteStrain	22	<b>1</b>	13
OliveOil	2499	<b>123</b>	287
OSULeaf	31181	2337	<b>154</b>
SonyAIBORobotSurface	34	<b>1</b>	6
SonyAIBORobotSurfaceII	44	<b>1</b>	10
SwedishLeaf	83656	<b>317</b>	3312
Symbols	3043	<b>69</b>	328
synthetic_control	2616	37	<b>18</b>
Trace	6104	<b>115</b>	185
Two_Patterns	11219	<b>601</b>	1241
TwoLeadECG	24	<b>1</b>	19
uWaveGestureLibrary_X	267727	5060	<b>274</b>
uWaveGestureLibrary_Y	373482	4429	<b>567</b>
uWaveGestureLibrary_Z	409494	4230	<b>619</b>
Wafer	2746	<b>217</b>	1585
WordsSynonyms	852394	<b>877</b>	2271
Yoga	4414	2388	<b>642</b>
# best (including ties)	0	24	18

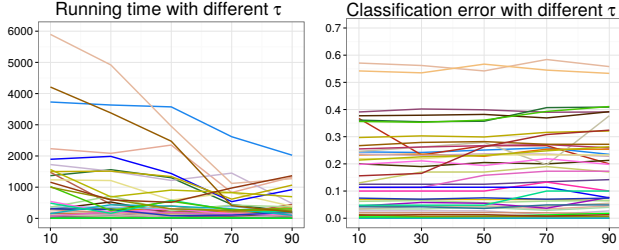


**Figure 8:** Runtime comparison between Representative Patterns, Fast Shapelets, and Learning Shapelets classifiers.

The average classification accuracy change with different  $\tau$  values is below 1%. That means this parameter does not affect the classification result too much. The user can set a

**Table 3:** The average running time and classification error changes for different similar threshold on 42 UCR data. Positive value means increase, negative value means decrease.

	10% - 30%	30% - 50%	50% - 70%	70% - 90%
Running Time Change (%)	-4.66	-12.38	-15.09	-1.17
Error Change (%)	-0.14	0.74	0.72	0.28



**Figure 9:** The running time and accuracy with different similarity threshold  $\tau$ .

higher value for this parameter to achieve a fast speed and still could maintain a high accuracy.

The  $\tau$  value used for the experiment results (classification error and running time) shown in Tables 1 and 2 is set as 30% because it gives the best accuracy and still has a fast running speed.

## 6. CASE STUDY

In this section we demonstrate the application of our method to classification of rotated time series data and Medical Alarm data. We compare results with 1NN Euclidean distance, 1NN DTW with the best warping window, SAX-VSM, Fast Shapelets, and Learning Shapelets classifiers.

### 6.1 Rotation invariance

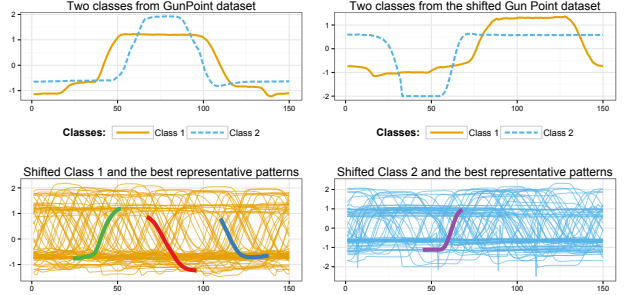
Many global-distance-based techniques do not work well if the time series data are shifted or out of phase. One type of time series data that is particularly susceptible to this kind of distortion is shape-converted time series [19], e.g. by radial scanning of the shape profile to convert the image into a “time series.” Several datasets in the UCR repository are shape-converted time series, e.g. OSU Leaf, Swedish Leaf, Shields, etc. In this section we demonstrate the rotation- or shift-invariance of our technique on a number of shifted datasets. To shift or “rotate” a time series, we randomly choose a cut point in the time series, and swap the sections before and after the cut point. This transformation is equivalent to starting the radial scanning of the shape at a different position on the shape profile. The out-of-phase phenomenon is also common in many other real-world time series data such as video. Figure 10 illustrates the original GunPoint dataset time series and their rotation.

In our experiments, we leave the training set unmodified, and shift only the test data. The rationale is that while it is not uncommon for one to pre-process and create a “cleaned” version of training data to build a good model, it is less reasonable to expect that the test data will be in the same cleaned format. In other words, we learn the patterns on existing training data, but modify the test data to create rotation distortion, in order to evaluate the robustness of our technique.

It is possible that the rotation cuts the best matching subsequence of the test time series. To handle this, we introduce

**Table 4:** Classification error rate on shifted time series

Dataset	1NN-ED	1NN-DTWB	SAX-VSM	LS	RPM
Coffee	0.536	0.460	<b>0.000</b>	0.036	<b>0.000</b>
Face Four	0.682	0.625	0.125	0.080	<b>0.045</b>
Gun point	0.460	0.493	<b>0.047</b>	0.200	<b>0.047</b>
Swedish Leaf	0.872	0.821	0.430	0.371	<b>0.246</b>
OSU Leaf	0.595	0.479	<b>0.107</b>	0.186	0.157
# best (including ties)	0	0	3	0	4



**Figure 10:** Shifted GunPoint dataset and the best representative patterns.

a new strategy to the test time series transformation step to make our algorithm rotation invariant. When transforming a raw time series into the new feature space of the closest match distances, our algorithm needs to calculate the distance between the time series to each representative pattern. In order to solve the aforementioned problem, our algorithm will build another time series. For example, when transforming a rotated time series  $A$ , we generate another new time series  $B$  by cutting  $A$  from its midpoint and swapping the first and the second halves. By doing so,  $B$  will contain the concatenation of  $A$ ’s tail and head. If the best-matching subsequence happens to be broken up due to the rotation ( $A$ ), then by rotating it again at the midpoint, one of  $A$  or  $B$  will contain the entirety of the best-matching subsequence. When computing the distance of  $A$  to a pattern  $p$ , besides calculating a distance  $d_a$  between  $A$  and  $p$ , the algorithm will also calculating another best match distance  $d_b$  between  $B$  and  $p$ . The minimal distance of these two will be used as the distance from  $A$  to  $p$ . This solution overcomes the potential problem that arises when the best matching pattern is cut into different parts due to the rotation.

The classification error rates of the rotated data are shown in Table 4. The accuracy of our method or SAX-VSM does not change very much from the unrotated version (though our technique seems to be more robust, with 4 wins), while the error rates of 1NN Euclidean distance and 1NN DTWB increase drastically.

### 6.2 Medical Alarm

In this case study, we use the medical alarm data from Intensive Care Unit (ICU) database (MIMIC II database from PhysioNet) [6]. We used arterial blood pressure (ABP) waveforms to create the dataset used in this work.

#### 6.2.1 Normal or Alarm

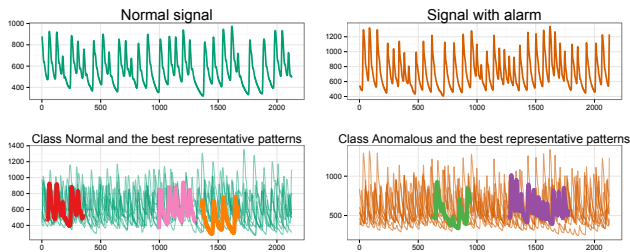
We selected two types of ABP series segments: those that triggered an alarm and those that did not. The data used are all from the same patient. The selected dataset contains 52 time series of length of 2126. Each of the time series represents a segment of arterial blood pressure (ABP) waveform

in a 17 second time range. The *Normal* class consists of segments without any alarms, whereas the *Alarm* class consists of segments that triggered the bedside patient monitors and were verified by the domain expert as true alarms. The dataset contains 26 time series in class *Normal* and 26 in class *Alarm*. Training and test data are split into sets of 16 and 36 time series respectively. Examples of medical alarm data from each class are shown in Figure 11.

The first row of Table 5 shows the accuracy of competing classification techniques. Figure 11 shows the representative patterns from medical alarm time series and their best matches on test data. Our method (RPM) achieves the best accuracy on this dataset.

**Table 5:** Classification error rate on Medical Alarm data

Dataset	1NN-ED	1NN-DTW	SAX-VSM	FS	LS	RPM
NormalOrAlarm	0.333	0.333	0.167	0.306	0.111	<b>0.056</b>
FiveAlarmTypes	0.760	0.360	0.350	0.485	<b>0.260</b>	0.300



**Figure 11:** Normal or Alarm data and the best representative patterns.

### 6.2.2 Five types of alarm

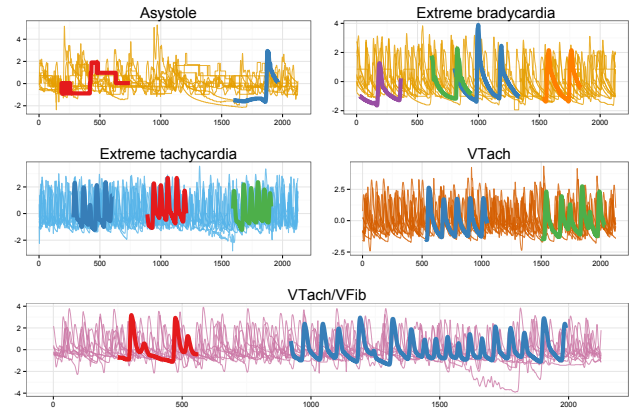
In PhysioNet’s MIMIC II database, there are five categories of critical arrhythmia alarms produced by a commercial ICU monitoring system: *Asystole*, *Extreme Bradycardia*, *Extreme Tachycardia*, *Ventricular Tachycardia*, and *Ventricular Fibrillation/Tachycardia*. We collected a dataset by taking the segments of an arterial blood pressure waveform. Each segment contains a verified alarm. The objective is to classify the alarm time series into one of the five types of alarms.

The training set has 50 examples, 10 for each class. The test set has 100 examples, 20 for each class. All time series have the same length of 2126 (17 seconds). The time series example of each class, the representative patterns, and the best matches are shown in Figure 12. Table 5 shows the accuracy of competing classification techniques for this dataset.

Our method (RPM) has the second best accuracy on this dataset. We lose slightly to Learning Shapelets since we have 30 incorrectly classified instances compare to 26 with LS. However, our method finished in 1373 seconds compare to 86195 seconds of LS. The speedup of our algorithm over LS is 63X on this dataset.

## 7. CONCLUSIONS

In this work, we propose a novel method to discover representative patterns of time series, specifically for the problem of classification. We demonstrate through extensive



**Figure 12:** Five type of Medical Alarm and the best representative patterns.

experimental evaluation that our technique achieves competitive classification accuracy on the standard UCR time series repository, and is able to discover meaningful sub-space patterns. The accuracy of our technique remains stable even when the data are shifted, while NN classifiers with global distance measures suffer from shift distortion. We also demonstrate that our technique outperforms existing techniques on a real-world medical alarm data that is extremely noisy.

In terms of efficiency, while our approach is competitive or better than other techniques, there are other optimization strategies that we can consider to speed up the algorithm even further. From profiling, we identified the bottleneck of the algorithm, which can be improved by adapting the state-of-the-art subsequence matching strategies [26]. Also, we used Euclidean distance as the base distance function for pattern matching. We will consider a more robust distance measure such as DTW in future work.

## 8. ACKNOWLEDGMENTS

This research is partially supported by the National Science Foundation under Grant No. 1218325 and 1218318.

## 9. REFERENCES

- [1] Webpage to download the code and dataset. <http://mason.gmu.edu/~xwang24/Projects/RPM.html>.
- [2] R. Briandet, E. K. Kemsley, and R. H. Wilson. Discrimination of arabica and robusta in instant coffee by fourier transform infrared spectroscopy and chemometrics. *Journal of agricultural and food chemistry*, 44(1):170–174, 1996.
- [3] B. Chiu, E. Keogh, and S. Lonardi. Probabilistic discovery of time series motifs. In *Proc. of 9th ACM SIGKDD Intl. Conf.*, 2003.
- [4] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proceedings of the VLDB Endowment*, 1(2):1542–1552, 2008.
- [5] P. Geurts. Pattern extraction for time series classification. In *Principles of Data Mining and Knowledge Discovery*, pages 115–127. Springer, 2001.

- [6] A. L. Goldberger, L. A. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. Physiobank, physiotoolkit, and physionet components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000.
- [7] J. Grabocka, N. Schilling, M. Wistuba, and L. Schmidt-Thieme. Learning time-series shapelets. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 392–401. ACM, 2014.
- [8] M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.
- [9] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery*, 15(1):55–86, 2007.
- [10] J. Hills, J. Lines, E. Baranauskas, J. Mapp, and A. Bagnall. Classification of time series by shapelet transformation. *Data Mining and Knowledge Discovery*, 28(4):851–881, 2014.
- [11] D. R. Jones, C. D. Perttunen, and B. E. Stuckman. Lipschitzian optimization without the lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181, 1993.
- [12] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and information Systems*, 3(3):263–286, 2001.
- [13] E. Keogh and S. Kasetty. On the need for time series data mining benchmarks: a survey and empirical demonstration. *DMKD*, 7(4):349–371, 2003.
- [14] E. Keogh, X. Xi, L. Wei, and C. A. Ratanamahatana. The UCR time series classification/clustering homepage. [http://www.cs.ucr.edu/~eamonn/time\\_series\\_data](http://www.cs.ucr.edu/~eamonn/time_series_data).
- [15] H. Kremer, S. Günnemann, A. Held, and T. Seidl. Effective and robust mining of temporal subspace clusters. In *ICDM*, pages 369–378, 2012.
- [16] X. Li, E. Keogh, L. Wei, and A. Mafra-Neto. Finding motifs in a database of shapes. In *SDM*, pages 249–260, 2007.
- [17] Y. Li, J. Lin, and T. Oates. Visualizing variable-length time series motifs. In *SDM*, pages 895–906, 2012.
- [18] J. Lin, E. Keogh, L. Wei, and S. Lonardi. Experiencing SAX: a novel symbolic representation of time series. *Data Mining and knowledge discovery*, 2007.
- [19] J. Lin, R. Khade, and Y. Li. Rotation-invariant similarity in time series using Bag-of-Patterns representation. *Journal of Intelligent Inform. Systems*, 39, 2012.
- [20] A. Mueen, E. Keogh, and N. Young. Logical-shapelets: an expressive primitive for time series classification. In *Proc. of 17th ACM SIGKDD Intl. Conf.*, 2011.
- [21] A. Nanopoulos, R. Alcock, and Y. Manolopoulos. Feature-based classification of time-series data. *International Journal of Computer Research*, 10(3), 2001.
- [22] S. Naoki. *Local feature extraction and its application using a library of bases*. Ph.D. thesis, Yale University, 1994.
- [23] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *J. Artif. Intell. Res. (JAIR)*, 7:67–82, 1997.
- [24] J. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1998.
- [25] M. Radovanovic, A. Nanopoulos, and M. Ivanovic. Time-series classification in many intrinsic dimensions. In *SDM*, pages 677–688, 2010.
- [26] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. 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, KDD '12*, pages 262–270, New York, NY, USA, 2012.
- [27] T. Rakthanmanon and E. Keogh. Fast shapelets: A scalable algorithm for discovering time series shapelets. In *Proc. of SDM*, 2013.
- [28] C. A. Ratanamahatana and E. Keogh. Making time-series classification more accurate using learned constraints. SIAM, 2004.
- [29] G. Salton. *The SMART Retrieval System; Experiments in Automatic Document Processing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1971.
- [30] P. Senin, J. Lin, X. Wang, T. Oates, S. Gandhi, A. P. Boedihardjo, C. Chen, S. Frankenstein, and M. Lerner. Grammarviz 2.0: a tool for grammar-based pattern discovery in time series. In *Proc. ECML/PKDD*. 2014.
- [31] P. Senin and S. Malinchik. SAX-VSM: Interpretable time series classification using sax and vector space model. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, pages 1175–1180. IEEE, 2013.
- [32] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, and E. Keogh. Experimental comparison of representation methods and distance measures for time series data. *Data Mining and Knowledge Discovery*, 26(2):275–309, 2013.
- [33] L. Wei and E. Keogh. Semi-supervised time series classification. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 748–753. ACM, 2006.
- [34] X. Xi, E. Keogh, C. Shelton, L. Wei, and C. A. Ratanamahatana. Fast time series classification using numerosity reduction. In *Proceedings of the 23rd international conference on Machine learning*, pages 1033–1040. ACM, 2006.
- [35] Z. Xing, J. Pei, S. Y. Philip, and K. Wang. Extracting interpretable features for early classification on time series. In *SDM*, volume 11, pages 247–258. SIAM, 2011.
- [36] L. Ye and E. Keogh. Time series shapelets: a new primitive for data mining. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 947–956. ACM, 2009.



# Interactive Temporal Association Analytics\*

Xiao Qin, Ramoza Ahsan, Xika Lin, Elke A. Rundensteiner, and Matthew O. Ward

Department of Computer Science  
Worcester Polytechnic Institute  
100 Institute Road, Worcester MA, USA

{xqin,rahsan,xika,rundenst,matt}@cs.wpi.edu

## ABSTRACT

Traditional temporal association mining systems, once supplied with a specific parameter setting such as time periods of interest, minimum support and confidence, generate the rule set from scratch. This one-at-a-time paradigm forces the analysts to perform successive trial-and-error iterations to finally discover interesting temporal patterns. This process is not only prohibitively time and resource consuming, but also ineffective in providing meaningful feedback for improving the desired rule outcome.

In this work, we introduce the first solution to interactively explore temporal associations from evolving data at multiple levels of abstraction, henceforth referred to as temporal association rule analytics (**TARA**). The offline rule preparation phase of the **TARA** infrastructure extracts the temporal associations from the raw data and compresses them into a knowledge-rich yet compact evolving parameter space (*EPS*) structure. The online exploration phase of **TARA** leverages this *EPS* structure to offer rich classes of novel exploration operations from parameter recommendations and time-travel queries to the discovery of hidden insights of associations with near instantaneous responsiveness. As demonstrated by our extensive experiments on real-world data sets, **TARA** accomplishes three to five orders of magnitude improvement over state-of-the-art approaches while offering a rich interactive exploration experience.

## 1. INTRODUCTION

### 1.1 Motivation

Nowadays batches of data are continuously transmitted from a rich variety of sources including websites, mobile devices and other data sources, henceforth referred to as *evolving datasets*. Discovering associations and their dynamics

\*This work was partly supported by the National Science Foundation under grants IIS-1117139, CRI-1305258, IIS-0812027 and CCF-0811510 and Fulbright program.

hidden in such large evolving datasets has been recognized as critical for domains ranging from market products analysis, stock trend monitoring, targeted advertising to weather forecasting.

For example, in the retail businesses, the arrival of new fashions or gadgets may boost unprecedented sales while seasonal products may gain or lose customers' interest. Some products are purchased together more frequently in the days leading to a large sports event or during a traditional holiday like Thanksgiving. Companies such as Amazon, eBay, Walmart and other retail businesses apply temporal association mining techniques to their transaction logs to identify popular product combination at specific times and their behavior over time. Such information is critical for deciding the times when products can be placed together on a web page or configured into attractive bundle-offers to be used for recommendations to encourage sales.

Interactive data mining models, crucial for discovering knowledge from data, enable analysts to actively engage in the analysis process. State-of-the-art temporal association mining systems [2, 9, 14, 18], once supplied with a specific parameter setting, tend to generate the ruleset for each request from scratch. This one-at-the-time request model suffers from severe limitations described below.

### 1.2 Limitations of State-of-the-Art

**Lack of instantaneous responsiveness.** Lag in responsiveness is known to risk losing an analyst's attention during the exploration process. In applications like targeted ad placement such delay in decision making may prove to be the cause of missed business opportunities and thus a potentially huge loss in profit. Unfortunately, *temporal association mining algorithms* [2, 14] are known to be computationally intensive. To overcome this challenge, [18] pre-generates the *intermediate itemsets* that are subsequently used to derive the temporal associations instead of extracting them from the huge raw data store. With this promising one-time preprocessing strategy, the response time has been shown to be greatly reduced. However, the process of the final rule derivation remains a query-time task. This results in the shortcoming that the response times for mining such requests are not sufficient to support truly interactive exploration as confirmed by our experiments (Sec. 8).

**Lack of parameter recommendations.** Temporal association mining algorithms are parametrized not only by traditional measures like *support* and *confidence* but also by *time-variant* measures [11, 16, 17]. Parameter settings

used for one batch of data may produce insignificant rules for a newly incoming data batch. Thus the data analysts often must perform numerous successive trial-and-error iterations to find an appropriate parameter configuration out of a seemingly infinite number of possible settings. Existing state-of-the-art models tend to correspond to a black-box [2, 6, 9, 14, 18] - providing little to no feedback about which parameter settings best capture the analyst’s interest. To tackle this, [10] incorporates an indexing technique to swiftly produce parameter recommendations. However it is restricted to static data and thus does not support *time variant operations* essential for temporal association mining.

**Lack of evolving ruleset comparison.** Analysis of the data in finer time granularity may reveal that associations exist only in certain time periods. Some may fluctuate as new data arrives while others may remain stable. Furthermore, two seemingly similar parameter settings can generate different results. Systems like [2, 6, 14, 18] independently generate the ruleset for each parameter settings. Worse yet, analysts then have to go through a tedious process to manually investigate the results generated by different parameter settings to extract their differences. This can be extremely tedious and impractical for large data sets.

**Lack of insights into the evolving associations.** Given a parameter configuration, a system often generates a huge number of rules. Analysts would benefit from being able to quickly identify the most interesting ones, such as the most stable rules [11] within the last week, the most significant rules that occur every weekend, or the rules concerning specific products. Offering such rich insights into time-variant rule behavior would provide the analysts with the opportunity to leverage their domain knowledge to drive the discovery process. Unfortunately, most existing parameter-driven exploration systems [10, 16, 18] do not support the analyst in the discovery of such useful *time-sensitive* insights.

### 1.3 Research Challenges

To develop an interactive temporal analytic system, the following research challenges must be tackled.

**Processing time-variant evolving data.** Given a time-variant data set containing  $n$  unique items, the maximum number of possible associations are bounded by  $3^n - 2^n + 1$  [10]. The significance of associations may vary over time, as newly incoming data may bring new items and associations. Being able to quickly extract these associations and their behavior w.r.t different time horizons to answer analysts’ requests is the key to providing an interactive mining experience. However, it is almost impossible to pre-generate all such information. Thus the system must have an efficient preprocessing strategy that pregenerates a minimal yet sufficient amount of information as its critical knowledge store to support interactive temporal association exploration.

**Managing temporal associations for all parameters.** Typical input parameters, such as *minimum support* and *confidence*, can be configured using any real number restricted to a certain range. Similarly, the time specification can be composed of one or multiple time periods along the continuous timeline. Clearly, an infinite number of possible parameter settings exists. Maintaining the corresponding ruleset for each parameter setting individually thus is impractical. Therefore, an efficient mechanism is needed to map the pregenerated temporal associations to the space of parameter settings.

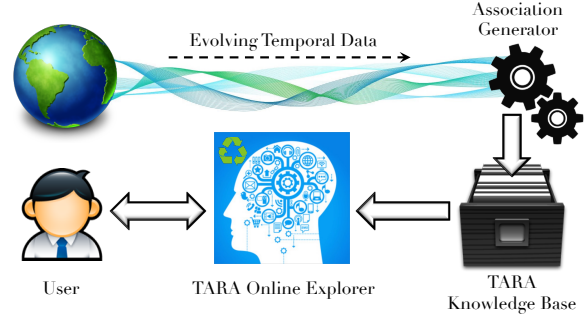


Figure 1: The TARA Approach

**Maintaining parameter values for different time periods.** The prominence level of an association may vary significantly for some associations while remaining stable for others. Such time-variant properties of parameter values may reveal important evolving patterns of an association in the evolving dataset. Yet keeping each single historical parameter value for each association is inefficient, resulting in large storage and search space. Therefore, a compact archive structure is needed to efficiently maintain the parameter values of the associations across time while supporting fast system access to retrieve any desired information.

**Supporting advanced temporal association exploration.** Rule mining algorithms tend to generate too many rules - making it extremely hard for the analysts to quickly identify the interesting ones. The problem of interestingness of temporal rules has been previously investigated [12, 17]. An interactive temporal association exploration system must integrate such interestingness measures to provide critical insights about the associations such as their evolving behaviors across time. The retrieved rules w.r.t particular parameter settings must be efficiently evaluated using these measures so that the instant responsiveness of the system is safeguarded.

### 1.4 The TARA Approach

We propose a novel temporal association rule analytics (TARA) framework that addresses the above challenges. The TARA infrastructure depicted in Fig. 1 employs an offline preprocessing phase composed of Association Generator and Knowledge Base Constructor followed by TARA Online Explorer that enables analysts to interactively explore the evolving data with support by the knowledge base.

The Association Generator extracts temporal associations from the evolving data and compactly stores them in the Temporal Association Rule Archive (*TAR Archive*) of TARA knowledge base. Later, by request, the parameter values of a particular association w.r.t various fine granularities can be quickly computed without processing the raw data again. These pregenerated temporal associations are compressed into a knowledge-rich yet compact *evolving parameter space (EPS)* that encodes the relationships among the temporal associations. Next, the TARA knowledge base explicitly extracts and then models the distribution of the pregenerated temporal associations with respect to their parameters, e.g. support, confidence and time periods.

Beyond achieving speedup in response time, the online processing strategies leverage the *EPS* index to offer analysts an innovative “rule-centric panorama” into the temporal associations present within the evolving dataset. The framework supports rich classes of novel exploration opera-

tions from time-travel queries and parameter recommendations to evolving ruleset comparisons.

## 1.5 Contributions

Key contributions of this work include:

- We propose the first *interactive temporal association rule mining* analytics framework called **TARA** that enables analysts to explore associations across time and pinpoint appropriate parameter settings in a systematic way.

- The **TARA** model organizes the temporal associations in the space of query parameters. It abstracts the temporal associations at the coarse granularity of *time-aware stable regions* across multiple time periods.

- The **TARA** model is supported by *evolving parameter space (EPS)* index structure that indexes *time-aware stable regions* along with the associated domination graph. TARA offers efficient algorithms for offline *EPS* index construction.

- For the rules generated, we design a temporal association rule archive, called *TAR Archive*, that compactly encodes the parameter values of each rule across time. Our specially designed encoding and decoding strategies achieve fast access to the requested information from this archive.

- We propose a rich set of novel temporal rule exploration operations beyond traditional temporal rule mining. Effective strategies for the online processing of the proposed operations that leverage our precomputed TARA index structures are provided.

- TARA framework supports the exploration of the associations at coarser or finer time granularities by roll-up and drill down operations. We provide a theoretical bound on the approximation of the solution under roll-up operations.

- Our extensive experiments using IBM Quest [1], *retail* [3] and *webdocs* [13] datasets demonstrate that **TARA** is 3 to 5 orders of magnitude faster than its state-of-the-art competitors for traditional temporal association mining, while in addition supporting novel analytics within milliseconds.

## 2. PRELIMINARIES OF TEMPORAL ASSOCIATION

$\mathcal{T} = \{\dots, t_i, \dots, t_j, \dots\}$  denotes a set of **times**, countably infinite, over which a linear order  $<_{\mathcal{T}}$  is defined, where  $t_i <_{\mathcal{T}} t_j$  means  $t_i$  occurs strictly before  $t_j$ . Let  $\mathcal{I} = \{i_1, i_2, \dots, i_n\}$  represent a set of **items**.  $\mathcal{D} = \{d_1, d_2, \dots, d_m\}$  is a collection of subsets of  $\mathcal{I}$  called the **transaction database**. Each **transaction**  $d_i$  in  $\mathcal{D}$  is a set of items such that  $d_i \subseteq \mathcal{I}$ . Each  $d_i$  has an associated timestamp  $t_j$ , denoted by  $d_i.time = t_j$ . Let  $\mathcal{X} \subseteq \mathcal{I}$  be a set of items, called **itemset**. If  $\mathcal{X} \subseteq d_i$ ,  $d_i$  **contains**  $\mathcal{X}$ . If the cardinality of  $\mathcal{X}$  is  $k$ ,  $\mathcal{X}$  is called a **k-itemset**. Given a closed **time period**  $[t_i, t_j]$  where  $t_i <_{\mathcal{T}} t_j$ , then the set of transactions in the range  $[t_i, t_j]$  of  $\mathcal{D}$  that **contain**  $\mathcal{X}$  is indicated by  $\mathcal{F}(\mathcal{X}, \mathcal{D}, [t_i, t_j]) = \{d_k \in \mathcal{D} \mid d_k \supseteq \mathcal{X} \wedge t_i \leq d_k.time \leq t_j\}$ .

**DEFINITION 1.** A **temporal association rule** is an expression of the form  $\mathcal{R}^{[t_i, t_j]} \equiv (\mathcal{X} \Rightarrow \mathcal{Y})$ , where  $\mathcal{X} \subseteq \mathcal{I}$ ,  $\mathcal{Y} \subseteq \mathcal{I} \setminus \mathcal{X}$ , and  $[t_i, t_j]$  indicates that  $\mathcal{R}$  is derived from all the transactions in  $\mathcal{D}$  whose timestamps fall into  $[t_i, t_j]$ .

A **temporal association rule** defaults to the traditional association rule if the *time period* is set to the entire timeline. This time restriction  $[t_i, t_j]$  empowers the data analysts to discover associations that are not significant throughout the entire data set. Moreover, an association may reappear

Table 1: Example of pregenerated temporal association rule

(a) Item set (min supp = 0.05)      (b) Rule (min conf = 0.25)

Itemset	Support		Rule	(Support, Confidence)	
	$\mathcal{T}_1$	$\mathcal{T}_2$		$\mathcal{T}_1$	$\mathcal{T}_2$
a	0.36	0.44	$\mathcal{R}_1: a \rightarrow b$	(0.18, 0.5)	(0.11, 0.25)
b	0.45	0.22	$\mathcal{R}_2: b \rightarrow a$	(0.18, 0.4)	(0.11, 0.5)
c	0.36	0.44	$\mathcal{R}_3: a \rightarrow c$	(0.18, 0.5)	(0.33, 0.75)
ab	0.18	0.11	$\mathcal{R}_4: c \rightarrow a$	(0.18, 0.5)	(0.33, 0.75)
ac	0.18	0.33	$\mathcal{R}_5: c \rightarrow b$	(0.09, 0.25)	(0.11, 0.25)
bc	0.09	0.11	$\mathcal{R}_6: b \rightarrow c$	-	(0.11, 0.5)

in multiple time periods expressing some periodicity. Furthermore, the association may behave differently in terms of its measured values. The evolution of the associations over time can lead to insightful observations [11].

Many measurements [17] have been proposed to evaluate the interestingness of associations. Out of these measurements, we work with the most common measures of *support* and *confidence* to demonstrate the key principles of our framework, though others can be plugged in the future.

$$Support(\mathcal{R}^{[t_i, t_j]}) = \frac{|\mathcal{F}(\mathcal{X} \cup \mathcal{Y}, \mathcal{D}, [t_i, t_j])|}{|\mathcal{F}(\emptyset, \mathcal{D}, [t_i, t_j])|} \quad (1)$$

$$Confidence(\mathcal{R}^{[t_i, t_j]}) = \frac{|\mathcal{F}(\mathcal{X} \cup \mathcal{Y}, \mathcal{D}, [t_i, t_j])|}{|\mathcal{F}(\mathcal{X}, \mathcal{D}, [t_i, t_j])|} \quad (2)$$

The **support** defined in Formula 1 describes the proportion of the transactions within the defined time period that **contains** all items in the association. The **confidence** defined in Formula 2 describes the probability of finding the **consequent**  $\mathcal{Y}$  of the association within the defined time period under the condition that these transactions also **contain** the **antecedent**  $\mathcal{X}$ .

## 3. THE TARA MODEL

We now introduce our **TARA** model framework for interactive exploration of associations from evolving data.

### 3.1 Time Dimension of the TARA Model

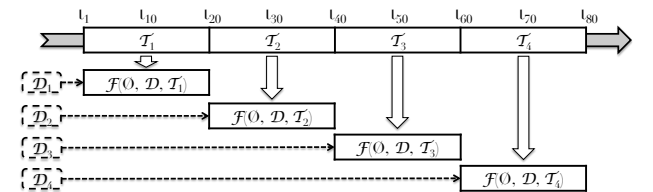


Figure 2: Tumbling Window Model of TARA

Data analysts often are interested in exploring the associations that hold in particular time periods, such as an hour or a day. Coarser time specifications can be broken down to ranges of smaller granularities. Moreover, the measures of an association in a longer time period can then be computed based on the measures of the associations in the shorter periods that are part of this longer period. Based on this observation, **TARA** partitions the data set into disjoint time periods, called *windows*. Mining queries with a coarser time granularity settings than this basic *window size* are then supported using roll-up operations.

Let  $\mathcal{D}$  be the evolving data set and  $w$  be the basic window

size that represents the minimum granularity. Therefore, the set of *times*  $\mathcal{T}$  contains disjoint but consecutive time periods each of size  $w$  denoted by  $\mathcal{T} = \{\dots, \mathcal{T}_i, \dots, \mathcal{T}_j, \dots\}$ , ( $\forall \mathcal{T}_i, \mathcal{T}_j$ ), if  $\mathcal{T}_i \neq \mathcal{T}_j$ , and  $\mathcal{T}_i \cap \mathcal{T}_j = \emptyset$ . The evolving data set  $\mathcal{D}$  is partitioned into small chunks according to each time period  $\mathcal{T}_i$  in  $\mathcal{T}$  denoted by  $\mathcal{D} = \{\dots, \mathcal{D}_i, \dots, \mathcal{D}_j, \dots\}$  where  $\mathcal{D}_i = \mathcal{F}(\emptyset, \mathcal{D}, \mathcal{T}_i)$ . In Fig. 2, for example we set the *window size*  $w = 20$ . That is, the time frame is partitioned into a set of time periods of length 20, e.g.  $\mathcal{T}_2 = [t_{21}, t_{40}]$ . The evolving data set  $\mathcal{D}$  is partitioned into time-oriented data subsets  $\mathcal{D}_i$  according to these time periods, e.g.  $\mathcal{D}_2 = \mathcal{F}(\emptyset, \mathcal{D}, \mathcal{T}_2)$ . For each data partition  $\mathcal{D}_i$ , **TARA** pregenerates the associations off-line (See Sec. 4). Table 1.(b) shows an example of the associations generated for the time periods  $\mathcal{T}_1$  and  $\mathcal{T}_2$ . **TARA** processes the raw data  $\mathcal{D}$  once to pregenerate the temporal associations held in these windows. A query with the coarser time specification can then be answered based on these pregenerated associations.

**DEFINITION 2. Time availability:** Let  $w$  be the finest time granularity. Then  $\mathcal{T}^w = \{\dots, \mathcal{T}_i^w, \dots, \mathcal{T}_j^w, \dots\}$  corresponds to the basic time periods of  $\mathcal{T}$  that are generated by **TARA** through time partitioning by  $w$ . A time specification  $\mathcal{T}_k$  supported in **TARA** thus is  $\mathcal{T}_k = \bigcup_{m=i}^j \mathcal{T}_m^w$ , where  $i \leq j$ .

This strategy allows us to support **roll-up** and **drill-down** of time periods at run time such as days, months or years to support long and short term goals.

### 3.2 Evolving Parameter Space Model

In association rule mining, the input parameter values of *minimum support* and *confidence* can be any real number within  $[0,1]$ . Each combination, referred to as **parameter setting**, corresponds to a set of rules generated by using this parameter setting. We now extend this into the notion of an *Evolving Parameter Space (EPS)* that models relationships and distribution of rules across the multi-dimensional temporal parameter space.

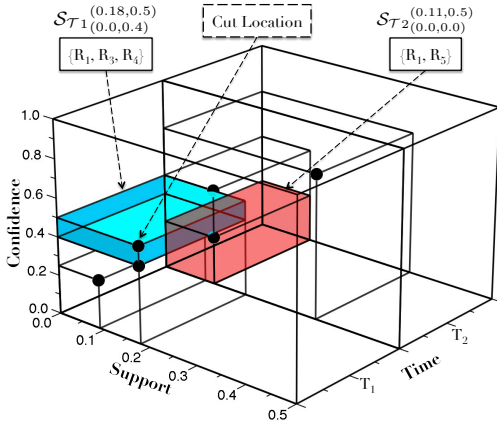


Figure 3: Evolving Parameter Space

**DEFINITION 3. Evolving Parameter Space:** Let  $\mathcal{D}$  be an evolving data set,  $\mathcal{D}_i$  be a *partition* of  $\mathcal{D}$  by a basic time granularity  $\mathcal{T}_i$ ,  $\forall \mathcal{T}_i \in \mathcal{T}$ . Let  $p_j$  be one of the  $n$  parameters. The  $(n+1)$ -dimensional space, denoted by  $\mathcal{E} = \{p_1, \dots, p_n, \mathcal{T}\}$  and called **Evolving Parameter Space (EPS)**, organizes the rules  $\{\mathcal{R}\}^{\mathcal{T}}$  where  $\{\mathcal{R}\}^{\mathcal{T}} = \bigcup_{i=1}^k \{\mathcal{R}\}^{\mathcal{T}_i}$  and  $k$  is the

total number of time partitions of  $\mathcal{D}$ . A temporal association rule  $\mathcal{R}$  is associated with its **temporal parametric locations**  $(\mathcal{R}.value(p_1), \dots, \mathcal{R}.value(p_n))^{\mathcal{T}_i}$  where  $\mathcal{R}.value(p_j)$  denotes the value of the  $j^{\text{th}}$  parameter for rule  $\mathcal{R}$  in time  $\mathcal{T}_i$ .

For simplicity, we use two parameters, namely *support* and *confidence* while others could be defined as well. Thus henceforth, the *EPS*  $\mathcal{E}$  is a 3-dimensional space with *support*, *confidence* and *time* as its dimensions. A *temporal parametric location* depicting a rule  $\mathcal{R}$  in time  $\mathcal{T}_i$ , denoted as  $\mathcal{R}(supp, conf)^{\mathcal{T}_i}$ , is represented as a line segment indicating the parameter values of  $\mathcal{R}$  in  $\mathcal{T}_i$ . Fig. 3 shows the *EPS* for the rules in Table 1(b). Rules  $\mathcal{R}_1, \mathcal{R}_3$  and  $\mathcal{R}_4$  map to the same *temporal parametric location*  $(0.18, 0.5)^{\mathcal{T}_1}$  in the time period  $\mathcal{T}_1$ . However in time  $\mathcal{T}_2$ ,  $\mathcal{R}_1$  travels in the space so that now it maps to same location as  $\mathcal{R}_5(0.11, 0.5)^{\mathcal{T}_2}$ .

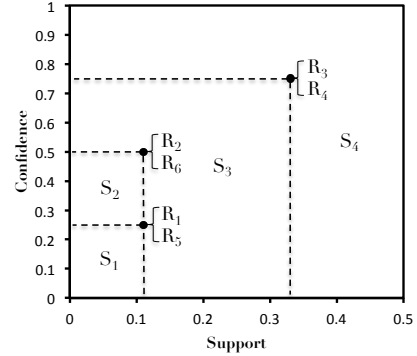


Figure 4: Evolving Parameter Space slice at Time  $\mathcal{T}_2$

**LEMMA 3.1.** Let  $\mathcal{L}$  denote a temporal parametric location in the *EPS*  $\mathcal{E}$ ,  $\mathcal{L}.p_i$  be the value of parameter  $p_i$  for location  $\mathcal{L}$ . Given a set of temporal parametric locations in the same time period  $\mathcal{T}_i$ ,  $\forall \mathcal{L}_m, \mathcal{L}_n \in \{\mathcal{L}\}$ , where  $m \neq n$ , if there exists a  $p_i$  such that  $\mathcal{L}_m.p_i \neq \mathcal{L}_n.p_i$ , then the temporal association rules that map to  $\mathcal{L}_m$  are guaranteed to be distinct from those that map to  $\mathcal{L}_n$ .

**PROOF.** Rules' temporal parametric locations in time  $\mathcal{T}_i$  are generated from the same data partition  $\mathcal{D}_i$ . Any given rule at time  $\mathcal{T}_i$  cannot have two distinct values for one parameter. Therefore, a rule  $\mathcal{R}$  cannot map to two distinct temporal parametric locations within the same time.  $\square$

Each rule's *temporal parametric location* can either remain steady or change over multiple time periods. We call this stream of locations the **trajectory of the association**.

**DEFINITION 4. Trajectory of an association:** Given a sequence of time periods  $\{\mathcal{T}\} = \{\mathcal{T}_1, \dots, \mathcal{T}_m\}$ , the **trajectory of an association**  $\mathcal{R}$  in  $\{\mathcal{T}\}$  is the set of temporal parametric locations that represent its parameter values in the time periods in  $\{\mathcal{T}\}$ .

This trajectory of a rule allows us to compute different measures about the rule that summarize its evolving patterns like *coverage* [16], *stability* [11] and *standard deviation*. These measures can be computed for each individual rule or even for a set of rules to provide individual or global summarization respectively.

Given a data set with  $n$  unique items, the maximum number of rules is finite, bounded by  $3^n - 2^n + 1$  [10]. Therefore, some set of parameter settings must correspond to same set

of rules. Fig. 4 shows a slice of the evolving parameter space for time  $\mathcal{T}_2$ . Rules with identical parameter values are represented by the same point in this space. These points partition the space into 4 regions marked by dashed lines. If a user specified *minimum support* and *confidence* configuration for mining falls into region  $\mathcal{S}_3$ , then regardless of its actual position within the region, the output ruleset is always  $\{\mathcal{R}_3, \mathcal{R}_4\}$ . This observation is inspired by the work presented in [10]. Thus the entire evolving parameter space at a time  $\mathcal{T}_i$  can be partitioned into a finite set of regions referred to as *time-aware stable regions*. This notion of *time-aware stable regions* forms our coarse granularity abstraction of the temporal association rules generated from an evolving data set  $\mathcal{D}$ .

**DEFINITION 5. Time-Aware Stable Regions:** Given an EPS  $\mathcal{E}$  of  $n$  parameters  $\{p_1, \dots, p_n\}$  and times  $\mathcal{T}$  as  $(n+1)$  dimensions for an evolving data set  $\mathcal{D}$ , then a **time-aware stable region** in a time period  $\mathcal{T}_i$  is a closed hyper-box denoted by  $\mathcal{S}_{\mathcal{T}_i}^{\{(upper(p_1), \dots, upper(p_n))\}}$  with its boundary specified by locations  $(\mathcal{S}.upper(p_1), \dots, \mathcal{S}.upper(p_n))^{\mathcal{T}_i}$  and  $\{(\mathcal{S}.lower(p_1), \dots, \mathcal{S}.lower(p_n))^{\mathcal{T}_i}\}$  within each of which no matter how the parameter values are adjusted, the set of rules generated from  $\mathcal{D}_i$  remains unchanged.

Considering the 3-dimensional EPS in Fig. 3, a *time-aware stable region* is bounded by an upper location  $(supp_u, conf_u)^{\mathcal{T}_i}$  and  $k$  lower locations  $\{(supp_{l_j}, conf_{l_j})^{\mathcal{T}_i}\}$  where  $j \in [1, k]$ . The *support* and *confidence* values of the upper location will always be greater than those of all its lower points, i.e.,  $\forall j (supp_u \geq supp_{l_j})$  and  $(conf_u \geq conf_{l_j})$ . The upper location of a *time-aware stable region* is called its **cut location**.

**DEFINITION 6. Cut Location:** Let EPS  $\mathcal{E}$  be a 3-dimensional space with support  $x$ , confidence  $y$  and time  $z$  as its dimensions,  $\{\mathcal{X}\}$  be a set of the intersections formed by the perpendicular projections of each temporal parametric location onto  $x$  and  $y$  planes. The cut locations within  $\mathcal{E}$  are then denoted by  $\{\mathcal{C}\}$ , where  $\{\mathcal{X}\} = \{\mathcal{C}\} \cup \{\mathcal{L}\}$ .

Fig. 3 depicts *time-aware stable regions*  $\mathcal{S}_{\mathcal{T}_1}^{(0.18, 0.5)}$  and  $\mathcal{S}_{\mathcal{T}_2}^{(0.11, 0.5)}$ . For region  $\mathcal{S}_{\mathcal{T}_1}^{(0.18, 0.5)}$ , the *cut location* is  $(0.18, 0.5)^{\mathcal{T}_1}$ . It is bounded by the parametric locations  $(0.18, 0.5)^{\mathcal{T}_1}$  and  $(0, 0.4)^{\mathcal{T}_1}$  and contains rules  $\mathcal{R}_1, \mathcal{R}_3$  and  $\mathcal{R}_4$ .

**LEMMA 3.2.** Given a set of *time-aware stable regions*  $\{\mathcal{S}\}$  for the same  $\mathcal{T}_i$ ,  $\forall \mathcal{S}_m, \mathcal{S}_n \in \{\mathcal{S}\}$ , where  $m \neq n$ , the associations that map to the cut location of  $\mathcal{S}_m$  are guaranteed to be distinct from the ones that map to the cut location of  $\mathcal{S}_n$ .

**PROOF.** By Lemma 3.1, rules generated in the same time period but map to different *temporal parametric locations* are guaranteed to be distinct. The locations in  $\{\mathcal{X}\}$  either belong to  $\{\mathcal{L}\}$  or have no rule. Therefore, within a time period  $\mathcal{T}_i$ , rules that map to different *time-aware stable regions* are guaranteed to be distinct.  $\square$

**DEFINITION 7. Dominating Stable Region:** A *time-aware stable region*  $\mathcal{S}_m$  **dominates** region  $\mathcal{S}_n$  where  $m \neq n$ , if and only if  $\forall p_i \in \mathcal{P} \mathcal{S}_m.C.p_i \leq \mathcal{S}_n.C.p_i$ , and  $\mathcal{S}_m$  and  $\mathcal{S}_n$  are in same  $\mathcal{T}_i$  where  $\mathcal{S}_m.C$  refers to the cut location of stable region  $\mathcal{S}_m$ .

**LEMMA 3.3.** Considering two *time-aware stable regions*  $\mathcal{S}_m$  and  $\mathcal{S}_n$  where  $m \neq n$ . If  $\mathcal{S}_m$  dominates  $\mathcal{S}_n$ , then rules valid within the dominated region  $\mathcal{S}_n$  are also valid in the dominating region  $\mathcal{S}_m$  but not vice versa.

**PROOF.** A temporal rule  $\mathcal{R}_i$  is in the final output ruleset if in the specified  $\mathcal{T}_k$ ,  $\forall p_j, \mathcal{R}_i.value(p_j) \geq \min \text{ parameters}$  where  $p_j \in \{p_1, \dots, p_n\}$ . If  $\mathcal{R}_i$  belongs to region  $\mathcal{S}_n$ , the *temporal parametric location* of  $\mathcal{R}_i$  is the upper location of  $\mathcal{S}_n$ . Because  $\mathcal{S}_m$  dominates  $\mathcal{S}_n$ ,  $\forall p_j, \mathcal{S}_m.upper(p_j) \leq \mathcal{S}_n.upper(p_j)$ , meaning  $\forall p_j, \mathcal{S}_m.upper(p_j) \leq \mathcal{R}_i.value(p_j)$ . So  $\mathcal{R}_i$  is valid in  $\mathcal{S}_m$  as well. However, vice versa is not true, as can be trivially shown.  $\square$

Consider  $\mathcal{S}_1 = \mathcal{S}_{\mathcal{T}_1}^{(0.18, 0.5)}$  and  $\mathcal{S}_2 = \mathcal{S}_{\mathcal{T}_1}^{(0.09, 0.25)}$  in Fig 3. Based on Def. 7,  $\mathcal{S}_2$  dominates  $\mathcal{S}_1$  because every parameter value in the upper location of  $\mathcal{S}_2$  is smaller than the corresponding value of  $\mathcal{S}_1$ .

If the rules in region  $\mathcal{S}_{\mathcal{T}_1}^{(0.09, 0.25)}$  are included in the final result, then region must also contain the rules that are valid in  $\mathcal{S}_{\mathcal{T}_1}^{(0.18, 0.5)}$ . By Lemma 3.3, given a user-specified parameter setting, once a region is identified as a valid region to produce the final ruleset, all its dominated regions should then also be included in the user output.

Using this concept of dominating stable regions [10], each rule is stored once in the stable region and by iterating over its dominating regions the final ruleset can simply be obtained for a given pair of *support* and *confidence* values.

### 3.3 Supported Queries on TARA Model

We now propose TARA operations that offer a rich classes of novel temporal analytical queries.

**Temporal Association Mining.** Given a parameter setting and time periods, Q1 returns the associations that satisfy the minimum parameters, such as *minimum support* and *confidence*. The evolving trajectory and measures of associations for each of the specified time periods are also returned. The Measures including *coverage* [16], *stability* [11] and *standard deviation* summarize the evolving patterns. *Exact match* option returns the associations that are consistently valid in *all* of the specified time periods. *Single match* returns the associations that are valid in exactly one of the specified time periods. *Fuzzy match* returns the associations that are valid in *at least* one or more of the specified time periods. Q2 returns the differences of associations with regards to two different parameter settings.

```
Q1 RETURN Rule, Trajectory, Measures
FROM Evolving Data Set D
PARAMETER  $\bigwedge_{i=1}^n \text{MinParameter}_i = P_i$ 
IN-TIME = Time Periods & Granularity
MATCH = Exact|Single(Time Period)|Fuzzy
```

**Use Case for Temporal Association Mining.** In the retail dataset [3], over the time period of one year broken in windows of a week, Q1 may return the rule (*turkey*  $\implies$  *pumpkin pie*) with relatively low support and confidence value (0.4, 0.5) throughout the year. However, this rule periodically rises to high support and confidence value of (0.6, 0.7) in the week before *Thanksgiving*. If we had examined this rule in the roll-up view over the complete data (whole year), it's support and confidence would overall have been too low. With Q1, an analyst can find the rules that are frequent

only within certain intervals. It also allows to find all time periods during which this rule was more popular.

```

Q2 RETURN Rule, Trajectory, Measures
FROM EPS E
PARAMETER  $\bigwedge_{i=1}^n \text{MinParameter}1_i = P1_i$ 
COMPARE-TO  $\bigwedge_{i=1}^n \text{MinParameter}2_i = P2_i$ 
IN-TIME = Time Periods & Granularity
MATCH = Exact|Single(Time Period)|Fuzzy

```

**Use Case of Unnoticeable Changes.** For a dense data set like webdocs [13], the size of the returned rulesets is huge, making it difficult for the analysts to perform manual inspection of the differences produced by two different parameter settings. To effectively tune the best parameter setting that includes the most important associations across the specified time intervals, analyst would benefit from being able to quickly explore the differences in the results. *Q2* allow to find all rules that were generated in last month by parameter configuration setting like (0.4, 0.8) but not by configuration setting (0.5, 0.95).

**Stable Region Exploration.** These queries provide meta information about the *time-aware stable regions*. In particular, *Q3* returns the sets of stable regions identified for the given parameter settings across the time periods specified by the *match* clause. *Q4* returns region parameters that contain the specified associations.

```

Q3 RETURN Stable Region
FROM EPS E
PARAMETER  $\bigwedge_{i=1}^n \text{MinParameter}_i = P_i$ 
IN-TIME = Time Periods & Granularity
MATCH = Exact|Single(Time Period)|Fuzzy

```

**Use Case of Parameter Recommendation.** For a sparse dataset like retail [3], often despite submitting several successive mining requests with different parameter settings, the system repeatedly returns the same set of rules due to a sparse distribution of rules. This trial and error process can be avoided by using query *Q3*. For example for the retail dataset [3] broken in windows of a week, *Q3* can easily inform an analyst that during the last week of a month, the same set of rules will be generated for all parameter configurations within (0.88, 0.76) and (0.91, 0.78).

```

Q4 RETURN Region
FROM Ruleset  $\bigcup_{i=1}^n R_i$ 
IN-TIME = Time Periods & Granularity
MATCH = Exact|Single(Time Period)|Fuzzy

```

**Time Specification: Roll-up and Drill-down.** For each of the above queries, a time granularity must be specified. For example, a time period  $\mathcal{T}_i$  can refer to a particular *hour*, a *day*, or a *week*.

**Rule Search.** As [12] pointed out, supporting content-based rule search can leverage analysts’ domain knowledge to efficiently narrow down the result into a more manageable smaller set. *Q5* allows analysts to filter a ruleset, generated by any of the above queries, based on the absence or presence of certain *items* given in the *MATCH* clause.

```

Q5 RETURN Rule
FROM Ruleset  $\bigcup_{i=1}^n R_i$ 
MATCH  $\{I\}^+ \cup \{I\}^-$ 

```

**Use Case of Content-based Association Exploration.** In the retail dataset using query *Q1* with *Single match* option and the time equal to week before *Christmas* returns a huge number of rules for the support and confidence values of (0.4, 0.5). With *Q5* an analyst can quickly find all the rules that contains “*Ipads*” and do not include “*Laptops*”.

## 4. OFF-LINE EPS-INDEX CONSTRUCTION

Our proposed *Evolving Parameter Space* (EPS) Index construction is composed of three tasks that are performed off-line. Task 1 generates the temporal association rules; task 2 computes the *time-aware stable regions* and constructs the domination graph, and task 3 constructs the *EPS-Index*. These tasks three are explained below (See Algo. 1).

**Task 1: Temporal Association Rule Generation.** **TARA** first pre-generates the rules from the evolving dataset using the lowest meaningful parameter settings, called the *primary support*  $\theta$  and the *confidence*  $\lambda$ , to prevent excess pre-generation [12]. Based on the *minimum time granularity*, **TARA** mines the rules from the current window, whose *support* and *confidence* value exceed  $\theta$  and  $\lambda$ . The rules and their parameter values w.r.t to the transactions within this window are then archived (See Sec. 5). In our work, we plug in FP-Tree [7] as the rule generation algorithm. The rule mining module in **TARA** framework is extendable to any incremental or parallel rule mining solution [8].

**Task 2: Time-Aware Stable Region.** To construct the *time-aware stable regions* (see Def. 5), **TARA** first computes the *cut locations* using a two step approach. In the first step, the *temporal parametric locations* of rules generated in the previous task are initialized as the first set of *cut locations*. Each *cut location* maintains a set of references to its rules. In the second step, let  $x$ ,  $y$ , and  $z$  be the axes representing *support*, *confidence* and *time* measures and  $o$  be the origin of all measures. The intersections formed by the perpendicular projections of each point onto the  $x$  and  $y$  planes are added to the *cut location* set.

By Lemma 3.2, each *cut location* identifies a unique *time-aware stable region*. For each *cut location*, the lower bounds are computed to form a complete *time-aware stable region*. Simultaneously, the *immediate dominated regions* are identified and connected to construct the domination graph. Upon user request, the *time-aware stable regions* can be constructed for the coarser time granularity over basic granularity used by the system. New parameter values of rules forming these *time-aware stable regions* are computed by Formulas 4 & 5.

**Task 3: EPS-Index Construction.** Within each *time-aware stable region*, all rules are indexed by their respective attributes, called a *region shard*. Next, the two layered *Evolving Parameter Space Index* (*EPS-Index*) is created to efficiently answer the **TARA** model requests. The top level of the *EPS-Index* facilitates the search to locate a particular *time-aware stable region* given its input parameters. Regions are indexed separately by different time periods. For regions within a single time period, a grid-based spatial index is utilized to partition the EPS into equal-sized grid cells. The time-aware stable regions are then allocated to their respective positions in the grid. The stable regions in

---

**Algorithm 1: Offline EPS Construction**


---

**Input:** Dataset  $\mathcal{D}$   
**begin**  
  for each  $\mathcal{D}_i$  from  $\mathcal{D}$  do  
     $\{\mathcal{R}\} \leftarrow \emptyset$ ;  $\{\mathcal{S}\} \leftarrow \emptyset$ ;  
     $\{\mathcal{R}\} \leftarrow \text{RuleGenerator}(\mathcal{D}_i, \theta, \lambda)$ ;  
     $\{\mathcal{S}\} \leftarrow \text{RegionAbstractor}(\{\mathcal{R}\})$ ;  
     $\mathcal{E} \leftarrow \text{EPSIndexConstructor}(\{\mathcal{S}^+\}, \{\mathcal{R}\})$ ;

---

**1.A: RuleGenerator**  
**Input:**  $\mathcal{D}_i, \theta, \lambda$   
**Output:** RuleSet  $\{\mathcal{R}\}$   
**begin**  
   $\{\mathcal{R}\} \leftarrow \emptyset$ ;  
   $\{\mathcal{R}\} \leftarrow \text{FP-Tree}(\mathcal{D}_i, \theta, \lambda)$ ;  
  Archive( $\{\mathcal{R}\}$ );

---

**1.B: RegionAbstractor**  
**Input:**  $\{\mathcal{R}\}$   
**Output:** RegionSet  $\{\mathcal{S}\}$   
**begin**  
   $\{\mathcal{C}\} \leftarrow \emptyset$ ; /\*Initial cut location set\*/  
   $\{\mathcal{S}\} \leftarrow \emptyset$ ;  
   $\{\mathcal{C}\} \leftarrow \text{GetCutLocation}(\{\mathcal{R}\})$ ;  
  for each  $C_i \in \{\mathcal{C}\}$  do  
     $S \leftarrow \text{ConstructShard}(C_i)$ ;  
     $S \leftarrow \text{GetRegion}(C_i, \{C\})$ ;  
     $S \leftarrow \text{ConnectDominatedRegion}(S, \{C\})$ ;  
   $\{\mathcal{S}\} \leftarrow \{\mathcal{S}\} \cup S$ ;

---

**1.C: EPSIndexConstructor**  
**Input:**  $\{\mathcal{S}\}$   
**Output:** EPS-Index  $\mathcal{E}$   
**begin**  
  GridIndexer( $\{\mathcal{S}\}$ );

---

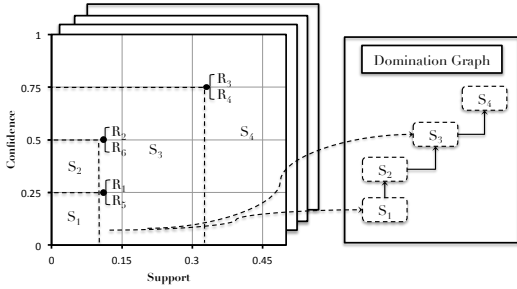


Figure 5: The EPS-Index

each cell point to the corresponding nodes in the next level of the *EPS-Index*. Fig. 5 shows the spatial index of the *time-aware stable regions* for  $\mathcal{T}_1$ .

Using the proposed grid structure, the online search for a stable region can be performed in near constant time. The online processing of our **TARA** exploration using this index is described in Sec. 7.

The second level of the *EPS-Index*, namely, the region domination graph (Fig. 5), is designed to expedite the collection of rules from dominating regions [10]. Each stable region forms a node in the graph with each node linked to its closest dominating neighbors. The region domination graph enables us to locate the closest dominating neighbor regions in near constant time and also produce complete rule sets in linear time in the number of rules involved.

## 5. TEMPORAL ASSOCIATION ARCHIVE

As explained before, the rules themselves can be huge. We describe an efficient storage structure for managing the rules generated across time called temporal association rule archive (*TAR Archive*).

## 5.1 TAR Archive Design

Our proposed archive structure consists of a directory and a number of index entries. The directory contains all temporal association rules. Each rule in the directory has a pointer to its index entry. The index entry stores the history of its parameters w.r.t a specific time granularity. The index entry can be implemented using a *time sequence* [6] as illustrated in Fig. 6. For simplicity only one parameter is shown. The size of the *time sequence* is equal to the number of basic windows. The advantage of using such naive *time sequence* is that every element in this structure maps to a parameter value of the rule within a specific window. Given a window and rule id, the search of the parameter value takes only  $O(1)$  time.

Rule Directory	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{T}_4$
$R_1$	0.5	0.5	0.5	0.6
$R_2$	0.4	0.3	0.3	0.3
$R_3$	0.6	0.6	0.4	0.5
$R_4$	0.5	0.3	0.2	0.3
$R_5$	0.7	0.0	0.0	0.0
$R_6$	0.0	0.0	0.0	0.8

Figure 6: Rule Index with Time Sequence

Interestingly, [15] observed that daily updates in transaction databases more often than not affect only a small part of the ruleset. [6] also indicates that parameter values of the frequent patterns mined from the data stream often remain stable within a period of time. In Fig. 6,  $\mathcal{R}_1, \mathcal{R}_2$  and  $\mathcal{R}_3$  are stable over several consecutive windows. Therefore, the time sequence structure may contain redundant values.

Rule Directory	Binary Encoding	Compressed Entry
$R_1$	1001(9)	0.5 0.6
$R_2$	11(3)	0.4 0.3
$R_3$	1101(13)	0.6 0.4 0.5
$R_4$	1111(15)	0.5 0.3 0.2 0.3
$R_5$	11(3)	0.7 0.0
$R_6$	1001(9)	0.0 0.8

Figure 7: Rule Index with Compact Time Sequence

To avoid the above problems while still achieving efficient access, we propose the *TAR Archive* with *compact time sequence* structure as depicted in Fig. 7. Each *compact time sequence* for one parameter consists of a binary code  $\mathcal{B}$  and an array of distinct parameter values, called *value sequence*, denoted as  $\mathcal{V} = \{v_1, \dots, v_n\}$ . The  $i$ th bit from the lowest order indicates whether or not the parameter value within the  $i$ th window is identical to the value within the previous window. The encoding strategy is shown in Formula 3.

$$\text{Bit}(\mathcal{R}.p_i^j) = \begin{cases} 0 & \text{if } \mathcal{R}.p_i^j = \mathcal{R}.p_i^{j-1} \\ 1 & \text{otherwise} \end{cases} \quad (3)$$

For example, in Fig. 6 the parameter values of  $R_1$  are stable in the first three windows with only the value in the latest window being different. “1001” denotes the evolution of this parameter across 4 windows. The lowest order digit represents the parameter value in the oldest window and the highest order digit represents the parameter value in the newest window. In this case, the parameter values in  $\mathcal{T}_1, \mathcal{T}_2$

and  $\mathcal{T}_3$  only need to be stored once in the structure because they are identical.

## 5.2 TAR Archive Operations

Next we discuss the supported operations, namely *append*, *access*, *purge* and *merge* on the *TAR Archive*.

**Append:** As the new window is being processed, rules are added into the *TAR Archive*. We distinguish between three cases when a rule is being appended: (1) The new rule already exists in the archive; (2) The new rule does not exist in the archive; (3) A rule that is in the archive does no longer appear in the newly generated ruleset.

For case 1, if the new value of the rule is different than in the last window, then “1” is added to the binary code otherwise no action is needed. To address case 2, first, this new rule is inserted in the rule directory. Second, for each parameter, if the current window is the very first window, the binary code is set to “1” and  $p_i^0$  is inserted into the empty *value sequence*. If  $j > 0$ , the binary code is first initialized as “1” and “0.0” is inserted into the value sequence. Then the procedure in case 1 is followed to process the new information. To address case 3, the system simply appends the parameter value “0.0” to all parameters of such rule. The append procedure is the same as the procedure for case 1.

**Access:** Our decoding strategy for *compact time sequence* structure allows to search *TAR Archive* takes  $O(1)$  time. Given a window id  $j$ , the parameter value  $p_i^j$  from  $\mathcal{V}_i$  is retrieved. For instance, in Fig. 7, for  $\mathcal{R}_1$ , if  $j = 3$ , the system needs to locate the correct value  $p^3$  in the value sequence. In  $B_i$ , the count of “1” is the length of the value sequence because every time a bit is set to “1”, a new value is then appended to the *value sequence*. Finding the offset of the element in the sequence value corresponding to the queried window  $j$  is equivalent to counting “1”s up until the  $j$ th bit in  $B_i$ . For the previous example, if  $j = 3$ , the binary code up until 3rd bit in  $B$  is “001” which only has 1 bit set to “1”. So  $s_1$  in the value sequence represents the value of  $p^3$ . Given a  $j$ , we calculate the offset of the element in the *value sequence* as follows  $Offset(j) = HammingWeight(B \text{ AND } (2^j - 1))$ . HammingWeight is a  $O(1)$  algorithm for bit counting.

**Purge:** As parameter values begin to accumulate, the size of the archive grows bigger and bigger. For some applications, historical information inserted at the very beginning may become insignificant. Therefore, an operation that purges such data is necessary. Specifically, the operator deletes the parameter values of the entire ruleset in the archive w.r.t a set of consecutive windows from  $\mathcal{T}_1$  to  $\mathcal{T}_k$  where  $k \leq total \text{ number of windows}$ . For each rule  $R$ , the purge operation performs a two-step procedure: (1) Delete the values that correspond to  $\langle \mathcal{D}_1, \dots, \mathcal{D}_k \rangle$  for each parameter from the *value sequence*. (2) Update the binary code so that it reflects the changes of the parameter within the rest of the windows. For an input  $k$ , if  $Offset(k)$  equals to  $Offset(k+1)$  which means the values of a parameter in  $\mathcal{D}_k$  and  $\mathcal{D}_{k+1}$  are mapped to the same  $s_i$  in  $S$ , then  $\{s_j | 1 \leq j \leq i-1\}$  are removed from  $S$ . The corresponding  $B$  is right shifted  $i-1$  times and the first bit of the modified  $B'$  is set to “1”. If the results of  $Offset(k)$  and  $Offset(k+1)$  are not identical, then  $\{s_j | 1 \leq j \leq i, i = Offset(k)\}$  are removed from  $S$ . The corresponding  $B$  vector is right shifted  $i$  times.

**Merge:** For some applications, analysts may be interested in recent changes at a fine granularity, but longer term changes at a coarse granularity. To support such time-sensitive roll-

up, also called *Natural tilted-time window* or *logarithmic tilted-time window* [6] an efficient merge operation is needed for the *TAR Archive*.

$\{\mathcal{R}\}_i$  denotes the ruleset generated from  $\mathcal{D}_i$ . The merge operation unions the rulesets,  $\bigcup_{w=i}^j \{\mathcal{R}\}_w$ , that correspond to the set of consecutive windows  $[\mathcal{T}_i, \mathcal{T}_j]$  and computes the new parameter value for each rule in the merged window.  $[\mathcal{T}_i, \mathcal{T}_j]$  denotes a set of consecutive windows that are requested to be merged,  $|\mathcal{D}_j|$  denotes the total number of transactions in  $\mathcal{T}_j$ . The *support* and *confidence* of a rule in the merged window is computed as follows:

$$Support(\mathcal{R}) = \frac{\sum_{w=i}^j |\mathcal{D}_w| \times \mathcal{R}.supp^w}{\sum_{w=i}^j |\mathcal{D}_w|} \quad (4)$$

$$Confidence(\mathcal{R}) = \frac{\sum_{w=i}^j |\mathcal{D}_w| \times \mathcal{R}.supp^w}{\sum_{w=i}^j |\mathcal{D}_w| \times \frac{\mathcal{R}.supp^w}{\mathcal{R}.conf^w}} \quad (5)$$

Based on the definition of *support* and *confidence*, Formula 4 describes the proportion of the transactions within the merged window that *contains* all items of the association; Formula 5 describes the ratio of the number of the transactions that *contains* all items of the association to the number of the transactions that *contains* antecedent items in the merged window. With these new values, the merge operator then further updates both the binary encodings  $\mathcal{B}_i$  and value sequences  $\mathcal{V}_i$  in the rule entry. For a particular parameter of a rule, if the windows from  $i$  to  $j$  are merged, then the bits from  $i$ th to  $j$ th position in its binary code are also merged into one bit. The value of this new bit and  $(i+1)$ th bit depends on whether the represented parameter value is different from its last parameter value. For example, in Fig. 7 when  $\mathcal{T}_2$  and  $\mathcal{T}_3$  are requested to be merged, the new binary code becomes “11” and value is 0.3.

## 6. ROLL-UP SUPPORT ACROSS TIME

In the rule generation step, rules with parameter values below minimal system thresholds are not maintained. Therefore, we may not have the exact parameter values for each rule when multiple windows are merged. The parameter value of a rule in a coarser time period (merged window) is computed based on its parameter values in all periods that compose the window. The calculation is described in Formula 4 & 5. For example, let  $\theta$  be 0.1, support values of  $\mathcal{R}$  in  $\mathcal{T}_1$  and  $\mathcal{T}_2$  be 0.2 and 0.08 respectively. If  $\mathcal{T}_1$  and  $\mathcal{T}_2$  are requested to be merged, the exact *support* of  $R$  in the new window is  $\frac{|\mathcal{D}_1| \times 0.2 + |\mathcal{D}_2| \times 0.08}{|\mathcal{D}_1| + |\mathcal{D}_2|}$ . However, let us assume that  $R$  had been withdrawn from  $\mathcal{T}_2$  because  $0.08 < \theta$ . Therefore the *support* value of  $R$  in  $\mathcal{T}_2$  becomes unknown and thus is treated as 0. As a result, we no longer have the exact parameter value of rules in a merged coarser-time-granularity window, rather only an approximate value. Rules withdrawn from a window may fall into 3 cases depending on whether they either fail to satisfy one of the system thresholds or both. The margin of error of the parameters value in merged windows varies case by case. Since the withdrawn rules are absent from the system, the exact reason of the withdrawal becomes unknown. Therefore, we introduce the worst sce-



nario below:

Let  $[\mathcal{T}_i, \mathcal{T}_j]$  be a set of consecutive windows that are requested to be merged,  $\mathcal{R}$  be a rule that at least appears once in any of these windows,  $\mathcal{T}'$  be the windows in which  $R$  is absent,  $\mathcal{T}''$  be the windows in which  $\mathcal{R}$  appears,  $S(\mathcal{R})$  and  $C(\mathcal{R})$  be the approximated *support* and *confidence* of  $\mathcal{R}$  calculated based on Formulas 4&5,  $\hat{S}(\mathcal{R})$  and  $\hat{C}(\mathcal{R})$  be the exact *support* and *confidence* of  $\mathcal{R}$ . The worst scenario arises where all absents of  $\mathcal{R}$  are caused by  $\mathcal{R}.supp \geq \theta$  and  $\mathcal{R}.conf < \lambda$ .

In this case,  $\mathcal{R}$  has a qualified *support*. In order to have a *confidence* value smaller than the threshold,  $\mathcal{R}$ 's *support* must be larger than  $\theta$ , however, smaller than  $\lambda$ . Therefore,

$$S(\mathcal{R}) \leq \hat{S}(\mathcal{R}) \leq \frac{\sum_{k=1}^m |\mathcal{D}'_k| \times \lambda + \sum_{k=1}^n |\mathcal{D}''_k| \times \mathcal{R}.supp_{\mathcal{D}''_k}}{\sum_{w=i}^j |\mathcal{D}_w|} \quad (6)$$

The error in confidence is caused by miss counting the withdrawn *support* of  $R$ , as well as miss counting of the withdrawn *support* of the antecedent of  $R$ . The *confidence* with maximum margin of errors is the following:

$$margin = \frac{\sum_{k=1}^m |\mathcal{D}'_k| \times \lambda + \sum_{k=1}^n |\mathcal{D}''_k| \times \mathcal{R}.supp_{\mathcal{D}''_k}}{\sum_{k=1}^m |\mathcal{D}'_k| + \sum_{k=1}^n |\mathcal{D}''_k| \times \frac{\mathcal{R}.supp_{\mathcal{D}''_k}}{\mathcal{R}.conf_{\mathcal{D}''_k}}} \quad (7)$$

Note that this value does not guarantee to be smaller or larger than  $\hat{C}(R)$ , because the errors are both introduced in the numerator and denominator.

Formula 6 gives the worst scenario for the approximation of *support* and Formula 7 the worst scenario for the approximation of *confidence*. The smaller  $\theta$  and  $\lambda$  are, the more accurate our approximation will be. Because with a smaller system threshold, less rules would be withdrawn from the window reducing the chance of miss counting.

## 7. ONLINE QUERY PROCESSING

In this section, we explain how the analytical TARA queries introduced in Section 3 are handled by the **TARA** framework (Algo. 2). Depending upon the query type, the appropriate subroutine is invoked. Q1, Q2 and Q3 are handled by subroutines 2.A, 2.B, 2.C respectively. For Q4 once the ruleset is selected by any of the subroutines, routine 2.C is used to return the stable region.

The **response time** for query processing mainly consists of 3 components, namely,  $Cost(SearchRegion)$ ,  $Cost(GetRule)$  and  $Cost(GetDominatedRegions)$ . The TARA storage structures namely *EPS-Index* and *TAR Archive* are in-memory structures. The cost for a region search against *EPS-Index* is  $\mathcal{O}(1)$ . As illustrated in Fig. 5, locating the regions within the same time interval takes  $\mathcal{O}(1)$ . By converting input parameters into offsets, the appropriate cell can be found in  $\mathcal{O}(1)$  as well. Thus  $Cost(SearchRegion) = \mathcal{O}(1)$ . Note that the system must generate the stable regions for such time periods in which they don't exist. To iteratively collect all the immediate dominated regions in the domination graph, a breadth-first search (BFS) is required starting at the node containing  $(minsupp, minconf)$  in  $\mathcal{T}$ . The time complexity of BFS is  $\mathcal{O}(|V| + |E|)$ . In our case,  $E \leq (2 \times V)$  as each vertex has a fanout of at most two edges. Thus,

---

### Algorithm 2: Online Temporal Association Exploration

---

#### 2.A: Temporal Association Mining Query

**Input:**  $s, c, \{\mathcal{T}\}, matchtype$

**Output:** Ruleset

```

begin
  {R} ← ∅; {S} ← ∅;
  switch matchtype do
    case single(T)
      {S} ← SearchRegion(s, c, T);
      {S} ← {S} ∪ GetDominatedRegion({S});
    case exact
      T ← any T ∈ {T};
      {S} ← SearchRegion(s, c, T);
      {S} ← {S} ∪ GetDominatedRegion({S});
    case fuzzy
      for each Ti ∈ {T} do
        {S} ← SearchRegion(s, c, Ti);
        {S} ← {S} ∪ GetDominatedRegion({S});
  {R} ← GetRule({S});

```

---

#### 2.B Ruleset Comparison Query

**Input:**  $s_1, c_1, s_2, c_2, \mathcal{T}, matchtype$

**Output:** Rulesets  $\{\mathcal{R}\}_1, \{\mathcal{R}\}_2$

```

begin
  sc ← max(s1, s2); cc ← max(c1, c2);
  {S}1 ← SearchRegion(s1, c1, T);
  {S}2 ← SearchRegion(s2, c2, T);
  //Collect the regions till it reaches a parameter value
  for i = 1; i ≤ 2; i++ do
    if si ≠ sc then
      {S}i ← GetDominatedRegion(Si, sc);
    if ci ≠ cc then
      {S}i ← GetDominatedRegion(Si, cc);
  {R}1 ← GetRule({S}1);
  {R}2 ← GetRule({S}2);

```

---

#### 2.C Stable Region Query

**Input:**  $s, c, \{\mathcal{T}\}, matchtype$

**Output:** Region S

```

begin
  {S} ← ∅;
  switch matchtype do
    case single(T)
      S ← SearchRegion(s, c, T);
    case exact or fuzzy
      for each Ti ∈ {T} do
        {S} ← SearchRegion(s, c, Ti);

```

---

$Cost(GetDominatedRegions) = \mathcal{O}(|V|)$ . If the trajectory of a rule cannot be obtained from the retrieved regions (*exact* or *single* match), *GetRule* searches the parameter values of the rule in the specified periods in the *TAR Archive*. If the granularity of the specified time periods is coarser than the ones available in the archive, *GetRule* finds the windows that are contained in coarser time periods and computes the parameter value for the merged window. See Sec. 5 for the complexity analysis of accessing the *TAR Archive* and the computation for obtaining the parameter value in the merged window.

## 8. EXPERIMENTAL EVALUATION

**Experimental Setup.** Experiments are conducted on a OS X machine with 2.4 GHz Intel Core i5 processor and 8 GB RAM. The system and its competitors are implemented in C++ using Qt Creator with Clang 64-bit compiler.

**Datasets.** We select a variety of datasets with diverse characteristics here. The benchmark datasets, *T5kL50N100* and *T2kL100N1k*, are generated by the *IBM Quest data generator* [1] modeling transactions in a retail store. We

Table 2: Datasets

	<i>100retail</i>	<i>T5k</i>	<i>T2k</i>	<i>webdocs</i>
Transactions	8,816,200	5,000,000	2,000,000	1,692,082
Unique Items	16,470	23,870	30,551	5,267,656
Avg Len of Tran	10	50	100	177
Size	416.8 MB	1.48 GB	1.38 GB	1.48 GB

Table 3: Thresholds for Indexes

Dataset	H-Mine	TARA&PARAS (supp, conf)
retail	0.0002	(0.0002, 0.1)
T5k	0.0012	(0.0012, 0.2)
T2k	0.001	(0.001, 0.2)
webdocs	0.1123	(0.1123, 0.2)

partition these datasets into 5 equal-sized batches to form the evolving data sources. The *retail* dataset [3] contains 88,163 transactions collected from a Belgian retail supermarket store in 5 months. To study scalability, we replicate this *retail* dataset 100 times. The *webdocs* dataset [13] is built from a spidered collection of web html documents. Both of these real datasets are partitioned into 10 equal-sized batches to form evolving data sources. The statistics of the datasets are summarized in Tab. 2.

**Alternate State-of-the-art Techniques.** The performance of **TARA** is compared against three competitors. **DCTAR** [9] derives the ruleset directly from the raw data given a parameter configuration. It computes the associations from scratch whenever a new batch of data arrives. **H-Mine** [18] instead pregenerates the intermediate frequent item sets offline. For specific parameter settings, the algorithm utilizes the itemsets to generate the associations online instead of extracting them from the raw data. **PARAS** [10] pregenerates frequent itemsets and rules offline for the entire data set assuming all data is static and given apriori. That is, time is ignored. For our experiments, we construct the PARAS index for a single time period. However at online time if request comes for different periods it then generates the associations from scratch.

**Experimental Methodologies:** The performance of our approach and state-of-the-art algorithms is measured by:

**Offline Preprocessing Time.** We measure the single and multiple data batches preprocessing time for **TARA**, **H-Mine** and **PARAS**. Since **DCTAR** does not involve any preprocessing, it is excluded from this measurement.

**Online Processing Time.** We measure the online processing time for a query averaged over multiple runs (explained in Sec 8.2) to evaluate the speedup.

**Size of Regenerated Information.** We compare the sizes of the preprocessed information. **DCTAR** is again excluded. The size of the tree structure in **H-Mine** and the size of the *TAR Archive* in **TARA** are thus compared.

## 8.1 Evaluation of Preprocessing Time

We first compare the preprocessing times for **H-Mine**, **PARAS** and **TARA**. In the offline step, as the window slides, **H-Mine** (1) precomputes the frequent item sets and (2) stores them along with their associated *support* value into a tree structure. Whereas **TARA** (1) precomputes the frequent item sets, (2) derives the ruleset, (3) archives them along with the associated *support* and *confidence* values and (4) updates the *EPS-Index*. **PARAS** proceeds with the same process as **TARA** except that it does not utilize the archive nor does it keep the pregenerated information from the previous windows. Therefore, the total preprocessing time of **PARAS** is similar to **TARA** except for the archival time.

Fig. 10 compares the preprocessing time of **H-Mine** and **TARA** for all windows for the *retail*, *T5kL50N100*, *T2kL100N1k* datasets, with the system threshold settings summarized in Tab. 3. As shown, frequent item set generation occupies a relatively large portion of the preprocessing time as compared to other tasks. This confirms prior works [7] that rule generation is more efficient compared to frequent itemset generation. Overall, the additional preprocessing tasks in **TARA** require no more than 20% extra time than **H-Mine**. This extra time gives significant advantage to **TARA** in terms of truly interactive online performance and support of many advanced exploration operations.

## 8.2 Evaluation of Online Processing Time

Next, we compare the online processing times (y-axis in log scale) for our proposed operations. The user-specified parameters, namely *minsupp*, *minconf* and *time periods*, are varied. The examined queries fall into two categories: (1) Rule trajectory and parameter recommendation queries and (2) Ruleset comparison queries. In the first experiment, we test the performance of **TARA** against the three competitors using several query types, namely *Q1* and *Q3* in *single match* mode. Second, we use *Q2* in *exact match* mode to test the performance of **TARA** against others. We choose *Q1*, *Q2* and *Q3* because they cover the major exploration operations and subroutines in the online processing phase.

### 8.2.1 Trajectory and Parameter Recommendation

To process *Q1*, the system needs to find the rules that satisfy *minsupp* and *conf* in a single *time period* and examine their parameter values in other specified time periods. For **DCTAR**, it mines the rules from the transactions that fall into the last window and examines their parameter values by processing the transactions that fall into the 3 previous windows. For **PARAS**, the process is identical except that the rules are retrieved from the **PARAS** index built based upon the newest window. For **H-Mine**, the rules are derived and examined by using its item set index.

**Impact of Varying Support and Confidence.** To determine the effect of *minsupp*, we conduct several experiments by fixing *minconf* to a constant value and varying the *minsupp* value. Fig. 8 illustrate the query processing times for *retail*, *T5kL50N100*, *T2kL100N1k* and *webdocs* datasets with fixed *minconf* 0.4, 0.2, 0.2 and 0.4, respectively.

We observe that, **TARA** consistently outperforms **DCTAR** and **PARAS** by 6,7,7 and 5 orders and **H-Mine** by 3, 4, 4 and 4 orders of magnitude for *retail*, *T5kL50N100*, *T2kL100N1k* and *webdocs* datasets respectively. **TARA-S** stands for the implementation of **TARA** with the rule index inside each *time sensitive stable region* to support content based exploration (*Q5*). The merging of indexes when *dominated regions* are being collected incurs extra costs as compared to the **TARA** system without these rule indexes. Especially when the number of rules in the result is small, this extra cost results in similar or slower response time compared to **H-Mine** as shown in Figs. 8(b) and (c). The reason of the fast response of **TARA** is that it prepares sufficient amount of information in the offline stage, so that answering such queries is simply about searching the **TARA** index.

**TARA-R** shows the response time of returning the *time-sensitive stable region* which answers *Q3*. Since **PARAS** always builds the index for the latest window, in this particular experiment, it achieves the same response time as

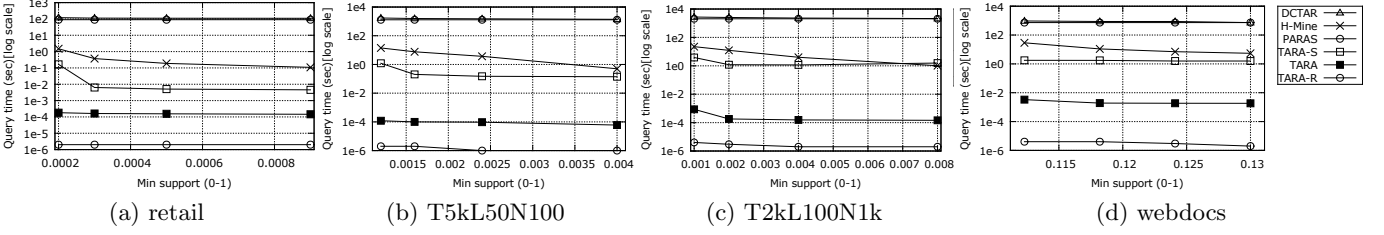


Figure 8: Rule Trajectory and Parameter Recommendation: Varying Support

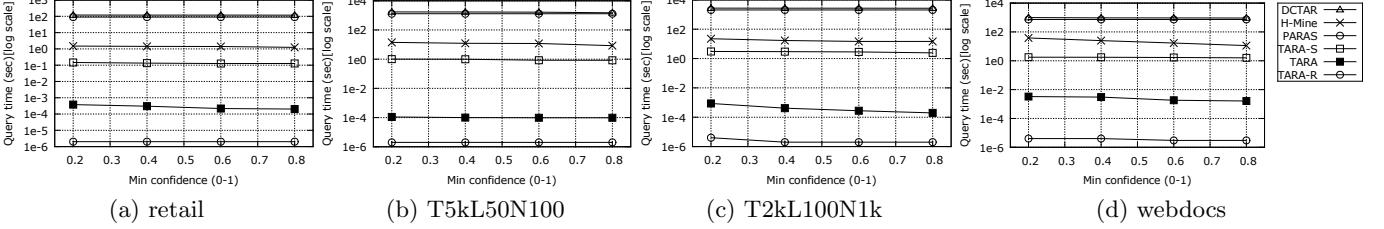


Figure 9: Rule Trajectory and Parameter Recommendation: Varying Confidence

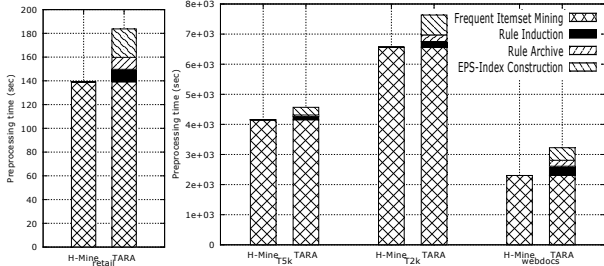


Figure 10: Preprocessing Time

**TARA** because only regions that fall into the latest window are considered. All other systems are not capable of answering  $Q3$ . That is, using DCTAR and H-Mine, an analyst would need to generate all possible rules in the specified time period and then investigate all to find the answer.

**Impact of Varying Confidence.** Next, we fix the  $minsupp$  to a constant value and vary the  $minconf$  value. Fig. 9 illustrates the query processing times for *retail*, *T5kL50N100*, *T2kL100N1k* and *webdocs* datasets with fixed  $minsupp$  0.0002, 0.0012, 0.0012 and 0.1123, respectively. Overall, both **TARA** and **TARA-S** consistently perform several orders of magnitude better than the three competitors.

### 8.2.2 Ruleset Comparison Queries

$Q2$  returns the differences of the rulesets w.r.t two parameter settings that share the same time specification. In this particular experiment, the query is configured with the *exact match* mode. It returns the differences of two parameter setting across 4 windows. Since the DCTAR and H-Mine do not support such query, we implement a subroutine in their rule derivation module to determine if the parameter value of the rule satisfies one setting but not the other. This subroutine is optimized so that it does not generate the overlapping ruleset w.r.t 2 different settings. In this experiment, we either fix  $minsupp$  or  $minconf$  and vary the other one.

**Impact of Varying  $2^{nd}$  Support.** Fig. 11 illustrates the query processing times for *retail*, *T5kL50N100*, *T2kL100N1k* and *webdocs* datasets. The fixed  $min$  parameters for these datasets are ( $minsupp_1$ ,  $minconf_1$ ,  $minsupp_2$ ): (0.0002, 0.4, 0.4), (0.0012, 0.2, 0.2), (0.0012, 0.2, 0.2) and (0.1123, 0.4, 0.4), respectively. The query processing times increase

with an increase in the  $minsupp$  because the increase of the deviation from  $minsupp_1$  to  $minsupp_2$  results in larger differences between the two parameter settings. In particular, **TARA** outperforms DCTAR and PARAS by 6,7,6 and 6 orders, H-Mine by 4, 5, 4 and 4 orders for *retail*, *T5kL50N100*, *T2kL100N1k* and *webdocs* datasets, respectively.

**Impact of Varying  $2^{nd}$  Confidence.** Fig. 12 illustrates the query processing times for *retail*, *T5kL50N100*, *T2kL100N1k* and *webdocs* datasets. The fixed  $min$  parameters for these four datasets are ( $minsupp_1$ ,  $minconf_1$ ,  $minsupp_2$ ): (0.0002, 0.4, 0.0002), (0.0012, 0.2, 0.0012), (0.0012, 0.2, 0.0012) and (0.1123, 0.4, 0.1123), respectively. **TARA** consistently performed several orders of magnitude better than the three competitors.

### 8.3 Evaluation of Archive Size

We compare the sizes of the pregenerated information in **TARA**, H-Mine and PARAS. For H-Mine, the size of the structure is determined by the number of frequent item sets times the number of processed partitions, while the size of pre-stored information in **TARA** is determined by the size of the *TAR Archive*. PARAS only pregenerates the association in a single window. Its maximum size is  $3^n - 2^n + 1$  where  $n$  is the unique items in that particular window. The actual index sizes can be estimated by multiplying the number of instances with the average space required per instance.

Fig 13 shows the size of the H-Mine Index, *TAR Archive* and the actual number of uncompressed rule parameter values for our four datasets with the system threshold settings summarized in Tab. 3. As **TARA** pre-generates rules instead of only the item sets, the size of the *TAR Archive* is larger than the H-Mine index. However, our encoding technique achieves favorable compression as compared to uncompressed rule parameter values.

## 9. RELATED WORK

**Temporal association mining.** Adding the time dimension in the context of association rules was first mentioned in [14]. However, while more follow-on works [6, 9, 18] improve the efficiency of temporal association mining by maintaining intermediate frequent item sets, all of these approaches require the user to input a specific parameter setting. This one-at-a-time approach not only limits efficiency,

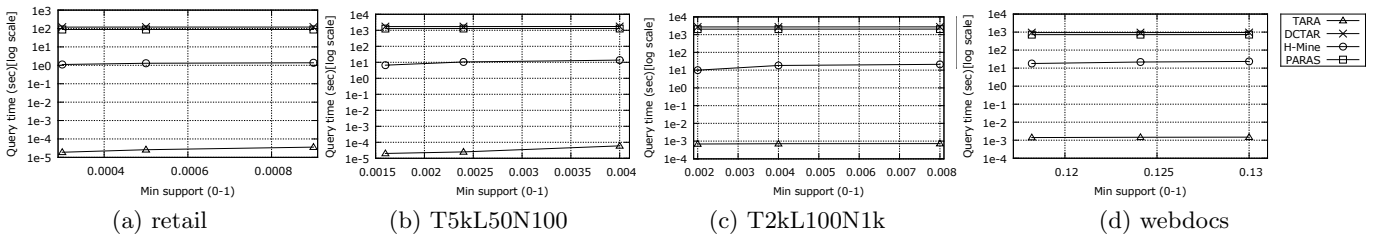


Figure 11: Ruleset Comparison: Varying 2<sup>nd</sup> Support

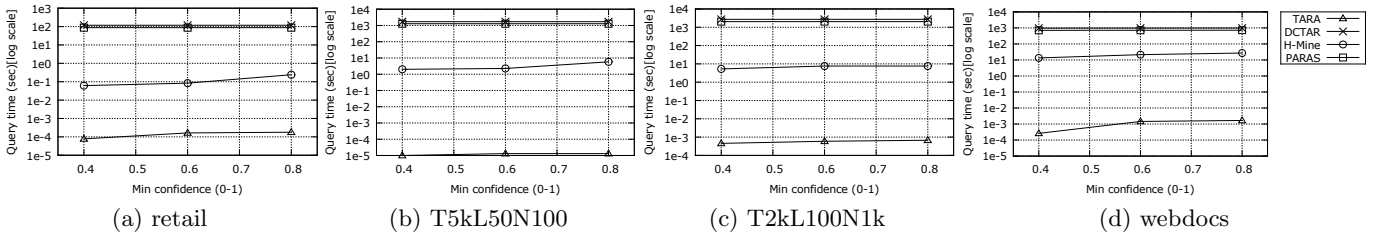


Figure 12: Ruleset Comparison: Varying 2<sup>nd</sup> Confidence

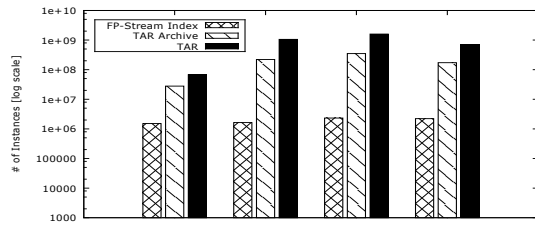


Figure 13: Size of the *TAR Archive*

but also provides very limited feedback for the user.

**Interestingness of temporal associations.** [12, 17] identify the importance of analyzing the interestingness measures of associations. In the context of time-variant data, [11] measures the changes of the interestingness of the association w.r.t its histories. It is suggested that the interest in the rule itself is primarily determined by the interestingness of its change over time. Neither of these works tackle interactive mining through precomputation. In contrast, we explore the space of interestingness parameters for prestore data mining results to facilitate fast online mining.

**Interactive association mining.** Prior works [4, 5, 10] have explored the space of parameters for handling data mining requests. However this work is restricted to static data. These approaches do not consider the time dimension as a property of the pattern. Instead we now study the problem of incorporating the time dimension into the association mining exploration process.

## 10. CONCLUSION

We present the first framework for interactive temporal association analytics. Our **TARA** framework employs a novel evolving parameter space model for pre-generating rules such that near real-time performance is guaranteed for online mining. In a variety of tested cases, **TARA** outperforms the three state-of-the-art competitor techniques, each by several orders of magnitude, while offering a holistic exploration experience supporting new classes of time-variant rule analytics.

## 11. REFERENCES

[1] R. Agrawal, M. Mehta, J. C. Shafer, R. Srikant, A. Arning, and T. Bollinger. The quest data mining system. In *KDD*,

volume 96, pages 244–249, 1996.

[2] J. M. Ale and G. H. Rossi. An approach to discovering temporal association rules. In *Proceedings of the 2000 ACM symposium on Applied computing-Volume 1*, pages 294–300. ACM, 2000.

[3] T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets. Using association rules for product assortment decisions: A case study. In *KDD*, pages 254–260, 1999.

[4] L. Cao, M. Wei, D. Yang, and E. A. Rundensteiner. Online outlier exploration over large datasets. In *Proceedings of the 21th ACM SIGKDD*, pages 89–98. ACM, 2015.

[5] S. Chaudhuri, H. Lee, and V. R. Narasayya. Variance aware optimization of parameterized queries. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 531–542. ACM, 2010.

[6] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu. Mining frequent patterns in data streams at multiple time granularities. *Next generation data mining*, 212:191–212, 2003.

[7] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Record*, volume 29, pages 1–12. ACM, 2000.

[8] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang. Pfp: parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM conference on Recommender systems*, pages 107–114. ACM, 2008.

[9] Y. Li, P. Ning, X. S. Wang, and S. Jajodia. Discovering calendar-based temporal association rules. *Data & Knowledge Engineering*, 44(2):193–218, 2003.

[10] X. Lin, A. Mukherji, E. A. Rundensteiner, C. Ruiz, and M. O. Ward. Paras: A parameter space framework for online association mining. *Proceedings of the VLDB Endowment*, 6(3):193–204, 2013.

[11] B. Liu, Y. Ma, and R. Lee. Analyzing the interestingness of association rules from the temporal dimension. In *Proceedings of IEEE ICDM*, pages 377–384. IEEE, 2001.

[12] B. Liu, K. Zhao, J. Benkler, and W. Xiao. Rule interestingness analysis using olap operations. In *Proceedings of the 12th ACM SIGKDD*, pages 297–306. ACM, 2006.

[13] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Webdocs: a real-life huge transactional dataset. In *FIMI*, 2004.

[14] B. Ozden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *Data Engineering, 1998. Proceedings., 14th International Conference on*, pages 412–421. IEEE, 1998.

[15] X. Qin, R. Ahsan, X. Lin, E. A. Rundensteiner, and M. O. Ward. Iparas: Incremental construction of parameter space for online association mining. In *Proceedings of the 3rd BigMine*, pages 149–165, 2014.

[16] S. Ramaswamy, S. Mahajan, and A. Silberschatz. On the discovery of interesting patterns in association rules. In *VLDB*, volume 98, pages 368–379. Citeseer, 1998.

[17] S. Sahar. Interestingness preprocessing. In *Proceedings of IEEE ICDM*, pages 489–496. IEEE, 2001.

[18] K. Verma and O. P. Vyas. Efficient calendar based temporal association rule. *ACM SIGMOD Record*, 34(3):63–70, 2005.

# Efficient Record Linkage Using a Compact Hamming Space

Dimitrios Karapiperis\*, Dinusha Vatsalan†, Vassilios S. Verykios\*, and Peter Christen†

\*Hellenic Open University

School of Science and Technology  
Patras, Greece

{dkarapiperis, verykios}@eap.gr

†The Australian National University

Research School of Computer Science  
Canberra ACT 0200, Australia

{dinusha.vatsalan, peter.christen}@anu.edu.au

## ABSTRACT

Record linkage, the process of identifying similar records that correspond to the same real-world entities across databases, is a well-established research problem in the database, data mining, and information retrieval communities. Computing distances between string values of records is the key component in order to determine the similarity of the represented entities. Due to the typically large volumes of records, a two-step process is followed. A blocking mechanism is first applied for grouping similar records together, and then a matching mechanism is performed for comparing the records which have been inserted into the same block. However, there does not exist any efficient blocking/matching mechanism which provides theoretical guarantees for identifying similar records which consist of strings. Towards this end, we put forth the novel notion of embedding string-based records into a Hamming space, where such a mechanism exists. The size of these embeddings is kept as small as needed in order to guarantee the correspondence of distances in that space to the types of errors that exist between strings, e.g., a missing or a modified character. We build embeddings whose size is 120 bits for representing accurately four fields of a publicly available data set. We also present a distance threshold-aware blocking technique for higher accuracy rates compared to blocking approaches which ignore the specified threshold. Our empirical study conducted on real-world data sets shows the efficacy achieved by our embedding method as compared to several existing solutions.

## 1. INTRODUCTION

The integration of data from disparate sources is increasingly being required as an important step towards the identification of similar entities across different sources. Known as entity resolution, record linkage, and data matching, the process of integrating data is an important problem in many data mining and knowledge engineering applications [10]. A wide range of real-world applications, including health-care,

government services, crime and fraud detection, national security, and businesses, require entity resolution techniques in order to enrich data quality and empower accurate decision making [2].

Since unique entity identifiers, which would allow a simple join between records, are often not available in databases, a common practice is to use personal identifying attributes, such as names and addresses. Due to the quadratic complexity of the number of comparisons required and the commonly large volumes of records, a two-step process is followed [29]. In the first step, a blocking mechanism is applied, which reduces the comparison space efficiently by creating blocks with potentially similar records. Then, in the second step, only the records within the same block are compared with each other. Moreover, the values of these attributes, which are well correlated with the entities being linked, often contain variations, errors, and misspellings which require the use of approximate matching solutions [2].

A widely adopted criterion that is used to determine the similarity between the string values of these attributes is their edit distance [20], which is the minimum number of character edit operations required to transform one string value into the other. Unfortunately, thus far there does not exist an efficient blocking/matching mechanism that works directly on records, which contain string values, and simultaneously provides theoretical guarantees for identifying all similar record pairs using the edit distance as the metric for determining their similarity. Furthermore, computing the edit distances for a large number of record pairs imposes a considerable non-negligible overhead. This can be tolerated for the traditional context of an off-line process but is not suitable for many emerging recent applications that require nearly real-time analysis, especially if they involve streaming data [5, 33].

For these reasons, a common practice is to embed string values into a metric space where such an efficient blocking/matching solution exists. For example, Hamming [17] and Jaccard [18] Locality-Sensitive Hashing (LSH) based blocking/matching mechanisms work in a Hamming and in a Jaccard space, respectively. Representing a string as a small-sized binary sequence in a Hamming space results in a particularly lightweight structure. Moreover, Hamming distance, which is the number of bits in which two binary sequences differ, can be computed very fast. These two features render those embeddings a perfect fit for distributed and real-time settings. One such example of a real-world application is a health surveillance system that continuously integrates data from hospitals and pharmacy stores by performing a large

number of distance computations in real-time.

Also, during the matching step, it is common practice to follow a decision model by applying to each record pair a rule, which classifies a pair of records as a matching or as a non-matching pair according to some distance thresholds specified for each attribute. Setting such a threshold arbitrarily or on an empirical basis may either impose additional unnecessary running time or generate incomplete results. LSH-based blocking mechanisms [17, 18] consider each record as an entity ignoring completely such classification rules during the blocking step. This record-level approach falls short in the presence of such rules, especially when different thresholds are specified for each attribute.

In this paper, we propose an embedding method of strings into a compact binary Hamming space where both the embeddings are of small size and the distances in that space correspond to certain types of errors, e.g., an accidentally deleted character, between strings. Therefore, one can specify accurately the threshold(s) required by the used blocking/matching mechanism in the embedding space. Furthermore, we adapt the blocking mechanism to the used classification rule and report the formal guarantees provided for identifying any similar pair by using the newly adapted mechanism. To the best of our knowledge, such an attribute-level LSH-based blocking/matching technique has not been proposed in the literature before.

The contributions of this paper are:

- An efficient embedding method of strings into a compact Hamming space resulting in lightweight, in terms of size, embeddings.
- A guaranteed correspondence of distances in the embedding space to certain types of errors between strings.
- An attribute-level LSH-based blocking scheme, which adapts to the used classification rule.
- An experimental evaluation of the proposed method compared with existing embedding solutions and using real-world data sets.

In the next section, we review the relevant literature, and in Section 3 we formulate the problem and motivate its importance. We outline the building components of our proposed method in Section 4, and in Section 5 we describe this method in detail. We empirically evaluate and compare our approach with existing embedding solutions in Section 6, and we conclude this paper with future research directions in Section 7.

## 2. RELATED WORK

A long line of research has been conducted in record linkage and various methods for computing similarities between records in an approximate manner have been proposed. We refer the interested reader to some recent surveys [2, 10]. Many techniques have been developed for the blocking and matching step aimed at reducing the comparison space and identifying as many similar record pairs as possible [3, 10].

For the blocking step, several approaches [6, 8, 12, 34] have been developed with the aim of being scalable to large data sets without sacrificing quality. Nevertheless, there are two methods which had great impact on the research community. The first is the sorted neighborhood method [12],

including all its variants, which first sorts all records from the participating data sets and then uses a fixed-sized sliding window over the sorted records in order to compare the pairs which are formulated within that window. The second is the canopy clustering technique [6] that relies on the idea of using a computationally cheap clustering approach to create high-dimensional overlapping clusters, from which blocks of candidate record pairs can then be generated. These methods though do not provide any guarantees for identifying record pairs that are similar nor scale well to large volumes of records.

Recently, randomized blocking/matching techniques, which mainly rely on Locality-Sensitive Hashing (LSH) [1], have received much attention [9, 15, 17, 18]. These techniques work in some metric space into which string values are embedded preserving their initial distances as accurately as possible. The most appealing property of these distance-based randomized techniques is that they provide theoretical guarantees for identifying each similar record pair in the embedding space with high probability.

For the matching step, a large body of work has also been conducted on similarity joins [35, 36, 11, 21, 24, 25, 30, 31, 32] where efficient and scalable approximate joins are facilitated by using several metrics during the matching step such as the edit, Jaccard, and cosine metrics [2]. Especially, the authors in [35, 21, 25, 30, 31] have devised efficient techniques for finding similar string values using the edit distance metric, however they focus on individual such values, whereas our work proposes a solution for finding similar records, which usually consist of multiple strings. All the above-mentioned metrics though use strings or high-dimensional vectors of integers, which are not suitable structures for highly demanding environments either in terms of communication or of computational cost. Three other state-of-the-art embedding methods, introduced in [14, 18, 27], are used as our competitors and are presented in detail in Section 6.

## 3. PROBLEM DEFINITION

Let us assume that two data custodians, who own databases  $A$  and  $B$ , respectively, engage themselves into a process for identifying the common entities among their records. They are allowed to make use of the services offered by an independent party, whom we call Charlie. The two data custodians agree to use a common set of  $n_f$  attributes, each denoted by  $f_i$  where  $i = 1, \dots, n_f$ , based on which they can exchange and compare their records. We denote by  $u_{\mathcal{E}}^{(f_i)}$  the distances between the values of attribute  $f_i$ , and by  $\vartheta_{\mathcal{E}}^{(f_i)}$  the specified threshold for this attribute. They also need to provide an additional attribute, let us call it  $Id$ , for the role of an identifier of each record. The data custodians submit their records to Charlie whose duty is to identify any pairs of *similar* records that belong to different data sets.

**DEFINITION 1 (A SIMILAR RECORD PAIR).** A record pair  $r_A \in A$  and  $r_B \in B$  is considered as similar if for each attribute  $f_i$ , it holds that  $u_{\mathcal{E}}^{(f_i)} \leq \vartheta_{\mathcal{E}}^{(f_i)}$  in the metric space  $(\mathcal{E}, d_{\mathcal{E}})$ , where  $\mathcal{E}$  is the original space in which the string values of all records of  $A$  and  $B$  exist, and  $d_{\mathcal{E}}$  is the edit distance used as the metric on  $\mathcal{E}$ .

Due to the generally large size of the data sets at hand, Charlie will use a randomized blocking/matching mecha-

---

**Algorithm 1** Mapping a  $q$ -gram to a position of a  $q$ -gram vector.

---

**Input:** a  $q$ -gram  $gr$ .  
**Output:** The index  $ind$  in the  $q$ -gram vector.  
1:  $ind = 0$   
2: **for**  $i = 1, \dots, q$  **do**  
3:    $ch = gr[i]$    // Extract each character  $ch$  from  $gr$ .  
4:    $ind = ind + ord(ch) \times |S|^{q-i}$    // Function  $ord(\cdot)$   
   returns the order (zero-based) of character  $ch$  in  $S$ .  
5: **end for**

---

nism, which both handles efficiently large volumes of data, and provides theoretical guarantees of performance in approximate matching by identifying each similar record pair with a specified (high) probability. However, such known mechanisms work in different metric spaces than  $\mathcal{E}$ . For this reason, Charlie embeds the string values into a Hamming metric space  $(\mathcal{H}, d_{\mathcal{H}})$ , where such a mechanism already exists. Additionally, there should exist a guaranteed correspondence between distances in  $\mathcal{H}$  and the type of errors in  $\mathcal{E}$  by keeping the size of the embeddings in  $\mathcal{H}$  as small as possible. Given this correspondence, the thresholds required can be easily specified in  $\mathcal{H}$ , which will result in identifying each pair of records regarded as similar in  $\mathcal{E}$ .

**DEFINITION 2 (AN EFFICIENT EMBEDDING METHOD).**  
Given an efficient blocking/matching mechanism in  $\mathcal{H}$ , construct an embedding method of strings from  $\mathcal{E}$  into  $\mathcal{H}$ , where errors in  $\mathcal{E}$  are identifiable in  $\mathcal{H}$  separately for each attribute  $f_i$  using embeddings whose sizes are as small as possible depending on the lengths of the strings in  $\mathcal{E}$ .

## 4. BACKGROUND

In this section, we outline the building blocks utilized by our proposed scheme.

### 4.1 $q$ -gram Vectors

A  $q$ -gram vector is a deterministic structure for representing a string value in the Hamming space. Such structures have been used in [18] and [19] for representing distinct attribute values and whole records, respectively. Each position of a  $q$ -gram vector represents a distinct  $q$ -gram which is a group of  $q$  consecutive characters in a string value. By assuming that the alphabet  $S$  of  $q$ -grams is the set of the upper-case letters, the size of a  $q$ -gram vector is  $m = |S|^q = 26^q$  positions. Let us denote a bijection  $F : \{gr_1, gr_2, \dots, gr_m\} \rightarrow \{0, \dots, m - 1\}$ , which maps each  $q$ -gram for a certain alphabet  $S$  to an integer, termed also as the index  $ind$  of that  $q$ -gram. The logic behind  $F$  is illustrated in Algorithm 1. Therefore, by this mapping, we obtain a set of indexes, denoted by  $U_s$ , that indicates which positions of the respective  $q$ -gram vector will be set to 1. Figure 1 illustrates how a string is represented by a bigram<sup>1</sup> vector. A record-level  $q$ -gram vector is built by concatenating the corresponding attribute-level  $q$ -gram vectors and its size is  $\bar{m} = n_f \times m$ . The space in which these record-level  $q$ -gram vectors exist is  $\mathcal{H} = \{0, 1\}^{\bar{m}}$ .

### 4.2 Hamming LSH-based Blocking/Matching

Due to the large number of records that occur in many of today’s databases, the randomized Hamming Locality-Sensitive Hashing (LSH) technique [1], denoted by HB, is

<sup>1</sup>Bigrams are the  $q$ -grams with  $q = 2$ .

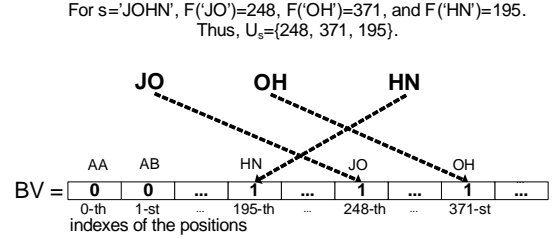


Figure 1: Representing the string ‘JOHN’ as a bigram vector denoted by  $BV$ .

Table 1: Interpretation of the most used variables throughout this paper.

$T_l$	An independent blocking group where $l = 1, \dots, L$ used by the HB.
$h_l$	A composite hash function used to specify the bucket of some $T_l$ into which a $c$ -vector, defined in Section 5.2, is stored.
$K$	The number of base hash functions used by a $h_l$ .
$U_s$	The set of indexes of the respective $q$ -grams of a string $s$ .
$d_{\mathcal{S}}$	The metric applied on space $\mathcal{S}$ where $\mathcal{S} \in \{\mathcal{H}, \widehat{\mathcal{H}}, \mathcal{E}, \mathcal{J}\}$ .
$u_{\mathcal{S}}$	Distance measured by using metric $d_{\mathcal{S}}$ .
$\vartheta_{\mathcal{S}}$	The specified distance threshold in space $\mathcal{S}$ .
$\bar{m}_{opt}$	The optimal size of a record-level $c$ -vector.
$n_f$	The number of common attributes which participate in the linkage process.
$f_i$	An attribute where $i = 1, \dots, n_f$ . When used as a superscript in parentheses, it denotes the attribute-level value, e.g., $u_{\mathcal{H}}^{(f_i)}$ , $K^{(f_i)}$ , $h_l^{(f_i)}$ etc.
$b^{(f_i)}$	The average number of $q$ -grams of the values of the corresponding attribute $f_i$ .

used as the blocking/matching technique in order to identify each similar pair of record-level  $q$ -gram vectors, which exist in  $\mathcal{H}$ , with high probability. Mechanism HB utilizes  $L$  independent hash tables, termed also as blocking groups. Each hash table, denoted by  $T_l$  where  $l = 1, \dots, L$ , consists of key-bucket pairs where a bucket hosts a linked list which is aimed at grouping similar  $q$ -gram vectors. Moreover, each hash table has been assigned a composite hash function  $h_l$  which consists of a fixed number  $K$  of base hash functions. A base hash function applied to a  $q$ -gram vector returns the value of its  $j$ -th position where  $j \in \{0, \dots, \bar{m} - 1\}$  chosen uniformly at random.

**DEFINITION 3 (A HAMMING LSH FAMILY).** A family  $\phi$  of composite hash functions has the following key property for any pair of record-level  $q$ -gram vectors denoted by  $QV_1, QV_2 \in \mathcal{H}$  whose Hamming distance is  $u_{\mathcal{H}}$  [1]:

$$\text{If } u_{\mathcal{H}} \leq \vartheta_{\mathcal{H}} \text{ then } \Pr[h_l(QV_1) = h_l(QV_2)] \geq p^K, \quad (1)$$

where  $p$  denotes the success probability of a base hash function and is equal to  $p = 1 - \frac{\vartheta_{\mathcal{H}}}{\bar{m}}$ .

Intuitively, the smaller the Hamming distance is, the higher the probability for a  $h_l$  to produce the same result. The result of a  $h_l$ , which constitutes the blocking key, applied to

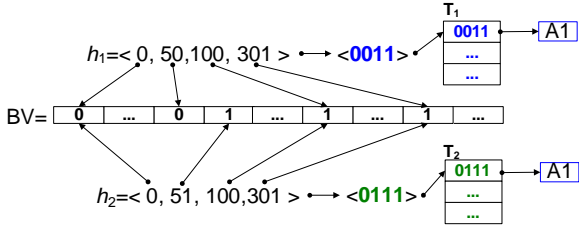


Figure 2: Hashing a record-level bigram vector  $BV$ , with  $Id='A1'$ , by  $h_1$  and  $h_2$ . For illustration purposes, we set  $K = 4$  and  $L = 2$ .

a  $q$ -gram vector specifies into which bucket, termed also as block, this  $q$ -gram vector will be stored<sup>2</sup>. Figure 2 illustrates how a record-level bigram vector is hashed.

During the matching step, we scan the buckets of each  $T_l$  and formulate pairs of  $q$ -gram vectors, which belong to different data sets. By using this redundant blocking scheme, we amplify the probability of identifying similar  $q$ -gram vectors, but we also increase both the utilized space and the running time in order to store the generated  $T_l$ 's. We therefore determine the optimal number of the  $T_l$ 's that should be utilized by setting [1]:

$$L = \lceil \frac{\ln(\delta)}{\ln(1 - p^K)} \rceil. \quad (2)$$

Each similar  $q$ -gram vector pair will be returned with high probability  $1 - \delta$ , as  $\delta$  is usually set to a small value, say  $\delta = 0.1$ . The value for  $K$  can be set empirically since the correctness of the scheme is guaranteed by setting  $L$  appropriately. In [16], a method for choosing the optimal value for  $K$  is presented, where the authors by sampling record pairs and by experimenting with several values for  $K$ , choose the value that minimizes the estimated running time. The value of  $K$  can be set empirically since the completeness, with respect to the identification of the matching pairs, of the mechanism is guaranteed by Equation (2), by deriving the optimal value for  $L$ . The value of  $K$  should be sufficiently large because otherwise the blocking keys will not reflect the variations of the bit sequences of the  $q$ -gram vectors. The direct side-effect of this deficiency will be the generation of a small number of buckets in each  $T_l$ , which will be overpopulated by mostly dissimilar pairs.

Not surprisingly though,  $q$ -gram vectors render inefficient and cumbersome the HB mainly due to their sparsity as will be explained in Section 5.2. In that section, towards mitigating this sparsity and reducing their size, we propose an alternative embedding scheme of the string values into  $\mathcal{H}$  in order to leverage the efficiency of HB.

## 5. AN EFFICIENT EMBEDDING METHOD

In this section, we instantiate our method, termed as cBV-HB, which relies on embedding string values by using their respective  $q$ -grams into a compact Hamming space. The reason for choosing this space is twofold. Firstly, we argue that by corresponding types of errors in  $\mathcal{E}$  to distances in  $\mathcal{H}$ , one can easily specify the distance threshold(s) required by HB. Secondly, the HB mechanism is a fast and accurate method as experimentally demonstrated in [17], and it

<sup>2</sup>More precisely, we store only the corresponding  $Id$ 's.

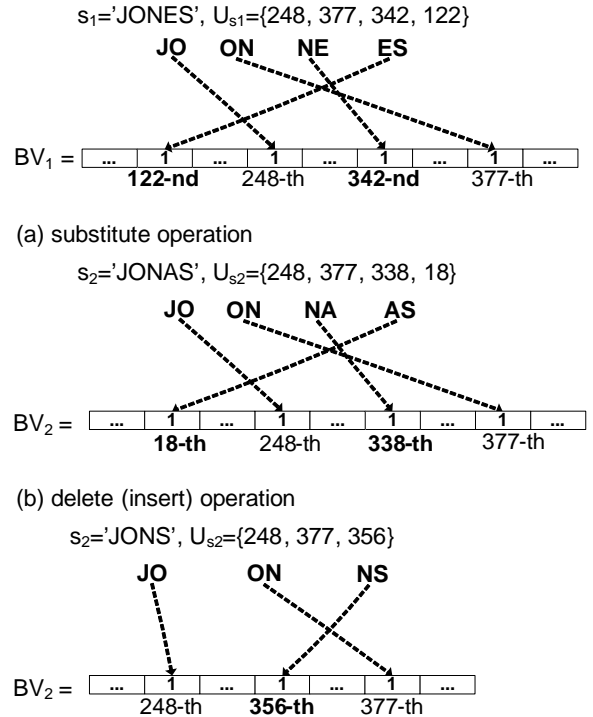


Figure 3: The indexes in bold indicate the differing bigrams between  $s_1$  and  $s_2$ , which result in distances being equal to 4 and 3 in  $\mathcal{H}$  for the substitute and delete (insert) operations, respectively ( $m = 676$ ).

works in Hamming spaces. We next (a) illustrate the correspondence between distances in  $\mathcal{H}$  and types of errors in  $\mathcal{E}$  by using the  $q$ -gram vectors, (b) propose an embedding method which can be used by HB for efficient identification of similar record pairs, and (c) present an attribute-level LSH-based blocking technique which brings the importance of the classification rule into the blocking step.

### 5.1 Corresponding Types of Errors in $\mathcal{E}$ to Distances in $\mathcal{H}$

An error between a pair of string values is formally quantified by edit distance. In the literature, several variants of edit distance exist using different perturbation operations, as the errors are termed in the edit distance context<sup>3</sup>. We consider the basic perturbation operations (substitute, insert, and delete) defined for the Levenshtein distance [20] and correspond these basic operations to distances in  $\mathcal{H}$  by using the  $q$ -gram vectors. We present an illustrative running example where the initial error-free string value is  $s_1='JONES'$ , unless otherwise stated and we set  $q = 2$  (bigrams). The perturbed values are stored in variable  $s_2$ . In this example, we use two bigram vectors, denoted by  $BV_1$  and  $BV_2$  for representing  $s_1$  and  $s_2$ , respectively.

**Substitute perturbation operation:** This type of perturbation operation changes a single character in  $s_1$  and materializes the main reason for errors and misspellings commonly found in string values. Assume the value  $s_2='JONAS'$ ,

<sup>3</sup>We use the terms *perturbation operation* and *error* interchangeably. Usually, a perturbation operation occurs intentionally and an error unintentionally.



the distance  $u_{\mathcal{H}}$  between  $BV_1$  and  $BV_2$  is 4, as shown in Figure 3, due to the 4 differing bigrams, which are ‘NE’ and ‘ES’ in  $s_1$  and ‘NA’ and ‘AS’ in  $s_2$ <sup>4</sup>. Distance may be smaller in case a differing bigram overlaps with a common bigram. For instance, by perturbing  $s_1 = \text{‘SHANNEN’}$  as  $s_2 = \text{‘SHENNEN’}$ , we produce two differing bigrams in  $s_1$ , which are ‘HA’ and ‘AN’, and two more in  $s_2$  namely ‘HE’ and ‘EN’. The latter though overlaps with a common bigram found in both  $s_1$  and  $s_2$ . Therefore, only bigrams ‘HA’, ‘AN’, and ‘HE’ affect  $u_{\mathcal{H}}$  which in this case is 3. We conclude that for this type of perturbation operation  $u_{\mathcal{H}} \leq 4 \times u_{\mathcal{E}}$ .

**Delete and insert perturbation operations:** Another common error that emerges when typing string values is the omission of a character. This causes the generation of a smaller number of bigrams for  $s_2$ . As an illustration, by setting  $s_2 = \text{‘JONS’}$ , we get two differing bigrams in  $s_1$ , which are ‘NE’ and ‘ES’ and only 1 in  $s_2$ , which is ‘NS’ (see Figure 3). Hence, for the delete operation, it holds that  $u_{\mathcal{H}} \leq 3 \times u_{\mathcal{E}}$ . Likewise, the insert is quite similar to the delete operation ( $s_2 = \text{‘JONEAS’}$ ) since it is essentially as a delete operation in  $s_1$ .

The above-mentioned observations, which hold for any  $q$ -gram vector pair with  $q \geq 2$ , lead to the definition of an upper bound for a distance  $u_{\mathcal{E}}$  which corresponds to a  $u_{\mathcal{H}}$  scaled by a constant factor  $\alpha$ :

$$u_{\mathcal{H}} \leq \alpha \times u_{\mathcal{E}}. \quad (3)$$

The factor  $\alpha$  depends on the type of the applied perturbation operation, as explained before, and determines the deviation between distances from  $\mathcal{E}$  to  $\mathcal{H}$ , commonly termed as *distortion* [13].

In the Jaccard space  $\mathcal{J}$ , which consists of the sets  $U_s$  where  $s$  stands for each possible string value, by using the Jaccard metric [2] the distance between  $s_1 = \text{‘JONES’}$  and  $s_2 = \text{‘JONAS’}$  is  $u_{\mathcal{J}} = 1 - |U_{s_1} \cap U_{s_2}| / |U_{s_1} \cup U_{s_2}| \simeq 0.667$ . Comparing though  $s_1 = \text{‘WASHINGTON’}$  with  $s_2 = \text{‘WASHANGTON’}$ , the distance is affected by the length of the strings resulting in  $u_{\mathcal{J}} \simeq 0.364$ . In contrast, the Hamming distance is constantly  $u_{\mathcal{H}} = 4$  in both cases. Hence using the Jaccard metric, one should take into account the length of strings in order to set the threshold appropriately. This task is not easy or sometimes is not feasible at all.

## 5.2 Embedding Strings into a Compact Space

In this subsection, we propose an embedding method, which preserves the distances from  $\mathcal{E}$ , into a compact space that consists of small-sized embeddings. These embeddings can be particularly useful in highly demanding distributed environments, which deal with large volumes of records, for efficient communication. Our method adapts the size of the  $q$ -gram vectors to the expected number of characters that a record holds as needed. For example, the average number of bigrams for the *LastName* attribute of the NCVR database [4], which is a large publicly available data set that we will use in our experiments in Section 6, is only 5.0 bigrams. Therefore, the generated attribute-level bigram vectors would be quite sparse due to the few bigrams produced by each string value. This sparsity though has negative effects during the application of the HB. The  $h_i$ ’s by

<sup>4</sup>We pad the first and the last character, e.g., ‘\_JONES\_’ in order to include all the characters in 2 bigrams.

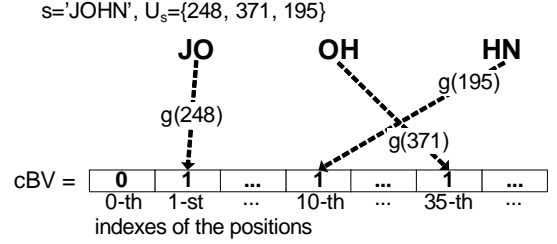


Figure 4: Representing the string ‘JOHN’ as a  $c$ -vector denoted by  $cBV$ .

sampling randomly bit positions from such  $q$ -gram vectors mostly choose 0’s, which has as a side-effect the formulation of a small number of overpopulated buckets. Thus, the HB boils down to an inefficient *all-pairs* comparison process. Moreover, by assuming  $n_f$  attributes, the size of a record-level  $q$ -gram vector would be quite large, namely  $\mathcal{O}(n_f \times m)$  bits. On account of these drawbacks, we embed the string values of each attribute into a new space  $\hat{\mathcal{H}}$  which also uses the Hamming metric. This space consists of compact  $q$ -gram vectors, termed as  $c$ -vectors, which are of size  $m^{(f_i)} \ll |S|^q$ , that will be exactly specified later, for each attribute  $f_i$ . We introduce the dependency of the size of  $c$ -vectors on the average number  $b^{(f_i)}$  of  $q$ -grams of the values of the corresponding attribute  $f_i$ . This dependency will allow us to be as efficient as possible by adjusting the sizes accordingly and simultaneously preserving the distances from  $\mathcal{H}$ . Towards this end, we hash the indexes in  $U_s$  of a string value  $s$  by randomly chosen, pairwise independent hash functions of the form  $g(x) = [(ax+b) \bmod P] \bmod m$ , where  $x \in U_s$ ,  $P$  is a large prime number (e.g.,  $2^{31} - 1$ ) and  $a, b$  are randomly chosen integers from  $(0, P)$ .

Figure 4 shows the creation of a  $c$ -vector by hashing the bigrams of a string. However, during the hash operations of the elements of  $U_s$ , a number of collisions may occur if two elements of  $U_s$  hash to the same index in the  $c$ -vector. This happens because the number of all possible  $q$ -grams is much larger than  $m^{(f_i)}$ . A collision is formally defined as  $g(x) = g(y)$  for any  $x, y \in U_s$  with  $x \neq y$  and the probability  $\Pr[g(x) = g(y)]$  is  $\frac{1}{m^{(f_i)}}$ . By considering the guarantees quoted in the previous section, the collisions, in which differing  $q$ -grams of a pair of  $c$ -vectors participate, affect the distances in  $\hat{\mathcal{H}}$ . These collisions result in misleadingly classifying non-matching as matching pairs.

As an illustration, let us assume the  $c$ -vectors with  $q = 2$ , generated by the values  $s_1 = \text{‘JONES’}$  and  $s_2 = \text{‘JONAS’}$  of an attribute  $f_i$ . We expect the distance to be  $u_{\mathcal{H}}^{(f_i)} = u_{\hat{\mathcal{H}}}^{(f_i)} = 4$  due to the differing bigrams generated by the suffixes ‘NES’ and ‘NAS’. However, if during the hash operations of  $s_2$ , the results corresponding to the bigrams ‘NA’ and ‘AS’ collide, then the  $u_{\hat{\mathcal{H}}}^{(f_i)}$  will be 1 bit less than the  $u_{\mathcal{H}}^{(f_i)}$ . Therefore, the value for  $m^{(f_i)}$  should be adequately specified so that both the HB can be efficiently applied and the distances should be preserved.

The phenomenon of collisions is described in the Birthday Paradox Problem [23] on which the following lemma relies. In the calculations below, we drop superscript  $(f_i)$  for better readability.

LEMMA 1. The expected number of collisions  $E[c]$  by hashing  $b$   $q$ -grams for attribute  $f_i$  to a  $c$ -vector with size  $m$  is:

$$E[c] = b - E[v], \quad (4)$$

where  $E[v]$  denotes the expected number of positions which both hold 1 and no collisions have occurred.

PROOF. The indexes of positions of a  $c$ -vector, representing a string  $s$  of an attribute  $f_i$ , are uniformly chosen by  $g(\cdot)$ , therefore the probability of choosing any position for each  $x \in U_s$  is  $\frac{1}{m}$ . Let the indicator variable  $I_j$  denote for the position with index  $j$ , where  $j \in \{0, \dots, m-1\}$ , the content in that position. The probability that a certain position with index  $j$  is not chosen ( $I_j = 0$ ) after hashing  $b$   $q$ -grams is:

$$\Pr[I_j = 0] = \left(1 - \frac{1}{m}\right)^b. \quad (5)$$

Thus, the probability that the position with index  $j$  is finally chosen is  $\Pr[I_j = 1] = 1 - \Pr[I_j = 0]$ . The expected number  $v$  of positions holding 1 is:

$$E[v] = \sum_{j=0}^{m-1} E[I_j] = mE[I_j] = m\left(1 - \left(1 - \frac{1}{m}\right)^b\right). \quad (6)$$

Then, by subtracting  $E[v]$  from  $b$ , we arrive at the desired result.  $\square$

Let us denote by  $\rho$  the maximum number of collisions we can tolerate during the generation of the  $c$ -vectors. Then, using  $\rho$  and the above lemma, we state the following theorem:

THEOREM 1. By expecting  $b$   $q$ -grams in the corresponding strings, the optimal size of the  $c$ -vectors for some attribute  $f_i$  is:

$$m_{opt} = \left\lceil \frac{b - \rho}{1 - e^{-r}} \right\rceil, \quad (7)$$

where  $E[c] \leq \rho$  with confidence  $1 - r$ .

PROOF. We first specify the value of  $\rho$  and expand Equation (4) by using Equation (6) as follows:

$$\begin{aligned} E[c] &= b - m\left(1 - \left(1 - \frac{1}{m}\right)^b\right) \leq \rho \Rightarrow \\ m\left(1 - \left(1 - \frac{1}{m}\right)^b\right) &\geq b - \rho. \end{aligned} \quad (8)$$

By using the fact that  $-\left(1 - \frac{1}{m}\right)^b \geq -e^{-\frac{b}{m}}$ , we derive an upper bound for the left hand side of the second inequality in (8) which is written as:

$$m\left(1 - e^{-\frac{b}{m}}\right) \geq b - \rho. \quad (9)$$

We then substitute  $\frac{b}{m}$  with a constant  $r$ , where  $r < 1$  since it always holds that  $\frac{b}{m} < m$ . This constant denotes the ratio between the number of  $q$ -grams to the size  $m$ . Intuitively, smaller values for  $r$  increase our confidence, quantified as  $1 - r$ , that collisions will not occur at the cost of a larger size  $m$ . Finally, we solve for  $m$  in (9), and derive the optimal size, as illustrated in Equation (7), where we keep only the equal sign, since we want to be as optimal as possible.  $\square$

In Section 6, we show experimentally that by setting  $r < 1/3$ , we just increase  $m_{opt}$  without earning a lot in terms of accuracy.

**Algorithm 2** Matching the  $c$ -vector pairs formulated in the buckets of the  $T_i$ 's.

---

```

Input:  $cBV_B \in B$ 
1:  $C \leftarrow \text{new UniqueCollection}()$ 
2: //  $C$  is a collection of unique  $Id$ 's.
3: for  $l = 1, \dots, L$  do
4:    $Id\_list \leftarrow T_l.get(h_l(cBV_B))$ 
5:   // Object  $Id\_list$  is a linked list of  $Id$ 's.
6:   for  $i = 1, \dots, Id\_list.size()$  do
7:      $Id \leftarrow Id\_list[i]$ 
8:     if ( $\text{not } C.contains(Id)$ ) then
9:        $cBV_A \leftarrow \text{retrieve}(Id)$ 
10:       $rule(cBV_A, cBV_B)$ 
11:      // A classification rule applied for each  $c$ -vector pair.
12:       $C.add(Id)$ 
13:     end if
14:   end for
15: end for

```

---

Table 2: Primitive operations used by Algorithm 2.

$get(x)$	Return the linked list, to which the specified key $x$ is mapped.
$size()$	Return the size of a linked list.
$contains(x)$	Return <i>true</i> if the unique collection contains element $x$ .
$retrieve(x)$	Retrieve a $c$ -vector with $Id = x$ from the data store.
$add(x)$	Add value $x$ to a unique collection.

---

For example, by assuming  $b^{(f_1)} = 5.1$  and  $b^{(f_3)} = 20.0$  from Table 3, by setting in (7)  $\rho = 1$  and  $r = 1/3$ , we derive values  $m_{opt}^{(f_1)} = 15$  and  $m_{opt}^{(f_3)} = 68$ , respectively. We use the ceiling function  $\lceil \cdot \rceil$  to  $m_{opt}^{(f_i)}$  because the size of a  $c$ -vector should be an integer.

For each attribute, Charlie transforms the strings he receives from Alice and Bob into  $c$ -vectors using the optimal size  $m_{opt}^{(f_i)}$  by sampling randomly and uniformly strings from the data sets and computing  $b^{(f_i)}$ . By concatenating the attribute-level  $c$ -vectors, Charlie then builds the record-level structures, whose size  $\bar{m}_{opt}$  is compact and adapted to the needs of each attribute.

### 5.3 Outline of the Blocking/Matching Step

Let us denote by  $cBV_A$  and  $cBV_B$  the record-level  $c$ -vectors which belong to data sets  $A$  and  $B$ , respectively. We first hash each  $cBV_A$  and store its  $Id$  in the buckets of the  $T_i$ 's. Then, we hash each  $cBV_B$  to the  $T_i$ 's in order to formulate  $c$ -vector pairs for performing the distance computations. Due to the *redundant* blocking model that we follow, certain pairs of  $c$ -vectors might be formulated in several  $T_i$ 's. On account of this redundancy, we incorporate a de-duplicating mechanism in HB in order to prevent the repetitive distance computations of duplicate pairs as can be seen in Algorithm 2. For each bucket that  $cBV_B$  maps to, we retrieve the  $Id$ 's already stored therein (line 4), and query them against a collection of unique elements<sup>5</sup> (line 8). If an  $Id$  is not found in that collection, then the corresponding distance computation is performed otherwise it is dropped. Table 2 quotes a description of each primitive operation used by Algorithm 2. We have to note that our method is capable of handling an arbitrary number of data

<sup>5</sup>This collection is instantiated by a *HashSet* object in Java programming language.

sets (two or more) belonging to different data custodians.

## 5.4 Attribute-level LSH-based Blocking

Mechanism HB assumes record-level  $c$ -vectors where by sampling randomly and uniformly their bits builds the  $T_i$ 's. During the matching step, a decision model is applied in order to classify the formulated record pairs as matching or as non-matching. In its simplest form, a decision model might be a classification rule, which applies a logical condition to the values of each attribute by comparing them to an attribute-level threshold. Therefore, there is no guarantee that  $c$ -vector pairs are formulated according to the classification rule during the blocking step. For instance, by using the attributes in Table 3, a classification rule might be  $(u^{f_1} \leq 4) \wedge (u^{f_2} \leq 8)^6$ . This rule is more strict to errors in the values of the first attribute while being more tolerant to errors in the values of the second attribute. HB though is unaware of that rule, and uses the underlying values of attributes on an equal basis. In this subsection, we propose a method for adjusting the HB mechanism to the classification rule. This adjustment has as a result the formulation of  $c$ -vector pairs which are much closer in terms of distance to the logic of the classification rule.

To begin with, the hash functions during the blocking step should use each attribute separately rather than sampling bits uniformly from the record-level  $c$ -vectors. To this end, we choose a value for each attribute-level  $K^{(f_i)}$ , in the same sense as we have described in Section 4.2. A  $K^{(f_i)}$  specifies the number of base hash functions for each  $h_l^{(f_i)}$  which work on the attribute-level  $c$ -vectors corresponding to attribute  $f_i$ . By assuming (a)  $n_c$  attributes, where  $n_c \leq n_f$ , that participate in each rule, (b) independence among the string values of each attribute, and (c) the attribute-level success probability of a base hash function is  $p^{(f_i)} = 1 - \frac{\vartheta^{(f_i)}}{m_{opt}^{(f_i)}}$ , we state the following definitions for any pair of record-level  $c$ -vectors that exhibit distances  $u^{(f_i)} \leq \vartheta^{(f_i)}$  as follow:

**DEFINITION 4 (AND OPERATOR).** *By using the AND operator ( $\wedge$ ) on certain attributes in the classification rule, the probability of a record-level  $c$ -vector pair to collide into the same bucket of a  $T_i$  is:*

$$p_{\wedge} \geq \prod_{i=1}^{n_c} (p^{(f_i)})^{K^{(f_i)}}. \quad (10)$$

Using the AND operator, the structure of the blocking groups used, described in Section 4.2, is maintained. The blocking keys for each  $f_i$  are concatenated resulting in a compound blocking key that is finally inserted into some  $T_i$ .

**DEFINITION 5 (OR OPERATOR).** *By using the OR operator ( $\vee$ ) on certain attributes in the classification rule, the probability of a record-level  $c$ -vector pair to collide into the same bucket of any  $T_l^{(f_i)}$  is:*

$$p_{\vee} \geq (p^{(f_1)})^{K^{(f_1)}} + (p^{(f_2)})^{K^{(f_2)}} - (p^{(f_1)})^{K^{(f_1)}} \times (p^{(f_2)})^{K^{(f_2)}}. \quad (11)$$

Without loss of generality, in Equation (11), we show only the case where  $n_c = 2$  attributes. For a larger number  $n_c$  of attributes, the inclusion-exclusion principle [22] should be used.

<sup>6</sup>We drop the space subscript from distances and thresholds, since from now on we focus on  $\hat{\mathcal{H}}$ .

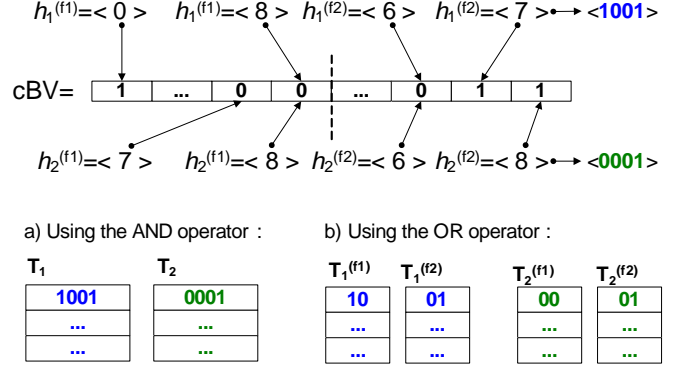


Figure 5: Applying attribute-level hashing to a  $c$ -vector  $cBV$  which consists of 2 attribute-level  $c$ -vectors. For illustration purposes, we set  $K = 4$  ( $K^{(f_i)} = 2$ ) and  $L = 2$ .

When using the OR operator, the structure of the blocking groups changes considerably. For each attribute  $f_i$ , which is part of the OR rule, for each blocking group, we build an independent hash table, denoted by  $T_l^{(f_i)}$ , that stores the blocking keys of this specific attribute. Therefore, given  $n_c$  attributes in the OR rule, we end up with  $L \times n_c$  hash tables. Intuitively, one would classify a  $c$ -vector pair as a matching one, if this pair is formulated in at least one  $T_l^{(f_i)}$ , regardless of the remaining outcomes.

**DEFINITION 6 (NOT OPERATOR).** *By using the NOT operator ( $\neg$ ) on a certain attribute, the probability of any pair of record-level  $c$ -vectors not to collide into the same bucket of a  $T_l^{(f_i)}$  is:*

$$p_{\neg} \geq 1 - (p^{(f_i)})^{K^{(f_i)}}. \quad (12)$$

The NOT operator assumes one attribute  $f_i$  and thus one hash table for each blocking group. One would assume the *true* value as outcome, if a certain pair has not been formulated in the corresponding  $T_l^{(f_i)}$ 's.

These revised probability bounds adjust the number  $L$  of the blocking groups<sup>7</sup> used by substituting  $p^K$  in Equation (2). The new value of  $L$  is larger using an AND rule, and smaller using an OR rule than the standard record-level LSH-based blocking approach. Using these operators, we build the basic classification rules which are depicted in Figure 5. The blocking group of the NOT operator does not include any modifications because we just change what we consider as a true outcome.

In addition, by using these basic rules and their corresponding blocking groups, we may compose compound classification rules which consist of several *subrules*. Such compound rules might be:

- $C_1 = [(u^{(f_1)} \leq \vartheta^{(f_1)}) \wedge (u^{(f_2)} \leq \vartheta^{(f_2)})] \vee [(u^{(f_3)} \leq \vartheta^{(f_3)}) \wedge (u^{(f_4)} \leq \vartheta^{(f_4)})]$  where two separate blocking structures for the AND subrules should be built. The first blocking structure comprises the blocking groups of attributes  $f_1$ , and  $f_2$ , while the second one contains the attributes  $f_3$ , and  $f_4$ . During the blocking mechanism, the blocking keys will be built from the

<sup>7</sup>The value of  $L$  may vary among the blocking structures used.

corresponding attribute-level  $c$ -vectors and will be inserted into the corresponding  $T_i$ 's. Since an OR operator joins the two subrules, a pair will be returned if it will be formulated in the blocking structure of either subrule. Thus, during the matching phase, for each  $c$ -vector from data set  $B$ , we formulate all possible pairs in all the corresponding buckets. Then, a pair is considered as a matching one, if it is formulated in either blocking structure.

- $C_2 = [(u^{(f_1)} \leq \vartheta^{(f_1)}) \vee [(u^{(f_2)} \leq \vartheta^{(f_2)})] \wedge [(u^{(f_3)} \leq \vartheta^{(f_3)}) \vee (u^{(f_4)} \leq \vartheta^{(f_4)})]$  where four separate blocking structures for the OR operators should be built. The main difference between  $C_1$  and  $C_2$  is that using  $C_2$ , a pair should be formulated in both blocking structures of the subrules in order to be considered as a matching pair.
- $C_3 = (u^{(f_1)} \leq \vartheta^{(f_1)}) \wedge [\neg(u^{(f_2)} \leq \vartheta^{(f_2)})]$ . Using  $C_3$ , a pair is returned if it is formulated in the blocking structure for  $f_1$ , but not found in the blocking structure for  $f_2$ .

The space needed for building the blocking groups of a rule using an AND operator is  $\mathcal{O}(L)$ , while using an OR operator is  $\mathcal{O}(n_c \times L)$ .

## 6. EVALUATION

In this section, we describe the experimental settings, the baseline methods, the data sets used, as well as the achieved results. We conducted experiments using two publicly available real-world databases which are (a) the NCVR database [4] and (b) the DBLP bibliography database<sup>8</sup>. By using both these data sets, which exhibit different properties such as the average lengths of string values, we obtain several insights through the experimental results. The attributes used are listed in Table 3.

We developed a software prototype which by using as input the above-mentioned databases, extracts records and creates two data sets, denoted by  $A$  and  $B$ , respectively, where one can specify the perturbation frequency, number of perturbation operations, and number of perturbed records in  $B$  for each chosen record in  $A$ . We apply (a) a light perturbation scheme, termed as  $PL$ , where we perturb the values of one randomly chosen attribute, and (b) a heavy scheme, termed as  $PH$ , where we apply one perturbation to the values of the first two attributes and two perturbations to the values of the third attribute. We notate the thresholds for each perturbation scheme as  $\vartheta_{PL}^{(f_i)}$  and  $\vartheta_{PH}^{(f_i)}$  regardless of the used space. The number of records in  $A$  (and  $B$ ) is 1,000,000, while the probability of choosing a record from  $A$  in order to apply a perturbation scheme and then place it in  $B$ , is set to 0.5. The experiments were executed on a dual-core Pentium PC with 32 GB of main memory. The software components are developed using the Java programming language (JDK 1.7) and are available from the authors.

**Quality measures.** The Pairs Completeness ( $PC$ ), Pairs Quality ( $PQ$ ), and Reduction Ratio ( $RR$ ) measures [3] are employed to evaluate the quality of both our method and the baseline methods which are discussed in detail below. The set of truly matching record pairs is denoted by  $M$

<sup>8</sup><http://dblp.uni-trier.de/xml/>

Table 3: Attribute-level parameters used for each type of data set by using bigrams.

	attribute	$b^{(f_i)}$	$m_{opt}^{(f_i)}$	$K^{(f_i)}$
<b>NCVR</b>	$f_1=FirstName$	5.1	15	5
	$f_2=LastName$	5.0	15	5
	$f_3=Address$	20.0	68	10
	$f_4=Town$	7.2	22	
			$\bar{m}_{opt} = 120$	
<b>DBLP</b>	$f_1=FirstName$	4.8	14	5
	$f_2=LastName$	6.2	19	5
	$f_3=Title$	64.8	226	12
	$f_4=Year$	3.0	8	
			$\bar{m}_{opt} = 267$	

and the set of identified matching pairs by  $\mathcal{M}$ . The accuracy in finding the matching record pairs is indicated by the  $PC$  measure, which is equal to  $PC = |\mathcal{M} \cap M|/|M|$ . The  $PQ$  measure shows the efficiency in generating mostly matching pairs with respect to candidate pairs, namely  $PQ = |\mathcal{M} \cap M|/|CR|$ , where  $CR$  is the set of candidate pairs. The  $RR$  metric indicates the percentage in the reduction of the comparison space  $A \times B$ , which is equal to  $RR = 1.0 - |CR|/|A \times B|$ . We ran each experiment 50 times and plotted the average values of these measures in the figures shown below.

### 6.1 Baseline Methods

We compare our approach  $cBV-HB$  with three state-of-the-art embedding approaches for record linkage. The first is the h-CC algorithm of **HARRA** [18], where two de-duplicated data sets are linked. In this approach, all attribute values of a record are represented by a single bigram vector. However, setting the same position of a bigram vector by identical bigrams, which belong to different attributes, may lead to ambiguous evaluation of distances and consequently to reduced accuracy. **HARRA** employs the Min-Hash LSH-based blocking/matching mechanism which uses the Jaccard metric, as described in Section 5.1, for performing the distance computations. Since **HARRA** selects arbitrary values for  $K$  and  $L$ , by experimenting for better results, we set  $L = 30$  and  $L = 90$  ( $K = 5$ ) for each perturbation scheme, respectively. We performed several distance computations using the vector space that **HARRA** works, where one cannot focus on separate attributes, using perturbed string values and ended up choosing  $\vartheta_{PL} = 0.35$  and  $\vartheta_{PH} = 0.45$ <sup>9</sup> as a nice balance between accuracy and efficiency. During the blocking phase, we hash those vectors by applying random permutations of their indexes and we choose the index of the minimum non-zero element of these permutations as the result of each base hash function. However, we mostly end up with an index holding 0, which implies that more elements of each permutation should be used until we find an index that is set to 1. As a result, similar records are inserted into different buckets. The blocking and matching mechanisms are conducted iteratively and separately for each  $T_i$ . When two records are classified as a matching pair, they are subsequently excluded from the remaining iterations.

<sup>9</sup>All thresholds are set after experimenting exhaustively using the initial and corresponding perturbed values.

Another method we compare our approach with is *BfH* presented in [17], which uses the HB, as described in Section 4.2, on Bloom filters. A Bloom filter is a data structure used to represent the elements of a set in order to support membership queries efficiently in terms of time and space required. It has been shown in [27] that by embedding string values into Bloom filters, distances from the original space are preserved. More specifically, a Bloom filter is a bitmap array initialized with zeros and created by hashing the bigrams of a string value by using independent composite cryptographic hash functions such as MD5 and SHA1 [26]. Field-level Bloom filters are created using a size of 500 bits by using 15 cryptographic hash functions for each bigram, as proposed in [27]. We set  $K = 30$  and  $\delta = 0.1$  while thresholds are set as  $\vartheta_{PL}^{(f_i)} = 45$  ( $L = 4$ ),  $\vartheta_{PH}^{(f_1)} = \vartheta_{PH}^{(f_2)} = 45$ , and  $\vartheta_{PH}^{(f_3)} = 90$  ( $L = 43$ ). We have to note that these attribute-level thresholds are used only during the matching step. A key observation for the Bloom filter space, which is a high-dimensional binary Hamming space, is that distances are affected by the number of bigrams. For example, using the field-level Bloom filters, described before, the Hamming distance between ‘JOHN’ and ‘JAHN’ is 54. In contrast, the Hamming distance between ‘SCALABILITY’ and ‘SCELABILITY’ is equal to 37. This variation in distance, although in both cases there exists a single error in the initial string values, causes difficulties in specifying effectively the distance threshold.

Finally, we compare *cBV-HB* with the *StringMap* algorithm [14] which is used to embed string values into a Euclidean space. Initially for each attribute, *StringMap* iterates the strings of both data sets in order to form  $d$  orthogonal directions (axes). Each such direction is specified by two strings, termed as pivots, whose distances are as far from each other as possible. Yet, the process of specifying the pivot values is quite expensive since it includes several iterations of the data sets. Then, for each string, we compute its coordinates on these  $d$  axes, which results in a vector of values of dimensionality  $d$ . As the authors suggest [14], dimensionality  $d$  is set to 20 for each attribute and thresholds  $\vartheta_{PL}^{(f_i)} = 4.5$ ,  $\vartheta_{PH}^{(f_1)} = \vartheta_{PH}^{(f_2)} = 4.5$ , and  $\vartheta_{PH}^{(f_3)} = 7.7$ , for each scheme respectively. We utilize the Euclidean LSH-based blocking/matching mechanism [17], specifically developed for finding similar points in Euclidean spaces. As in *BfH*, the above-mentioned thresholds are used only during the matching step. The value of  $K$  is set to 5 which generates  $L = 29$  and  $L = 194$  blocking groups [7] for each perturbation scheme, respectively. We call this method *SM-EB* due to the combination of *StringMap* and the blocking/matching mechanism used.

## 6.2 Experimental Results

**Accuracy.** In the first series of experiments, for the rules:

- $C_1 = (u^{(f_1)} \leq \vartheta^{(f_1)}) \wedge (u^{(f_2)} \leq \vartheta^{(f_2)}) \wedge (u^{(f_3)} \leq \vartheta^{(f_3)})$ ,
- $C_2 = [(u^{(f_1)} \leq \vartheta^{(f_1)}) \wedge (u^{(f_2)} \leq \vartheta^{(f_2)})] \vee (u^{(f_3)} \leq \vartheta^{(f_3)})$ , and
- $C_3 = (u^{(f_1)} \leq \vartheta^{(f_1)}) \wedge [\neg(u^{(f_2)} \leq \vartheta^{(f_2)})]$ ,

we measured the *PC* and *PQ* rates of our attribute-level blocking and compare them with the rates of the standard LSH-based approach, which, as described in Section 4.2, during the blocking phase samples bits uniformly from the

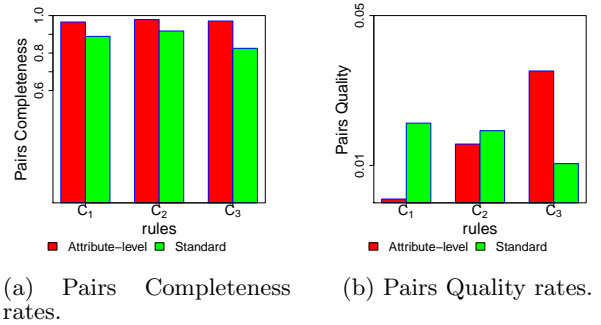


Figure 6: Attribute-level Pairs Completeness and Pairs Quality evaluation using the NCVR-based data sets.

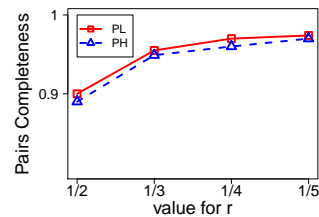
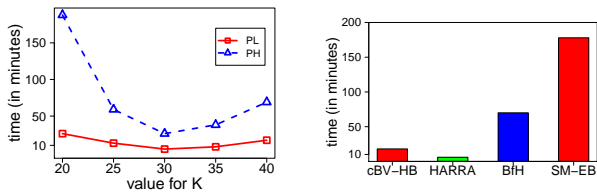


Figure 7: Evaluation of accuracy by setting several values to confidence  $r$ , from Equation (7), and measuring the *PC* rates using the NCVR-based data sets ( $K = 35$ ).

whole record-level  $c$ -vector. Clearly, Figure 6(a) shows that by using this rule-aware blocking phase, the *PC* rates are constantly higher than those using the standard LSH-based blocking approach. Nevertheless, the largest difference lies in  $C_3$  where the standard approach is unable to articulate the NOT operator, comparing and discarding rather late any such non-matching pairs. Conversely, those pairs, in the rule-aware blocking phase, are not formulated at all and are never brought for comparison. The *PQ* rates, illustrated in Figure 6(b), for  $C_1$  are lower than the standard approach due to the larger number of blocking groups required. For  $C_2$ , the utilization of two hash tables in a blocking group, because of the OR operator, drops initially the *PQ* rates which balance later, during the matching phase, due to the better quality of the formulated pairs. We then experimented by setting several values to confidence  $r$ , for less collisions from Equation (7), and measured the corresponding *PC* rates. Since we want to be as optimal as possible, the choice of  $r = 1/3$  exhibits both high *PC* rates and the sizes of the  $c$ -vectors are kept to a desired level. As can be seen in Figure 7, we do not gain a lot in terms of accuracy by setting  $r < 1/3$ .

**Running time.** By choosing several values for  $K$  we measure the elapsed running time. Specifically, for both perturbation schemes we vary  $K$  between 20 and 40, which results in generating different values for  $L^{10}$  (blocking groups). Figure 8(a) clearly illustrates that there is a near-optimal value of  $K$ , which is 30 for both perturbation schemes, that minimizes the running time. This is quite reasonable because by increasing  $K$  we adjust the *selectivity* of our block-

<sup>10</sup>As shown in Section 4.2,  $L$  depends on  $K$ ,  $\delta$ ,  $\vartheta_{\hat{H}}$ , and  $\overline{m}_{opt}$ .



(a) Applying *cBV-HB* by using several values for  $K$ . (b) Time needed for converting the data sets into the respective embeddings of each method.

Figure 8: Running time evaluation using the NCVR-based data sets.

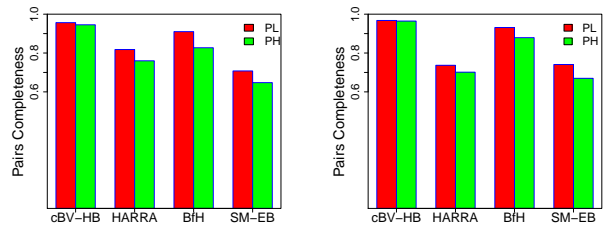
ing mechanism, i.e., buckets are populated with  $Id$ 's corresponding mostly to similar  $c$ -vectors. The consequence of higher selectivity is a decrease in running time due to the smaller number of the formulated  $c$ -vector pairs. Nevertheless, there is a value for  $K$  ( $K = 35$ ) where the time needed for building the blocking groups dominates the total running time which starts to increase.

**Comparison with the baseline methods.** For the next set of experiments, we first measured the average number  $b^{(f_i)}$  of bigrams of string values for each attribute listed in Table 3. We underline the difference in the values of  $b^{(f_i)}$  between the two sources of data sets and evaluate its impact on the experimental results which follow below. For scheme *PL*, since there is a single perturbation operation, we set  $K = 30$ ,  $\delta = 0.1$ , and  $\vartheta_{PL}^{(f_i)} = 4$ , which generate  $L = 6$ , and  $L = 3$  blocking groups, without applying attribute-level blocking, for each source of data sets used. For scheme *PH*, we apply attribute-level blocking by using the rule  $C_1$ , as defined previously, and the parameters in Table 3. We set the thresholds as  $\vartheta_{PH}^{(f_1)} = \vartheta_{PH}^{(f_2)} = 4$ , and  $\vartheta_{PH}^{(f_3)} = 8$ , which yield  $L = 178$  and  $L = 62$  blocking groups, respectively.

In Figure 8(b), we evaluate the time needed in order to embed the data sets into the space required by each method. The bigram vectors used by *HARRA* require the least amount of time because a single vector is used for the bigrams generated of the whole record with the side-effects in accuracy, which will be discussed below. The vectors utilized by *SM-EB* exhibit a large amount of time due to the distance computations performed for specifying the pivots.

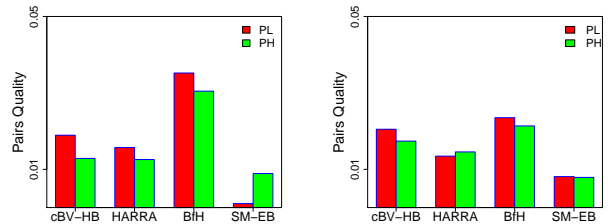
Figures 9(a) and 9(b) show that the *PC* rates of our method are constantly above 95% by using both sources of data sets. These figures also indicate that *cBV-HB* is the only method which exhibits stable *PC* rates regardless of the source of data sets used. Furthermore, by applying *PH* in the presence of a certain classification rule our method adjusts to it during the blocking step by generating the required number of blocking groups separately for each attribute as the rule defines. However, this attribute-level adjustment requires a larger number of blocking groups, which results in reduced *PQ* rates due to the larger number of the formulated pairs (Figures 10(a) and 10(b)).

The *PC* rates of *SM-EB* are rather low, as Figure 9(a) suggests, especially when using *PH*. This happens due to the insufficient distance-preserving property of the used embedding method. This insufficiency has also another drawback, which is the population of blocks with truly non-matching



(a) Using the NCVR-based data sets. (b) Using the DBLP-based data sets.

Figure 9: Pairs Completeness (accuracy) evaluation.



(a) Using the NCVR-based data sets. (b) Using the DBLP-based data sets.

Figure 10: Pairs Quality evaluation.

pairs. These pairs, although being similar in the Euclidean space, exhibit large distances in the original space, which results in low *PQ* rates, as can be clearly seen in Figures 10(a) and 10(b).

In *HARRA*, the early removal of records in each iteration may lead to missed matching pairs as well, since those records do not participate in the subsequent iterations. Initially, the *PC* rates of *HARRA* were below 0.77. We had to increase considerably the number of blocking groups<sup>11</sup> in order to achieve better rates, which were approximately equal to 0.82 as shown in Figure 9(a). However, the side-effect of increasing the number of blocking groups was the low *PQ* rates as illustrated in Figures 10(a) and 10(b). Especially by using the DBLP-based data sets, the larger number of bigrams combined with the utilization of a single bigram vector for all attributes in a record increased considerably the probability of comparing bigram vectors with identical bigrams belonging to different attributes. This disambiguation, as expected, deteriorated the *PC* rates which fell below 0.75 (Figure 9(b)).

The Bloom filters, which are used by *BfH*, seem to preserve the initial distances from the original space, as confirmed by the high *PC* rates. However, the accuracy guarantees, provided by the HB, refer to the Bloom filter space and there is no study in the literature that corresponds distances from that space to distances in  $\mathcal{E}$  with a specified distortion. The authors in [17] provide only some empirical observations with respect to this correspondence without any rigorous justifications. The dependency of distances, in the Bloom filter space, on the length of the initial string

<sup>11</sup>We actually doubled the number of blocking groups in order to give more chances to similar records to be grouped together.

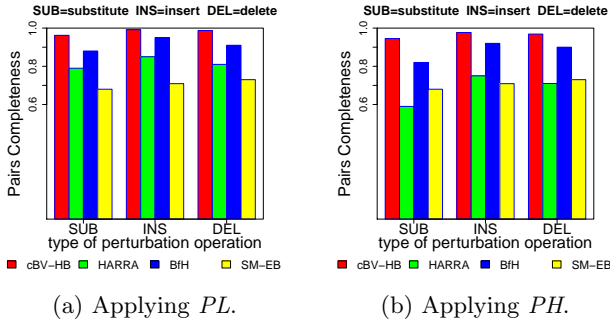


Figure 11: Evaluating Pairs Completeness by focusing on each type of perturbation operation by using the NCVR-based data sets.

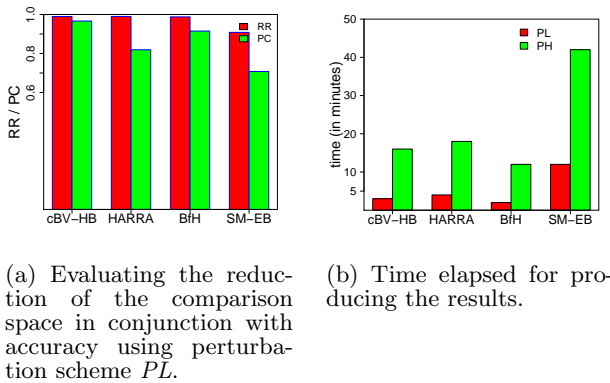


Figure 12: Evaluating the efficiency of each method by using the NCVR-based data sets.

values, as demonstrated previously, explains the increased accuracy by using the DBLP-based data sets (Figure 9(b)). On the contrary, distances in  $\hat{H}$  depend only on the type of error and by no means on the length of the initial string values. The  $PQ$  rates of *BfH* are slightly higher than *cBV-HB* mainly due to the larger number of the cryptographic hash functions used for creating the Bloom filters, which results in a larger number of positions set to 1. These bit patterns hashed by the  $h_i$ 's produce a larger number of buckets in each  $T_i$ , which host a smaller number of pairs than *cBV-HB*.

Another factor that does not affect the  $PC$  rates of our method is the type of the applied perturbation operation as demonstrated in Figures 11(a) and 11(b) by applying  $PL$  and  $PH$ , respectively. Our method attains excellent performance and the  $PC$  rate barely falls below 0.95 only when applying the substitute operations. In general, we observe that all methods exhibit a lower  $PC$  rate for pairs which have been perturbed by the substitute operation, which indicates a higher distortion in all spaces. For  $PH$ , only *BfH* exhibits comparable performance, as can be seen in Figure 11(b). However, the two operations performed in the values of the *Address* attribute resulted in missing some pairs especially when both were substitute operations.

Figure 12(a) illustrates together the  $RR$  and the  $PC$  rates so that one can easily evaluate the efficiency of each method.  $RR$  is high for all the compared methods except for *SM-EB*

*EB* where the formulated blocks are overwhelmed by non-matching pairs. The reduction of the comparison space though keeps up with high accuracy only for *cBV-HB* and *BfH*, where our method performs better than *BfH* in terms of accuracy by at least 5%. Overall, this provides additional validation of the robustness and practicality of our method. These high  $RR$  rates affect the running time positively which is below 5 minutes for  $PL$  for both methods, as depicted in Figure 12(b). By applying  $PH$  though, the running time increases due to the larger number of blocking groups generated for this perturbation scheme. In *HARRA*, the early pruning of records in each iteration reduces the running time but the results are far from accurate. As expected, *SM-EB* exhibits the highest running time by a large margin among all the compared methods due to the large number of the formulated vector pairs.

## 7. CONCLUSIONS AND FUTURE EXTENSIONS

In this paper we have proposed a method to embed strings into a compact binary Hamming space in order to apply HB which is an efficient blocking/matching mechanism. The embeddings are of small size, e.g., a record of four strings is represented by 120 bits, and simultaneously the initial distances are preserved as the supporting set of experiments confirmed. Furthermore, we have adapted the LSH-based blocking mechanism to the used classification rule for highly accurate results. We have considered and provided formal guarantees for rules using the AND, OR, and NOT operators. In addition, we have also demonstrated the use of compound classification rules, which include several sub-rules. For the future, we aim to investigate a distance-preserving and lightweight embedding method for the Jaro-Winkler metric, which was specifically developed for measuring distances between attributes that denote personal information such as names, surnames, or addresses. We also aim to extend the experimental part by comparing the effectiveness of our method with the baselines in identifying records with missing or non-standardized values. The initial results indicate that by applying  $PH$ , the gain in accuracy compared to the baselines is larger. Another interesting research avenue could be the adaptation of our method to the privacy-preserving context by applying two-party techniques [28]. The compact data structures used for representing the records could be an ideal fit in the protocols introduced in [17, 19] which are used for comparing those records in a secure manner.

## 8. REFERENCES

- [1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *CACM*, 51(1):117 – 122, 2008.
- [2] P. Christen. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer, Data-Centric Sys. and Appl., 2012.
- [3] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *TKDE*, 24(9):1537 – 1555, 2012.
- [4] P. Christen. Preparation of a real voter data set for record linkage and duplicate detection research. Tech. Rep., The Australian National Univ., 2013.

- [5] P. Christen, R. Gayler, and D. Hawking. Similarity-aware indexing for real-time entity resolution. In *CIKM*, pages 1565–1568, 2009.
- [6] W. W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *SIGKDD*, pages 475–480, 2002.
- [7] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG*, pages 253–262, 2004.
- [8] T. de Vries, H. Ke, S. Chawla, and P. Christen. Robust Record Linkage Blocking Using Suffix Arrays and Bloom Filters. *TKDD*, 5(2), 2011.
- [9] E. Durham. *A Framework For Accurate Efficient Private Record Linkage*. PhD thesis, Vanderbilt Univ., US, 2012.
- [10] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1):1–16, 2007.
- [11] L. Gravano, P. Ipeirotis, P. Koudas, and D. Srivastava. Text joins for data cleansing and integration in an RDBMS. In *ICDE*, pages 729–731, 2003.
- [12] M. Hernandez and S. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, pages 127–138, 1995.
- [13] G. Hjaltson and H. Samet. Properties of embedding methods for similarity searching in metric spaces. *TPAMI*, 25(5):530–549, 2003.
- [14] L. Jin, C. Li, and S. Mehrotra. Efficient Record Linkage In Large Data Sets. In *DASFAA*, pages 137–146, 2003.
- [15] D. Karapiperis and V. Verykios. A distributed framework for scaling up LSH-based computations in privacy preserving record linkage. In *BCI*, pages 102–109, 2013.
- [16] D. Karapiperis and V. Verykios. A Distributed Near-Optimal LSH-based Framework for Privacy-Preserving Record Linkage. *COMSIS*, 11(2):745–763, 2014.
- [17] D. Karapiperis and V. Verykios. An LSH-based Blocking Approach with a Homomorphic Matching Technique for Privacy-Preserving Record Linkage. *TKDE*, 27(4):909–921, 2015.
- [18] H. Kim and D. Lee. Fast Iterative Hashed Record Linkage for Large-Scale Data Collections. In *EDBT*, pages 525–536, 2010.
- [19] M. Kuzu, M. Kantarcioglu, A. Inan, E. Bertino, E. Durham, and B. Malin. Efficient privacy-aware record integration. In *EDBT*, pages 167–178, 2013.
- [20] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.
- [21] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: a partition-based method for similarity joins. In *PVLDB*, pages 253–264, 2011.
- [22] I. Miller and J. Freund. *Probability and Statistics for Engineers*. Prentice-Hall, 1977.
- [23] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge Univ. Press, 1995.
- [24] A. M. N. Koudas and D. Srivastava. Flexible string matching against large databases in practice. In *Vldb*, pages 1086–1094, 2004.
- [25] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD*, pages 1033–1044, 2011.
- [26] B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. Wiley, 1996.
- [27] R. Schnell, T. Bachteler, and J. Reiher. Privacy-preserving record linkage using Bloom filters. *Central Medical Inf. and Decision Making*, 9, 2009.
- [28] D. Vatsalan, P. Christen, and V. Verykios. Efficient two-party private blocking based on sorted nearest neighborhood clustering. In *CIKM*, pages 1949–1958, 2013.
- [29] D. Vatsalan, P. Christen, and V. Verykios. A taxonomy of privacy-preserving record linkage techniques. *JIS*, 38(6):946–969, 2013.
- [30] J. Wang, J. Feng, J. Wang, and G. Li. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. In *PVLDB*, pages 1219–1230, 2010.
- [31] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD*, pages 759–770, 2009.
- [32] M. Weis, F. Naumann, U. Jehle, J. Lufter, , and H. Schuster. Industry-scale duplicate detection. In *PVLDB*, pages 1253–1264, 2008.
- [33] S. Whang, D. Marmaros, and H. Garcia-Molina. Pay-as-you-go entity resolution. *TKDE*, 25(5):1111–1124, 2013.
- [34] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD*, pages 219–232, 2009.
- [35] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. In *PVLDB*, pages 933–944, 2008.
- [36] C. Xiao, W. Wang, X. Lin, and J. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.



# Scaling Entity Resolution to Large, Heterogeneous Data with Enhanced Meta-blocking

George Papadakis<sup>§</sup>, George Papastefanatos<sup>#</sup>, Themis Palpanas<sup>◇</sup>, Manolis Koubarakis<sup>§</sup>

<sup>◇</sup> Paris Descartes University, France [themis@mi.parisdescartes.fr](mailto:themis@mi.parisdescartes.fr)

<sup>#</sup>IMIS, Research Center “Athena”, Greece [gapas@imis.athena-innovation.gr](mailto:gapas@imis.athena-innovation.gr)

<sup>§</sup>Dep. of Informatics & Telecommunications, Uni. Athens, Greece [{gpadadis, koubarak}@di.uoa.gr](mailto:{gpadadis, koubarak}@di.uoa.gr)

## ABSTRACT

Entity Resolution constitutes a quadratic task that typically scales to large entity collections through blocking. The resulting blocks can be restructured by Meta-blocking in order to significantly increase precision at a limited cost in recall. Yet, its processing can be time-consuming, while its precision remains poor for configurations with high recall. In this work, we propose new meta-blocking methods that improve precision by up to an order of magnitude at a negligible cost to recall. We also introduce two efficiency techniques that, when combined, reduce the overhead time of Meta-blocking by more than an order of magnitude. We evaluate our approaches through an extensive experimental study over 6 real-world, heterogeneous datasets. The outcomes indicate that our new algorithms outperform all meta-blocking techniques as well as the state-of-the-art methods for block processing in all respects.

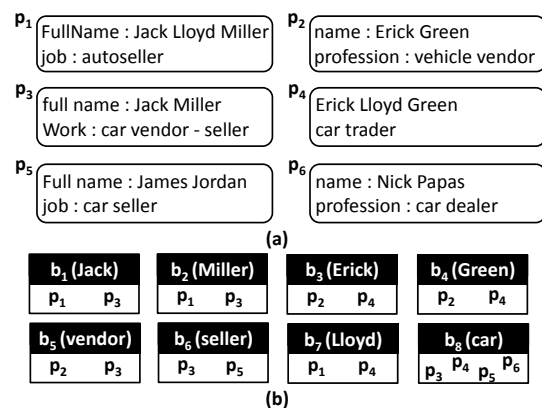
## 1. INTRODUCTION

A common task in the context of Web Data is *Entity Resolution* (ER), i.e., the identification of different entity profiles that pertain to the same real-world object. ER suffers from low efficiency, due to its inherently quadratic complexity: every entity profile has to be compared with all others. This problem is accentuated by the continuously increasing volume of heterogeneous Web Data; *LOD-Stats*<sup>1</sup> recorded around 1 billion triples for Linked Open Data in December, 2011, which had grown to 85 billion by September, 2015. Typically, ER scales to these volumes of data through *blocking* [4].

The goal of blocking is to boost precision and time efficiency at a controllable cost in recall [4, 5, 21]. To this end, it groups similar profiles into clusters (called *blocks*) so that it suffices to compare the profiles within each block [7, 8]. Blocking methods for Web Data are confronted with high levels of noise, not only in attribute values, but also in attribute names. In fact, they involve an unprecedented schema heterogeneity: Google Base<sup>2</sup> alone encompasses 100,000 distinct schemata that correspond to 10,000 entity types [17]. Most blocking methods deal with these high levels of

<sup>1</sup><http://stats.lod2.eu>

<sup>2</sup><http://www.google.com/base>



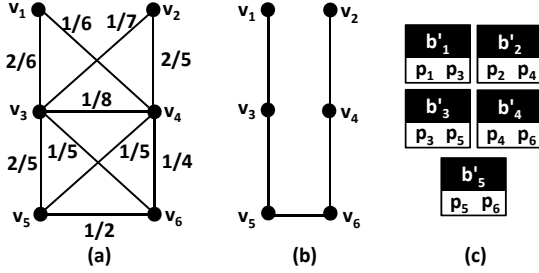
**Figure 1: (a) A set of entity profiles, and (b) the corresponding blocks produced by Token Blocking.**

schema heterogeneity through a schema-agnostic functionality that completely disregards schema information and semantics [5]. They also rely on *redundancy*, placing every entity profile into multiple blocks so as to reduce the likelihood of missed matches [4, 22].

The simplest method of this type is Token Blocking [21]. In essence, it splits the attribute values of every entity profile into tokens based on whitespace; then, it creates a separate block for every token that appears in at least two profiles. To illustrate its functionality, consider the entity profiles in Figure 1(a), where  $p_1$  and  $p_2$  match with  $p_3$  and  $p_4$ , respectively; Token Blocking clusters them in the blocks of Figure 1(b). Despite the schema heterogeneity and the noisy values, both pairs of duplicates co-occur in at least one block. Yet, the total cost is 13 comparisons, which is rather high, given that the brute-force approach executes 15 comparisons.

This is a general trait of redundancy-based blocking methods: in their effort to achieve high recall in the context of noisy and heterogeneous data, they produce a large number of unnecessary comparisons. These come in two forms [22, 23]: the *redundant* ones repeatedly compare the same entity profiles across different blocks, while the *superfluous* ones compare non-matching profiles. In our example,  $b_2$  and  $b_4$  contain one redundant comparison each, which are repeated in  $b_1$  and  $b_3$ , respectively; all other blocks entail superfluous comparisons between non-matching entity profiles, except for the redundant comparison  $p_3$ - $p_5$  in  $b_8$  (it is repeated in  $b_6$ ). In total, the blocks of Figure 1(b) involve 3 redundant and 8 superfluous out of the 13 comparisons.

**Block Processing.** To improve the quality of redundancy-based blocks, methods such as Meta-blocking [5, 7, 22], Comparison Propagation [21] and Iterative Blocking [27] aim to process them in the optimal way (see Section 2 for more details). Among these methods, Meta-blocking achieves the best balance between preci-



**Figure 2:** (a) A blocking graph extracted from the blocks in Figure 1(b), (b) one of the possible edge-centric pruned blocking graphs, and (c) the new blocks derived from it.

sion and recall [22, 23], and is the focus of this work.

Meta-blocking restructures a block collection  $B$  into a new one  $B'$  that contains a significantly lower number of unnecessary comparisons, while detecting almost the same number of duplicates. It operates in two steps [7, 22, 23]: first, it transforms  $B$  into the blocking graph  $G_B$ , which contains a vertex  $v_i$  for every entity profile  $p_i$  in  $B$ , and an edge  $e_{i,j}$  for every pair of *co-occurring profiles*  $p_i$  and  $p_j$  (i.e., entity profiles sharing at least one block). Figure 2(a) depicts the graph for the blocks in Figure 1(b). As no parallel edges are constructed, every pair of entities is compared at most once, thus eliminating all *redundant* comparisons.

Second, it annotates every edge with a weight analogous to the likelihood that the incident entities are matching. For instance, the edges in Figure 2(a) are weighted with the Jaccard similarity of the lists of blocks associated with their incident entity profiles. The lower the weight of an edge, the more likely it is to connect non-matching entities. Therefore, Meta-blocking discards most *superfluous* comparisons by pruning the edges with low weights. A possible approach is to discard all edges with a weight lower than the overall mean weight (1/4). This yields the pruned graph in Figure 2(b). The restructured block collection  $B'$  is formed by creating a new block for every retained edge – as depicted in Figure 2(c). Note that  $B'$  maintains the original recall, while reducing the comparisons from 13 to just 5.

**Open issues.** Despite the significant enhancements in efficiency, Meta-blocking suffers from two drawbacks:

(i) There is plenty of room for raising its precision, especially for the configurations that are more robust to recall. The reason is that they retain a considerable portion of redundant and superfluous comparisons. This is illustrated in our example, where the restructured blocks of Figure 2(c) contain 3 superfluous comparisons in  $b'_3$ ,  $b'_4$  and  $b'_5$ .

(ii) The processing of voluminous datasets involves a significant overhead. The corresponding blocking graphs comprise millions of nodes that are strongly connected with billions of edges. Inevitably, the pruning of such graphs is very time-consuming; for example, a graph with 3.3 million nodes and 35.8 billion edges requires 16 hours, on average, on commodity hardware (see Section 6.3).

**Proposed Solution.** In this paper, we describe novel techniques for overcoming both weaknesses identified above.

First, we speed up Meta-blocking in two ways:

(i) We introduce *Block Filtering*, which intelligently removes profiles from blocks, in which their presence is unnecessary. This acts as a pre-processing technique that shrinks the blocking graph, discarding more than half of its unnecessary edges, on average. As a result, the running time is also reduced to half, on average.

(ii) We accelerate the creation and the pruning of the blocking graph by minimizing the computational cost for edge weighting, which is the bottleneck of Meta-blocking. Our approach reduces its running time by 30% to 70%.

In combination, these two techniques restrict drastically the overhead of Meta-blocking even on commodity hardware. For example, the blocking graph mentioned earlier is now processed within just 3 hours, instead of 16.

Second, we enhance the precision of Meta-blocking in two ways:

(i) We redefine two pruning algorithms so that they produce restructured blocks with no redundant comparisons. On average, they save 30% more comparisons for the same recall.

(ii) We introduce two new pruning algorithms that rely on a generic property of the blocking graph: the reciprocal links. That is, our algorithms retain only the edges that are important for both incident profiles. Their recall is slightly lower than the existing techniques, but precision raises by up to an order of magnitude.

We analytically examine the performance of our methods using 6 real-world established benchmarks, which range from few thousands to several million entities. Our experimental results designate that our algorithms consistently exhibit the best balance between recall, precision and run-time for the main types of ER applications among all meta-blocking techniques. They also outperform the best relevant methods in the literature to a significant extent.

**Contributions & Paper Organization.** In summary, we make the following contributions:

- We improve the running time of Meta-blocking by an order of magnitude in two complementary ways: by cleaning the blocking graph from most of its noisy edges, and by accelerating the estimation of edge weights.
- We present four new pruning algorithms that raise precision by 30% to 100% at a small (if any) cost in recall.
- We experimentally verify the superior performance of our new methods through an extensive study over 6 datasets with different characteristics. In this way, our experimental results provide insights into the best configuration for Meta-blocking, depending on the data and the application at hand. The code and the data of our experiments are publicly available for any interested researcher.<sup>3</sup>

The rest of the paper is structured as follows: Section 2 delves into the most relevant works in the literature, while Section 3 elaborates on the main notions of Meta-blocking. In Section 4, we introduce two methods for minimizing its running time, and in Section 5, we present new pruning algorithms that boost the precision of Meta-blocking at no or limited cost in recall. Section 6 presents our thorough experimental evaluation, while Section 7 concludes the paper along with directions for future work.

## 2. RELATED WORK

Entity Resolution has been the focus of numerous works that aim to tame its quadratic complexity and scale it to large volumes of data [4, 8]. Blocking is the most popular among the proposed approximate techniques [5, 7]. Some blocking methods produce disjoint blocks, such as Standard Blocking [9]. Their majority, though, yields overlapping blocks with redundant comparisons in an effort to achieve high recall in the context of noisy and heterogeneous data [4]. Depending on the interpretation of redundancy, blocking methods are distinguished into three categories [22]:

(i) The *redundancy-positive* methods ensure that the more blocks two entity profiles share, the more likely they are to be matching. In this category fall the Suffix Arrays [1], Q-grams Blocking [12], Attribute Clustering [21] and Token Blocking [21].

(ii) The *redundancy-negative* methods ensure that the most similar entity profiles share just one block. In Canopy Clustering [19], for instance, the entity profiles that are highly similar to the cur-

<sup>3</sup>See <http://sourceforge.net/projects/erframework> for both the code and the datasets.

rent seed are removed from the pool of candidate matches and are exclusively placed in its block.

(iii) The *redundancy-neutral* methods yield overlapping blocks, but the number of common blocks between two profiles is irrelevant to their likelihood of matching. As such, consider the single-pass Sorted Neighborhood [13]: all pairs of profiles co-occur in the same number of blocks, which is equal to the size of the sliding window.

Another line of research focuses on developing techniques that optimize the processing of an existing block collection, called *block processing methods*. In this category falls Meta-blocking [7], which operates exclusively on top of redundancy-positive blocking methods [20, 21]. Its pruning can be either unsupervised [22] or supervised [23]. The latter achieves higher accuracy than the former, due to the composite pruning rules that are learned by a classifier trained over a set of labelled edges. In practice, though, its utility is limited, as there is no effective and efficient way for extracting the required training set from the input blocks. For this reason, we exclusively consider unsupervised Meta-blocking in this work.

Other prominent block processing methods are the following:

(i) *Block Purging* [21] aims for discarding oversized blocks that are dominated by redundant and superfluous comparisons. It automatically sets an upper limit on the comparisons that can be contained in a valid block and purges those blocks that exceed it. Its functionality is coarser and, thus, less accurate than Meta-blocking, because it targets entire blocks instead of individual comparisons. However, it is complementary to Meta-blocking and is frequently used as a pre-processing step [22, 23].

(ii) *Comparison Propagation* [21] discards all redundant comparisons from a block collection without any impact on recall. In a small scale, this can be accomplished directly, using a central data structure  $H$  that hashes all executed comparisons; then, a comparison is executed only if it is not contained in  $H$ . Yet, in the scale of billions of comparisons, Comparison Propagation can only be accomplished indirectly: the input blocks are enumerated according to their processing order and the Entity Index is built. This is an inverted index that points from entity ids to block ids. Then, a comparison  $p_i-p_j$  in block  $b_k$  is executed (i.e., non-redundant) only if it satisfies the Least Common Block Index condition (LeCoBI for short). That is, if the id  $k$  of the current block  $b_k$  equals the least common block id of the profiles  $p_i$  and  $p_j$ . Comparison Propagation is competitive to Meta-blocking, but targets only redundant comparisons. We compare their performance in Section 6.4.

(ii) *Iterative Blocking* [27] propagates all identified duplicates to the subsequently processed blocks so as to save repeated comparisons and to detect more duplicates. Hence, it improves both precision and recall. It is competitive to Meta-blocking, too, but it targets exclusively redundant comparisons between matching profiles. We employ it as our second baseline method in Section 6.4.

### 3. PRELIMINARIES

**Entity Resolution.** An *entity profile*,  $p$ , is defined as a uniquely identified collection of name-value pairs that describe a real-world object. A set of profiles is called *entity collection*,  $E$ . Given  $E$ , the goal of ER is to identify all profiles that describe the same real-world object; two such profiles,  $p_i$  and  $p_j$ , are called *duplicates* ( $p_i \equiv p_j$ ) and their comparison is called *matching*. The set of all duplicates in the input entity collection  $E$  is denoted by  $D(E)$ , with  $|D(E)|$  symbolizing its size (i.e., the number of existing duplicates).

Depending on the input entity collection(s), we identify two ER tasks [4, 21, 22]: (i) *Dirty ER* takes as input a single entity collection with duplicates and produces as output a set of equivalence clusters. (ii) *Clean-Clean ER* receives two duplicate-free, but overlapping entity collections,  $E_1$  and  $E_2$ , and identifies the matching

entity profiles between them. In the context of Databases, the former task is called *Deduplication* and the latter *Record Linkage* [4].

Blocking improves the run-time of both ER tasks by grouping similar entity profiles into blocks so that comparisons are limited between co-occurring profiles. Placing an entity profile into a block is called *block assignment*. Two profiles,  $p_i$  and  $p_j$ , assigned to the same block are called *co-occurring* and their comparison is denoted by  $c_{i,j}$ . An individual block is symbolized by  $b$ , with  $|b|$  denoting its *size* (i.e., number of profiles) and  $\|b\|$  denoting its *cardinality* (i.e., number of comparisons). A set of blocks  $B$  is called *block collection*, with  $|B|$  denoting its size (i.e., number of blocks) and  $\|B\|$  its cardinality (i.e., total number of comparisons):  $\|B\| = \sum_{b \in B} \|b\|$ .

**Performance Measures.** To assess the effectiveness of a blocking method, we follow the best practice in the literature, which treats entity matching as an orthogonal task [4, 5, 7]. We assume that two duplicate profiles can be detected using any of the available matching methods as long as they co-occur in at least one block.  $D(B)$  stands for the set of co-occurring duplicate profiles and  $|D(B)|$  for its size (i.e., the number of detected duplicates).

In this context, the following measures are typically used for estimating the *effectiveness* of a block collection  $B$  that is extracted from the input entity collection  $E$  [4, 5, 22]:

(i) *Pairs Quality (PQ)* corresponds to precision, assessing the portion of comparisons that involve a *non-redundant* pair of duplicates. In other words, it considers as true positives the matching comparisons and as false positives the superfluous and the redundant ones (given that some of the redundant comparisons involve duplicate profiles,  $PQ$  offers a pessimistic estimation of precision). More formally,  $PQ = |D(B)|/\|B\|$ .  $PQ$  takes values in the interval  $[0, 1]$ , with higher values indicating higher precision for  $B$ .

(ii) *Pairs Completeness (PC)* corresponds to recall, assessing the portion of existing duplicates that can be detected in  $B$ . More formally,  $PC = |D(B)|/|D(E)|$ .  $PC$  is defined in the interval  $[0, 1]$ , with higher values indicating higher recall.

The goal of blocking is to maximize both  $PC$  and  $PQ$  so that the overall effectiveness of ER exclusively depends on the accuracy of the entity matching method. This requires that  $|D(B)|$  is maximized, while  $\|B\|$  is minimized. However, there is a clear trade-off between  $PC$  and  $PQ$ : the more comparisons are executed (higher  $\|B\|$ ), the more duplicates are detected (higher  $|D(B)|$ ), thus increasing  $PC$ ; given, though, that  $\|B\|$  increases quadratically for a linear increase in  $|D(B)|$  [10, 11],  $PQ$  is reduced. Hence, a blocking method is effective if it achieves a good balance between  $PC$  and  $PQ$ .

To assess the *time efficiency* of a block collection  $B$ , we use two measures [22, 23]:

(i) *Overhead Time (OTime)* measures the time required for extracting  $B$  either from the input entity collection  $E$  or from another block collection  $B'$ .

(ii) *Resolution Time (RTime)* is equal to  $OTime$  plus the time required to apply an entity matching method to all comparisons in the restructured blocks. As such, we use the Jaccard similarity of all tokens in the values of two entity profiles for entity matching – this approach does not affect the relative efficiency of the examined methods and is merely used for demonstration purposes.

For both measures, the lower their value, the more efficient is the corresponding block collection.

**Meta-blocking.** The redundancy-positive block collections place every entity profile into multiple blocks, emphasizing recall at the cost of very low precision. Meta-blocking aims for improving this balance by restructuring a redundancy-positive block collection  $B$  into a new one  $B'$  that contains a small part of the original unnecessary comparisons, while retaining practically the same recall [22]. More formally,  $PC(B') \approx PC(B)$  and  $PQ(B') \gg PQ(B)$ .

Weighting Schemes	Pruning Algorithms
1) Aggregate Reciprocal Comparisons (ARCS)	1) Cardinality Edge Pruning (CEP)
2) Common Blocks (CBS)	2) Cardinality Node Pruning (CNP)
3) Enhanced Common Blocks (ECBS)	3) Weighted Edge Pruning (WEP)
4) Jaccard Similarity (JS)	4) Weighted Node Pruning (WNP)
5) Enhanced Jaccard Similarity (EJS)	

**Figure 3: All configurations for the two parameters of Meta-blocking: the weighting scheme and the pruning algorithm.**

Aggregate Reciprocal Comparisons Scheme	$ARCS(p_i, p_j, B) = \sum_{b_k \in B_{ij}} \frac{1}{ b_k }$
Common Blocks Scheme	$CBS(p_i, p_j, B) =  B_{ij} $
Enhanced Common Blocks Scheme	$ECBS(p_i, p_j, B) = CBS(p_i, p_j, B) \cdot \log \frac{ B }{ B_i } \cdot \log \frac{ B }{ B_j }$
Jaccard Scheme	$JS(p_i, p_j, B) = \frac{ B_{ij} }{ B_i  +  B_j  -  B_{ij} }$
Enhanced Jaccard Scheme	$EJS(p_i, p_j, B) = JS(p_i, p_j, B) \cdot \log \frac{ V_B }{ v_i } \cdot \log \frac{ V_B }{ v_j }$

**Figure 4: The formal definition of the five weighting schemes.**  $B_i \subseteq B$  denotes the set of blocks containing  $p_i$ ,  $B_{i,j} \subseteq B$  the set of blocks shared by  $p_i$  and  $p_j$ , and  $|v_i|$  the degree of node  $v_i$ .

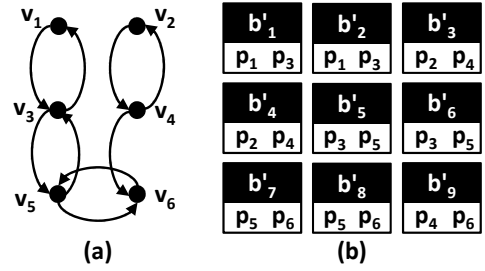
Central to this procedure is the blocking graph  $G_B$ , which captures the co-occurrences of profiles within the blocks of  $B$ . Its nodes correspond to the profiles in  $B$ , while its undirected edges connect the co-occurring profiles. The number of edges in  $G_B$  is called *graph size* ( $|E_B|$ ) and the number of nodes *graph order* ( $|V_B|$ ).

Meta-blocking prunes the edges of the blocking graph in a way that leaves the matching profiles connected. Its functionality is configured by two parameters: (i) the scheme that assigns weights to the edges, and (ii) the pruning algorithm that discards the edges that are unlikely to connect duplicate profiles. The two parameters are independent in the sense that every configuration of the one is compatible with any configuration of the other (see Figure 3).

In more detail, five schemes have been proposed for weighting the edges of the blocking graph [22]. Their formal definitions are presented in Figure 4. They all normalize their weights to  $[0, 1]$  so that the higher values correspond to edges that are more likely to connect matching profiles. The rationale behind each scheme is the following: ARCS captures the intuition that the smaller the blocks two profiles share, the more likely they are to be matching; CBS expresses the fundamental property of redundancy-positive block collections that two profiles are more likely to match, when they share many blocks; ECBS improves CBS by discounting the effect of the profiles that are placed in a large number of blocks; JS estimates the portion of blocks shared by two profiles; EJS improves JS by discounting the effect of profiles involved in too many non-redundant comparisons (i.e., they have a high node degree).

Based on these weighting schemes, Meta-blocking discards part of the edges of the blocking graph using an *edge-* or a *node-centric* pruning algorithm. The former iterates over the edges of the blocking graph and retains the globally best ones, as in Figure 2(b); the latter iterates over the nodes of the blocking graph and retains the locally best edges. An example of node-centric pruning appears in Figure 5(a); for each node in Figure 2(a), it has retained the incident edges that exceed the average weight of the neighborhood. For clarity, the retained edges are directed and outgoing, since they might be preserved in the neighborhoods of both incident profiles. Again, every retained edge forms a new block, yielding the restructured block collection in Figure 5(b).

Every pruning algorithm relies on a *pruning criterion*. Depending on its scope, this can be either a *global* criterion, which applies to the entire blocking graph, or a *local* one, which applies to an



**Figure 5: (a) One of the possible node-centric pruned blocking graphs for the graph in Figure 2(a). (b) The new blocks derived from the pruned graph.**

individual node neighborhood. With respect to its functionality, the pruning criterion can be a *weight threshold*, which specifies the minimum weight of the retained edges, or a *cardinality threshold*, which determines the maximum number of retained edges.

Every combination of a pruning algorithm with a pruning criterion is called *pruning scheme*. The following four pruning schemes were proposed in [22] and were experimentally verified to achieve a good balance between  $PC$  and  $PQ$ :

(i) *Cardinality Edge Pruning* (CEP) couples the edge-centric pruning with a global cardinality threshold, retaining the top- $K$  edges of the entire blocking graph, where  $K = \lfloor \sum_{b \in B} |b| / 2 \rfloor$ .

(ii) *Cardinality Node Pruning* (CNP) combines the node-centric pruning with a global cardinality threshold. For each node, it retains the top- $k$  edges of its neighborhood, with  $k = \lfloor \sum_{b \in B} |b| / |E| - 1 \rfloor$ .

(iii) *Weighted Edge Pruning* (WEP) couples edge-centric pruning with a global weight threshold equal to the average edge weight of the entire blocking graph.

(iv) *Weighted Node Pruning* (WNP) combines the node-centric pruning with a local weight threshold equal to the average edge weight of every node neighborhood.

The *weight-based* schemes, WEP and WNP, discard the edges that do not exceed their weight threshold and typically perform a shallow pruning that retains high recall [22]. The *cardinality-based* schemes, CEP and CNP, rank the edges of the blocking graph in descending order of weight and retain a specific number of the top ones. For example, if CEP retained the 4 top-weighted edges of the graph in Figure 2(a), it would produce the pruned graph of Figure 2(b), too. Usually, CEP and CNP perform deeper pruning than WEP and WNP, trading higher precision for lower recall [22].

**Applications of Entity Resolution.** Based on their performance requirements, we distinguish ER applications into two categories:

(i) The *efficiency-intensive* applications aim to minimize the response time of ER, while detecting the vast majority of the duplicates. More formally, their goal is to maximize precision ( $PQ$ ) for a recall ( $PC$ ) that exceeds 0.80. To this category belong real-time applications or applications with limited temporal resources, such as Pay-as-you-go ER [26], entity-centric search [25] and crowd-sourcing ER [6]. Ideally, their goal is to identify a new pair of duplicate entities with every executed comparison.

(ii) The *effectiveness-intensive* applications can afford a higher response time in order to maximize recall. At a minimum, recall ( $PC$ ) should not fall below 0.95. Most of these applications correspond to off-line batch processes like data cleaning in data warehouses, which practically call for almost perfect recall [2]. Yet, higher precision ( $PQ$ ) is pursued even in off-line applications so as to ensure that they scale to voluminous datasets.

Meta-blocking accommodates the effectiveness-intensive applications through the weight-based pruning schemes (WEP, WNP) and the efficiency-intensive applications through the cardinality-based schemes (CEP, CNP).

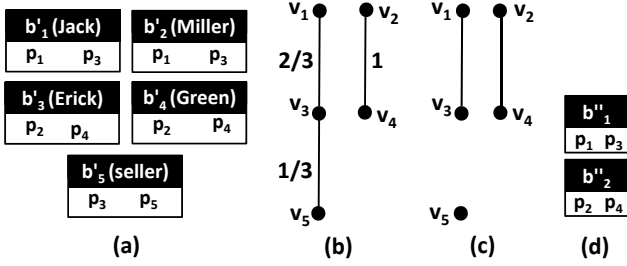


Figure 6: (a) The block collection produced by applying Block Filtering to the blocks in Figure 1(b), (b) the corresponding blocking graph, (c) the pruned blocking graph produced by WEP, and (d) the corresponding restructured block collection.

#### 4. TIME EFFICIENCY IMPROVEMENTS

We now propose two methods for accelerating the processing of Meta-blocking, minimizing its *OTime*: (i) Block Filtering, which operates as a pre-processing step that reduces the size of the blocking graph, and (ii) Optimized Edge Weighting, which minimizes the computational cost for the weighting of individual edges.

##### 4.1 Block Filtering

This approach is based on the idea that each block has a different importance for every entity profile it contains. For example, a block with thousands of profiles is usually superfluous for most of them, but it may contain a couple of matching entity profiles that do not co-occur in another block; for them, this particular block is indispensable. Based on this principle, Block Filtering restructures a block collection by removing profiles from blocks, in which their presence is not necessary. The *importance* of a block  $b_k$  for an individual entity profile  $p_i \in b_k$  is implicitly determined by the maximum number of blocks  $p_i$  participates in.

Continuing our example with the blocks in Figure 1(b), assume that their importance is inversely proportional to their id; that is,  $b_1$  and  $b_8$  are the most and the least important blocks, respectively. A possible approach to Block Filtering would be to remove every entity profile from the least important of its blocks, i.e., the one with the largest block id. The resulting block collection appears in Figure 6(a). We can see that Block Filtering reduces the 15 original comparisons to just 5. Yet, there is room for further improvements, due to the presence of 2 redundant comparisons, one in  $b'_2$  and another one in  $b'_4$ , and 1 superfluous in block  $b'_5$ . Using the JS weighting scheme, the graph corresponding to these blocks is presented in Figure 6(b) and the pruned graph produced by WEP appears in Figure 6(c). In the end, we get the 2 matching comparisons in Figure 6(d). This is a significant improvement over the 5 comparisons in Figure 2(c), which were produced by applying the same pruning scheme directly to the blocks of Figure 1(b).

In more detail, the functionality of Block Filtering is outlined in Algorithm 1. First, it orders the blocks of the input collection  $B$  in descending order of importance (Line 3). Then, it determines the maximum number of blocks per entity profile (Line 4). This requires an iteration over all blocks in order to count the block assignments per entity profile. Subsequently, it iterates over all blocks in the specified order (Line 5) and over all profiles in each block (Line 6). The profiles that have more block assignments than their threshold are discarded, while the rest are retained in the current block (Lines 7-10). In the end, the current block is retained only if it still contains at least two entity profiles (Lines 11-12).

The time complexity of this procedure is dominated by the sorting of blocks, i.e.,  $O(|B| \cdot \log |B|)$ . Its space complexity is linear with respect to the size of the input,  $O(|E|)$ , because it maintains a threshold and a counter for every entity profile.

#### Algorithm 1: Block Filtering.

---

**Input:**  $B$  the input block collection  
**Output:**  $B'$  the restructured block collection

```

1  $B' \leftarrow \{\}$ ;
2  $\text{counter}[] \leftarrow \{\}$ ; // count blocks per profile
3  $\text{orderBlocks}(B)$ ; // sort in descending importance
4  $\text{maxBlocks}[] \leftarrow \text{getThresholds}(B)$ ; // limit per profile
5 foreach  $b_k \in B$  do // check all blocks
6   foreach  $p_i \in b_k$  do // check all profiles
7     if  $\text{counter}[i] > \text{maxBlocks}[i]$  then
8        $b_k \leftarrow b_k \setminus p_i$ ; // remove profile
9     else
10       $\text{counter}[i]++$ ; // increment counter
11 if  $|b_k| > 1$  then // retain blocks with
12    $B' \leftarrow B' \cup b_k$ ; // at least 2 profiles
13 return  $B'$ ;

```

---

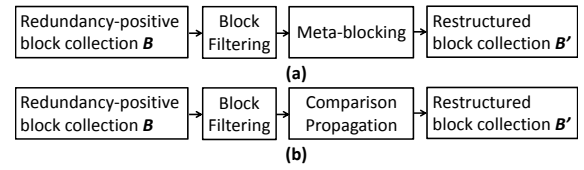


Figure 7: (a) Using Block Filtering for pre-processing the blocking graph of Meta-blocking, and (b) using Block Filtering as a graph-free Meta-blocking method.

The performance of Block Filtering is determined by two factors:

(i) The criterion that specifies the importance of a block  $b_i$ . This can be defined in various ways and ideally should be different for every profile in  $b_i$ . For higher efficiency, though, we use a criterion that is common for all profiles in  $b_i$ . It is also generic, applying to any block collection, independently of the underlying ER task or the schema heterogeneity. This criterion is the cardinality of  $b_i$ ,  $|b_i|$ , presuming that the less comparisons a block contains, the more important it is for its entities. Thus, Block Filtering sorts a block collection from the smallest block to the largest one.

(ii) The *filtering ratio* ( $r$ ) that determines the maximum number of block assignments per profile. It is defined in the interval  $[0, 1]$  and expresses the portion of blocks that are retained for each profile. For example,  $r=0.5$  means that each profile remains in the first half of its associated blocks, after sorting them in ascending cardinality. We experimentally fine-tune this parameter in Section 6.2.

Instead of a local threshold per entity profile, we could apply the same global threshold to all profiles. Preliminary experiments, though, demonstrated that this approach exhibits low performance, as the number of blocks associated with every profile varies largely, depending on the quantity and the quality of information it contains. This is particularly true for Clean-Clean ER, where  $E_1$  and  $E_2$  usually differ largely in their characteristics. Hence, it is difficult to identify the break-even point for a global threshold that achieves a good balance between recall and precision for all profiles.

Finally, it is worth noting that Block Filtering can be used in two fundamentally different ways, which are compared in Section 6.4.

(i) As a pre-processing method that prunes the blocking graph before applying Meta-blocking – see Figure 7(a).

(ii) As a graph-free Meta-blocking method that is combined only with Comparison Propagation – see Figure 7(b).

The latter workflow skips the blocking graph, operating on the level of individual profiles instead of profile pairs. Thus, it is expected to be significantly faster than all graph-based algorithms. If it achieves higher precision, as well, it outperforms the graph-based workflow in all respects, rendering the blocking graph unnecessary.

---

**Algorithm 2: Original Edge Weighting.**

---

**Input:**  $B$  the input block collection  
**Output:**  $W$  the set of edge weights

```
1  $W \leftarrow \{\}$ ;  
2  $EI \leftarrow \text{buildEntityIndex}(B)$ ;  
3 foreach  $b_k \in B$  do // check all blocks  
4   foreach  $c_{i,j} \in b_k$  do // check all comparisons  
5      $B_i \leftarrow EI.\text{getBlockList}(p_i)$ ;  
6      $B_j \leftarrow EI.\text{getBlockList}(p_j)$ ;  
7      $\text{commonBlocks} \leftarrow 0$ ;  
8     foreach  $m \in B_i$  do  
9       foreach  $n \in B_j$  do  
10        if  $m < n$  then break; // repeat until  
11        if  $n < m$  then continue; // finding common id  
12        if  $\text{commonBlocks} = 0$  then // 1st common id  
13          if  $m \neq k$  then // it violates LeCoBl  
14            break to next comparison;  
15           $\text{commonBlocks}++$ ;  
16      $w_{i,j} \leftarrow \text{calculateWeight}(\text{commonBlocks}, B_i, B_j)$ ;  
17      $W \leftarrow W \cup \{w_{i,j}\}$ ;  
18 return  $W$ ;
```

---

## 4.2 Optimized Edge Weighting

A complementary way of speeding up Meta-blocking is to accelerate its bottleneck, i.e., the estimation of edge weights. Intuitively, we want to minimize the computational cost of the procedure that derives the weight from every individual edge. Before we explain in detail our solution, we give some background, by describing how the existing Edge Weighting algorithm operates.

The blocking graph cannot be materialized in memory in the scale of million nodes and billion edges. Instead, it is implemented implicitly. The key idea is that every edge  $e_{i,j}$  in the blocking graph  $G_B$  corresponds to a non-redundant comparison  $c_{i,j}$  in the block collection  $B$ . In other words, a comparison  $c_{i,j}$  in  $b_k \in B$  defines an edge  $e_{i,j}$  in  $G_B$  as long as it satisfies the LeCoBl condition (see Section 2). The condition is checked with the help of the Entity Index during the core process that derives the blocks shared by  $p_i$  and  $p_j$ .

In more detail, the original implementation of Edge Weighting is outlined in Algorithm 2. Note that  $B_i$  stands for the *block list* of  $p_i$ , i.e., the set of block ids associated with  $p_i$ , sorted from the smallest to the largest one. The core process appears in Lines 7-15 and relies on the established process of Information Retrieval for intersecting the posting lists of two terms while answering a keyword query [18]: it iterates over the blocks lists of two co-occurring profiles in parallel, incrementing the counter of common blocks for every id they share (Line 15). This process is terminated in case the first common block id does not coincide with the id of the current block  $b_k$ , thus indicating a redundant comparison (Lines 12-14).

Our observation is that since this procedure is repeated for every comparison in  $B$ , a more efficient implementation would significantly reduce the run time of Meta-blocking. To this end, we develop a filtering technique inspired by similarity joins [14].

*Prefix Filtering* [3, 14] is a prominent method, which prunes dissimilar pairs of strings with the help of the minimum similarity threshold  $t$  that is determined *a-priori*;  $t$  can be defined with respect to various similarity metrics that are essentially equivalent, due to a set of transformations [14]. Without loss of generality, we assume in the following that  $t$  is normalized in  $[0, 1]$ , just like the edge weights, and that it expresses a Jaccard similarity threshold.

Adapted to edge weighting, Prefix Filtering represents every profile  $p_i$  by the  $\lfloor (1-t) \cdot |B_i| \rfloor + 1$  smallest blocks of  $B_i$ . The idea is that pairs having disjoint representations cannot exceed the similarity

---

**Algorithm 3: Optimized Edge Weighting.**

---

**Input:**  $B$  the input block collection,  $E$  the input entity collection  
**Output:**  $W$  the set of edge weights

```
1  $W \leftarrow \{\}$ ;  $\text{commonBlocks}[] \leftarrow \{\}$ ;  $\text{flags}[] \leftarrow \{\}$ ;  
2  $EI \leftarrow \text{buildEntityIndex}(B)$ ;  
3 foreach  $p_i \in E$  do // check all profiles  
4    $B_i \leftarrow EI.\text{getBlockList}(p_i)$ ;  
5    $\text{neighbors} \leftarrow \{\}$ ; // set of co-occurring profiles  
6   foreach  $b_k \in B_i$  do // check all associated blocks  
7     foreach  $p_j (\neq p_i) \in b_k$  do // co-occurring profile  
8       if  $\text{flags}[j] \neq i$  then  
9          $\text{flags}[j] = i$ ;  
10         $\text{commonBlocks}[j] = 0$ ;  
11         $\text{neighbors} \leftarrow \text{neighbors} \cup \{p_j\}$ ;  
12         $\text{commonBlocks}[j]++$ ;  
13   foreach  $p_j \in \text{neighbors}$  do  
14      $B_j \leftarrow EI.\text{getBlockList}(p_j)$ ;  
15      $w_{i,j} \leftarrow \text{calculateWeight}(\text{commonBlocks}[j], B_i, B_j)$ ;  
16      $W \leftarrow W \cup \{w_{i,j}\}$ ;  
17 return  $W$ ;
```

---

threshold  $t$ . For example, for  $t=0.8$  an edge  $e_{i,j}$  could be pruned using 1/5 of  $B_i$  and  $B_j$ , thus speeding up the nested loops in Lines 8-9 of Algorithm 2. Yet, there are 3 problems with this approach:

(i) For the weight-based algorithms, the pruning criterion  $t$  can only be determined *a-posteriori* – after averaging all edge weights in the entire graph (WEP), or in a node neighborhood (WNP). As a result, the optimizations of Prefix Filtering apply only to the pruning phase of WEP and WNP and not to the initial construction of the blocking graph.

(ii) For the cardinality-based algorithms CEP and CNP,  $t$  equals the minimum edge weight in the sorted stack with the top-weighted edges. Thus, its value is continuously modified and cannot be used for a-priori building optimized entity representations.

(iii) Preliminary experiments demonstrated that  $t$  invariably takes very low values, below 0.1, for all combinations of pruning algorithms and weighting schemes. These low thresholds force all versions of Prefix Filtering to consider the entire block lists  $B_i$  and  $B_j$  as entity representations, thus ruining their optimizations.

For these reasons, we propose a novel implementation that is independent of the similarity threshold  $t$ . Our approach is outlined in Algorithm 3. Instead of iterating over all comparisons in  $B$ , it iterates over all input profiles in  $E$  (Line 3). The core procedure in Lines 6-12 works as follows: for every profile  $p_i$ , it iterates over all co-occurring profiles in the associated blocks and records their frequency in an array. At the end of the process,  $\text{commonBlocks}[j]$  indicates the number of blocks shared by  $p_i$  and  $p_j$ . This information is then used in Lines 13-16 for estimating the weight  $w_{i,j}$ . This method is reminiscent of *ScanCount* [16]. Note that the array  $\text{flags}$  helps us to avoid reallocating memory for  $\text{commonBlocks}$  in every iteration, a procedure that would be costly, due to its size,  $|E|$  [16];  $\text{neighbors}$  is a hash set that stores the unique profiles that co-occur with  $p_i$ , gathering the distinct neighbors of node  $n_i$  in the blocking graph without evaluating the LeCoBl condition.

## 4.3 Discussion

The average time complexity of Algorithm 2 is  $O(2 \cdot BPE \cdot \|B\|)$ , where  $BPE(B) = \sum_{b \in B} |b|/|E|$  is the average number of blocks associated with every profile in  $B$ ;  $2 \cdot BPE$  corresponds to the average computational cost of the nested loops in Lines 8-9, while  $\|B\|$  stems from the nested loops in Lines 3-4, which iterate over all comparisons in  $B$ .

Block Filtering improves this time complexity in two ways:

(i) It reduces  $\|B\|$  by discarding a large part of the redundant and superfluous comparisons in  $B$ .

(ii) It reduces  $2\cdot BPE$  to  $2\cdot r\cdot BPE$  by removing every profile from  $(1-r)\cdot 100\%$  of its associated blocks, where  $r$  is the filtering ratio.<sup>4</sup>

The computational cost of Algorithm 3 is determined by two procedures that yield an average time complexity of  $O(\|B\| + |\bar{v}|\cdot|E|)$ :

(i) The three nested loops in Lines 3-7. For every block  $b$ , these loops iterate over  $|b|-1$  of its entity profiles (i.e., over all profiles except  $p_i$ ) for  $|b|$  times – once for each entity profile. Therefore, the process in Lines 8-12 is repeated  $|b|\cdot(|b|-1)=\|b\|$  times and the overall complexity of the three nested loops is  $O(\|B\|)$ .

(ii) The loop in Lines 13-16. Its cost is analogous to the average node degree  $|\bar{v}|$ , i.e., the average number of neighbors per profile. It is repeated for every profile and, thus, its overall cost is  $O(|\bar{v}|\cdot|E|)$ .

Comparing the two algorithms, we observe that the optimized implementation minimizes the computational cost of the process that is applied to each comparison: instead of intersecting the associated block lists, it merely updates 2-3 cells in 2 arrays and adds an entity id in the set of neighboring profiles. The former process involves two nested loops with an average cost of  $O(2\cdot BC)$ , while the latter processes have a constant complexity,  $O(1)$ . Note that Algorithm 3 incorporates the loop in Lines 13-16, which has complexity of  $O(|\bar{v}|\cdot|E|)$ . In practice, though, this is considerably lower than both  $O(2\cdot BPE\cdot\|B\|)$  and  $O(\|B\|)$ , as we show in Section 6.3.

## 5. PRECISION IMPROVEMENTS

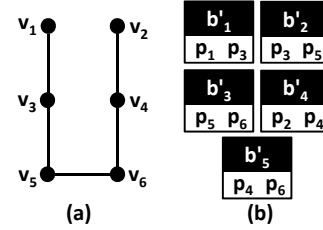
We now introduce two new pruning algorithms that significantly enhance the effectiveness of Meta-blocking, increasing precision for similar levels of recall: (i) Redefined Node-centric Pruning, which removes all redundant comparisons from the restructured blocks of CNP and WNP, and (ii) Reciprocal Pruning, which infers the most promising matches from the reciprocal links of the directed pruned blocking graphs of CNP and WNP.

There are several reasons why we exclusively focus on improving the precision of CNP and WNP:

(i) Node-centric algorithms are quite robust to recall, since they retain the most likely matches for each node and, thus, guarantee to include every profile in the restructured blocks. Edge-centric algorithms do not provide such guarantees, because they retain the overall best edges from the entire graph.

(ii) Node-centric algorithms are more flexible, in the sense that their performance can be improved in generic, algorithmic ways. Instead, edge-centric algorithms improve their performance only with threshold fine-tuning. This approach, though, is application-specific, as it is biased by the characteristics of the block collection at hand. Another serious limitation is that some superfluous comparisons cannot be pruned without discarding part of the matching ones. For instance, the superfluous edge  $e_{5,6}$  in Figure 2(a) has a higher weight than both matching edges  $e_{1,3}$  and  $e_{2,4}$ .

(iii) There is more room for improvement in node-centric algorithms, because they exhibit lower precision than the edge-centric ones. They process every profile independently of the others and they often retain the same edge for both incident profiles, thus yielding redundant comparisons. They also retain more superfluous comparisons than the edge-centric algorithms in most cases [22]. As an example, consider Clean-Clean ER: every profile from both



**Figure 8: (a) The undirected pruned blocking graph corresponding to the directed one Figure 5(a), and (b) the corresponding block collection.**

entity collections retains its connections with several incident nodes, even though only one of them is matching with it.

Nevertheless, our experiments in Section 6.4 demonstrate that our new node-centric algorithms outperform the edge-centric ones, as well. They also cover both efficiency- and effectiveness-intensive ER applications, enhancing both CNP and WNP.

### 5.1 Redefined Node-centric Pruning

We can enhance the precision of both CNP and WNP without any impact on recall by discarding all the redundant comparisons they retain. Assuming that the blocking graph is materialized, a straightforward approach is to convert the directed pruned graph into an undirected one by connecting every pair of neighboring nodes with a single undirected edge – even if they are reciprocally linked. In the extreme case where every retained edge has a reciprocal link, this saves 50% more comparisons and doubles precision.

As an example, the directed pruned graph in Figure 5(a) can be transformed into the undirected pruned graph in Figure 8(a); the resulting blocks, which are depicted in Figure 8(b), reduce the retained comparisons from 9 to 5, while maintaining the same recall as the blocks in Figure 5(b):  $p_1-p_3$  co-occur in  $b'_1$  and  $p_2-p_4$  in  $b'_4$ .

Yet, it is impossible to materialize the blocking graph in memory in the scale of billions of edges. Instead, the graph is implicitly implemented as explained in Section 4.2. In this context, a straightforward solution for improving CNP and WNP is to apply Comparison Propagation to their output. This approach, though, entails a significant overhead, as it evaluates the LeCoBI condition for every retained comparison; on average, its total cost is  $O(2\cdot BPE\cdot\|B\|)$ .

The best solution is to redefine CNP and WNP so that Comparison Propagation is integrated into their functionality. The new implementations are outlined in Algorithms 4 and 5, respectively. In both cases, the processing consists of two phases:

(i) The first phase involves a node-centric functionality that goes through the nodes of the blocking graph and derives the pruning criterion from their neighborhood. A central data structure stores the top- $k$  nearest neighbors (CNP) or the weight threshold (WNP) per node neighborhood. In total, this phase iterates twice over every edge of the blocking graph – once for each incident node.

(ii) The second phase operates in an edge-centric fashion that goes through all edges, retaining those that satisfy the pruning criterion for at least one of the incident nodes. Thus, every edge is retained at most once, even if it is important for both incident nodes.

In more detail, Redefined CNP iterates over all nodes of the blocking graph to extract their neighborhood and calculate the corresponding cardinality threshold  $k$  (Lines 2-4 in Algorithm 4). Then it iterates over the edges of the current neighborhood and places the top- $k$  weighted ones in a sorted stack (Lines 5-8). In the second phase, it iterates over all edges and retains those contained in the sorted stack of either of the incident profiles (Lines 10-13).

Similarly, Redefined WNP first iterates over all nodes of the blocking graph to extract their neighborhood and to estimate the

<sup>4</sup>This seems similar to the effect of Prefix Filtering, but there are fundamental differences: (i) Prefix Filtering does not reduce the number of executed comparisons; it just accelerates their pruning. (ii) Prefix Filtering relies on a similarity threshold for pairs of profiles, while the filtering ratio  $r$  pertains to individual profiles.

---

**Algorithm 4: Redefined Cardinality Node Pruning.**

---

**Input:** (i)  $G_B^{in}$  the blocking graph, and  
(ii)  $ct$  the function defining the local cardinality thresholds.  
**Output:**  $G_B^{out}$  the pruned blocking graph

```
1 SortedStacks[]  $\leftarrow$  {}; // sorted stack per node
2 foreach  $v_i \in V_B$  do // for every node
3    $G_{v_i} \leftarrow$  getNeighborhood( $v_i, G_B$ );
4    $k \leftarrow ct(G_{v_i})$ ; // get local cardinality threshold
5   foreach  $e_{i,j} \in E_{v_i}$  do // add every adjacent edge
6     SortedStacks[i].push( $e_{i,j}$ ); // in sorted stack
7     if  $k < SortedStacks[i].size()$  then
8       SortedStacks[i].pop(); // remove last edge
9  $E_B^{out} \leftarrow$  {}; // the set of retained edges
10 foreach  $e_{i,j} \in E_B$  do // for every edge
11   if  $e_{i,j} \in SortedStacks[i]$ 
12   OR  $e_{i,j} \in SortedStacks[j]$  then // retain if in
13      $E_B^{out} \leftarrow E_B^{out} \cup \{e_{i,j}\}$ ; // top-k for either node
14 return  $G_B^{out} = \{V_B, E_B^{out}, WS\}$ ;
```

---

corresponding weight threshold (Lines 2-4 in Algorithm 5). Then, it iterates once over all edges and retains those exceeding the weight thresholds of either of the incident nodes (Lines 6-9).

For both algorithms, the function *getNeighborhood* in Line 3 implements the Lines 4-16 of Algorithm 3. Note also that both algorithms use the same configuration as their original implementations:  $k = \lfloor \sum_{b \in B} |b|/|E| - 1 \rfloor$  for Redefined CNP and the average weight of each node neighborhood for Redefined WNP. Their time complexity is  $O(|V_B| \cdot |E_B|)$  in the worst-case of a complete blocking graph, and  $O(|E_B|)$  in the case of a sparse graph, which typically appears in practice. Their space complexity is dominated by the requirements of Entity Index and the number of retained comparisons, i.e.,  $O(BPE \cdot |V_B| + |B'|)$ , on average.

## 5.2 Reciprocal Node-centric Pruning

This approach treats the redundant comparisons retained by CNP and WNP as strong indications for profile pairs with high chances of matching. As explained above, these comparisons correspond to reciprocal links in the blocking graph. For example, the edges  $e_{1,3}^{\rightarrow}$  and  $e_{3,1}^{\leftarrow}$  in Figure 5(a) indicate that  $p_1$  is highly likely to match with  $p_3$  and vice versa, thus reinforcing the likelihood that the two profiles are duplicates. Based on this rationale, Reciprocal Pruning retains one comparison for every pair of profiles that are reciprocally connected in the directed pruned blocking graph of the original node-centric algorithms; profiles that are connected with a single edge, are not compared in the restructured block collection.

In our example, Reciprocal Pruning converts the directed pruned blocking graph in Figure 5(a) into the undirected pruned blocking graph in Figure 9(a). The corresponding restructured blocks in Figure 9(b) contain just 4 comparisons, one less than the blocks in Figure 8(b). Compared to the blocks in Figure 5(b), the overall efficiency is significantly enhanced at no cost in recall.

In general, Reciprocal Pruning yields restructured blocks with no redundant comparisons and less superfluous ones than both the original and the redefined node-centric pruning. In the worst case, all pairs of nodes are reciprocally linked and Reciprocal Pruning coincides with Redefined Node-centric Pruning. In all other cases, its precision is much higher, while its impact on recall depends on the strength of co-occurrence patterns in the blocking graph.

Reciprocal Pruning yields two new node-centric algorithms: Reciprocal CNP and Reciprocal WNP. Their functionality is almost

---

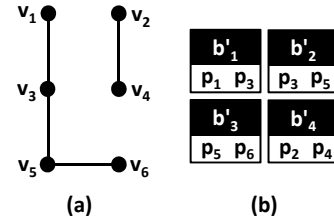
**Algorithm 5: Redefined Weighted Node Pruning.**

---

**Input:** (i)  $G_B^{in}$  the blocking graph, and  
(ii)  $wt$  the function defining the local weight thresholds.  
**Output:**  $G_B^{out}$  the pruned blocking graph

```
1 weights[]  $\leftarrow$  {}; // thresholds per node
2 foreach  $v_i \in V_B$  do // for every node
3    $G_{v_i} \leftarrow$  getNeighborhood( $v_i, G_B$ );
4    $weights[i] \leftarrow wt(G_{v_i})$ ; // get local threshold
5  $E_B^{out} \leftarrow$  {}; // the set of retained edges
6 foreach  $e_{i,j} \in E_B$  do // for every edge
7   if  $weights[i] \leq e_{i,j}.weight$ 
8   OR  $weights[j] \leq e_{i,j}.weight$  then // retain if it
9      $E_B^{out} \leftarrow E_B^{out} \cup \{e_{i,j}\}$ ; // exceeds either threshold
10 return  $G_B^{out} = \{V_B, E_B^{out}, WS\}$ ;
```

---



**Figure 9: (a) The pruned blocking graph produced by applying Reciprocal Pruning to the graph in Figure 5(a), and (b) the restructured blocks.**

identical to Redefined CNP and Redefined WNP, respectively. The only difference is that they use conjunctive conditions instead of disjunctive ones: the operator OR in Lines 11-12 and 7-8 of Algorithms 4 and 5, respectively, is replaced by the operator AND. Both algorithms use the same pruning criteria as redefined methods, while sharing the same average time and space complexities:  $O(|E_B|)$  and  $O(BPE \cdot |V_B| + |B'|)$ , respectively.

## 6. EXPERIMENTAL EVALUATION

We now examine the performance of our techniques through a series of experiments. We present their setup in Section 6.1 and in Section 6.2, we fine-tune Block Filtering, assessing its impact on blocking effectiveness. Its impact on time efficiency is measured in Section 6.3 together with that of Optimized Edge Weighting. Section 6.4 assess the effectiveness of our new pruning algorithms and compares them to state-of-the-art block processing methods.

### 6.1 Setup

We implemented our approaches in Java 8 and tested them on a desktop machine with Intel i7-3770 (3.40GHz) and 16GB RAM, running Ubuntu 15.04 (kernel version 3.19.0). We repeated all time measurements 10 times and report the average value so as to minimize the effect of external parameters.

**Datasets.** In our experiments, we use 3 real-world entity collections. They are established benchmarks [15, 21, 24] with significant variations in their size and characteristics. They pertain to Clean-Clean ER, but are used for Dirty ER, as well; we simply merge their clean entity collections into a single one that contains duplicates in itself. In total, we have 6 block collections that lay the ground for a thorough experimental evaluation of our techniques.

To present the technical characteristics of the entity collections, we use the following notation:  $|E|$  stands for the number of profiles they contain,  $|D(E)|$  for the number of existing duplicates,  $|N|$  for the number of distinct attribute names,  $|P|$  for the total number of name-value pairs,  $|\bar{p}|$  for the mean number of name-value pairs



	Original Block Collections						After Block Filtering					
	D <sub>1C</sub>	D <sub>2C</sub>	D <sub>3C</sub>	D <sub>1D</sub>	D <sub>2D</sub>	D <sub>3D</sub>	D <sub>1C</sub>	D <sub>2C</sub>	D <sub>3C</sub>	D <sub>1D</sub>	D <sub>2D</sub>	D <sub>3D</sub>
B	6,877	40,732	1,239,424	44,097	76,327	1,499,534	6,838	40,708	1,239,066	44,096	76,317	1,499,267
B	1.92·10 <sup>6</sup>	8.11·10 <sup>7</sup>	4.23·10 <sup>10</sup>	9.49·10 <sup>7</sup>	5.03·10 <sup>8</sup>	8.00·10 <sup>10</sup>	6.98·10 <sup>5</sup>	2.77·10 <sup>7</sup>	1.30·10 <sup>10</sup>	2.38·10 <sup>7</sup>	1.37·10 <sup>8</sup>	2.31·10 <sup>10</sup>
BPE	4.65	28.17	17.56	10.67	32.86	14.79	3.63	22.54	14.05	8.54	26.29	11.83
PC(B)	0.994	0.980	0.999	0.997	0.981	0.999	0.990	0.976	0.998	0.994	0.976	0.997
PQ(B)	1.19·10 <sup>-3</sup>	2.76·10 <sup>-4</sup>	2.11·10 <sup>-5</sup>	2.43·10 <sup>-5</sup>	4.46·10 <sup>-5</sup>	1.12·10 <sup>-5</sup>	3.28·10 <sup>-3</sup>	8.06·10 <sup>-4</sup>	6.86·10 <sup>-5</sup>	9.62·10 <sup>-5</sup>	1.62·10 <sup>-4</sup>	3.86·10 <sup>-5</sup>
RR	0.988	0.873	0.984	0.953	0.610	0.986	0.637	0.659	0.693	0.749	0.727	0.711
V <sub>B</sub>	61,399	50,720	3,331,647	63,855	50,765	3,333,356	60,464	50,720	3,331,641	63,855	50,765	3,333,355
E <sub>B</sub>	1.83·10 <sup>6</sup>	6.75·10 <sup>7</sup>	3.58·10 <sup>10</sup>	7.98·10 <sup>7</sup>	2.70·10 <sup>8</sup>	6.65·10 <sup>10</sup>	6.69·10 <sup>5</sup>	2.52·10 <sup>7</sup>	1.14·10 <sup>10</sup>	2.11·10 <sup>7</sup>	9.76·10 <sup>7</sup>	2.00·10 <sup>10</sup>
OTime(B)	2.1 sec	5.6 sec	4 min	2.2 sec	5.7 sec	5 min	2.3 sec	6.4 sec	5 min	2.5 sec	6.5 sec	6 min
RTime(B)	19 sec	65 min	~350 hrs	13 min	574 min	~660 hrs	9 sec	24 min	~110 hrs	3 min	174 min	~190 hrs

Table 1: Technical characteristics of (a) the original block collections, and (b) the ones restructured by Block Filtering with  $r=0.80$ .

	E	D(E)	N	P	p	E	RT(E)
D <sub>1C</sub>	2,516	2,308	4	1.01·10 <sup>4</sup>	4.0	1.54·10 <sup>8</sup>	26 min
	61,353	2,308	4	1.98·10 <sup>5</sup>	3.2	1.54·10 <sup>8</sup>	26 min
D <sub>2C</sub>	27,615	22,863	4	1.55·10 <sup>5</sup>	5.6	6.40·10 <sup>8</sup>	533 min
	23,182	22,863	7	8.16·10 <sup>5</sup>	35.2	6.40·10 <sup>8</sup>	533 min
D <sub>3C</sub>	1,190,733	892,579	30,688	1.69·10 <sup>7</sup>	14.2	2.58·10 <sup>12</sup>	~21,000 hrs
	2,164,040	892,579	52,489	3.50·10 <sup>7</sup>	16.2	2.58·10 <sup>12</sup>	~21,000 hrs

(a) Entity Collections for Clean-Clean ER

	E	D(E)	N	P	p	E	RT(E)
D <sub>1D</sub>	63,869	2,308	4	2.08·10 <sup>5</sup>	3.3	2.04·10 <sup>9</sup>	272 min
D <sub>2D</sub>	50,797	22,863	10	9.71·10 <sup>5</sup>	19.1	1.29·10 <sup>9</sup>	1,505 min
D <sub>3D</sub>	3,354,773	892,579	58,590	5.19·10 <sup>7</sup>	15.5	5.63·10 <sup>12</sup>	~47,000 hrs

(b) Entity Collections for Dirty ER

Table 2: Technical characteristics of the entity collections. For  $D_{3C}$  and  $D_{3D}$ ,  $RT(E)$  was estimated from the average time required for comparing two of its entity profiles: 0.03 msec.

per profile,  $||E||$  for the number of comparisons executed by the brute-force approach and  $RT(E)$  for its resolution time; in line with  $RTime(B)$ ,  $RT(E)$  is computed using the Jaccard similarity of all tokens in the values of two profiles as the entity matching method.

Tables 2(a) and (b) present the technical characteristics of the real entity collections for Clean-Clean and Dirty ER, respectively.  $D_{1C}$  contains bibliographic data from DBLP ([www.dblp.org](http://www.dblp.org)) and Google Scholar (<http://scholar.google.gr>) that were matched manually [15, 24].  $D_{2C}$  matches movies from IMDB ([www.imdb.com](http://www.imdb.com)) and DBPedia (<http://dbpedia.org>) based on their ImdbId [22, 23].  $D_{3C}$  involves profiles from two snapshots of English Wikipedia (<http://en.wikipedia.org>) Infoboxes, which were automatically matched based on their URL [22, 23]. For Dirty ER, the datasets  $D_{x,D}$  with  $x \in [1, 6]$  were derived by merging the profiles of the individually clean collections that make up  $D_{x,C}$ , as explained above.

**Measures.** To assess the *effectiveness* of a restructured block collection  $B'$ , we use four established measures [4, 5, 22]: (i) its cardinality  $||B'||$ , i.e., total number of comparisons, (ii) its recall  $PC(B')$ , (iii) its precision  $PQ(B')$ , and (iv) its *Reduction Ratio* ( $RR$ ), which expresses the relative decrease in its cardinality in comparison with the original block collection,  $B$ :  $RR(B, B') = 1 - ||B'||/||B||$ . The last three measures take values in the interval  $[0, 1]$ , with higher values, indicating better effectiveness; the opposite is true for  $||B'||$ , as effectiveness is inversely proportional to its value (cf. Section 3).

To assess the *time efficiency* of a restructured block collection  $B'$ , we use the two measures that were defined in Section 3: (i) its *Overhead Time*  $OTime(B')$ , which measures the time required by Meta-blocking to derive it from the input block collection, and (ii) its *Resolution Time*  $RTime(B')$ , which adds to  $OTime(B')$  the time taken to apply an entity matching method to all comparisons in  $B'$ .

## 6.2 Block Collections

**Original Blocks.** From all datasets, we extracted a redundancy-positive block collection by applying Token Blocking [21]. We also applied Block Purging [21] in order to discard those blocks that

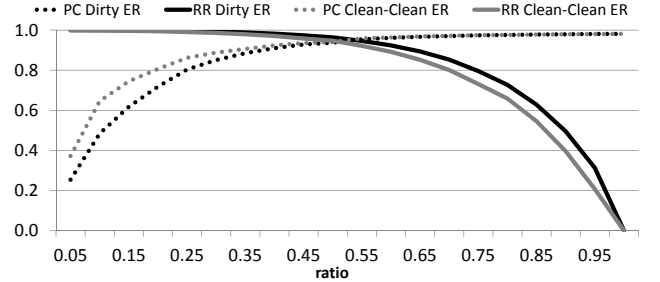


Figure 10: The effect of Block Filtering's ratio  $r$  on the blocks of  $D_{5C}$  and  $D_{5D}$  with respect to  $RR$  and  $PC$ .

contained more than half of the input entity profiles. The technical characteristics of the resulting block collections are presented in Table 1(a). Remember that  $|V_B|$  and  $|E_B|$  stand for the order and the size of the corresponding blocking graph, respectively.  $RR$  has been computed with respect to  $||E||$  in Table 2, i.e.,  $RR=1-||B'||/||E||$ .

We observe that all block collections exhibit nearly perfect recall, as their  $PC$  consistently exceeds 0.98. They also convey significant gains in efficiency, executing an order of magnitude less comparisons than the brute-force approach ( $RR>0.9$ ) in most cases. They reduce the resolution time to a similar extent, due to their very low  $OTime$ . Still, their precision is significantly lower than 0.01 in all cases. This means that on average, more than 100 comparisons have to be executed in order to identify a new pair of duplicates. The corresponding blocking graphs vary significantly in size, ranging from tens of thousands edges to tens of billions, whereas their order ranges from few thousands nodes to few millions.

Note that we experimented with additional redundancy-positive blocking methods, such as Q-grams Blocking. All of them involved a schema-agnostic functionality that tackles effectively the schema heterogeneity. They all produced blocks with similar characteristics as Token Blocking and are omitted for brevity. In general, the outcomes of our experiments are independent of the schema-agnostic, redundancy-positive methods that yield the input blocks.

**Block Filtering.** Before using Block Filtering, we have to fine-tune its *filtering ratio*  $r$ , which determines the portion of the most important blocks that are retained for each profile. To examine its effect, we measured the performance of the restructured blocks using all values of  $r$  in  $[0.05, 1]$  with a step of 0.05. We consider two evaluation measures: recall ( $PC$ ) and reduction ratio ( $RR$ ).

Figure 10 presents the evolution of both measures over the original blocks of  $D_{5C}$  and  $D_{5D}$  – the other datasets exhibit similar patterns and are omitted for brevity. We observe that there is a clear trade-off between  $RR$  and  $PC$ : the smaller the value of  $r$ , the less blocks are retained for each profile and the lower is the total cardinality of the restructured blocks  $||B'||$ , thus increasing  $RR$ ; this reduces the number of detected duplicates, thus decreasing  $PC$ . The opposite is true for large values of  $r$ . Most importantly, Block Filtering exhibits a robust performance with respect to  $r$ , with small

	Original Block Collections						After Block Filtering					
	D <sub>1C</sub>	D <sub>2C</sub>	D <sub>3C</sub>	D <sub>1D</sub>	D <sub>2D</sub>	D <sub>3D</sub>	D <sub>1C</sub>	D <sub>2C</sub>	D <sub>3C</sub>	D <sub>1D</sub>	D <sub>2D</sub>	D <sub>3D</sub>
$\ B'\ $	1.43·10 <sup>5</sup>	7.14·10 <sup>5</sup>	2.63·10 <sup>7</sup>	3.41·10 <sup>5</sup>	8.34·10 <sup>5</sup>	2.47·10 <sup>7</sup>	1.10·10 <sup>5</sup>	5.72·10 <sup>5</sup>	2.11·10 <sup>7</sup>	2.73·10 <sup>5</sup>	6.67·10 <sup>5</sup>	1.97·10 <sup>7</sup>
$PC(B')$	0.966	0.860	0.724	0.765	0.522	0.535	0.948	0.871	0.748	0.823	0.641	0.566
$PQ(B')$	0.016	0.028	0.025	0.005	0.014	0.019	0.020	0.035	0.032	0.007	0.022	0.026
$OT(B')$	395 ms	49 sec	9.4 hrs	22 sec	16 min	17.2 hrs	142 ms	12 sec	2.0 hrs	5 sec	4 min	3.7 hrs
<b>(a) Cardinality Edge Pruning (CEP)</b>												
$\ B'\ $	2.34·10 <sup>5</sup>	1.41·10 <sup>6</sup>	4.95·10 <sup>7</sup>	6.38·10 <sup>5</sup>	1.62·10 <sup>6</sup>	4.63·10 <sup>7</sup>	1.69·10 <sup>5</sup>	1.10·10 <sup>6</sup>	3.95·10 <sup>7</sup>	5.10·10 <sup>5</sup>	1.31·10 <sup>6</sup>	3.63·10 <sup>7</sup>
$PC(B')$	0.975	0.946	0.973	0.949	0.880	0.951	0.955	0.942	0.965	0.936	0.888	0.942
$PQ(B')$	0.010	0.015	0.018	0.003	0.012	0.018	0.013	0.020	0.022	0.004	0.016	0.023
$OT(B')$	899 ms	83 sec	18.6 hrs	58 sec	17 min	35.2 hrs	310 ms	25 sec	4.7 hrs	13 sec	6 min	8.6 hrs
<b>(b) Cardinality Node Pruning (CNP)</b>												
$\ B'\ $	4.32·10 <sup>5</sup>	1.48·10 <sup>7</sup>	6.64·10 <sup>9</sup>	1.38·10 <sup>7</sup>	7.81·10 <sup>7</sup>	1.19·10 <sup>10</sup>	1.63·10 <sup>5</sup>	5.50·10 <sup>6</sup>	2.11·10 <sup>9</sup>	3.99·10 <sup>6</sup>	2.26·10 <sup>7</sup>	4.06·10 <sup>9</sup>
$PC(B')$	0.977	0.963	0.977	0.987	0.970	0.973	0.953	0.947	0.967	0.964	0.944	0.965
$PQ(B')$	1.08·10 <sup>-2</sup>	2.62·10 <sup>-3</sup>	6.66·10 <sup>-4</sup>	2.99·10 <sup>-4</sup>	7.30·10 <sup>-4</sup>	3.54·10 <sup>-4</sup>	2.92·10 <sup>-2</sup>	6.54·10 <sup>-3</sup>	1.49·10 <sup>-3</sup>	9.51·10 <sup>-4</sup>	1.62·10 <sup>-3</sup>	7.27·10 <sup>-4</sup>
$OT(B')$	588 ms	92 sec	17.1 hrs	40 sec	26 min	31.7 hrs	193 ms	23 sec	3.7 hrs	8 sec	7 min	6.9 hrs
<b>(c) Weighted Edge Pruning (WEP)</b>												
$\ B'\ $	1.11·10 <sup>6</sup>	2.81·10 <sup>7</sup>	1.60·10 <sup>10</sup>	3.41·10 <sup>7</sup>	1.54·10 <sup>8</sup>	3.00·10 <sup>10</sup>	4.64·10 <sup>5</sup>	1.05·10 <sup>7</sup>	5.38·10 <sup>9</sup>	9.84·10 <sup>6</sup>	5.29·10 <sup>7</sup>	9.49·10 <sup>9</sup>
$PC(B')$	0.988	0.972	0.997	0.993	0.971	0.995	0.979	0.964	0.997	0.979	0.959	0.992
$PQ(B')$	2.32·10 <sup>-3</sup>	1.14·10 <sup>-3</sup>	1.44·10 <sup>-4</sup>	1.13·10 <sup>-4</sup>	3.11·10 <sup>-4</sup>	7.63·10 <sup>-5</sup>	5.13·10 <sup>-3</sup>	3.01·10 <sup>-3</sup>	3.56·10 <sup>-4</sup>	3.42·10 <sup>-4</sup>	6.79·10 <sup>-4</sup>	1.94·10 <sup>-4</sup>
$OT(B')$	862 ms	85 sec	18.6 hrs	55 sec	16 min	35.0 hrs	303 ms	24 sec	4.7 hrs	13 sec	5 min	9.0 hrs
<b>(d) Weighted Node Pruning (WNP)</b>												

**Table 3: Performance of the existing pruning schemes, averaged across all weighting schemes, before and after Block Filtering.**

variations in its value leading to small differences in  $RR$  and  $PC$ .

To use Block Filtering as a pre-processing method, we should set its ratio to a value that increases precision at a low cost in recall. We quantify this constraint by requiring that  $r$  decreases  $PC$  by less than 0.5%, while maximizing  $RR$  and, thus,  $PQ$ . The ratio that satisfies this constraint across all datasets is  $r=0.80$ . Table 1(b) presents the characteristics of the restructured block collections corresponding to this configuration. Note that  $RR$  has been computed with respect to the cardinality of the original blocks.

We observe that the number of blocks is almost the same as in Table 1(a). Yet, their total cardinality is reduced by 64% to 75%, while recall is reduced by less than 0.5% in most cases. As a result,  $PQ$  rises from 2.7 to 4.0 times, but still remains far below 0.01. There is an insignificant increase in  $OTime$  and, thus,  $RTIME$  decreases to the same extent as  $RR$ . The same applies to the order of the blocking graph  $|E_B|$ , while its size  $|V_B|$  remains almost the same. Finally,  $BPE$  is reduced by  $(1-r) \cdot 100\% = 20\%$  across all datasets.

### 6.3 Time Efficiency Improvements

Table 3 presents the performance of the four existing pruning schemes, averaged across all weighting schemes. Their original performance appears in the left part, while the right part presents their performance after Block Filtering.

**Original Performance.** We observe that CEP reduces the executed comparisons by 1 to 3 orders of magnitude for the smallest and the largest datasets, respectively. It increases precision ( $PQ$ ) to a similar extent at the cost of much lower recall in most of the cases. This applies particularly to Dirty ER, where  $PC$  drops consistently below 0.80, the minimum acceptable recall of efficiency-intensive ER applications. The reason is that Dirty ER is more difficult than Clean-Clean ER, involving much larger blocking graphs with many more noisy edges between non-matching entity profiles.

CNP is more robust to recall than CEP, as its  $PC$  lies well over 0.80 across all datasets. Its robustness stems from its node-centric functionality, which retains the best edges per node, instead of the globally best ones. This comes, though, at the cost of a much higher computational cost: its overhead time is larger than that of CEP by 44%, on average. Further, CNP retains almost twice as many comparisons and yields a slightly lower precision than CEP.

For WEP, we observe that its recall consistently exceeds 0.95, the minimum acceptable  $PC$  of effectiveness-intensive ER applications. At the same time, it executes almost an order of magni-

tude less comparisons than the original blocks in Table 1(a) and enhances  $PQ$  to a similar extent. These patterns apply to all datasets.

Finally, WNP saves 60% of the brute-force comparisons, on average, retaining twice as many comparisons as WEP. Its recall remains well over 0.95 across all datasets, exceeding that of WEP to a minor extent. As a result, its precision is half that of WEP, while its overhead time is slightly higher.

In summary, these experiments verify previous findings about the relative performance of pruning schemes [22]: the cardinality-based ones excel in precision, being more suitable for efficiency-intensive ER applications, while the weight-based schemes excel in recall, being more suitable for effectiveness-intensive applications. In both cases, the node-centric algorithms trade higher recall for lower precision and higher overhead.

**Block Filtering.** Examining the effect of Block Filtering in the right part of Table 3, we observe two patterns:

(i) Its impact on overhead time depends on the dataset at hand, rather than the pruning scheme applied on top of it. In fact,  $OTIME$  is reduced by 65% ( $D_{1C}$ ) to 78% ( $D_{1D}$ ), on average, across all pruning schemes. This is close to  $RR$  and the reduction in the order of the blocking graph  $|E_B|$ , but higher than them, because Block Filtering additionally reduces  $BPE$  by 20%.

(ii) Its impact on blocking effectiveness depends on the type of the pruning criterion used by Meta-blocking. For cardinality thresholds, Block Filtering conveys a moderate decrease in the retained comparisons, with  $\|B'\|$  dropping by 20%, on average. The reason is that both CEP and CNP use thresholds that are proportional to  $BPE$ , which is reduced by  $(1-r) \cdot 100\%$ . At the same time, their recall is either reduced to a minor extent (<2%), or increases by up to 10%. The latter case appears in half the datasets and indicates that Block Filtering cleans the blocking graph from noisy edges, enabling CEP and CNP to identify more duplicates with fewer retained comparisons.

For weight thresholds, Block Filtering reduces the number of retained comparisons to a large extent:  $\|B'\|$  drops by 62% to 71% for both WEP and WNP. The reason is that their pruning criteria depends directly on the size and the structure of the blocking graph. At the same time, their recall gets lower by less than 3% in all cases, an affordable reduction that is caused by two factors: (i) Block Filtering discards some matching edges itself, and (ii) Block Filtering reduces the extent of co-occurrence for some matching entity profiles, thus lowering the weight of their edges.

	Redefined Node-centric Pruning						Reciprocal Node-centric Pruning					
	D <sub>1C</sub>	D <sub>2C</sub>	D <sub>3C</sub>	D <sub>1D</sub>	D <sub>2D</sub>	D <sub>3D</sub>	D <sub>1C</sub>	D <sub>2C</sub>	D <sub>3C</sub>	D <sub>1D</sub>	D <sub>2D</sub>	D <sub>3D</sub>
$\ B'\ $	$1.63 \cdot 10^5$	$8.52 \cdot 10^5$	$3.36 \cdot 10^7$	$3.91 \cdot 10^5$	$1.02 \cdot 10^6$	$2.92 \cdot 10^7$	$6.54 \cdot 10^3$	$2.50 \cdot 10^5$	$5.88 \cdot 10^6$	$1.19 \cdot 10^5$	$2.86 \cdot 10^5$	$7.12 \cdot 10^6$
$PC(B')$	0.955	0.942	0.965	0.936	0.888	0.942	0.880	0.886	0.912	0.847	0.650	0.868
$PQ(B')$	0.014	0.025	0.026	0.006	0.020	0.029	0.312	0.084	0.142	0.017	0.057	0.111
$OT(B')$	339 ms	5 sec	2.1 hrs	5 sec	23 sec	4.9 hrs	329 ms	5 sec	2.1 hrs	5 sec	23 sec	4.9 hrs
(a) Redefined Cardinality Node Pruning						(b) Reciprocal Cardinality Node Pruning						
$\ B'\ $	$3.72 \cdot 10^5$	$7.52 \cdot 10^6$	$3.96 \cdot 10^9$	$7.23 \cdot 10^6$	$3.26 \cdot 10^7$	$6.96 \cdot 10^9$	$9.30 \cdot 10^4$	$2.96 \cdot 10^6$	$1.41 \cdot 10^9$	$2.61 \cdot 10^6$	$2.02 \cdot 10^7$	$2.54 \cdot 10^9$
$PC(B')$	0.979	0.964	0.994	0.979	0.959	0.992	0.953	0.949	0.977	0.964	0.924	0.971
$PQ(B')$	$6.53 \cdot 10^{-3}$	$4.79 \cdot 10^{-3}$	$5.01 \cdot 10^{-4}$	$4.91 \cdot 10^{-4}$	$1.09 \cdot 10^{-3}$	$2.74 \cdot 10^{-4}$	$3.16 \cdot 10^{-2}$	$8.78 \cdot 10^{-3}$	$1.47 \cdot 10^{-3}$	$1.14 \cdot 10^{-3}$	$1.78 \cdot 10^{-3}$	$7.88 \cdot 10^{-4}$
$OT(B')$	582 ms	10 sec	5.6 hrs	9 sec	45 sec	10.7 hrs	576 ms	10 sec	5.4 hrs	9 sec	45 sec	10.5 hrs
(c) Redefined Weighted Node Pruning						(d) Reciprocal Weighted Node Pruning						

**Table 4: Performance of the new pruning schemes on top of Block Filtering over all datasets, averaged across all weighting schemes.**

	D <sub>1C</sub>	D <sub>2C</sub>	D <sub>3C</sub>	D <sub>1D</sub>	D <sub>2D</sub>	D <sub>3D</sub>
CEP	117 ms	4 sec	1.5 hrs	3 sec	14 sec	1.8 hrs
CNP	246 ms	6 sec	2.2 hrs	6 sec	25 sec	4.3 hrs
WEP	150 ms	6 sec	2.7 hrs	5 sec	25 sec	3.8 hrs
WNP	257 ms	8 sec	4.4 hrs	7 sec	33 sec	7.5 hrs

**Table 5:  $OTime$  of Optimized Edge Weighting for each pruning scheme, averaged across all weighting schemes, over the datasets in Table 1(b), i.e., after Block Filtering.**

We can conclude that Block Filtering enhances the scalability of Meta-blocking to a significant extent, accelerating the processing of all pruning schemes almost by 4 times, on average. It also achieves much higher precision, while its impact on recall is either negligible or beneficial. Thus, it constitutes an indispensable pre-processing step for Meta-blocking. For this reason, the following experiments are carried out on top of Block Filtering.

**Optimized Edge Weighting.** Table 5 presents the overhead time of the four pruning schemes when combined with Block Filtering and Optimized Edge Weighting (cf. Algorithm 3). Comparing it with  $OTime$  in the right part of Table 3, we observe significant enhancements in efficiency. Again, they depend on the dataset at hand, rather than the pruning scheme. In fact, the higher the  $BPE$  of a dataset after Block Filtering, the larger is the reduction in overhead time:  $OTime$  is reduced by 19% for  $D_{1C}$ , where  $BPE$  takes the lowest value across all datasets (3.63), and by 92% for  $D_{2D}$ , where  $BPE$  takes the highest one (26.29). The reason is that Optimized Edge Weighting minimizes the computational cost of the process that is applied to every comparison by Original Edge Weighting (cf. Algorithm 2) from  $O(2 \cdot BPE)$  to  $O(1)$ .

Also interesting is the comparison between  $OTime$  in Table 5 and  $OTime$  in the left part of Table 3, i.e., before applying Block Filtering to the input blocks. On average, across all datasets and pruning schemes,  $OTime$  is reduced by 87%, which is almost an order of magnitude. Again, the lowest (72%) and the highest (98%) average reductions correspond to  $D_{1C}$  and  $D_{2D}$ , respectively, which exhibit the minimum and the maximum  $BPE$  before Block Filtering. We can conclude, therefore, that the two efficiency optimizations are complementary and indispensable for scalable Meta-blocking.

## 6.4 Precision Improvements

To estimate the performance of Redefined and Reciprocal Node-centric Pruning, we applied the four pruning schemes they yield to the datasets in Table 1(b). Their performance appears in Table 4.

**Cardinality-based Pruning.** Starting with Redefined CNP, we observe that it maintains the recall of the original CNP, while conveying a moderate increase in efficiency. On average, across all weighting schemes and datasets, it retains 18% less comparisons, increasing  $PQ$  by 1.2 times. With respect to  $OTime$ , there is no clear winner, as the implementation of both algorithms is highly similar, relying on Optimized Edge Weighting. Hence, the original CNP is just 2% faster, on average, because it does not store all

retained edges per node in memory.

For Reciprocal CNP,  $OTime$  is practically identical with that of Redefined CNP, as they only differ in a single operator. Yet, Reciprocal CNP consistently achieves the highest precision among all versions of CNP at the cost of the lowest recall. On average, it retains 82% and 78% less comparisons than CNP and Redefined CNP, respectively, while increasing precision by 7.9 and 6.9 times, respectively; it also decreases recall by 11%, but exceeds the minimum acceptable  $PC$  for efficiency-intensive applications (0.80) to a significant extent in most cases. The only exception is  $D_{2D}$ , where  $PC$  drops below 0.80 for all weighting schemes. Given that the corresponding Clean-Clean ER dataset ( $D_{2C}$ ) exhibits much higher  $PC$ , the poor recall for  $D_{2D}$  is attributed to highly similar (i.e., noisy) entity profiles in one of the duplicate-free entity collections.

It is worth comparing at this point the new pruning schemes with their edge-centric counterpart: CEP coupled with Block Filtering (see right part of Table 3). We observe three patterns: (i) On average, CEP keeps 33% less comparisons than Redefined CNP, but lowers recall by 18%. Its recall is actually so low in most datasets that Redefined CNP achieves higher precision in all cases except  $D_{1C}$  and  $D_{2C}$ . (ii) Reciprocal CNP typically outperforms CEP in all respects of effectiveness. On average, it executes 67% less comparisons, while increasing recall by 8%.  $D_{1C}$  is the exception that proves this rule: Reciprocal CNP saves more than an order of magnitude more comparisons, but CEP achieves slightly higher recall. (iii) The overhead time of CEP is lower by 44%, on average, than both algorithms, because it iterates once instead of twice over all edges in the blocking graph.

On the whole, we can conclude that Reciprocal CNP offers the best choice for efficiency-intensive applications, as it consistently achieves the highest precision among all cardinality-based pruning schemes with  $PC \geq 0.8$ . However, in datasets with high levels of noise, where many entity profiles share the same information, Redefined CNP should be preferred; it is more robust to recall than CEP and Reciprocal CNP, while saving 30% more comparisons than CNP. In any case, Block Filtering is indispensable.

**Weight-based Pruning.** As expected, Redefined WNP maintains the same recall as the original implementation of WNP, while conveying major enhancements in efficiency. On average, across all weighting schemes and datasets, it reduces the retained comparisons by 28% and increases precision by 1.5 times. It is also faster than WNP by 7%, but in practice, the method with the lowest  $OTime$  varies across the datasets.

Reciprocal WNP performs a deeper pruning that consistently trades a lower number of executed comparisons for a lower recall. On average, it reduces the total cardinality of WNP by 72% and the recall by 2%. Its mean  $PC$  drops slightly below 0.95 in  $D_{2C}$ , but this does not apply to all weighting schemes; two of them exceed the minimum acceptable recall of effectiveness-intensive applications even for this dataset. As a result, Reciprocal WNP enhances the precision of WNP by 3.9 times. Its overhead is slightly lower than

	R <sub>1D</sub>	R <sub>2D</sub>	R <sub>3D</sub>	R <sub>1C</sub>	R <sub>2C</sub>	R <sub>3C</sub>
$\ B'\ $	4.22·10 <sup>4</sup>	7.47·10 <sup>5</sup>	4.86·10 <sup>8</sup>	4.53·10 <sup>5</sup>	2.10·10 <sup>6</sup>	2.12·10 <sup>9</sup>
$PC(B')$	0.870	0.862	0.963	0.862	0.804	0.965
$PQ(B')$	0.048	0.026	0.002	0.004	0.009	4.07·10 <sup>-4</sup>
$OTime(B')$	24 msec	86 msec	1 min	62 msec	150 msec	2 min
<b>(a) Efficiency-intensive Graph-free Meta-blocking (<math>r=0.25</math>)</b>						
$\ B'\ $	2.21·10 <sup>5</sup>	6.16·10 <sup>6</sup>	8.52·10 <sup>9</sup>	4.73·10 <sup>6</sup>	2.36·10 <sup>7</sup>	3.55·10 <sup>10</sup>
$PC(B')$	0.973	0.959	0.998	0.980	0.954	0.965
$PQ(B')$	1.02·10 <sup>-2</sup>	3.56·10 <sup>-3</sup>	1.05·10 <sup>-4</sup>	4.78·10 <sup>-4</sup>	9.24·10 <sup>-4</sup>	2.51·10 <sup>-5</sup>
$OTime(B')$	30 msec	221 msec	6 min	146 msec	650 msec	25 min
<b>(b) Effectiveness-intensive Graph-free Meta-blocking (<math>r=0.55</math>)</b>						
$\ B'\ $	1.76·10 <sup>6</sup>	1.32·10 <sup>7</sup>	2.34·10 <sup>10</sup>	9.07·10 <sup>7</sup>	4.08·10 <sup>8</sup>	4.81·10 <sup>10</sup>
$PC(B')$	0.994	0.980	0.999	0.997	0.981	0.999
$PQ(B')$	1.31·10 <sup>-3</sup>	1.70·10 <sup>-3</sup>	3.81·10 <sup>-5</sup>	2.54·10 <sup>-5</sup>	5.49·10 <sup>-5</sup>	1.85·10 <sup>-5</sup>
$OTime(B')$	76 msec	2 sec	1.9 hrs	5 sec	1 min	11.1 hrs
<b>(c) Iterative Blocking</b>						

**Table 6: Performance of the baseline methods over all datasets.**

Redefined WNP, because it retains less comparisons in memory. Thus, it is faster than WNP by 9%, on average.

Compared to WEP, Redefined WNP exhibits lower precision, but is more robust to recall, maintaining  $PC$  well above 0.95 under all circumstances. In contrast, WEP violates this constraint in all datasets for at least one weighting scheme. Reciprocal WNP scores higher precision than WEP, while being more robust to recall, as well. For this reason, it is the optimal choice for effectiveness-intensive applications. Again, for datasets with high levels of noise, Redefined WNP should be preferred.

**Baseline Methods.** We now compare our techniques with the state-of-the-art block processing method Iterative Blocking [27] and with Graph-free Meta-blocking (see end of Section 4.1). The functionality of the former was optimized by ordering the blocks from the smallest to the largest cardinality; its functionality was further optimized for Clean-Clean ER by assuming the ideal case where two matching entities are not compared to other co-occurring entities after their detection. The configuration of Graph-free Meta-blocking was also optimized by setting  $r$  to the smallest values in [0.05, 1.0] with a step of 0.05 that ensures a recall higher than 0.80 and 0.95 across all real datasets; this resulted in  $r=0.25$  and  $r=0.55$  for efficiency- and effectiveness-intensive applications, respectively. Table 6 presents the performance of the two methods.

Juxtaposing Efficiency-intensive Graph-free Meta-blocking and Reciprocal CNP, we notice a clear trade-off between precision and recall. The latter approach emphasizes  $PQ$ , retaining 85% less comparisons at the cost of 5% lower  $PC$ , on average. The only advantage of Graph-free Meta-blocking is its minimal overhead: its lightweight functionality is able to process datasets with millions of entities within few minutes even on commodity hardware. Similar patterns arise when comparing Reciprocal WNP with Effectiveness-intensive Graph-free Meta-blocking: the former executes 58% less comparisons at the cost of a 2% decrease in recall, thus achieving higher precision. This behaviour can be explained by the fine-grained functionality of Reciprocal Pruning: unlike Graph-free Meta-blocking, which operates on the level of individual entities, it considers pairwise comparisons, thus being more accurate in their pruning.

Compared to Iterative Blocking, Reciprocal WNP retains less comparisons by a whole order of magnitude at the cost of slightly lower recall. Thus, it achieves significantly higher precision. The overhead of Iterative Blocking is significantly lower in most cases, but it does not scale well to large datasets, even though Iterative Blocking does not involve a blocking graph. The reason is that it goes through the input block collection several times, updating the representation of the duplicate entities in all blocks that contain them, whenever a new match is detected.

## 7. CONCLUSIONS

In this paper, we introduced two techniques for boosting the efficiency of Meta-blocking along with two techniques for enhancing its effectiveness. Our thorough experimental analysis verified that in combination, our methods go well beyond the existing Meta-blocking techniques in all respects and simplify its configuration, depending on the data and the application at hand. For efficiency-intensive ER applications, Reciprocal CNP processes a large heterogeneous dataset with 3 millions entities and 80 billion comparisons within 2 hours even on commodity hardware; it also manages to retain recall and precision above 0.8 and 0.1, respectively. For effectiveness-intensive ER applications, Reciprocal WNP processes the same voluminous dataset within 5 hours on commodity hardware, while retaining recall above 0.95 and precision close to 0.01. In both cases, Block Filtering and Optimized Edge Weighting are indispensable. In the future, we plan to adapt our techniques for Enhanced Meta-blocking to Incremental Entity Resolution.

## References

- [1] A. N. Aizawa and K. Oyama. A fast linkage detection scheme for multi-source information integration. In *WIRI*, pages 30–39, 2005.
- [2] Y. Altowim, D. V. Kalashnikov, and S. Mehrotra. Progressive approach to relational entity resolution. *PVLDB*, 7(11):999–1010, 2014.
- [3] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.
- [4] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Trans. Knowl. Data Eng.*, 24(9):1537–1555, 2012.
- [5] V. Christophides, V. Efthymiou, and K. Stefanidis. *Entity Resolution in the Web of Data*. Morgan & Claypool Publishers, 2015.
- [6] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. Large-scale linked data integration using probabilistic reasoning and crowdsourcing. *VLDB J.*, 22(5):665–687, 2013.
- [7] X. L. Dong and D. Srivastava. *Big Data Integration*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2015.
- [8] A. Elmagarmid, P. Ipeirotis, and V. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.
- [9] I. Fellegi and A. Sunter. A theory for record linkage. *Journal of American Statistical Association*, pages 1183–1210, 1969.
- [10] L. Getoor and A. Machanavajjhala. Entity resolution: Theory, practice & open challenges. *PVLDB*, 5(12):2018–2019, 2012.
- [11] L. Getoor and A. Machanavajjhala. Entity resolution for big data. In *KDD*, 2013.
- [12] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.
- [13] M. Hernández and S. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, pages 127–138, 1995.
- [14] Y. Jiang, G. Li, J. Feng, and W. Li. Similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.
- [15] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1):484–493, 2010.
- [16] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.
- [17] J. Madhavan, S. Cohen, X. L. Dong, A. Y. Halevy, S. R. Jeffery, D. Ko, and C. Yu. Web-scale data integration: You can afford to pay as you go. In *CIDR*, pages 342–350, 2007.
- [18] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.
- [19] A. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD*, pages 169–178, 2000.
- [20] G. Papadakis, E. Ioannou, C. Niederée, T. Palpanas, and W. Nejdl. Beyond 100 million entities: large-scale blocking-based resolution for heterogeneous data. In *WSDM*, pages 53–62, 2012.
- [21] G. Papadakis, E. Ioannou, T. Palpanas, C. Niederée, and W. Nejdl. A blocking framework for entity resolution in highly heterogeneous information spaces. *IEEE Trans. Knowl. Data Eng.*, 25(12):2665–2682, 2013.
- [22] G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. Meta-blocking: Taking entity resolution to the next level. *IEEE Trans. Knowl. Data Eng.*, 2014.
- [23] G. Papadakis, G. Papastefanatos, and G. Koutrika. Supervised meta-blocking. *PVLDB*, 7(14):1929–1940, 2014.
- [24] A. Thor and E. Rahm. Moma - a mapping-based object matching system. In *CIDR*, pages 247–258, 2007.
- [25] M. J. Welch, A. Sane, and C. Drome. Fast and accurate incremental entity resolution relative to an entity knowledge base. In *CIKM*, pages 2667–2670, 2012.
- [26] S. E. Whang, D. Marmaros, and H. Garcia-Molina. Pay-as-you-go entity resolution. *IEEE Trans. Knowl. Data Eng.*, 25(5):1111–1124, 2013.
- [27] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD*, pages 219–232, 2009.

# Practical Query Answering in Data Exchange Under Inconsistency-Tolerant Semantics

Balder ten Cate  
UC Santa Cruz  
Google  
btencate@ucsc.edu

Richard L. Halpert  
UC Santa Cruz  
rhalpert@ucsc.edu

Phokion G. Kolaitis  
UC Santa Cruz  
IBM Research - Almaden  
kolaitis@ucsc.edu

## ABSTRACT

Exchange-repair semantics (or, XR-Certain semantics) is a recently proposed inconsistency-tolerant semantics in the context of data exchange. This semantics makes it possible to provide meaningful answers to target queries in cases in which a given source instance cannot be transformed into a target instance satisfying the constraints of the data exchange specification. It is known that computing the answers to conjunctive queries under XR-Certain semantics is a coNP-complete problem in data complexity. Moreover, this problem can be reduced in a natural way to cautious reasoning over stable models of a disjunctive logic program.

Here, we explore how to effectively perform XR-Certain query answering for practical data exchange settings by leveraging modern sophisticated solvers for disjunctive logic programming. We first present a new reduction, accompanied by an optimized implementation, of XR-Certain query answering to disjunctive logic programming. We then evaluate this approach on a benchmark that we introduce here and which is modeled after a practical data exchange problem in computational genomics. Specifically, we present a benchmark scenario that mimicks a portion of the UCSC Genome Browser data import process. Our initial results, based on real genomic data, suggest that the solvers we apply fail to take advantage of some critical exploitable structural properties of the specific instances at hand. We then develop an improved encoding to take advantage of these properties using techniques inspired by the notion of a repair envelope. The improved implementation utilizing these techniques computes query answers ten to one thousand times faster for large instances, and exhibits promising scalability with respect to the size of instances and the rate of target constraint violations.

## Categories and Subject Descriptors

H.2 [Database Management]: Systems—*relational databases, rule based databases, query processing*

©2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0  
EDBT 2016 Bordeaux, France

## General Terms

Theory, Algorithms, Design

## Keywords

Data exchange, certain answers, repairs, consistent query answering, disjunctive logic programming, stable models

## 1. INTRODUCTION

Data exchange is the task of transforming data structured under a source schema into data structured under a target schema in such a way that all constraints in a fixed set of source-to-target constraints and in a fixed set of target constraints are satisfied. During the past decade, there has been an extensive and multifaceted investigation of data exchange (see the monograph [1]). There are two main algorithmic problems in data exchange: the problem of materializing an instance that, together with a given source instance satisfies all constraints (such an instance is called a *solution* of the given source instance) and the problem of computing the *certain answers* to a query over the target schema, i.e., the intersection of the answers to the query over all solutions of a given source instance. In data exchange settings with a non-empty set of target constraints, it frequently happens that a given source instance has no solution. In particular, this may happen when the source instance at hand contains inconsistencies or conflicting information that is exposed by the target constraints. The standard data exchange frameworks are not able to provide meaningful answers to target queries in such circumstances; in fact, the certain answers to every target query trivialize. To address this problem and to give meaningful answers to target queries, we recently introduced the framework of *exchange-repair* semantics (or, XR-Certain semantics) [8]. This is an inconsistency-tolerant framework that is based on the notion of *source* repairs, where, informally, a source repair is a source instance that differs minimally from the original source data, but has a solution. In turn, source repairs give rise to the notion of the XR-Certain *answers* to target queries, which, by definition, are the intersection of the answers to the query over all solutions of all source repairs of the given source instance. It should be noted that inconsistency-tolerant semantics have also been investigated in the context of data integration (see, e.g., [7, 17]) and in the context of ontology-based data access (OBDA) (see, e.g., the recent survey [5]). In [9], which is the full version of [8], we provided a detailed comparison between the XR-Certain semantics and the inconsistency-tolerant semantics in these two other frameworks. In partic-

ular, we showed that, as regards consistent query answering, the exchange-repairs framework and the OBDA framework can simulate each other.

A data exchange task is specified using a *schema mapping*  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ , where  $\mathbf{S}$  is the source schema,  $\mathbf{T}$  is the target schema,  $\Sigma_{st}$  is a set of constraints between  $\mathbf{S}$  and  $\mathbf{T}$ , and  $\Sigma_t$  is a set of constraints on  $\mathbf{T}$ . The most extensively studied schema mappings are the ones in which  $\Sigma_{st}$  is a set of source-to-target tuple-generating dependencies (s-t tgds) and  $\Sigma_t$  is a weakly-acyclic set of target tuple-generating dependencies (target tgds) and target equality-generating dependencies (target egds) [11]. Note that tuple-generating dependencies are also known as GLAV (global-and-local as view) constraints; as special cases, they contain the classes of GAV (global-as-view) constraints and LAV (local-as-view) constraints. In [8], it was shown that computing the XR-Certain answers to target conjunctive queries is a coNP-complete problem in data complexity; in fact, this intractability persists even in the case in which  $\Sigma_{st}$  is a set of GAV constraints and  $\Sigma_t$  is a set of egds. Moreover, a connection with disjunctive logic programming was unveiled in [8] by showing that the XR-Certain answers of conjunctive queries can be rewritten as the cautious answers of a union of conjunctive queries with respect to the stable models of a disjunctive logic program over a suitably defined expansion of the source schema.

Here, our main aim is to explore how to effectively perform XR-Certain query answering for practical data exchange settings by leveraging modern sophisticated solvers for disjunctive logic programming. Disjunctive logic programming is a suitable formalism for coping with the intractability of XR-Certain query answering because it goes beyond the natural expressiveness of SQL while still remaining in a declarative framework. Our first technical result is a new improved reduction, accompanied by an optimized implementation, of the problem of computing the XR-Certain answers of queries in the context of data exchange to the problem of computing the certain answers of queries with respect to the stable models of disjunctive logic programs. We then evaluate this approach on a benchmark that we introduce here and which is modeled after a practical data exchange problem in computational genomics. Specifically, we present a benchmark scenario that mimics a portion of the data import process of the UCSC Genome Browser (<https://genome.ucsc.edu/>), a widely used genomics resource that “contains the reference sequence and working draft assemblies for a large collection of genomes.” We believe that data sets from computational sciences, such as computational genomics, are particularly in need of concepts and techniques that, like XR-Certain answers, eliminate the unquantifiable uncertainty that arise from constraint violations.

We carry out two experimental evaluations using real genomic data. The first is based on what we call a *monolithic* approach, which generates a disjunctive logic program from a given query and source instance, and then runs the *clingo* solver from the Potassco collection [12]. The results of this evaluation suggest that the solver fails to take advantage of some critical exploitable structural properties of the specific instances at hand. Intuitively, the cost of transforming the data from the source schema into the target schema is embedded in the execution cost of running each individual query, which causes large instances to become unworkable even for simple queries. In view of this, we develop a dif-

ferent *segmentary* approach that utilizes an improved encoding to take advantage of the aforementioned structural properties using techniques inspired by the notion of a *repair envelope*. The improved implementation utilizing these techniques computes query answers ten to one thousand times faster than the monolithic approach for large instances, and exhibits promising scalability with respect to the size of instances and the rate of target constraint violations.

## 2. PRELIMINARIES

This section contains definitions of basic notions and a minimum amount of background material on data exchange and on disjunctive logic programming. More detailed information about schema mappings and certain answers can be found in [1, 11].

*Instances, Queries, and Homomorphisms.* Fix an infinite set  $\text{Const}$  of elements, and an infinite set  $\text{Nulls}$  of elements such that  $\text{Const}$  and  $\text{Nulls}$  are disjoint. A *schema*  $\mathbf{R}$  is a finite set of relation symbols, each having a designated arity. An  *$\mathbf{R}$ -instance* is a finite database  $I$  over the schema  $\mathbf{R}$  whose active domain is a subset of  $\text{Const} \cup \text{Nulls}$ . A *fact* of an  $\mathbf{R}$ -instance  $I$  is an expression of the form  $R(a_1, \dots, a_k)$ , where  $R$  is a relation symbol of arity  $k$  in  $\mathbf{R}$  and  $(a_1, \dots, a_k)$  is a member of the relation  $R^I$  on  $I$  that interprets the symbol  $R$ . Every  $\mathbf{R}$ -instance can be identified with the set of its facts. We say that an  $\mathbf{R}$ -instance  $I'$  is a *sub-instance* of an  $\mathbf{R}$ -instance  $I$  if  $I' \subseteq I$ , where  $I'$  and  $I$  are viewed as sets of facts. If  $I$  is an  $\mathbf{R}$ -instance and  $\mathbf{R}' \subseteq \mathbf{R}$ , then by the  *$\mathbf{R}'$ -restriction* of  $I$  we will mean the subinstance of  $I$  containing only those facts that involve relations from  $\mathbf{R}'$ .

We assume familiarity with *conjunctive queries* (CQs) and *unions of conjunctive queries* (UCQs). The answers to a query  $q$  in an instance  $I$  are denoted by  $q(I)$ , and we denote by  $q \downarrow(I)$  the answers of  $q$  on  $I$  that contain only values from  $\text{Const}$ .

The *active domain* of an instance  $I$  is the set of values from  $\text{Const} \cup \text{Nulls}$  that occur in facts of  $I$ . By a *homomorphism* from an  $\mathbf{R}$ -instance  $I$  to another  $\mathbf{R}$ -instance  $I'$ , we mean a map  $h$  from the active domain of  $I$  to the active domain of  $I'$ , such that  $h(c) = c$  for all  $c \in \text{Const}$ , and such that for every fact  $R(v_1, \dots, v_n) \in I$  we have that  $R(h(v_1), \dots, h(v_n)) \in I'$ .

*Schema Mappings.* A *tuple-generating dependency* (tgd) over a schema  $\mathbf{R}$  is an expression of the form  $\forall \mathbf{x}(\phi(\mathbf{x}) \rightarrow \exists \mathbf{y}\psi(\mathbf{x}, \mathbf{y}))$ , where  $\phi(\mathbf{x})$  and  $\psi(\mathbf{x}, \mathbf{y})$  are conjunctions of atoms over  $\mathbf{R}$ . Tgds are also known as GLAV (global-and-local-as-view) constraints. Two important special cases are the GAV constraints and the LAV constraints. A GAV constraint is a tgd of the form  $\forall \mathbf{x}(\phi(\mathbf{x}) \rightarrow P(\mathbf{x}))$  (that is, the right-hand side of the implication consists of a single atom without existential quantifiers) and a LAV constraint is a tgd of the form  $\forall \mathbf{x}(R(\mathbf{x}) \rightarrow \exists \mathbf{y}\psi(\mathbf{x}, \mathbf{y}))$  (that is, the left-hand side of the implication consists of a single atom).

Let  $\mathbf{S}$  and  $\mathbf{T}$  be disjoint schemas, called the *source* schema and the *target* schema. A *source-to-target tgd* (s-t tgd, or, source-to-target GLAV constraint) is a tgd as defined above, where  $\phi(\mathbf{x})$  is a conjunction of atoms over  $\mathbf{S}$  and  $\psi(\mathbf{x}, \mathbf{y})$  is a conjunction of atoms over  $\mathbf{T}$ .

An *equality-generating dependency* (egd) over a schema  $\mathbf{R}$  is an expression of the form  $\forall \mathbf{x}(\phi(\mathbf{x}) \rightarrow x_i = x_j)$  with  $\phi(\mathbf{x})$  a conjunction of atoms over  $\mathbf{R}$ .

For the sake of readability, we will frequently drop universal quantifiers when writing tgds and egds.

A *schema mapping* is a quadruple  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{\text{st}}, \Sigma_t)$ , where  $\mathbf{S}$  is a source schema,  $\mathbf{T}$  is a target schema,  $\Sigma_{\text{st}}$  is a finite set of s-t tgds, and  $\Sigma_t$  is a finite set of tgds and/or egds over the target schema. We will also call such schema mappings GLAV+(GLAV, EGD) schema mappings. In the special case where  $\Sigma_{\text{st}}$  consists of (source-to-target) GAV constraints and  $\Sigma_t$  consists of GAV constraints and/or egds, we will say that  $\mathcal{M}$  is a GAV+(GAV, EGD) schema mapping.

**Universal Solutions and Certain Answers.** Let  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{\text{st}}, \Sigma_t)$  be a schema mapping, and let  $I$  be a source instance. As usual in data exchange, we will assume that the source instances we consider do not contain null values.

A target instance  $J$  is a *solution* for a source instance  $I$  w.r.t.  $\mathcal{M}$  if the pair  $(I, J)$  satisfies the constraints of  $\mathcal{M}$ , that is,  $I$  and  $J$  together satisfy  $\Sigma_{\text{st}}$ , while  $J$  satisfies  $\Sigma_t$ . In general, a source instance may have many solutions. A *universal solution* for  $I$  (with respect to  $\mathcal{M}$ ) is a solution  $J$  for  $I$  such that for all solutions  $J'$  of  $I$ , there is a homomorphism  $h$  from  $J$  to  $J'$ . Universal solutions are considered the preferred solutions in data exchange. One reason for this is that universal solutions can be used to compute certain answers to target queries.

If  $q$  is a query over the target schema  $\mathbf{T}$ , then the *certain answers* of  $q$  with respect to  $I$  and  $\mathcal{M}$  are defined as

$$\text{certain}(q, I, \mathcal{M}) = \bigcap \{q(J) : J \text{ is a solution for } I \text{ w.r.t. } \mathcal{M}\}$$

It was shown in [11] that, if  $J$  is a universal solution for a source instance  $I$  w.r.t. a schema mapping  $\mathcal{M}$ , then for every conjunctive query  $q$ , it holds that  $\text{certain}(q, I, \mathcal{M}) = q \downarrow(J)$ .

**Weak Acyclicity and the Chase.** If  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{\text{st}}, \Sigma_t)$  is an arbitrary GLAV+(GLAV, EGD) schema mapping, then a given source instance may have no solution or it may have a solution, but no universal solution. For this reason, in [11] the concept of *weak acyclicity* was introduced, and it was shown that, when  $\Sigma_t$  is the union of a *weakly acyclic* set of target tgds and a set of egds, then, for all source instances  $I$ , a solution exists if and only if a universal solution exists. Moreover, the *chase procedure* can be used to determine in polynomial time (data complexity) whether a solution for  $I$  exists and, if so, to construct a universal solution for  $I$  in time polynomial in the size of  $I$ . The obtained solution, which we will denote by  $\text{chase}(I, \mathcal{M})$  (when it exists), is known as the *canonical universal solution* of  $I$ . We refer to [11] for more details, including the definition of weak acyclicity and of the chase procedure.

By a GLAV+(WA-GLAV, EGD) schema mapping we will mean a schema mapping  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{\text{st}}, \Sigma_t)$ , where  $\Sigma_{\text{st}}$  is a finite set of s-t tgds, and  $\Sigma_t$  is the union of a weakly acyclic set of target tgds and a set of egds. We spell out here two basic facts that are used in several arguments in this paper: let  $\mathcal{M}$  be any GLAV+WA-GLAV schema mapping (without egds). Then (i) every source instance  $I$  has a solution (and hence has a canonical universal solution), and (ii) whenever  $I' \subseteq I$ , then  $\text{chase}(I', \mathcal{M}) \subseteq \text{chase}(I, \mathcal{M})$ . The latter is also known as the *monotonicity of the chase*.

**Disjunctive Logic Programming.** A *disjunctive logic program* (DLP program)  $\Pi$  over a schema  $\mathbf{R}$  is a finite collection of rules of the form

$$\alpha_1 \vee \dots \vee \alpha_n \leftarrow \beta_1, \dots, \beta_m, \neg\gamma_1, \dots, \neg\gamma_k.$$

where  $n, m, k \geq 0$  and  $\alpha_1, \dots, \alpha_n, \beta_1, \dots, \beta_m, \gamma_1, \dots, \gamma_k$  are atoms formed from the relations in  $\mathbf{R} \cup \{=, \neq\}$ , using the constants in  $\text{Const}$  and first-order variables. A DLP program is said to be *ground* if it consists of rules that do not contain any first-order variables. A *model* of  $\Pi$  is an  $\mathbf{R}$ -instance  $I$  over the domain  $\text{Const}$  that satisfies all rules of  $\Pi$  (viewed as universally quantified first-order sentences). A *minimal model* of  $\Pi$  is a model  $M$  of  $\Pi$  such that there does not exist a model  $M'$  of  $\Pi$  where the facts of  $M'$  form a strict subset of the facts of  $M$ . For a ground DLP  $\Pi$  over a schema  $\mathbf{R}$  and an  $\mathbf{R}$ -instance  $M$  over the domain  $\text{Const}$ , the *reduct*  $\Pi^M$  of  $\Pi$  with respect to  $M$  is the DLP containing, for each rule  $\alpha_1 \vee \dots \vee \alpha_n \leftarrow \beta_1, \dots, \beta_m, \neg\gamma_1, \dots, \neg\gamma_k$ , with  $M \not\models \gamma_i$  for all  $i \leq k$ , the rule  $\alpha_1 \vee \dots \vee \alpha_n \leftarrow \beta_1, \dots, \beta_m$ . A *stable model* of a ground DLP  $\Pi$  is an  $\mathbf{R}$ -instance  $M$  over the domain  $\text{Const}$  such that  $M$  is a minimal model of the reduct  $\Pi^M$ . See [13] for more details. The *cautious answers* to a query  $q$ , w.r.t. a DLP program  $\Pi$  (under the stable model semantics) are defined as

$$\bigcap \{q(s) \mid s \text{ is a stable model of } \Pi\}.$$

The stable model semantics is the most widely used semantics of DLP programs, and many solvers have been developed that support reasoning over stable models. In particular, stable models of disjunctive logic programs have been well-studied as a way to compute database repairs ([18] provides a thorough treatment).

### 3. EXCHANGE REPAIR FRAMEWORK

We briefly recall here the exchange-repair framework that was introduced in [8]. The development of this framework was motivated by the observation that the definition of  $\text{certain}(q, I, \mathcal{M})$  trivializes when a source instance  $I$  has no solution w.r.t. a given schema mapping  $\mathcal{M}$ . XR-Certain answers were proposed as a semantics that provides meaningful answers to queries in such cases.

*Definition 1.* [8] Let  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{\text{st}}, \Sigma_t)$  be a schema mapping, let  $I$  be an  $\mathbf{S}$ -instance.

1. A source instance  $I'$  is said to be a *source repair* of  $I$  with respect to  $\mathcal{M}$  if  $I' \subseteq I$ ,  $I'$  has a solution with respect to  $\mathcal{M}$ , and no instance  $I''$  with  $I' \subsetneq I'' \subseteq I$  has a solution with respect to  $\mathcal{M}$ .
2. We say that a pair  $(I', J')$  is an *exchange-repair solution* (or XR-Solution) for  $I$  with respect to  $\mathcal{M}$  if  $I'$  is a source repair of  $I$  with respect to  $\mathcal{M}$  and  $J'$  is a solution for  $I'$  with respect to  $\mathcal{M}$ .
3. For a query  $q$  over the target schema  $\mathbf{T}$ , the *XR-certain answers* to  $q$  in  $I$  w.r.t.  $\mathcal{M}$  is the set

$$\text{XR-Certain}(q, I, \mathcal{M}) = \bigcap \{q(J') \mid (I', J') \text{ is an XR-Solution for } I \text{ w.r.t. } \mathcal{M}\}.$$

Note that, whenever a source instance  $I$  *does* have solutions w.r.t.  $\mathcal{M}$ , then, for all queries  $q$ , we have that

$\text{XR-Certain}(q, I, \mathcal{M}) = \text{certain}(q, I, \mathcal{M})$ . While the XR-Certain semantics takes inspiration from the well-established notions of *database repairs* and *consistent query answers* [2, 4], the precise definition of the semantics reflects important assumptions that are specific to the context of data exchange. Specifically, the definitions of XR-Solution and XR-Certain reflect the fact that in a data exchange setting, it is preferred to make tgds satisfied by deriving additional facts, rather than by deleting facts; and the data used to answer target queries should derive from coherent sets of source facts.

It was shown in [8] that, in the case of GLAV+(WA-GLAV, EGD) schema mappings, the data complexity of XR-Certain query answering for conjunctive queries is coNP-complete (note that the restriction to weakly acyclic schema mappings is necessary here, since it follows from results in [11] that the same problem is undecidable for arbitrary GLAV+(GLAV, EGD) schema mappings). Furthermore, several approaches to query answering were studied in [8]. In particular, it was shown that, for GLAV+(WA-GLAV, EGD) schema mappings, XR-certain answers can be computed by means of a disjunctive logic program. We discuss this approach in the next section.

## 4. BASIC APPROACH TO QUERY ANSWERING

Based on the initial results in [8], we pursue here the development of a practical system for XR-Certain query answering via disjunctive logic programming, leveraging modern sophisticated solvers. Note that the emergence of such powerful solvers for NP-hard problems has already enabled practical solutions for many other computationally hard problems in industry. Concretely, in this section, we present a first implementation of XR-Certain query answering based on a translation along the lines of [8], that takes as input a schema mapping  $\mathcal{M}$  and a source instance  $I$ , and produces a single, typically large, DLP program whose stable models describe the XR-solutions of  $I$  w.r.t.  $\mathcal{M}$ . We refer to this as the *monolithic approach*, in order to contrast it with another approach, which will be presented in Section 6, involving multiple DLP programs of smaller size.

The first step in this approach consists of a reduction from the general case of GLAV+(WA-GLAV, EGD) schema mappings to the case of GAV+(GAV, EGD) schema mappings.

**THEOREM 1** ([8]). *If  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$  is a GLAV+(WA-GLAV, EGD) schema mapping and  $q$  a conjunctive query over  $\mathbf{T}$ , then there exist a GAV+(GAV, EGD) schema mapping  $\hat{\mathcal{M}}$  and union of conjunctive queries  $\hat{q}$  such that  $\text{XR-Certain}(q, I, \mathcal{M}) = \text{XR-Certain}(\hat{q}, I, \hat{\mathcal{M}})$ .*

The resulting schema mapping may in general be exponentially larger than the original. However, as we will see in Section 5.2, our implementation incurs only a modest increase when applied to our benchmark.

Next, in [8], we present a very natural and concise encoding of XR-Certain for GAV+(GAV, EGD) schema mappings as the cautious answers over the parallel circumscription of a disjunctive logic program  $\Pi$ . In [15] it was shown that the problem of finding such models can be subsequently reduced to that of computing stable models of a translation of  $\Pi$  into a new disjunctive logic program. However, the translation involved requires the explicit representation of the herbrand

base of the source instance, which can be prohibitively large even for small source instances. We now present an improved, direct reduction of XR-Certain to the cautious answers over stable models of a disjunctive logic program.

In order to facilitate the discussion below, it is convenient to introduce the notion of a *canonical* XR-Solution. An XR-Solution  $(I', J')$  for  $I$  w.r.t.  $\mathcal{M}$  is said to be a *canonical* XR-Solution if  $J'$  is a canonical universal solution for  $I'$  w.r.t.  $\mathcal{M}$ , that is,  $J' = \text{chase}(I', \mathcal{M})$ .

For any GAV+(GAV, EGD) schema mapping  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ , let  $\Pi_{\mathcal{M}}$  be the DLP program given in Figure 1. Observe that the schema of the program  $\Pi_{\mathcal{M}}$  contains multiple distinct copies of each relation from  $\mathbf{S} \cup \mathbf{T}$ . The program in Theorem 2 intends to describe the canonical XR-Solutions of a source instance. Suppose  $(I', J')$  is a canonical XR-Solution for some source instance  $I$  w.r.t. a GAV+(GAV, EGD) schema mapping  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ , and let  $J$  be the canonical universal solution for  $I$  w.r.t. the tgds of  $\mathcal{M}$ . The program introduces three predicates for each source relation, one with the original name meant to contain the facts of  $I$ , one subscripted with **d** (“deleted”) meant to contain the facts of  $I \setminus I'$ , and one subscripted with **r** (“remains”) meant to contain the facts of  $I'$ . The program also introduces these same three predicates for each target relation, plus a fourth subscripted with **i** (“incidentally deleted”) meant to contain the target facts in  $J \setminus J'$  that are also contained in some subset  $J'' \supseteq J'$  of  $J$  that is consistent with  $\Sigma_t$  (that is, they are not in a canonical XR-Solution but they may appear in some other XR-Solution).

For every stable model  $M$  of  $\Pi_{\mathcal{M}}$ , we denote by  $I^M$  and  $J^M$  the **S**-instance and **T**-instance consisting of those facts  $R(\mathbf{a})$  in the relevant schema for which  $R_r(\mathbf{a}) \in M$ .

**THEOREM 2.** *Let  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$  be a GAV+(GAV, EGD) schema mapping and  $I$  a source instance. The XR-solutions of  $I$  w.r.t.  $\mathcal{M}$  are precisely those pairs of instances that are of the form  $(I^M, J^M)$  for some stable model  $M$  of  $\Pi_{\mathcal{M}} \cup I$ .*

**COROLLARY 1.** *Let  $\mathcal{M}$  be a GAV+(GAV, EGD) schema mapping. For every union of conjunctive queries  $q$  and for every source instance  $I$ ,*

$$\text{XR-Certain}(q, I, \mathcal{M}) =$$

$$\bigcap \{q_r(s) \mid s \text{ a stable model of } \Pi_{\mathcal{M}} \cup I\}$$

where  $q_r$  is the query formed from  $q$  by replacing every occurrence of a relation  $R$  with  $R_r$ .

With this new encoding, we can now compute XR-Certain using the disjunctive logic program  $\Pi_{\mathcal{M}} \cup I$ , whose size is linear in the combined size of the source instance and schema mapping. The DLP generated using the prior approach of [8], in contrast, is lower-bounded in size by  $|\text{adom}(I)|^s$ , where  $s$  is the maximum arity of the relations in the source schema.

## 5. SCENARIO: GENOME BROWSER

Scientific data is a potentially important application of XR-Certain query answering because research decisions and discoveries are often made based on data drawn from a variety of sources, and because unquantifiable uncertainty must



For each tgd  $(R1(\mathbf{x}, \mathbf{y}) \wedge \dots \wedge Rn(\mathbf{x}, \mathbf{y}) \rightarrow T(\mathbf{x})) \in (\Sigma_{st} \cup \Sigma_t)$ , construct a *chase rule*, *deletion rule*, and *target remainder rule* respectively:

$$\begin{aligned} T(\mathbf{x}) &\leftarrow R1(\mathbf{x}, \mathbf{y}), \dots, Rn(\mathbf{x}, \mathbf{y}). \\ R1_d(\mathbf{x}, \mathbf{y}) \vee \dots \vee Rn_d(\mathbf{x}, \mathbf{y}) &\leftarrow T_d(\mathbf{x}), R1(\mathbf{x}, \mathbf{y}), \dots, Rn(\mathbf{x}, \mathbf{y}), \\ &\quad \neg R1_i(\mathbf{x}, \mathbf{y}), \dots, \neg Rn_i(\mathbf{x}, \mathbf{y}). \\ T_r(\mathbf{x}) &\leftarrow R1_r(\mathbf{x}, \mathbf{y}), \dots, Rn_r(\mathbf{x}, \mathbf{y}). \end{aligned}$$

For each egd  $(R1(\mathbf{x}) \wedge \dots \wedge Rn(\mathbf{x}) \rightarrow x_i = x_j) \in \Sigma_t$ , construct a *deletion rule*:

$$R1_d(\mathbf{x}) \vee \dots \vee Rn_d(\mathbf{x}) \leftarrow R1(\mathbf{x}), \dots, Rn(\mathbf{x}), x_i \neq x_j, \neg R1_i(\mathbf{x}), \dots, \neg Rn_i(\mathbf{x}).$$

For each relation  $R \in \mathbf{S}$ , construct a *source remainder rule*:

$$R_r(\mathbf{x}) \leftarrow R(\mathbf{x}), \neg R_d(\mathbf{x})$$

For each relation  $R \in \mathbf{T}$ , construct an *incidental deletion rule*, and the *one-of-three rules*:

$$\begin{aligned} R_i(\mathbf{x}) &\leftarrow R(\mathbf{x}), \neg R_r(\mathbf{x}), \neg R_d(\mathbf{x}) \\ \perp &\leftarrow R_r(\mathbf{x}), R_d(\mathbf{x}) \\ \perp &\leftarrow R_r(\mathbf{x}), R_i(\mathbf{x}) \\ \perp &\leftarrow R_d(\mathbf{x}), R_i(\mathbf{x}) \end{aligned}$$

Figure 1: Procedure to construct the disjunctive logic program  $\Pi_{\mathcal{M}}$

be eliminated from the data before a decision or discovery is made. The guarantee given by XR-Certain query answers – that all possible repairs of the source instance agree on them – is a natural fit for these circumstances.

Inspired by the UCSC Genome Browser, we present a benchmark that uses real data: a loose simulation of the genome browser data import process. The UCSC Genome Browser database is constructed using a variety of algorithms and public data sources. Our benchmark focuses on the human gene model, a set of genomic sequences which putatively capture the portion of the human genome that encodes proteins. The UCSC Genome Browser algorithms compute these sequences from a reference genome by computing alignments for known/observed proteins and transcripts from the UniProt and GenBank databases [14]. For our purposes, we treat the set of transcripts as given (that is, as a source instance rather than the result of a computation), and we provide a schema mapping mimicking how this data, plus a significant volume of data from the RefSeq, Entrez Gene, and UniProt databases, are combined and transformed into the UCSC Genome Browser database. Our schema mapping makes several loose approximations of scientific reality which serve to maximize the portion of the genome browser schema that we populate, but which also introduce some inconsistency to the data. It is for this reason that we say our schema mapping merely *mimicks* the true UCSC Genome Browser data import process, even though our target schema is faithful to the real database.

The RefSeq and EntrezGene databases are not available for download in a flat relational format. The RefSeq database is offered in a text file format, within which the data are arranged in a nested fashion with transcripts as the top-level elements. Every transcript has associated source and reference information (documenting how and by whom

Table 1: Source Instances

Database	Relations	total # of Attributes	total # of Tuples
UCSC*	2	13	165,920
RefSeq	5	38	706,923
EntrezGene	1	3	431,114
UniProt	1	3	4,405,573

\*Transcript alignments and crossreference only.

it was observed), and each may have subsections specifying what protein it encodes and what gene it is transcribed from. These subsections often link to other databases using external protein and gene identifiers. Our ETL step places transcripts, sources, references, genes, and proteins into five separate respective relations, all keyed by transcript accession identifier. The EntrezGene database is available in ASN.1 format, which we first convert to xml using the `gene2xml` tool from the NCBI ToolBox [19]. From the resulting xml, we extract the desired fields into a single table using the `xtract` tool from NCBI Entrez Direct [20].

Our complete schema mapping is available for download at <https://users.soe.ucsc.edu/~rhalpert/xr-benchmarks/>. Table 1 summarizes the data sources.

We represent the given part of UCSC’s gene model with two tables, `ComputedAlignments`, which holds data about the transcripts themselves, and `ComputedCrossref`, which holds a cross-reference between UCSC “known gene IDs” (referred to here as “transcripts”) and the closest corresponding external database transcript identifier (usually a RefSeq accession) and protein identifier (usually a UniProt or RefSeq accession). Our hand-written schema mapping specifies how these tables and the RefSeq, EntrezGene, and UniProt databases are used to populate the target schema. It also applies a key constraint to each target relation, per industry best-practice. Many of the key constraints are specified by the Genome Browser’s schema, while some are not specified but are reasonable constraints that are in fact satisfied by the Genome Browser data.

The true Genome Browser’s process computes a single coherent truth which may differ from that represented in the external databases. Our schema mapping, on the other hand, consolidates these sources into the target schema, and this gives rise to some inconsistency. The key constraints on the `knownGene` and `kgXref` tables prove critical in this regard: they enforce that each transcript have exactly one value for the exon count, and one gene symbol, respectively. Since values from both the UCSC gene model and from the other sources are used to populate the relevant attributes, this effectively gives rise to constraint violations when the UCSC gene model disagrees with RefSeq on the number of exons, and when RefSeq and EntrezGene collectively list more than one gene symbol. These two circumstances are expected to arise a small fraction of the time. The relevant parts of the schema mapping are depicted in Figure 2.

The `knownIsoforms` relation groups transcripts into clusters, where each cluster represents a gene. The Genome Browser computes this relation based on genomic coordinates [14]. Our schema mapping populates the `knownIsoforms` relation using a naive simplification of this approach: transcripts that share either an Entrez Gene ID or a gene symbol are made to reside in the same cluster. These two

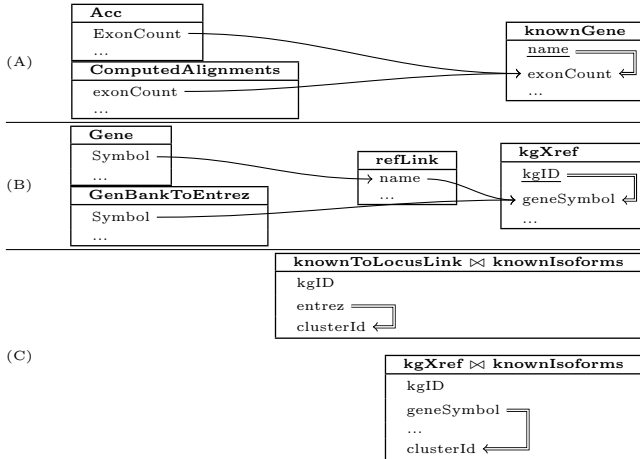


Figure 2: Critical parts of the schema mapping. Single arrows represent value propagation via tgds, and double arrows represent functional dependencies (egds). Keys (also egds) are underlined. (A) Competing values for the exon count. (B) Competing values for the gene symbol. (C) Clustering of transcripts according to Entrez Gene ID and gene symbol. The indicated egds give rise to equalities between nulls.

approaches are incomparable. Ours relies on existing gene symbol annotations from Entrez and UniProt, as well as crossreference annotations on UCSC transcripts, all of which have different levels of rigor and completeness. The knownIsoforms table is thus included in our schema mapping primarily to exercise the interaction of existentially quantified values with egds, which is a differentiating feature of weakly acyclic schema mappings versus other types of syntactic restrictions of GLAV+(GLAV,EGD) schema mappings (e.g., GAV+(GAV,EGD) or separable [6] schema mappings). This part of the schema mapping is depicted in Figure 2.

## 5.1 Benchmark Data and Queries

We wish to test the scalability of our implementation relative to two factors: the instance size and the proportion of the source tuples “involved” in egd violations (called “suspect” tuples), which we make precise with the notion of a *source repair envelope* defined in Section 6.2). To this end, we define a set of instances having specified sizes and ratios of suspect to total source tuples. In order to construct our test instances, we chase the unmodified source instances with the schema mapping and compute which source tuples are suspect. We call this the *raw result*. Then, each test instance is produced by selecting a randomized subset of the source instances with the desired size and ratio of suspect to total tuples in one particular table (ComputedCrossref), which we use as a rough proxy for the ratio for the entire source. Thus, for the largest size (“full”), the maximum ratio is the ratio found in the raw result (2.9%), and there is only one such instance. For the smaller instances, we are able to select enough suspect tuples to produce larger ratios. We use a randomized selection procedure to materialize a set of instances for each profile; six (at 3% suspect) for small and medium instances, three per ratio for large instances, and, necessarily, just one full instance (at 2.9% suspect). The characteristics of these instances are given in Table 2.

Table 2: Test instances have sizes small (S), medium (M), large (L), and full (F), and 0, 3, 9, or 20 percent of their transcripts suspect.

instance:	L0	L3	L9	L20
source tuples	321k	322k	316k	301k
total tuples	716k	724k	731k	748k
suspect transcripts	0%	3%	9%	20%
suspect tuples*	0%	2.0%	5.8%	13.4%
instance:	S3	M3	L3	F3
source tuples	3.5k	36k	322k	1,846k
total tuples	7.9k	77k	724k	5,354k
suspect transcripts	3%	3%	3%	2.9%
suspect tuples*	1.8%	1.8%	2.0%	3.4%

\*includes source and target

Table 3 lists our query suite. Queries labeled “epN” are adapted from the EQUIP query suite [16]: five of the 21 queries given there are applicable to our target schema. Additional queries labeled “xrN” are new queries created to exercise the critical parts of the schema mapping, including what is XR-Certain knowledge in the knownGene relation, and what pairs of transcripts reside in the same cluster in the knownIsoforms relation. In our experiments, we run the queries sequentially.

## 5.2 Monolithic Implementation and Results

Using the reduction given in Theorem 2, we have implemented a *monolithic* approach to XR-Certain query answering. Our monolithic implementation takes as input a GLAV+(WA-GLAV,EGD) schema mapping (encoded as text), an arbitrary source instance (via a JDBC connection string), and a union of conjunctive queries over the target schema (also text). The schema mapping is transformed into a GAV+(GAV,EGD) schema mapping using an optimized version of the reduction in Theorem 1, and the query is transformed into a new union of conjunctive queries using the same reduction. These transformations take an average of 18.7 seconds combined, and the resulting schema mapping is approximately seven times larger than the original (from 33 tgds and 26 egds to 339 tgds and 67 egds). We generate a separate disjunctive logic program for each query and instance, and run them using clingo 4.4.0, a solver from the Potsdam Answer Set Solving Collection[12] (Potassco).

All experiments are run on an 8-core Intel Core i7-2720QM CPU @2.20GHz with 16GB RAM, running Ubuntu 14.04 LTS (Linux 3.13.0 SMP x86\_64). The plots of Figure 3 depict the runtime for these programs versus the percentage of suspect tuples and versus the instance size, respectively, with a line for each query. The latter is a log-log plot, since each instance size is an order of magnitude larger than the last.

These results illustrate a significant problem with this *monolithic* logic program approach: the cost of transforming the data from the source schema into the target schema is embedded in the execution cost of each individual query, which causes large instances to become unworkable even for simple queries. Additionally, the rapid increase in query runtimes as instance size increases, even for queries whose answers should be easy to compute, suggest that this implementation fails to take advantage of some exploitable structure in the instance and schema mapping. This is perhaps a symptom of the fact that the disjunctive logic program’s rules are no simpler for areas of the source and target in-

Table 3: Query Suite, with approximate answer counts for the large-size instances.

Query	Answers
ep1() :- refLink(symbol, -, acc, protacc, -, -, -, -), kgXref(ucscid, -, spid, -, symbol, -, -, -, -)	1*
ep2(protacc) :- refLink(symbol, -, acc, protacc, -, -, -, -), kgXref(ucscid, -, spid, -, symbol, -, -, -, -)	6,000
ep3(protacc,spid) :- refLink(symbol, -, acc, protacc, -, -, -, -), kgXref(ucscid, -, spid, -, symbol, -, -, -, -)	12,000
ep15(symbol) :- kgXref(ucscid, -, -, -, symbol, refseq, -, -, -, -), refLink(-, product, refseq, -, -, -, entrez, -)	1,500
ep16(symbol,entrez) :- kgXref(ucscid, -, -, -, symbol, refseq, -, -, -, -), refLink(-, product, refseq, -, -, -, entrez, -)	1,500
xr1() :- knownGene(kgid, ch, sd, txs, txe, cs, ce, exc, exs, exe, pac, alignid)	1*
xr2(kgid) :- knownGene(kgid, ch, sd, txs, txe, cs, ce, exc, exs, exe, pac, alignid)	10,000
xr3(kgid, ch, sd, txs, txe, cs, ce, exc, exs, exe, pac, ai) :- knownGene(kgid, ch, sd, txs, txe, cs, ce, exc, exs, exe, pac, ai)	10,000**
xr4() :- knownIsoforms(cluster, transcript1), knownIsoforms(cluster, transcript2)	1*
xr5(transcript1) :- knownIsoforms(cluster, transcript1), knownIsoforms(cluster, transcript2)	10,000
xr6(transcript1, transcript2) :- knownIsoforms(cluster, transcript1), knownIsoforms(cluster, transcript2)	35,000

\*boolean \*\*projection-free

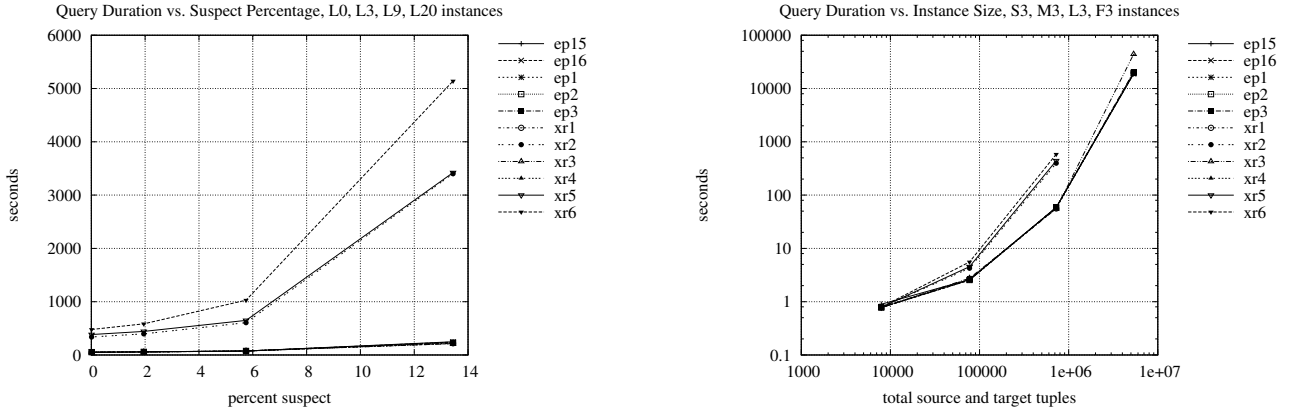


Figure 3: Performance of XR-Certain query answering using clingo.

stances that are unaffected by egd violations than for those that are affected. In the following section, we will develop techniques to identify and exploit such structure by grounding the egds.

## 6. ENHANCED APPROACH TO QUERY ANSWERING

Although the monolithic approach serves as a precise, straightforward specification of XR-Certain answers, it does not lend itself to fine-grained optimization. In this section, we present practical adaptations and optimizations to XR-Certain query answering that are motivated by our experimentation with the monolithic implementation, and that draw on techniques described in the literature for query answering over inconsistent databases, specifically, the notion of *repair envelopes* introduced by Eiter *et al.* [10]. We split query answering into two phases. The first, the “exchange phase”, is a tractable-time query-independent preprocessing step, which enables the second, the “query phase”, in which XR-Certain answers to a particular query are computed by solving a collection of small disjunctive logic programs. Although the problem at hand is coNP-complete, this new *segmentary approach* allows us to answer queries by solving many small hard problems rather than one large one. At the end of this section, we evaluate an implementation based on this enhanced approach.

### 6.1 Candidate Answers

The following definition of *candidate answers* effectively

provides an upper bound on the set of XR-Certain answers of a query, in the sense that the latter is always a subset of the former.

*Definition 2.* Let  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$  be a GAV+(GAV, EGD) schema mapping.

1. We denote by  $\mathcal{M}^{tgd}$  the schema mapping  $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t^{tgd})$  where  $\Sigma_t^{tgd}$  consists of the tgds from  $\Sigma_t$ . That is, all egds are dropped.
2. The *canonical quasi-solution* of a source instance  $I$  w.r.t.  $\mathcal{M}$  is the canonical universal solution of  $I$  w.r.t.  $\mathcal{M}^{tgd}$ .
3. For every UCQ  $q$  over  $\mathbf{T}$ , the *candidate answers* to  $q$  w.r.t.  $I$  and  $\mathcal{M}$  are  $q(J)$ , where  $J$  is the canonical quasi-solution of  $I$  w.r.t.  $\mathcal{M}$ .

*PROPOSITION 1.* Let  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$  be a GAV+(GAV, EGD) schema mapping. Let  $I$  be an  $\mathbf{S}$ -instance. Let  $J$  be the canonical quasi-solution of  $I$  w.r.t.  $\mathcal{M}$ .

1. For all canonical XR-Solutions  $(I', J')$  for  $I$  w.r.t.  $\mathcal{M}$ , it holds that  $J' \subseteq J$ .
2. For every UCQ  $q$ , we have that  $\text{XR-Certain}(q, I, \mathcal{M}) \subseteq q(J)$ . That is, every XR-Certain answer is indeed a candidate answer.

*PROOF.* Let  $(I', J')$  be a canonical XR-Solution for  $I$  w.r.t.  $\mathcal{M}$ . Since  $\mathcal{M}$  is GAV+(GAV, EGD) and  $J'$  is the canonical universal solution for  $I'$  w.r.t.  $\mathcal{M}$ ,  $J'$  is simply the closure

of  $I'$  w.r.t. the tgds of  $\mathcal{M}$ . Therefore  $J'$  is also the canonical universal solution for  $I'$  w.r.t.  $\mathcal{M}^{tgd}$ . Finally, since the chase procedure is monotone [11] and  $I' \subseteq I$ , we have that  $J' \subseteq J$ .

The second item follows directly from the first, since UCQs are monotone queries.  $\square$

## 6.2 Source Repair Envelopes

It is often possible to exclude a large portion of the database from high-complexity computations. We will define a notion similar to a *repair envelope* from [10] but suited to the setting of data exchange.

*Definition 3.* Let  $\mathcal{M}$  be a GLAV+(WA-GLAV, EGD) schema mapping and  $I$  a source instance. A subset  $E$  of  $I$  is a *source repair envelope* if  $(I \setminus I') \subseteq E$  for all source repairs  $I'$ .

We will now see that we can restrict our attention within a source repair envelope when computing source repairs.

**PROPOSITION 2.** *Let  $\mathcal{M}$  be a GLAV+(WA-GLAV, EGD) schema mapping,  $I$  a source instance, and  $E \subseteq I$  a source repair envelope for  $I$  w.r.t.  $\mathcal{M}$ . Then  $\{I' \mid I' \text{ is a source repair of } I \text{ w.r.t. } \mathcal{M}\} = \{E' \cup (I \setminus E) \mid E' \text{ is a source repair of } E \text{ w.r.t. } \mathcal{M}\}$ .*

**PROOF.** Claim:  $I' \cap E$  is a source repair of  $E$ .

Suppose for the sake of contradiction that  $I' \cap E$  is not a source repair of  $E$ . Then either  $I' \cap E$  has no solution (which cannot be the case due to monotonicity of the chase) or  $I' \cap E$  is strictly contained in a source repair  $E''$  of  $E$ . But then  $E''$  must be contained in a source repair  $I''$  of  $I$ . However, the definition of a source repair envelope tells us that  $I''$  contains all of  $E'' \cup (I \setminus E)$ . Hence it contains  $I'$ , so  $I'$  wasn't a source repair of  $I$  after all.

Claim: if  $E'$  is a source repair of  $E$  then  $E' \cup (I \setminus E)$  is a source repair of  $I$ .

Indeed,  $E'$  must be contained in a source repair of  $I$ , and by the definition of a source repair envelope, that source repair of  $I$  must contain all of  $(I \setminus E)$ . Therefore it contains  $E' \cup (I \setminus E)$ . However, it cannot be a strict superset of  $E' \cup (I \setminus E)$  because then  $E'$  could be extended to a larger source repair of  $E$ .  $\square$

There are many ways to calculate a source repair envelope (e.g.,  $I$  is a trivial source repair envelope). Consider the *ideal source repair envelope*, given by  $I - \bigcap \{I' \mid I' \text{ is a source repair for } I \text{ w.r.t. } \mathcal{M}\}$ . Equivalently, the ideal source repair envelope is the minimal source repair envelope for  $I$  w.r.t.  $\mathcal{M}$ . The next result tells us that computing this envelope is hard.

**THEOREM 3.** *Fix a schema mapping  $\mathcal{M}$ . Let the intersection of source repairs membership problem be the following decision problem: given a source instance  $I$ , and a fact  $f$  of  $I$ , is  $f$  contained in the intersection of all source repairs (that is, is  $f \in \bigcap \{I' \mid I' \text{ is a source repair for } I \text{ w.r.t. } \mathcal{M}\}$ )? There is a GAV+(GAV, EGD) schema mapping for which this problem is coNP-hard.*

**PROOF.** The result is proven by reduction from the complement of 3-colorability. Let  $G = (V, E)$  be a graph, with edge set  $E = \{e_1, \dots, e_n\}$ .  $I_G$  is the source instance, with active domain  $V \cup \{1, \dots, n\}$ , containing, for each edge  $e_i = (a, b)$  the fact  $E(a, b, i, i+1)$ ; for each vertex  $a \in V$ , the

facts  $C_r(a), C_g(a), C_b(a)$ ; and one additional fact, namely  $F(n, 1)$ .

$\mathcal{M}$  is the schema mapping consisting of the source-to-target tgds

- $E(x, y, u, v) \wedge C_z(x) \rightarrow E'(x, y)$  (for  $z \in \{r, g, b\}$ )
- $E(x, y, u, v) \wedge C_z(x) \rightarrow F'(u, v)$  (for  $z \in \{r, g, b\}$ )
- $P_z(x) \rightarrow P'_z(x)$  (for  $z \in \{r, g, b\}$ )
- $F(u, v) \rightarrow F'(u, v)$

and target constraints

- $E'(x, y) \wedge P'_z(x) \wedge P'_z(y) \wedge F'(u, v) \rightarrow u = v$  (for all  $z \in \{r, g, b\}$ )
- $F'(u, v) \wedge F'(v, w) \rightarrow F'(u, w)$
- $F'(u, u) \wedge F'(v, w) \rightarrow v = w$

Note that  $I$  has no solutions with respect to  $\mathcal{M}$ , regardless of whether  $G$  is 3-colorable. This is because a solution  $J$  of  $I$  has to contain a directed cycle of  $F'$ -edges (of length  $n$ ), and  $F'$  must be transitive, which means that  $J$  would have to include facts of the form  $F'(i, i)$ , leading to an egd violation. Indeed, every source-repair of  $I$  must either (i) omit at least one of the  $E$ -facts, or (ii) omit all  $P_z$ -facts ( $z \in \{r, g, b\}$ ) for some vertex or (iii) omit the  $F(n, 1)$  fact. It is then not hard to see that  $G$  is 3-colorable if and only if some source repair omits  $F(n, 1)$ . Note that, if  $G$  is 3-colorable, then there is a source repair that retains all  $E$ -facts and that retains at least one  $P_z$ -fact for each vertex ( $z \in \{r, g, b\}$ ). This source repair must then omit the  $F(n, 1)$  fact. If, on the other hand,  $G$  is not 3-colorable, then every source repair has to satisfy (i) or (iii) and, consequently, will include  $F(n, 1)$ .  $\square$

The hardness established in Theorem 3 shows that computing the ideal source repair envelope is not helpful, given that the purpose of a source repair envelope is to help reduce the need for high-complexity computations. Our next result pertains to a source repair envelope that *can* be computed in PTIME.

First, we introduce the notion of *support sets* for a target fact. For an egd or GAV tgd  $\sigma$  and an instance  $I$ , we denote by  $\text{ground}(\sigma, J)$  the set of all groundings of  $\sigma$  using values from the active domain of  $I$  (that is, quantifier-free formulas that can be obtained from  $\sigma$  by replacing universally quantified variables by values from the active domain of  $I$ ). Note that, if  $J$  is a canonical quasi-solution for a source instance  $I$  w.r.t. a GAV+(GAV, EGD) schema mapping, then the active domain of  $J$  is already included in the active domain of  $I$ .

*Definition 4.* Let  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{\text{st}}, \Sigma_t)$  be a GAV+(GAV, EGD) schema mapping. Let  $I$  be an  $\mathbf{S}$ -instance with canonical quasi-solution  $J$ , and let  $f \in J$  be a fact.

- A *support set* for  $f$  is a set of the form  $\{f_1, \dots, f_n\}$  where  $(f_1 \wedge \dots \wedge f_n \rightarrow f) \in \text{ground}(\Sigma_{\text{st}} \cup \Sigma_t, I)$ , and  $(I, J) \models f_1 \wedge \dots \wedge f_n$ . The set of all support sets of  $f$  is denoted by  $\text{support\_sets}(f, I, \mathcal{M})$ .
- The *support closure* for a set  $F$  of facts, denoted as  $\text{support}^*(F, I, \mathcal{M})$ , is the smallest set containing  $F$  such that whenever  $g \in \text{support}^*(F, I, \mathcal{M})$ , then all facts that belong to a support set of  $g$  belong to  $\text{support}^*(F, I, \mathcal{M})$  as well.

*Definition 5.* Let  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{\text{st}}, \Sigma_t)$  be a GAV+(GAV, EGD) schema mapping. Let  $J$  be the canonical quasi-solution of  $I$ . Let

$$\text{violations}(I, \mathcal{M}) = \left\{ f \mid \begin{array}{l} f \text{ occurs in the body of some} \\ \sigma \in \text{ground}(\Sigma_t, I) \text{ with } J \not\models \sigma \end{array} \right\}$$

We say that a source fact  $f \in I$  is *suspect* (w.r.t.  $\mathcal{M}$ ) if it belongs to  $\text{support}^*(\text{violations}(I, \mathcal{M}), I, \mathcal{M})$ , and that it is *safe* otherwise. The set of suspect facts of  $I$  is denoted by  $I_{\text{suspect}}$  and the set of safe facts of  $I$  is denoted by  $I_{\text{safe}}$ .

The violation set is, intuitively, the set of facts that are directly involved in an egd violation, while the violation closure is, intuitively, the set of facts that are, possibly indirectly, involved in an egd violation. The notation  $I_{\text{safe}}$  and  $I_{\text{suspect}}$  assumes that it is clear from the context which schema mapping is being referred to.

*PROPOSITION 3.* Let  $\mathcal{M}$  be a GAV+(GAV, EGD) schema mapping and  $I$  a source instance. Then  $I_{\text{suspect}}$  is a source repair envelope for  $I$  (w.r.t.  $\mathcal{M}$ ). Moreover,  $I_{\text{suspect}}$  can be computed in polynomial time (data complexity).

To see that this proposition holds, suppose that some  $f \in I$  is omitted by a source repair  $I'$  of  $I$ . Then  $I' \cup \{f\}$  has no solution. It follows that  $f$  must be in the violation closure of  $I$  w.r.t.  $\mathcal{M}$  (with the steps of the chase of  $I' \cup \{f\}$  serving as proof of such). Therefore,  $f$  belongs to  $I_{\text{suspect}}$ .

The following example reminds us that  $I_{\text{suspect}}$  is not necessarily a *minimal* source repair envelope.

*Example 1.* Let  $I = \{P(a, b), P(a, c), Q(b, c)\}$ , and let  $\mathcal{M} = (\{P, Q\}, \{P', Q'\}, \{P(x, y) \rightarrow P'(x, y), Q(x, y) \rightarrow Q'(x, y)\}, \{P'(x, y) \wedge P'(x, y') \rightarrow y = y', P'(x, y) \wedge P'(x, y') \wedge Q'(y, y') \rightarrow y = y'\})$ . Then

$$I_{\text{suspect}} = \{P(a, b), P(a, c), Q(b, c)\}$$

However, the key constraint on  $P'$  forces every XR-Solution to have at most one of  $P(a, b), P(a, c)$ , so the second egd in  $\Sigma_t$  is satisfied without eliminating  $Q(b, c)$ . Indeed, the ideal source repair envelope for  $I$  is  $\{P(a, b), P(a, c)\}$ .

Nonetheless,  $I_{\text{suspect}}$  is a potentially useful source repair envelope: we vary the proportion of facts in  $I_{\text{suspect}}$  versus  $I_{\text{safe}}$  in our test instances in order to evaluate the importance of this measure.

We will now extend the notion of a source repair envelope to include target instances.

*Definition 6.* Let  $\mathcal{M}$  be a GAV+(GAV, EGD) schema mapping, and  $I$  a source instance. Let  $J$  be the canonical quasi-solution of  $I$ . Two sets  $E \subseteq I$  and  $F \subseteq J$  of facts, together comprise an *exchange repair envelope*  $(E, F)$  if, for all canonical XR-Solutions  $(I', J')$  of  $I$  w.r.t.  $\mathcal{M}$ , we have that  $(I \setminus I') \subseteq E$  and  $(J \setminus J') \subseteq F$ .

It follows from Theorem 3 that computing the minimal exchange repair envelope is hard. We will now see how to extend an existing source repair envelope to the target, in polynomial time.

*Definition 7.* Let  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{\text{st}}, \Sigma_t)$  be a GAV+(GAV, EGD) schema mapping, and  $I$  be an  $\mathbf{S}$ -instance. Let  $J$  be the canonical quasi-solution of  $I$  w.r.t.  $\mathcal{M}$ . Define the *influence* of a set of facts  $E \subseteq I$ , denoted  $\text{influence}(E, I, \mathcal{M})$ ,

as the smallest set containing  $E$  such that whenever  $g \in \text{influence}(E, I, \mathcal{M})$ , every fact  $f$  that has a support set containing  $g$  also belongs to  $\text{influence}(E, I, \mathcal{M})$ .

*PROPOSITION 4.* Let  $\mathcal{M}$  be a GAV+(GAV, EGD) schema mapping and  $I$  a source instance. Let  $J$  be the canonical quasi-solution of  $I$  w.r.t.  $\mathcal{M}$ . Let  $E$  be a source repair envelope for  $I$  w.r.t.  $\mathcal{M}$ , and let  $F = \text{influence}(E, I, \mathcal{M})$ . Then  $(E, F)$  is an exchange repair envelope for  $I$  w.r.t.  $\mathcal{M}$ .

In particular,  $(I_{\text{suspect}}, J_{\text{suspect}})$  is an exchange repair envelope for  $I$ , where  $J_{\text{suspect}} = \text{influence}(I_{\text{suspect}}, I, \mathcal{M})$ .

*PROOF.* Let  $(I', J')$  be a canonical XR-Solution for  $I$  w.r.t.  $\mathcal{M}$ . Let  $t$  be a fact in  $J \setminus J'$ . Since  $t$  is not in  $(I', J')$ , there must be some fact  $f$  in  $\text{support}^*(\{t\}, I, \mathcal{M})$  in  $I \setminus I'$ . Therefore,  $f$  is contained in the source repair envelope  $E$ , so by definition we have that  $t \in \text{influence}(E, I, \mathcal{M})$ .  $\square$

*Fact 1.* The following two statements hold, where  $F$  is an arbitrary set of target facts:

- The influence of the source restriction of the support closure of  $F$  contains the support closure of  $F$ ; and
- A support closure of  $F$  and its influence are equal over their source restrictions.

In light of the above, we can refer to violation influences instead of violation closures whenever we wish to work with exchange repair envelopes rather than source repair envelopes. It is important to notice that a fact may have one support set which places it in a violation influence, but also another support set whose facts are not contained in any violation influence. Such facts lie in the difference between the violation influence and the ideal exchange repair envelope, but are nonetheless easy to identify.

### 6.3 Violation Clusters

Violation clusters are a concept that will help us further reduce the combinatorial complexity of computing XR-Certain answers. We start with a motivating example.

*Example 2.* Let  $I = \{P_1(a, b), P_1(a, c), P_2(a, b), P_2(a, c), \dots, P_n(a, b), P_n(a, c)\}$ , and let

$$\begin{aligned} \mathcal{M} = (\{P_1, \dots, P_n\}, \{Q_1, \dots, Q_n\}, \\ \{P_1(x, y) \rightarrow Q_1(x, y), \dots, P_n(x, y) \rightarrow Q_n(x, y)\}, \\ \{Q_1(x, y) \wedge Q_1(x, y') \rightarrow y = y', \dots, \\ Q_n(x, y) \wedge Q_n(x, y') \rightarrow y = y'\}) \end{aligned}$$

There are  $2^n$  source repairs, which can be built by choosing one source atom from each of the  $n$  relations, in every possible combination. In this sense, the set of source repairs for this example is highly structured.

Now consider the query  $q(x)$ :-  $Q_1(x, y)$ . Every source repair of  $I$  w.r.t.  $\mathcal{M}$  contains either  $P_1(a, b)$  or  $P_1(a, c)$ , so we can conclude that  $q(a) \in \text{XR-Certain}(q, I, \mathcal{M})$  by considering just the two possibilities for the the  $P_1$  relation. In so doing, we ignore the other  $n - 1$  relations and avoid having to consider  $2^n$  source repairs.

In this section, we will generalize the above observation and demonstrate that it can be used to reduce the size of instances and schema mappings for which we must explore

all source repairs. To do so, we introduce a notion of *independence* that captures when particular egd violations are sufficiently isolated from each other to be processed separately.

To simplify the presentation, it will be convenient to consider schema mappings  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$  in which  $\Sigma_t$  may contain *grounded egds* (where universally quantified variables have been replaced by constants). This will allow us to more easily describe how an instance is carved up into segments that can be processed independently. Intuitively, each grounding of an egd describes one potential violation of that egd. The notions of solutions, universal solutions, source repairs, and exchange repair solutions all apply without modification to schema mappings containing grounded egds.

As a slight abuse of notation, when  $D$  is a set of target constraints, we will write  $\mathcal{M} \cup D$  to denote the schema mapping obtained by adding the constraints in  $D$  to the target constraint set of a schema mapping  $\mathcal{M}$ .

*Definition 8.* Let  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$  be a GAV+(GAV, EGD) schema mapping. Let  $I$  be an  $\mathbf{S}$ -instance. Let  $J$  be the canonical quasi-solution for  $I$  w.r.t.  $\mathcal{M}$ . Let  $\sigma_1, \sigma_2 \in \text{ground}(\Sigma_t, I)$  be distinct grounded egds with  $J \not\models \sigma_1$  and  $J \not\models \sigma_2$ . Let  $E_1$  be the ideal source repair envelope for  $I$  w.r.t.  $(\mathcal{M}^{tgd} \cup \{\sigma_1\})$ , and let  $E_2$  be the ideal source repair envelope for  $I$  w.r.t.  $(\mathcal{M}^{tgd} \cup \{\sigma_2\})$ . We say  $\sigma_1$  and  $\sigma_2$  are *pairwise-independent* if  $\text{source\_repairs}(I, \mathcal{M}^{tgd} \cup \{\sigma_1, \sigma_2\}) = \{(I \setminus (E_1 \cup E_2)) \cup E'_1 \cup E'_2 \mid E'_1 \in \text{source\_repairs}(I \cap E_1, \mathcal{M}^{tgd} \cup \{\sigma_1\}) \text{ and } E'_2 \in \text{source\_repairs}(I \cap E_2, \mathcal{M}^{tgd} \cup \{\sigma_2\})\}$ . We say  $\sigma_1$  and  $\sigma_2$  are *pairwise-dependent* if they are not *pairwise-independent*.

Consider the graph of all egd violations in the quasi-solution and connect each pair of pairwise dependent egd violations by an edge. Each connected component of this graph is called a *violation cluster*. If  $\sigma_1$  and  $\sigma_n$  reside in distinct violation clusters then we say  $\sigma_1$  and  $\sigma_n$  are *independent*.

If a pair of violations is independent, by definition, their XR-Solutions can be processed separately, then suitably recombined. Note that the above definition of violation clusters does not provide us with a way to compute them efficiently, because the definition involves ideal source repair envelopes. The following proposition provides an approximation.

*PROPOSITION 5.* Let  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$  be a GAV+(GAV, EGD) schema mapping. Let  $I$  be an  $\mathbf{S}$ -instance. Let  $J$  be the canonical quasi-solution for  $I$  w.r.t.  $\mathcal{M}$ . Let  $\sigma_1, \sigma_2$  be distinct grounded egds in  $\text{ground}(\Sigma_t, I)$  such that  $J \not\models \sigma_1$  and  $J \not\models \sigma_2$ . Let  $E_1, E_2$  be ideal source repair envelopes for  $I$  w.r.t.  $\mathcal{M} \cup \sigma_1$  and  $\mathcal{M} \cup \sigma_2$ , respectively. If  $E_1$  and  $E_2$  are disjoint, then  $\sigma_1$  and  $\sigma_2$  are *pairwise-independent*.

*PROOF.* Suppose  $E_1 \cap E_2 = \emptyset$ , and let  $(I', J')$  be a canonical XR-Solution for  $I$  w.r.t.  $\mathcal{M} \cup \{\sigma_1, \sigma_2\}$ . Let  $E'_1 = E_1 \cap I'$  and let  $E'_2 = E_2 \cap I'$ . By definition of a source repair envelope,  $(I \setminus E_1) \cup E'_1$  is an XR-Solution for  $I$  w.r.t.  $\mathcal{M} \cup \{\sigma_1\}$ , and likewise  $(I \setminus E_2) \cup E'_2$  is an XR-Solution for  $I$  w.r.t.  $\mathcal{M} \cup \{\sigma_2\}$ , and since  $E_1$  and  $E_2$  are disjoint, it is easy to see that  $E'_1$  is an XR-Solution for  $E_1$  w.r.t.  $\mathcal{M} \cup \{\sigma_1, \sigma_2\}$  and  $E'_2$  is an XR-Solution for  $E_2$  w.r.t.  $\mathcal{M} \cup \{\sigma_1, \sigma_2\}$ . Finally, since GAV chase is monotone, we have that  $(I \setminus E_1 \setminus E_2) \cup (E'_1) \cup (E'_2)$

has a solution w.r.t.  $\mathcal{M} \cup \{\sigma_1, \sigma_2\}$ , and there is no instance  $I'' \subset I$  which strictly contains  $(I \setminus E_1 \setminus E_2) \cup (E'_1) \cup (E'_2)$  and also has a solution w.r.t.  $\mathcal{M} \cup \{\sigma_1, \sigma_2\}$ .  $\square$

We can, in polynomial time, compute the support closure for each egd violation, then compute an overapproximation of the violation clusters based on the restriction to the source schema of those closures. The next proposition follows simply from the definition of a support closure, but provides some intuition and gives a shortcut for computing a source repair envelope for a violation cluster.

*PROPOSITION 6.* Let  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$  be a GAV+(GAV, EGD) schema mapping. Let  $I$  be an  $\mathbf{S}$ -instance. Let  $J$  be the canonical quasi-solution for  $I$  w.r.t.  $\mathcal{M}$ . Let  $\sigma_1, \dots, \sigma_n$  be a violation cluster (so each  $\sigma_i$  is a ground egd where  $J \not\models \sigma_i$ ), with support closures  $E_1, \dots, E_n$ . Then  $E_1 \cup \dots \cup E_n$  is the support closure of the facts in  $\sigma_1, \dots, \sigma_n$ , and its  $\mathbf{S}$ -restriction is a source repair envelope for  $I$  w.r.t.  $\mathcal{M}^{tgd} \cup \{\sigma_1, \dots, \sigma_n\}$

We have now seen how to compute a conservative approximation of the violation clusters for an instance w.r.t. a given schema mapping. Proposition 6 makes clear the fact that distinct violation clusters have disjoint source repair envelopes, and are therefore pairwise-independent themselves. Definition 8 tells us that we can thus compute the source repairs for an entire instance by computing separately the source repairs for the envelope of each violation cluster. We will now see how this supports query answering.

## 6.4 Answering Queries

We now show how to use the techniques introduced in the previous sections to compute XR-Certain answers for unions of conjunctive queries. In fact, without loss of generality, we can restrict attention to projection-free atomic queries: Let  $q(\mathbf{x}) :- \phi_1(\mathbf{x}, \mathbf{y}) \vee \dots \vee \phi_n(\mathbf{x}, \mathbf{y})$  be a union of conjunctive queries with  $n$  clauses. Define  $t_1, \dots, t_n$  to be a set of new GAV tgds, where the head of each tgd is the head of  $q$ , and the body of each tgd  $t_k$  is  $\phi_k(\mathbf{x}, \mathbf{y})$ . If we extend a canonical quasi-solution  $J$  with the new relation symbol  $q$ , and chase with  $t_1, \dots, t_n$ , the result will be  $J \cup \{q(J)\}$ . We will use the notation  $\text{XR-Certain}(I, \mathcal{M} \cup \{t_1, \dots, t_n\})$  to denote the set of facts in the intersection of all exchange repair solutions for  $I$  w.r.t.  $\mathcal{M} \cup \{t_1, \dots, t_n\}$ , and the term *candidate facts* to refer to the tuples of any relation in  $\mathbf{T} \cup q$ .

Consider the definition of *support sets* in Section 6.2, and suppose  $f$  is a candidate fact. By definition,  $f$  is XR-Certain if it is contained in every exchange repair solution. Since exchange repair solutions satisfy the constraints of the schema mapping, it is easy to see that  $f$  is XR-Certain if it has at least one support set in every XR-Solution, or equivalently, in every canonical XR-Solution.

Proposition 6 naturally extends to exchange repair envelopes. Thus we define a *violation cluster influence* to refer to the union of the influences of the violations in a cluster.

*Example 3.* This example illustrates that a candidate fact  $f$  may belong to the influences of the influences of multiple distinct violation clusters. Let  $I = \{P(a_1, a_2), P(a_1, a_3), Q(a_1, a_2), Q(a_1, a_3)\}$ , and let  $\mathcal{M} = (\{P, Q\}, \{R, S, T\}, \{P(x, y) \rightarrow R(x, y), Q(x, y) \rightarrow S(x, y)\}, \{R(x, y) \wedge S(x, z) \rightarrow T(x, y, z), R(x, y) \wedge R(x, y') \rightarrow$

$y = y', S(x, y), S(x, y') \rightarrow y = y'\}$ . Then the violation cluster influence for  $\{R(a_1, a_2), R(a_1, a_3)\}$  is

$$\left\{ \begin{array}{l} P(a_1, a_2), P(a_1, a_3), R(a_1, a_2), R(a_1, a_3), \\ T(a_1, a_2, a_3), T(a_1, a_3, a_2), T(a_1, a_2, a_2), T(a_1, a_3, a_3) \end{array} \right\}$$

and the violation cluster influence for  $S(a_1, a_2), S(a_1, a_3)$  is

$$\left\{ \begin{array}{l} Q(a_1, a_2), Q(a_1, a_3), S(a_1, a_2), S(a_1, a_3), \\ T(a_1, a_2, a_3), T(a_1, a_3, a_2), T(a_1, a_2, a_2), T(a_1, a_3, a_3) \end{array} \right\}$$

which are disjoint in their restriction to the source schema, yet both contain the target facts  $T(a_1, a_2, a_3)$ ,  $T(a_1, a_3, a_2)$ ,  $T(a_1, a_2, a_2)$ , and  $T(a_1, a_3, a_3)$ .

This example illustrates how distinct target violations with non-overlapping source repair envelopes may jointly affect target tuples; we cannot determine the status of tuples in  $T$  without considering violations of both key constraints.

Suppose  $f$  is a candidate fact in  $T$ . Each support set for  $f$  may be contained in only certain combinations of XR-Solutions from the violation cluster influences containing  $f$ . So, to determine if  $f$  has at least one support set in every XR-Solution w.r.t. the broader schema mapping, it is necessary to consider **all** combinations of XR-Solutions from the violation cluster influences containing  $f$ . We call the set of violation clusters whose influences contain  $f$  the *signature* of  $f$ , denoted  $\text{signature}(f)$ . Recall that  $I_{\text{safe}}$  denotes the set of source facts that are *safe*, that is, not *suspect* (Definition 5). In the following, let  $J_{\text{safe}}$  denote  $\text{chase}(I_{\text{safe}}, \mathcal{M})$ .

**THEOREM 4.** *Let  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$  be a GAV+(GAV, EGD) schema mapping. Let  $I$  be an  $\mathbf{S}$ -instance. Let  $J$  be the canonical quasi-solution for  $I$  w.r.t.  $\mathcal{M}$ . Let  $f$  be a candidate fact in  $J$ . Let  $I_{f\text{-focus}}$  and  $J_{f\text{-focus}}$  be, respectively, the source and target parts of  $\bigcup\{V \mid V \text{ the influence for a violation cluster in } \text{signature}(f)\}$ . Then  $f \in \text{XR-Certain}(I, \mathcal{M}) \leftrightarrow f \in \bigcap\{\text{chase}(J_{f\text{-focus}} \cup J_{\text{safe}}, \Sigma_t) \mid J_{f\text{-focus}} \in \text{XR-Solution}(I_{f\text{-focus}}, \mathcal{M})\}$ .*

**PROOF SKETCH.** By definition, the violations in the clusters in  $\text{signature}(f)$  are contained in  $J_{f\text{-focus}}$ . Thus  $(I_{f\text{-focus}}, J_{f\text{-focus}})$  is an exchange repair envelope for  $I$  w.r.t.  $\mathcal{M}^{\text{tgd}}$  augmented with those violations. Furthermore, all of the violations in  $J_{f\text{-focus}}$  are pairwise independent of all of the violations in  $J \setminus J_{f\text{-focus}}$ , from which we conclude that every fact in every support set for  $f$  is contained in  $(I_{f\text{-focus}}, J_{f\text{-focus}})$  or in  $(I_{\text{safe}}, J_{\text{safe}})$ .  $\square$

This result gives us the following algorithm for computing XR-Certain for an instance  $I$  and schema mapping  $\mathcal{M}$ , using a (hopefully large)  $J_{\text{safe}}$  combined with, for each signature, a (hopefully small)  $J_{\text{focus}}$ . For the exchange phase: Chase  $I$  with  $\mathcal{M}^{\text{tgd}}$ , compute the violation set of  $I$  w.r.t.  $\mathcal{M}$ , and compute the support closure of each violation. Then mark source facts safe if they do not reside in any violation closure, chase  $I_{\text{safe}}$  with  $\mathcal{M}^{\text{tgd}}$ , and mark every resulting fact safe. Lastly, compute violation clusters, and the influence of each cluster. For the query phase: Compute the candidate facts, marking safe those with support sets in  $J_{\text{safe}}$ . Next, compute the signature of each unmarked candidate fact. Finally, generate and solve a grounded disjunctive logic program to compute XR-Certain w.r.t.  $\mathcal{M}$  for  $(I_{\text{focus}}, J_{\text{focus}})$  for each signature. This program is the restriction to  $(I_{\text{focus}}, J_{\text{focus}})$  of the grounding of the program from Theorem 2. Facts in  $(I_{\text{focus}}, J_{\text{focus}}) \cap (I_{\text{safe}}, J_{\text{safe}})$  may be represented by the value TRUE in the program.

## 6.5 Segmentary Implementation and Results

Using the techniques developed in this section, we have implemented a *segmentary* approach to XR-Certain query answering using Java 1.7.0\_80, MySQL 5.5.42, and clingo 4.4.0. Our segmentary implementation takes as input a GLAV+(WA-GLAV, EGD) schema mapping (encoded as text), an arbitrary source instance (via a JDBC connection string), and a union of conjunctive queries over the target schema (also text). As with the monolithic implementation, the schema mapping is transformed into a GAV+(GAV, EGD) schema mapping. Additionally, the query is transformed into an atomic query using the reduction described at the beginning of Section 6.4.

Using the above algorithm, query answering is done in two phases: the exchange phase, and the query phase. The exchange phase materializes the target instance in MySQL using a chase procedure written in Java. The detailed implementation of the chase procedure is immaterial: here, we use a semi-naïve chase. The exchange phase next computes violation cluster influences, fact signatures, and the “safe” part of the source and target instances (also in MySQL, run from Java). The query phase appends candidate answers to the target instance, marks “safe” candidates, and generated disjunctive logic programs as explained in Section 6.4, which are then solved using clingo. The results obtained from clingo are used to mark each candidate either “accepted” or “rejected”. The “safe” candidates and “accepted” candidates together comprise the XR-Certain query answers.

Table 4: Duration of the exchange phase, in seconds.

instance	L0	L3	L9	L20
duration	150.7	196.7	235.7	297.3
instance	S3	M3	L3	F3
duration	36.5	50.8	196.7	2229.7

Table 4 gives the runtime of the exchange phase for each instance. Notice that for large instances, the exchange phase compares very favorably against the per-query runtime of the monolithic approach described in Section 5.2. The plots in Figure 4 give the performance of each query as we scale the rate of violations and the instance size, respectively, with the latter on a log-log scale. These results improve considerably on the monolithic approach: the query phase runtimes are between ten times and one thousand times faster for large and full instances.

## 7. CONCLUDING REMARKS

We have implemented XR-Certain query answering and evaluated it on real data using a benchmark that mimics the UCSC Genome Browser data import process. Our experiments suggest that using the reduction from XR-Certain to disjunctive logic programming to create a monolithic logic program is not a viable approach using today’s best-of-breed solvers. However, by efficiently computing a suitable overapproximation, our segmentary approach computes query answers ten to one thousand times faster for larger instances, and exhibits promising scalability with respect to both instance size and the rate of target constraint violations.

We note that significant progress has recently been made toward broadly applicable, reproducible schema mapping benchmarks, in particular iBench [3]. We intend as fu-

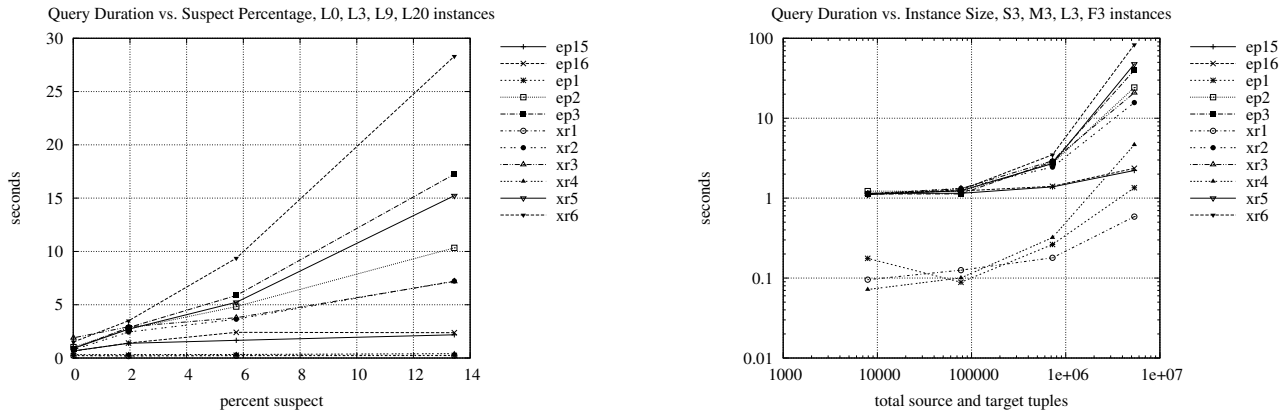


Figure 4: Performance of XR-Certain query answering using MySQL along with clingo.

ture work to further evaluate our segmentary implementation on such benchmarks. Nonetheless, our success with our Genome Browser benchmark serves as evidence that XR-Certain query answering may be efficiently computable in practice for realistic applications.

## 8. ACKNOWLEDGMENTS

The research of all authors was partially supported by NSF Grant IIS-1217869. Kolaitis' research was also supported by the project "Handling Uncertainty in Data Intensive Applications" under the program THALES.

## 9. REFERENCES

- [1] M. Arenas, P. Barceló, L. Libkin, and F. Murlak. *Relational and XML Data Exchange*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.
- [2] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In V. Vianu and C. H. Papadimitriou, editors, *PODS*, pages 68–79. ACM Press, 1999.
- [3] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The iBench integration metadata generator. *PVLDB*, 9(3):108–119, 2015.
- [4] L. E. Bertossi. *Database Repairing and Consistent Query Answering*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.
- [5] M. Bienvenu and M. Ortiz. Ontology-mediated query answering with data-tractable description logics. In *Reasoning Web. Web Logic Rules - 11th Int. Summer School*, pages 218–307, 2015.
- [6] A. Cali, M. Console, and R. Frosini. Deep separability of ontological constraints. *CoRR*, abs/1312.5914, 2013.
- [7] A. Cali, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In F. Neven, C. Beeri, and T. Milo, editors, *PODS*, pages 260–271. ACM, 2003.
- [8] B. ten Cate, R. L. Halpert, and P. G. Kolaitis. Exchange-repairs: Managing inconsistency in data exchange. In *RR*, volume 8741 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2014.
- [9] B. ten Cate, R. L. Halpert, and P. G. Kolaitis. Exchange-repairs: Managing inconsistency in data exchange, September 2015. arXiv <http://arxiv.org/abs/1509.06390>, 29 pages; to appear in the *Journal of Data Semantics*.
- [10] T. Eiter, M. Fink, G. Greco, and D. Lembo. Repair localization for query answering from inconsistent databases. *ACM Trans. Database Syst.*, 33(2), 2008.
- [11] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.
- [12] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):107–124, 2011.
- [13] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.
- [14] F. Hsu, W. J. Kent, H. Clawson, R. M. Kuhn, M. Diekhans, and D. Haussler. The UCSC known genes. *Bioinformatics*, 22(9):1036–1046, 2006.
- [15] T. Janhunen and E. Oikarinen. Capturing parallel circumscription with disjunctive logic programs. In J. J. Alferes and J. A. Leite, editors, *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pages 134–146. Springer, 2004.
- [16] P. G. Kolaitis, E. Pema, and W.-C. Tan. Efficient querying of inconsistent databases with binary integer programming. *PVLDB*, 6(6):397–408, 2013.
- [17] D. Lembo, M. Lenzerini, and R. Rosati. Source inconsistency and incompleteness in data integration. In *KRDB*, 2002.
- [18] M. C. Marileo and L. E. Bertossi. The consistency extractor system: Answer set programs for consistent query answering in databases. *Data Knowl. Eng.*, 69(6):545–572, 2010.
- [19] National Center for Biotechnology Information. NCBI ToolBox, 2001. <http://www.ncbi.nlm.nih.gov/IEB/ToolBox/>.
- [20] National Center for Biotechnology Information. Entrez Programming Utilities, 2013. <http://www.ncbi.nlm.nih.gov/books/NBK25501/>.



# Querying RDF Data Using A Multigraph-based Approach

Vijay Ingalalli  
LIRMM, IRSTEA  
Montpellier, France  
vijay@lirmm.fr

Dino Ienco  
IRSTEA  
Montpellier, France  
dino.ienco@irstea.fr

Pascal Poncelet  
LIRMM  
Montpellier, France  
pascal.poncelet@lirmm.fr

Serena Villata  
CNRS, I3S Laboratory  
Sophia Antipolis, France  
villata@i3s.unice.fr

## ABSTRACT

RDF is a standard for the conceptual description of knowledge, and SPARQL is the query language conceived to query RDF data. The RDF data is cherished and exploited by various domains such as life sciences, Semantic Web, social network, etc. Further, its integration at Web-scale compels RDF management engines to deal with complex queries in terms of both size and structure. In this paper, we propose AMBER (Attributed Multigraph Based Engine for RDF querying), a novel RDF query engine specifically designed to optimize the computation of complex queries. AMBER leverages subgraph matching techniques and extends them to tackle the SPARQL query problem. First of all RDF data is represented as a multigraph, and then novel indexing structures are established to efficiently access the information from the multigraph. Finally a SPARQL query is represented as a multigraph, and the SPARQL querying problem is reduced to the subgraph homomorphism problem. AMBER exploits structural properties of the query multigraph as well as the proposed indexes, in order to tackle the problem of subgraph homomorphism. The performance of AMBER, in comparison with state-of-the-art systems, has been extensively evaluated over several RDF benchmarks. The advantages of employing AMBER for complex SPARQL queries have been experimentally validated.

## 1. INTRODUCTION

In the recent years, structured knowledge represented in the form of RDF data has been increasingly adopted to improve the robustness and the performances of a wide range of applications with various purposes. Popular examples are provided by Google, that exploits the so called *knowledge graph* to enhance its search results with semantic information gathered from a wide variety of sources, or by Facebook, that implements the so called *entity graph* to empower its search engine and provide further information extracted, for

instance by Wikipedia. Another example is supplied by recent question-answering systems [4, 15] that automatically translate natural language questions in SPARQL queries and successively retrieve answers considering the available information in the different Linked Open Data sources. In all these examples, complex queries (in terms of size and structure) are generated to ensure the retrieval of all the required information. Thus, as the use of large knowledge bases, that are commonly stored as RDF triplets, is becoming a common way to ameliorate a wide range of applications, efficient querying of RDF data sources using SPARQL is becoming crucial for modern information retrieval systems.

All these different scenarios pose new challenges to the RDF query engines for two vital reasons: firstly, the automatically generated queries cannot be bounded in their structural complexity and size (e.g., the DBPEDIA SPARQL Benchmark [12] contains some queries having more than 50 triplets [1]); secondly, the queries generated by retrieval systems (or by any other applications) need to be efficiently answered in a reasonable amount of time. Modern RDF data management, such as *x-RDF-3X* [13] and *Virtuoso* [7], are designed to address the scalability of SPARQL queries but they still have problems to answer big and structurally complex SPARQL queries [2]. Our experiments with state-of-the-art systems demonstrate that they fail to efficiently manage such kind of queries (Table 1).

Systems	AMBER	<i>gStore</i>	<i>Virtuoso</i>	<i>x-RDF-3X</i>
Time (sec)	1.56	11.96	20.45	>60

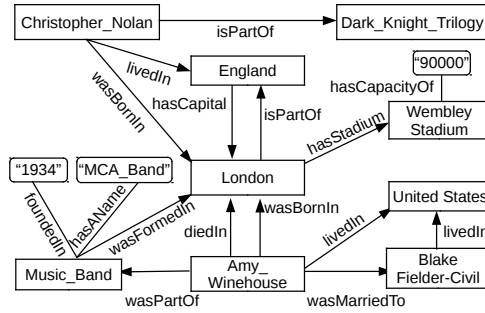
Table 1: Average Time (seconds) for a sample of 200 complex queries on *DBPEDIA*. Each query has 50 triplets.

In order to tackle these issues, in this paper, we introduce AMBER (Attributed Multigraph Based Engine for RDF querying), which is a graph-based RDF engine that involves two steps: an offline stage where RDF data is transformed into multigraph and indexed, and an online step where an efficient approach to answer SPARQL query is proposed. First of all RDF data is represented as a multigraph where subjects/objects constitute vertices and multiple edges (predicates) can appear between the same pair of vertices. Then, new indexing structures are conceived to efficiently access RDF multigraph information. Finally, by representing the SPARQL queries also as multigraphs, the query answering

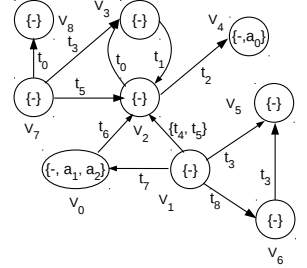
Prefixes: x= <http://dbpedia.org/resource/>; y=<http://dbpedia.org/ontology/>

Subject	Predicate	Object
x:London	y:isPartOf	x:England
x:England	y:hasCapital	x:London
x:Christopher_Nolan	y:wasBornIn	x:London
x:Christopher_Nolan	y:LivedIn	x:England
x:Christopher_Nolan	y:isPartOf	x:Dark_Knight_Triology
x:London	y:hasStadium	x:WembleyStadium
x:WembleyStadium	y:hasCapacityOf	"90000"
x:Amy_Winehouse	y:wasBornIn	x:London
x:Amy_Winehouse	y:diedIn	x:London
x:Amy_Winehouse	y:wasPartOf	x:Music_Band
x:Music_Band	y:hasName	"MCA_Band"
x:Music_Band	y:FoundedIn	"1994"
x:Music_Band	y:wasFormedIn	X:London
x:Amy_Winehouse	y:livedIn	x:United States
x:Amy_Winehouse	y:wasMarriedTo	x:Blake Fielder-Civil
x:Blake Fielder-Civil	y:livedIn	x:United States

(a) RDF tripleset



(b) Graph representation of RDF data



(c) Equivalent multigraph  $G$

Figure 1: (a) RDF data in n-triple format; (b) graph representation (c) attributed multigraph  $G$

task can be reduced to the problem of subgraph homomorphism. To deal with this problem, AMBER employs an efficient approach that exploits structural properties of the multigraph query as well as the indices previously built on the multigraph structure. Experimental evaluation over popular RDF benchmarks show the quality in terms of time performances and robustness of our proposal.

In this paper, we focus only on the SELECT/WHERE clause of the SPARQL language<sup>1</sup>, that constitutes the most important operation of any RDF query engines. It is out of the scope of this work to consider operators like FILTER, UNION and GROUP BY or manage RDF update. Such operations can be addressed in future as extensions of the current work.

The paper is organized as follows. Section 2 introduces the basic notions about RDF and SPARQL language. In Section 3 AMBER is presented. Section 4 describes the indexing strategy while Section 5 presents the query processing. Related works are discussed in Section 6. Section 7 provides the experimental evaluation. Section 8 concludes.

## 2. BACKGROUND AND PRELIMINARIES

In this section we provide basic definitions on the interplay between RDF and its multigraph representation. Later, we explain how the task of answering SPARQL queries can be reduced to multigraph homomorphism problem.

### 2.1 RDF Data

As per the W3C standards<sup>2</sup>, RDF data is represented as a set of triples  $\langle S, P, O \rangle$ , as shown in Figure 1a, where each triple  $\langle s, p, o \rangle$  consists of three components: a *subject*, a *predicate* and an *object*. Further, each component of the RDF triple can be of any two forms; an *IRI* (Internationalized Resource Identifier) or a literal. For brevity, an *IRI* is usually written along with a prefix (e.g.,  $\langle \text{http://dbpedia.}$

$\text{org/resource/isPartOf} \rangle$  is written as ‘x:isPartOf’), whereas a literal is always written with double quotes (e.g., “90000”). While a subject  $s$  and a predicate  $p$  are always an *IRI*, an object  $o$  is either an *IRI* or a literal.

RDF data can also be represented as a directed graph where, given a triple  $\langle s, p, o \rangle$ , the subject  $s$  and the object  $o$  can be treated as vertices and the predicate  $p$  forms a directed edge from  $s$  to  $o$ , as depicted in Figure 1b. Further, to underline the difference between an *IRI* and a literal, we use standard rectangles and arc for the former while we use beveled corner and edge (no arrows) for the latter.

#### 2.1.1 Data Multigraph Representation

Motivated by the graph representation of RDF data (Figure 1b), we take a step further by transforming it to a data multigraph  $G$ , as shown in Figure 1c.

Let us consider an RDF triple  $\langle s, p, o \rangle$  from the RDF tripleset  $\langle S, P, O \rangle$ . Now to transform the RDF tripleset into data multigraph  $G$ , we set four protocols: we always treat the subject  $s$  as a vertex; a predicate  $p$  is always treated as an edge; we treat the object  $o$  as a vertex only if it is an *IRI* (e.g., vertex  $v_2$  corresponds to object ‘x:London’); when the object is a literal, we combine the object  $o$  and the corresponding predicate  $p$  to form a tuple  $\langle p, o \rangle$  and assign it as an attribute to the subject  $s$  (e.g.,  $\langle \text{‘y:hasCapacityOf’, ‘90000’} \rangle$  is assigned to vertex  $v_4$ ). Every vertex is assigned a null value  $\{-\}$  in the attribute set. However, to realize this in the realms of graph management techniques, we maintain three different dictionaries, whose elements are a pair of ‘key’ and ‘value’, and a mapping function that links them. The three dictionaries depicted in Table 2 are: a vertex dictionary (Table 2a), an edge-type dictionary (Table 2b) and an attribute dictionary (Table 2c). In all the three dictionaries, an RDF entity represented by a ‘key’ is mapped to a corresponding ‘value’, which can be a vertex/edge/attribute identifier. Thus by using the mapping functions -  $\mathcal{M}_v$ ,  $\mathcal{M}_e$ , and  $\mathcal{M}_a$  for vertex, edge-type and attribute mapping respectively, we obtain a directed, vertex attributed data multi-

<sup>1</sup><http://www.w3.org/TR/sparql11-overview/>

<sup>2</sup><http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/>

graph  $G$  (Figure 1c), which is formally defined as follows.

**DEFINITION 1.** Directed, Vertex Attributed Multigraph. A directed, vertex attributed multigraph  $G$  is defined as a 4-tuple  $(V, E, L_V, L_E)$  where  $V$  is a set of vertices,  $E \subseteq V \times V$  is a set of directed edges with  $(v, v') \neq (v', v)$ ,  $L_V$  is a labelling function that assigns a subset of vertex attributes  $A$  to the set of vertices  $V$ , and  $L_E$  is a labelling function that assigns a subset of edge-types  $T$  to the edge set  $E$ .

To summarise, an RDF tripleset is transformed into a data multigraph  $G$ , whose elements are obtained by using the mapping functions as already discussed. Thus, the set of vertices  $V = \{v_0, \dots, v_m\}$  is the set of mapped subject/object *IRI*, and the labelling function  $L_V$  assigns a set of vertex attributes  $A = \{-, a_0, \dots, a_n\}$  (mapped tuple of predicate and object-literal) to the vertex set  $V$ . The set of directed edges  $E$  is a set of pair of vertices  $(v, v')$  that are linked by a predicate, and the labelling function  $L_E$  assigns the set of edge types  $T = \{t_0, \dots, t_p\}$  (mapped predicates) to these set of edges. The edge set  $E$  maintains the topological structure of the RDF data. Further, mapping of object-literals and the corresponding predicates as a set of vertex attributes, results in a compact representation of the multigraph. For example (in Fig. 1c), all the object-literals and the corresponding predicates are reduced to a set of vertex attributes.

## 2.2 SPARQL Query

A SPARQL query usually contains a set of triple patterns, much like RDF triples, except that any of the subject, predicate and object may be a variable, whose bindings are to be found in the RDF data<sup>3</sup>. In the current work, we address the SPARQL queries with ‘SELECT/WHERE’ option, where the predicate is always instantiated as an *IRI* (Figure 2a). The SELECT clause identifies the variables to appear in the query results while the WHERE clause provides triple patterns to match against the RDF data.

### 2.2.1 Query Multigraph Representation

In any valid SPARQL query (as in Figure 2a), every triplet has at least one unknown variable  $?X$ , whose bindings are to be found in the RDF data. It should now be easy to observe that a SPARQL query can be represented in the form of a graph as in Figure 2b, which in turn is transformed into query multigraph  $Q$  (as in Figure 2c).

In the query multigraph representation, each unknown variable  $?X_i$  is mapped to a vertex  $u_i$  that forms the vertex set  $U$  component of the query multigraph  $Q$  (e.g.,  $?X_6$  is mapped to  $u_6$ ). Since a predicate is always instantiated as an *IRI*, we use the edge-type dictionary in Table 2b, to map the predicate to an edge-type identifier  $t_i \in T$  (e.g., ‘isMarriedTo’ is mapped as  $t_8$ ). When an object  $o_i$  is a literal, we use the attribute dictionary (Table 2c), to find the attribute identifier  $a_i$  for the predicate-object tuple  $\langle p_i, o_i \rangle$  (e.g.,  $\{a_0\}$  forms the attribute for vertex  $u_4$ ). Further, when a subject or an object is an *IRI*, which is a not a variable, we use the vertex dictionary (2a), to map it to an *IRI*-vertex  $u_i^{iri}$  (e.g., ‘x:United.States’ is mapped to  $u_0^{iri}$ ) and maintain a set of *IRI* vertices  $R$ . Since this vertex is not a variable and a real vertex of the query, we portray it differently by a shaded square shaped vertex. When a query vertex  $u_i$  does

<sup>3</sup><http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>

$s/o$	$\mathcal{M}_v(s/o)$	$p$	$\mathcal{M}_e(p)$
x:Music_Band	$v_0$	y:isPartOf	$t_0$
x:Amy_Winehouse	$v_1$	y:hasCapital	$t_1$
x:London	$v_2$	y:hasStadium	$t_2$
x:England	$v_3$	y:livedIn	$t_3$
x:WembleyStadium	$v_4$	y:diedIn	$t_4$
x:United States	$v_5$	y:wasBornIn	$t_5$
x:Blake Fielder-Civil	$v_6$	y:wasFormedIn	$t_6$
x:Christopher_Nolan	$v_7$	y:wasPartOf	$t_7$
x:Dark_Knight_Triology	$v_8$	y:wasMarriedTo	$t_8$

(a) Vertex Dictionary

(b) Edge-type Dictionary

$\langle p, o \rangle$	$\mathcal{M}_a(\langle p, o \rangle)$
$\langle y:\text{hasCapacityOf}, "90000" \rangle$	$a_0$
$\langle y:\text{wasFoundedIn}, "1994" \rangle$	$a_1$
$\langle y:\text{hasName}, "MCA\_Band" \rangle$	$a_2$

(c) Attribute Dictionary

Table 2: Dictionary look-up tables for vertices, edge-types and vertex attributes

not have any vertex attributes associated with it (e.g.,  $u_0, u_1, u_2, u_3, u_6$ ), a null attribute  $\{-\}$  is assigned to it. On the other hand, an *IRI*-vertex  $u_i^{iri} \in R$  does not have any attributes. Thus, a SPARQL query is transformed into a query multigraph  $Q$ .

In this work, we always use the notation  $V$  for the set of vertices of  $G$ , and  $U$  for the set of vertices of  $Q$ . Consequently, a data vertex  $v \in V$ , and a query vertex  $u \in U$ . Also, an incoming edge to a vertex is positive (default), and an outgoing edge from a vertex is labelled negative ( $'-$ ).

## 2.3 SPARQL Querying by Adopting Multigraph Homomorphism

As we recall, the problem of SPARQL querying is addressed by finding the solutions to the unknown variables  $?X$ , that can be bound with the RDF data entities, so that the relations (predicates) provided in the SPARQL query are respected. In this work, to harness the transformed data multigraph  $G$  and the query multigraph  $Q$ , we reduce the problem of SPARQL querying to a sub-multigraph homomorphism problem. The RDF data is transformed into data multigraph  $G$  and the SPARQL query is transformed into query multigraph  $Q$ . Let us now recall that finding SPARQL answers in the RDF data is equivalent to finding all the sub-multigraphs of  $Q$  in  $G$  that are homomorphic. Thus, let us now formally introduce homomorphism for a vertex attributed, directed multigraph.

**DEFINITION 2.** Sub-multigraph Homomorphism. Given a query multigraph  $Q = (U, E^Q, L_U, L_E^Q)$  and a data multigraph  $G = (V, E, L_V, L_E)$ , the sub-multigraph homomorphism from  $Q$  to  $G$  is a surjective function  $\psi : U \rightarrow V$  such that:

- $\forall u \in U, L_U(u) \subseteq L_V(\psi(u))$
- $\forall (u_m, u_n) \in E^Q, \exists (\psi(u_m), \psi(u_n)) \in E$ , where  $(u_m, u_n)$  is a directed edge, and  $L_E^Q(u_m, u_n) \subseteq L_E(\psi(u_m), \psi(u_n))$ .

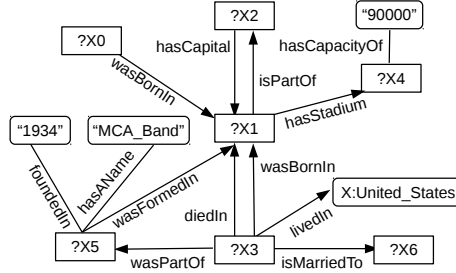
Thus, by finding all the sub-multigraphs in  $G$  that are homomorphic to  $Q$ , we enumerate all possible homomorphic embeddings of  $Q$  in  $G$ . These embeddings contain the solution for each of the query vertex that is an unknown variable. Thus, by using the inverse mapping function  $\mathcal{M}_v^{-1}(v_i)$  (introduced already), we find the bindings for the SPARQL

```

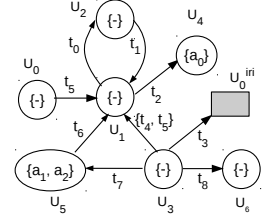
SELECT ?X0?X1 ?X2 ?X3 ?X4 ?X5 ?X6 WHERE {
?X0 y:livedIn ?X1.
?X1 y:isPartOf ?X2.
?X2 y:hasCapital ?X1.
?X1 y:hasStadium ?X4.
?X3 y:wasBornIn ?X1.
?X3 y:diedIn ?X1.
?X3 y:isMarriedTo ?X6.
?X3 y:wasPartOf ?X5.
?X5 y:wasFormedIn ?X1.
?X4 y:hasCapacity ?X4.
?X5 y:hasName "MCA_Band".
?X5 y:foundedIn "1934".
?X3 y:livedIn x:United States . }

```

(a) SPARQL Query



(b) Graph representation of SPARQL

(c) Equivalent Multigraph  $Q$ Figure 2: (a) SPARQL query representation; (b) graph representation (c) attributed multigraph  $Q$ 

query. The decision problem of subgraph homomorphism is NP-complete. This standard subgraph homomorphism problem can be seen as a particular case of sub-multigraph homomorphism, where both the labelling functions  $L_E$  and  $L_E^Q$  always return the same subset of edge-types for all the edges in both  $Q$  and  $G$ . Thus the problem of sub-multigraph homomorphism is at least as hard as subgraph homomorphism. Further, the subgraph homomorphism problem is a generic scenario of subgraph isomorphism problem where, the injectivity constraints are slackened [11].

### 3. AMBER: A SPARQL QUERYING ENGINE

We now present an overview of our proposal - AMBER (Attributed Multigraph Based Engine for RDF querying). AMBER contains two different stages: (i) an offline stage during which, RDF data is transformed into multigraph  $G$  and then a set of index structures  $\mathcal{I}$  is constructed that captures the necessary information contained in  $G$ ; (ii) an online stage during which, a SPARQL query is transformed into a multigraph  $Q$ , and then by exploiting the subgraph matching techniques along with the already built index structures  $\mathcal{I}$ , the homomorphic matches of  $Q$  in  $G$  are obtained.

Given a multigraph representation  $Q$  of a SPARQL query, AMBER decomposes the query vertices  $U$  into a set of core vertices  $U_c$  and satellite vertices  $U_s$ . Intuitively, a vertex  $u \in U$  is a core vertex, if the degree of the vertex is more than one; on the other hand, a vertex  $u$  with degree one is a satellite vertex. For example, in Figure 2c,  $U_c = \{u_1, u_3, u_5\}$  and  $U_s = \{u_0, u_2, u_4, u_6\}$ . Once decomposed, we run the sub-multigraph matching procedure on the query structure spanned only by the core vertices. However, during the procedure, we also process the satellite vertices (if available) that are connected to a core vertex that is being processed. For example, while processing the core vertex  $u_1$ , we also process the set of satellite vertices  $\{u_0, u_2, u_4\}$  connected to it; whereas, the core vertex  $u_5$  has no satellite vertices to be processed. In this way, as the matching proceeds, the entire structure of the query multigraph  $Q$  is processed to find the homomorphic embeddings in  $G$ . The set of indexing structures  $\mathcal{I}$  are extensively used during the process of sub-multigraph matching. The homomorphic embeddings are finally translated back to the RDF entities using the inverse mapping function  $\mathcal{M}_v^{-1}$  as discussed in Section 2.

## 4. INDEX CONSTRUCTION

Given a data multigraph  $G$ , we build the following three different indices: (i) an inverted list  $\mathcal{A}$  for storing the set of data vertex for each attribute in  $a_i \in A$  (ii) a trie index structure  $\mathcal{S}$  to store features of all the data vertices  $V$  (iii) a set of trie index structures  $\mathcal{N}$  to store the neighbourhood information of each data vertex  $v \in V$ . For brevity of representation, we ensemble all the three index structures into  $\mathcal{I} := \{\mathcal{A}, \mathcal{S}, \mathcal{N}\}$ .

During the query matching procedure (the online step), we access these indexing structures to obtain the candidate solutions for a query vertex  $u$ . Formally, for a query vertex  $u$ , the candidate solutions are a set of data vertices  $C_u = \{v | v \in V\}$  obtained by accessing  $\mathcal{A}$  or  $\mathcal{S}$  or  $\mathcal{N}$ , denoted as  $C_u^A$ ,  $C_u^S$  and  $C_u^N$  respectively.

### 4.1 Attribute Index

The set of vertex attributes is given by  $A = \{a_0, \dots, a_n\}$  (Section 2), where a data vertex  $v \in V$  might have a subset of  $A$  assigned to it. We now build the vertex attribute index  $\mathcal{A}$  by creating an inverted list where a particular attribute  $a_i$  has the list of all the data vertices in which it appears.

Given a query vertex  $u$  with a set of vertex attributes  $u.A \subseteq A$ , for each attribute  $a_i \in u.A$ , we access the index structure  $\mathcal{A}$  to fetch a set of data vertices that have  $a_i$ . Then we find a common set of data vertices that have the entire attribute set  $u.A$ . For example, considering the query vertex  $u_5$  (Fig. 2c), it has an attribute set  $\{a_1, a_2\}$ . The candidate solutions for  $u_5$  are obtained by finding all the common data vertices, in  $\mathcal{A}$ , between  $a_1$  and  $a_2$ , resulting in  $C_{u_5}^A = \{v_0\}$ .

### 4.2 Vertex Signature Index

The index  $\mathcal{S}$  captures the edge type information from the data vertices. For a lucid understanding of this indexing schema we formally introduce the notion of vertex signature that is defined for a vertex  $v \in V$ , which encapsulates the edge information associated with it.

**DEFINITION 3.** Vertex signature. For a vertex  $v \in V$ , the vertex signature  $\sigma_v$  is a multiset containing all the directed multi-edges that are incident on  $v$ , where a multi-edge between  $v$  and a neighbouring vertex  $v'$  is represented by a set that corresponds to the edge types. Formally,  $\sigma_v = \bigcup_{v' \in N(v)} L_E(v, v')$  where  $N(v)$  is the set of neighbourhood vertices of  $v$ , and  $\cup$  is the union operator for multiset.

Data vertex	Signature	Synopsis							
		$f_1^+$	$f_2^+$	$f_3^+$	$f_4^+$	$f_1^-$	$f_2^-$	$f_3^-$	$f_4^-$
$v$	$\sigma_v$								
$v_0$	$\{\{-t_6\}, \{t_7\}\}$	1	1	-7	7	1	1	-6	6
$v_1$	$\{\{-t_3\}, \{-t_7\}, \{-t_8\}, \{-t_4, -t_5\}\}$	0	0	0	0	2	5	-3	8
$v_2$	$\{\{-t_0\}, \{t_1\}, \{-t_2\}, \{t_5\}, \{t_6\}, \{t_4, t_5\}\}$	2	4	-1	6	1	2	0	2
$v_3$	$\{\{t_0\}, \{t_3\}, \{-t_1\}\}$	1	2	0	3	1	1	-1	1
$v_4$	$\{\{t_2\}\}$	1	1	-2	2	0	0	0	0
$v_5$	$\{\{t_3\}, \{t_3\}\}$	1	1	-3	3	0	0	0	0
$v_6$	$\{\{t_8\}, \{-t_3\}\}$	1	1	-8	8	1	1	-3	3
$v_7$	$\{\{-t_0\}, \{-t_3\}, \{-t_5\}\}$	0	0	0	0	1	3	0	5
$v_8$	$\{\{t_0\}\}$	1	1	0	0	0	0	0	0

Table 3: Vertex signatures and the corresponding synopses for the vertices in the data multigraph  $G$  (Figure 1c)

The index  $\mathcal{S}$  is constructed by tailoring the information supplied by the vertex signature of each vertex in  $G$ . To extract some interesting features, let us observe the vertex signature  $\sigma_{v_2}$  as supplied in Table 3. To begin with, we can represent the vertex signature  $\sigma_{v_2}$  separately for the incoming and outgoing multi-edges as  $\sigma_{v_2}^+ = \{\{t_1\}, \{t_5\}, \{t_6\}, \{t_4, t_5\}\}$  and  $\sigma_{v_2}^- = \{\{-t_0\}\{-t_2\}\}$  respectively. Now we observe that  $\sigma_{v_2}^+$  has four distinct multi-edges and  $\sigma_{v_2}^-$  has two distinct multi-edges. Now, lets think that we want find candidate solutions for a query vertex  $u$ . The data vertex  $v_2$  can be a match for  $u$  only if the signature of  $u$  has at most four incoming ('+') edges and at most two outgoing ('-') edges; else  $v_2$  can not be a match for  $u$ . Thus, more such features (e.g., maximum cardinality of a set in the vertex signature) can be proposed to filter out irrelevant candidate vertices. Thus, for each vertex  $v$ , we propose to extract a set of features by exploiting the corresponding vertex signature. These features constitute a *synopses*, which is a surrogate representation that approximately captures the vertex signature information.

The synopsis of a vertex  $v$  contains a set of features  $F$ , whose values are computed from the vertex signature  $\sigma_v$ . In this background, we propose four distinct features:  $f_1$  - the maximum cardinality of a set in the vertex signature;  $f_2$  - the number of unique dimensions in the vertex signature;  $f_3$  - the minimum index value of the edge type;  $f_4$  - the maximum index value of the edge type. For  $f_3$  and  $f_4$ , the index values of edge type are nothing but the position of the sequenced alphabet. These four basic features are replicated separately for outgoing (negative) and incoming (positive) edges, as seen in Table 3. Thus for the vertex  $v_2$ , we obtain  $f_1^+ = 2$ ,  $f_2^+ = 4$ ,  $f_3^+ = -1$  and  $f_4^+ = 7$  for the incoming edge set and  $f_1^- = 1$ ,  $f_2^- = 2$ ,  $f_3^- = 0$  and  $f_4^- = 2$  for the outgoing edge set. Synopses for the entire vertex set  $V$  for the data multigraph  $G$  are depicted in Table 3.

Once the synopses are computed for all data vertices, an R-tree is constructed to store all the synopses. This R-tree constitutes the vertex signature index  $\mathcal{S}$ . A synopsis with  $|F|$  fields forms a leaf in the R-tree.

When a set of possible candidate solutions are to be obtained for a query vertex  $u$ , we create a vertex signature  $\sigma_u$  in order to compute the synopsis, and then obtain the possible solutions from the R-tree structure.

The general idea of using an R-tree is as follows. A synopsis  $F$  of a data vertex spans an axes-parallel rectangle in an  $|F|$ -dimensional space, where the maximum co-ordinates of the rectangle are the values of the synopses

fields  $(f_1, \dots, f_{|F|})$ , and the minimum co-ordinates are the origin of the rectangle (filled with zero values). For example, a data vertex represented by a synopses with two features  $F(v) = [2, 3]$  spans a rectangle in a 2-dimensional space in the interval range  $([0, 2], [0, 3])$ . Now, if we consider synopses of two query vertices,  $F(u_1) = [1, 3]$  and  $F(u_2) = [1, 4]$ , we observe that the rectangle spanned by  $F(u_1)$  is wholly contained in the rectangle spanned by  $F(v)$  but  $F(u_2)$  is not wholly contained in  $F(v)$ . Thus,  $u_1$  is a candidate match while  $u_2$  is not.

LEMMA 1. *Querying the vertex signature index  $\mathcal{S}$  constructed with synopses, guarantees to output at least the entire set of candidate solutions.*

PROOF. Consider the field  $f_1^\pm$  in the synopses that represents the maximum cardinality of the neighbourhood signature. Let  $\sigma_u$  be the signature of the query vertex  $u$  and  $\{\sigma_{v_1}, \dots, \sigma_{v_n}\}$  be the set of signatures on the data vertices. By using  $f_1$  we need to show that  $C_u^S$  has at least all the valid candidate matches. Since we are looking for a superset of query vertex signature, and we are checking the condition  $f_1^\pm(u) \leq f_1^\pm(v_i)$ , where  $v_i \in V$ , a vertex  $v_i$  is pruned if it does not match the inequality criterion since, it can never be an eligible candidate. This analogy can be extended to the entire synopses, since it can be applied disjunctively.  $\square$

Formally, the candidates solutions for a vertex  $u$  can be written as  $C_u^S = \{v | \forall_{i \in [1, \dots, |F|]} f_i^\pm(u) \leq f_i^\pm(v)\}$ , where the constraints are met for all the  $|F|$ -dimensions. Since we apply the same inequality constraint to all the fields, we negate the fields that refer to the minimal index value of the edge type ( $f_3^+$  and  $f_3^-$ ) so that the rectangular containment problem still holds good. Further to respect the rectangular containment, we populate the synopses fields with '0' values, in case, the signature does not have either positive or negative edges in it, as seen for  $v_1, v_3, v_4, v_5$  and  $v_7$ .

For example, if we want to compute the possible candidates for a query vertex  $u_0$  in Figure 2c, whose signature is  $\sigma_{u_0} = \{-t_5\}$ , we compute the synopsis which is  $[0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 5 \ 5]$ . Now we look for all those vertices that subsume this synopsis in the R-tree, whose elements are depicted in Table 3, which gives us the candidate solutions  $C_{u_0}^S = \{v_1, v_7\}$ , thus pruning the rest of the vertices.

The  $\mathcal{S}$  index helps to prune the vertices that do not respect the edge type constraints. This is crucial since this pruning is performed for the initial query vertex, and hence many candidates are cast away, thereby avoiding unnecessary recursion during the matching procedure. For example,

for the initial query vertex  $u_0$ , whose candidate solutions are  $\{v_1, v_7\}$ , the recursion branch is run only on these two starting vertices instead of the entire vertex set  $V$ .

### 4.3 Vertex Neighbourhood Index

The *vertex neighbourhood index*  $\mathcal{N}$  captures the topological structure of the data multigraph  $G$ . The index  $\mathcal{N}$  comprises of 1-neighbourhood trees built for each data vertex  $v \in V$ . Since  $G$  is a directed multigraph, and each vertex  $v \in V$  can have both the incoming and outgoing edges, we construct two separate index structures  $\mathcal{N}^+$  and  $\mathcal{N}^-$  for incoming and outgoing edges respectively, that constitute the structure  $\mathcal{N}$ .

To understand the index structure, let us consider the data vertex  $v_2$  from Figure 1c, shown separately in Figure 3a. For this vertex  $v_2$ , we collect all the neighbourhood information (vertices and multi-edges), and represent this information by a tree structure, built separately for incoming ('+') and outgoing ('-') edges. Thus, the tree representation of a vertex  $v$  contains the neighbourhood vertices and the corresponding multi-edges, as shown in Figure 3b, where the vertices of the tree structure are represented by the edge types.

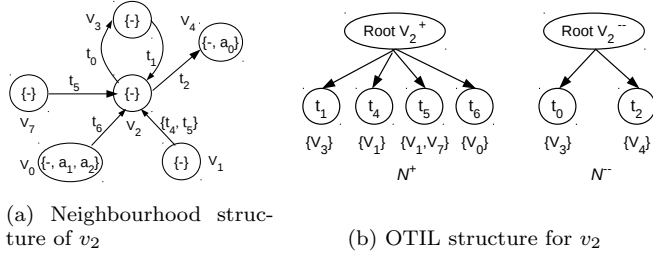
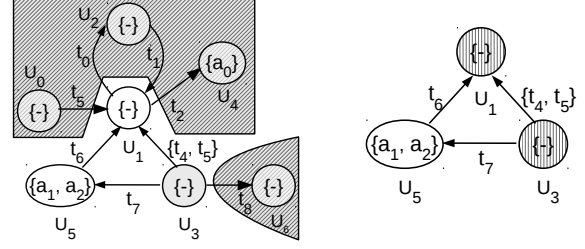


Figure 3: Building Neighbourhood Index for data vertex  $v_2$

In order to construct an efficient tree structure, we take inspiration from [14] to propose the structure - Ordered Trie with Inverted List (OTIL). To construct the OTIL index as shown in Figure 3b, we insert each ordered multi-edge that is incident on  $v$  at the root of the trie. Consider a data vertex  $v_i$ , with a set of  $n$  neighbourhood vertices  $N(v_i)$ . Now, for every pair of incoming edge  $(v_i, N^j(v_i))$ , where  $j \in \{1, \dots, n\}$ , there exists a multi-edge  $\{t_i, \dots, t_j\}$ , which is inserted into the OTIL structure  $\mathcal{N}^+$ . Similarly for every pair of outgoing edge  $(N^j(v_i), v_i)$ , there exists a multi-edge  $\{t_m, \dots, t_n\}$ , which is inserted into the OTIL structure  $\mathcal{N}^-$  maintaining two OTIL structures that constitute  $\mathcal{N}$ . Each multi-edge is ordered (w.r.t. increasing edge type indexes), before inserting into the respective OTIL structure, and the order is universally maintained for all data vertices. Further, for every edge type  $t_i$ , we maintain a *list* that contains all the neighbourhood vertices  $N^+(v_i)/N^-(v_i)$ , that have the edge type  $t_i$  incident on them.

To understand the utility of  $\mathcal{N}$ , let us consider an illustrative example. Considering the query multigraph  $Q$  in Figure 2c, let us assume that we want to find the matches for the query vertices  $u_1$  and  $u_0$  in order. Thus, for the initial vertex  $u_1$ , let us say, we have found the set of candidate solutions which is  $\{v_2\}$ . Now, to find the candidate solutions for the next query vertex  $u_0$ , it is important to maintain the structure spanned by the query vertices, and this is where the indexing structure  $\mathcal{N}$  is accessed. Thus to retain the structure of the query multigraph (in this case, the struc-



(a) Query graph  $Q$  highlighted with satellite vertices (b) Query graph spanned by core vertices

Figure 4: Decomposing the query multigraph into *core* and *satellite* vertices

ture between  $u_1$  and  $u_0$ ), we have to find the data vertices that are in the neighbourhood of already matched vertex  $v_2$  (a match for vertex  $u_1$ ), that has the same structure (edge types) between  $u_1$  and  $u_0$  in the query graph. Thus to fetch all the data vertices that have the edge type  $t_5$ , which is directed towards  $v_2$  and hence '+', we access the neighbourhood index trie  $\mathcal{N}^+$  for vertex  $v_2$ , as shown in Figure 3. This gives us a set of candidate solutions  $C_{u_0}^{\mathcal{N}} = \{v_1, v_7\}$ . It is easy to observe that, by maintaining two separate indexing structures  $\mathcal{N}^+$  and  $\mathcal{N}^-$ , for both incoming and outgoing edges, we can reduce the time to fetch the candidate solutions.

Thus, in a generic scenario, given an already matched data vertex  $v$ , the edge direction '+' or '-', and the set of edge types  $T' \subseteq T$ , the index  $\mathcal{N}$  will find a set of neighbourhood data vertices  $\{v' | (v', v) \in E \wedge T' \subseteq L_E(v', v)\}$  if the edge direction is '+' (incoming), while  $\mathcal{N}$  returns  $\{v' | (v, v') \in E \wedge T' \subseteq L_E(v, v')\}$  if the edge direction is '-' (outgoing).

## 5. QUERY MATCHING PROCEDURE

In order to follow the working of the proposed query matching procedure, we formalize the notion of *core* and *satellite* vertices. Given a query graph  $Q$ , we decompose the set of query vertices  $U$  into a set of *core* vertices  $U_c$  and a set of *satellite* vertices  $U_s$ . Formally, when the degree of the query graph  $\Delta(Q) > 1$ ,  $U_c = \{u | u \in U \wedge deg(u) > 1\}$ ; however, when  $\Delta(Q) = 1$ , i.e, when the query graph is either a vertex or a multiedge, we choose one query vertex at random as a *core* vertex, and hence  $|U_c| = 1$ . The remaining vertices are classified as satellite vertices, whose degree is always 1. Formally,  $U_s = \{U \setminus U_c\}$ , where for every  $u \in U_s$ ,  $deg(u) = 1$ . The decomposition for the query multigraph  $Q$  is depicted in Figure 4, where the satellite vertices are separated (vertices under the shaded region in Fig. 4a), in order to obtain the query graph that is spanned only by the core vertices (Fig. 4b).

The proposed AMBER-Algo (Algorithm 3) performs recursive sub-multigraph matching procedure only on the query structure spanned by  $U_c$  as seen in Figure 4b. Since the entire set of satellite vertices  $U_s$  is connected to the query structure spanned by the core vertices, AMBER-Algo processes the satellite vertices while performing sub-multigraph matching on the set of core vertices. Thus during the recursion, if the current *core* vertex has *satellite* vertices connected to it, the algorithm retrieves directly a list of possible matching for such *satellite* vertices and it includes them in

the current partial solution. Each time the algorithm executes a recursion branch with a solution, the solution not only contains a data vertex match  $v_c$  for each query vertex belonging to  $U_c$ , but also a set of matched data vertices  $V_s$  for each query vertex belonging to  $U_s$ . Each time a solution is found, we can generate not only one, but a set of embeddings through the Cartesian product of the matched elements in the solution.

Since finding SPARQL solutions is equivalent to finding homomorphic embeddings of the query multigraph, the homomorphic matching allows different query vertices to be matched with the same data vertices. Recall that there is no injectivity constraint in sub-multigraph homomorphism as opposed to sub-multigraph isomorphism [11]. Thus during the recursive matching procedure, we do not have to check if the potential data vertex has already been matched with previously matched query vertices. This is an advantage when we are processing satellite vertices: we can find matches for each satellite vertex independently without the necessity to check for a repeated data vertex.

Before getting into the details of the AMBER-Algo, we first explain how a set of candidate solutions is obtained when there is information associated only with the vertices. Then we explain how a set of candidate solutions is obtained when we encounter the satellite vertices.

## 5.1 Vertex Level Processing

To understand the generic query processing, it is necessary to understand the matching process at vertex level. Whenever a query vertex  $u \in U$  is being processed, we need to check if  $u$  has a set of attributes  $A$  associated with it or any IRIs are connected to it (recall Section 2.2).

---

### Algorithm 1: PROCESSVERTEX( $u, Q, \mathcal{A}, \mathcal{N}$ )

---

```

1 if  $u.A \neq \emptyset$  then
2    $C_u^A = \text{QUERYATTINDEX}(\mathcal{A}, u.A)$ 
3 if  $u.R \neq \emptyset$  then
4    $C_u^I = \bigcap_{u_i^{iri} \in u.R} (\text{QUERYNEIGHINDEX}(\mathcal{N}, L_E^Q(u, u_i^{iri}), u_i^{iri}))$ 
5  $CandAtt_u = C_u^A \bar{\cap} C_u^I$  /* Find common candidates */
6 return  $CandAtt_u$ 

```

---

To process an arbitrary query vertex, we propose a procedure PROCESSVERTEX, depicted in Algorithm 1. This algorithm is invoked only when a vertex  $u$  has at least, either a set of vertex attributes or any IRI associated with it. The PROCESSVERTEX procedure returns a set of data vertices  $CandAtt_u$ , which are matchable with  $u$ ; in case  $CandAtt_u$  is empty, then the query vertex  $u$  has no matches in  $V$ .

As seen in Lines 1-2, when a query vertex  $u$  has a set of vertex attributes i.e.,  $u.A \neq \emptyset$ , we obtain the candidate solutions  $C_u^A$  by invoking QUERYATTINDEX procedure, that accesses the index  $\mathcal{A}$  as explained in Section 4.1. For example, the query vertex  $u_5$  with vertex attributes  $\{a_1, a_2\}$ , can only be matched with the data vertex  $v_0$ ; thus  $C_{u_5}^A = \{v_0\}$ .

When a query vertex  $u$  has IRIs associated with it, i.e.,  $u.R \neq \emptyset$  (Lines 3-4), we find the candidate solutions  $C_u^I$  by invoking the QUERYNEIGHINDEX procedure. As we recall from Section 2.2, a vertex  $u$  is connected to an IRI vertex  $u_i^{iri}$  through a multi-edge  $L_E^Q(u, u_i^{iri})$ . An IRI vertex  $u_i^{iri}$  always has only one data vertex  $v$ , that can match. Thus, the candidate solutions  $C_u^I$  are obtained by invoking the QUERYNEIGHINDEX procedure, that fetches all the neigh-

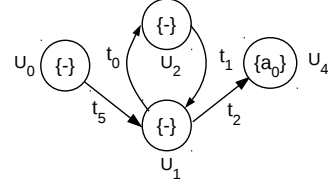


Figure 5: A star structure in the query multigraph  $Q$

bourhood vertices of  $v$  that respect the multi-edge  $L_E^Q(u, u_i^{iri})$ . The procedure is invoked until all the IRI vertices  $u.R$  are processed (Line 4). Considering the example in Figure 2c,  $u_3$  is connected to an IRI-vertex  $u_0^{iri}$ , which has a unique data vertex match  $v_5$ , through the multi-edge  $\{-t_3\}$ . Using the neighbourhood index  $\mathcal{N}$ , we look for the neighbourhood vertices of  $v_5$ , that have the multi-edge  $\{-t_3\}$ , which gives us the candidate solutions  $C_{u_3}^I = \{v_1\}$ .

Finally in Line 5, the merge operator  $\bar{\cap}$  returns a set of common candidates  $CandAtt_u$ , only if  $u.A \neq \emptyset$  and  $u.R \neq \emptyset$ . Otherwise,  $C_u^A$  or  $C_u^I$  are returned as  $CandAtt_u$ .

## 5.2 Processing Satellite Vertices

In this section, we provide insights on processing a set of satellite vertices  $U_{sat} \subseteq U_s$  that are connected to a core vertex  $u_c \in U_c$ . This scenario results in a structure that appears frequently in SPARQL queries called star structure [8, 10].

A typical star structure depicted in Figure 5, has a core vertex  $u_c = u_1$ , and a set of satellite vertices  $U_{sat} = \{u_0, u_2, u_4\}$  connected to the core vertex. For each candidate solution of the core vertex  $u_1$ , we process  $u_0, u_2, u_4$  independently of each other, since there is no structural connectivity (edges) among them, although they are only structurally connected to the core vertex  $u_1$ .

LEMMA 2. For a given star structure in a query graph, each satellite vertex can be independently processed if a candidate solution is provided for the core vertex  $u_c$ .

PROOF. Consider a core vertex  $u_c$  that is connected to a set of satellite vertices  $U_{sat} = \{u_0, \dots, u_s\}$ , through a set of edge-types  $T' = \{t_0, \dots, t_s\}$ . Let us assume  $v_c$  is a candidate solution for the core vertex  $u_c$ , and we want to find candidate solutions for  $u_i \in U_{sat}$  and  $u_j \in U_{sat}$ , where  $i \neq j$ . Now, the candidate solutions for  $u_i$  and  $u_j$  can be obtained by fetching the neighbourhoods of already matched vertex  $v_c$  that respect the edge-type  $t_i \in T'$  and  $t_j \in T'$  respectively. Since two satellite vertices  $u_i$  and  $u_j$  are never connected to each other, the candidate solutions of  $u_i$  are independent of that of  $u_j$ . This analogy applies to all the satellite vertices.  $\square$

Given a core vertex  $u_c$ , we initially find a set of candidate solutions  $Cand_{u_c}$ , by using the index  $\mathcal{S}$ . Then, for each candidate solution  $v_c \in Cand_{u_c}$ , the set of solutions for all the satellite vertices  $U_{sat}$  that are connected to  $u_c$  are returned by the MATCHSATVERTICES procedure, described in Algorithm 2. The set of solution tuple  $M_{sat}$  defined in Line 1, stores the candidate solutions for the entire set of satellite vertices  $U_{sat}$ . Formally,  $M_{sat} = \{[u_s, V_s]\}_{s=1}^{|U_{sat}|}$ , where  $u_s \in U_{sat}$  and  $V_s$  is a set of candidate solutions for  $u_s$ . In order to obtain candidate solutions for  $u_s$ , we query the

**Algorithm 2: MATCHSATVERTICES( $\mathcal{A}, \mathcal{N}, Q, U_{sat}, v_c$ )**


---

```

1 SET:  $M_{sat} = \emptyset$ , where  $M_{sat} = \{[u_s, V_s]\}_{s=1}^{|U_{sat}|}$ 
2 for all  $u_s \in U_{sat}$  do
3    $Cand_{u_s} = \text{QUERYNEIGHINDEX}(\mathcal{N}, L_E^Q(u_c, u_s), v_c)$ 
4    $Cand_{u_s} = Cand_{u_s} \cap \text{PROCESSVERTEX}(u_s, Q, \mathcal{A}, \mathcal{N})$ 
5   if  $Cand_{u_s} \neq \emptyset$  then
6      $M_{sat} = M_{sat} \cup (u_s, Cand_{u_s})$  /* Satellite solutions */
7   else
8     return  $M_{sat} := 0$  /* No solutions possible */
9 return  $M_{sat}$  /* Matches for satellite vertices */

```

---

neighbourhood index  $\mathcal{N}$  (Line 3); the QUERYNEIGHINDEX function obtains all the neighbourhood vertices of already matched  $v_c$ , that also considers the multi-edge in the query multigraph  $L_E^Q(u_c, u_s)$ . As every query vertex  $u_s \in U_{sat}$  is processed, the solution set  $M_{sat}$  that contains candidate solutions grows until all the satellite vertices have been processed (Lines 2-8).

In Line 4, the set of candidate solutions  $Cand_{u_s}$  are refined by invoking Algorithm 1 (VERTEXPROCESSING). After the refinement, if there are finite candidate solutions, we update the solution  $M_{sat}$ ; else, we terminate the procedure as there can be no matches for a given matched vertex  $v_c$ . The MATCHSATVERTICES procedure performs two tasks: firstly, it checks if the candidate vertex  $v_c \in Cand_{u_s}$  is a valid matchable vertex and secondly, it obtains the solutions for all the satellite vertices.

### 5.3 Arbitrary Query Processing

Algorithm 3 shows the generic procedure we develop to process arbitrary queries.

Recall that for an arbitrary query  $Q$ , we define two different types of vertexes: a set of core vertices  $U_c$  and a set of satellite vertices  $U_s$ . The QUERYDECOMPOSE procedure in Line 1 of Algorithm 3, performs this decomposition by splitting the query vertices  $U$  into  $U_c$  and  $U_s$ , as observed in Figure 4.

To process arbitrary query multigraphs, we perform recursive sub-multigraph matching procedure on the set of core vertices  $U_c \subseteq U$ ; during the recursion, satellite vertexes connected to a specific core vertex are processed too. Since the recursion is performed on the set of core vertices, we propose a few heuristics for ordering the query vertices.

Ordering of the query vertices forms one of the vital steps for subgraph matching algorithms [11]. In any subgraph matching algorithm, the embeddings of a query subgraph are obtained by exploring the solution space spanned by the data graph. But since the solution space itself can grow exponentially in size, we are compelled to use intelligent strategies to traverse the solution space. In order to achieve this, we propose a heuristic procedure VERTEXORDERING (Line 2, Algorithm 3) that employs two ranking functions.

The first ranking function  $r_1$  relies on the number of satellite vertices connected to the core vertex, and the query vertices are ordered with the decreasing rank value. Formally,  $r_1(u) = |U_{sat}|$ , where  $U_{sat} = \{u_s | u_s \in U_s \wedge (u, u_s) \in E(Q)\}$ . A vertex with more satellite vertices connected to it, is rich in structure and hence it would probably yield fewer candidate solutions to be processed under recursion. Thus, in Figure 4,  $u_1$  is chosen as an initial vertex. The second ranking function  $r_2$  relies on the number of incident edges on a query vertex. Formally,  $r_2(u) = \sum_{j=1}^m |\sigma(u)^j|$ , where  $u$

has  $m$  multiedges and  $|\sigma(u)^j|$  captures the number of edge types in the  $j^{th}$  multiedge. Again,  $U_c^{ord}$  contains the ordered vertices with the decreasing rank value  $r_2$ . Further, when there are no satellite vertices in the query  $Q$ , this ranking function gets the priority. Despite the usage of any ranking function, the query vertices in  $U_c^{ord}$ , when accessed in sequence, should be structurally connected to the previous set of vertices. If two vertices tie up with the same rank, the rank with lesser priority determines which vertex wins. Thus, for the example in Figure 4, the set of ordered core vertices is  $U_c^{ord} = \{u_1, u_3, u_5\}$ .

**Algorithm 3: AMBER-Algo ( $\mathcal{I}, Q$ )**


---

```

1 QUERYDECOMPOSE: Split  $U$  into  $U_c$  and  $U_s$ 
2  $U_c^{ord} = \text{VERTEXORDERING}(Q, U_c)$ 
3  $u_{init} = u | u \in U_c^{ord}$ 
4  $CandInit = \text{QUERYSYNINDEX}(u_{init}, \mathcal{S})$ 
5  $CandInit = CandInit \cap \text{PROCESSVERTEX}(u_{init}, Q, \mathcal{A}, \mathcal{N})$ 
6 FETCH:  $U_{init}^{sat} = \{u | u \in U_s \wedge (u_{init}, u) \in E(Q)\}$ 
7 SET:  $Emb = \emptyset$ 
8 for  $v_{init} \in CandInit$  do
9   SET:  $M = \emptyset, M_s = \emptyset, M_c = \emptyset$ 
10  if  $U_{init}^{sat} \neq \emptyset$  then
11     $M_{sat} = \text{MATCHSATVERTICES}(\mathcal{A}, \mathcal{N}, Q, U_{init}^{sat}, v_{init})$ 
12    if  $M_{sat} \neq \emptyset$  then
13      for  $[u_s, V_s] \in M_{sat}$  do
14        UPDATE:  $M_s = M_s \cup [u_s, V_s]$ 
15        UPDATE:  $M_c = M_c \cup [u_{init}, v_{init}]$ 
16         $Emb = Emb \cup \text{HOMOMORPHICMATCH}(M, \mathcal{I}, Q, U_c^{ord})$ 
17  else
18    UPDATE:  $M_c = M_c \cup (u_{init}, v_{init})$ 
19     $Emb = Emb \cup \text{HOMOMORPHICMATCH}(M, \mathcal{I}, Q, U_c^{ord})$ 
20 return  $Emb$  /* Homomorphic embeddings of query multigraph */

```

---

The first vertex in the set  $U_c^{ord}$  is chosen as the initial vertex  $u_{init}$  (Line 3), and subsequent query vertices are chosen in sequence. The candidate solutions for the initial query vertex  $CandInit$  are returned by QUERYSYNINDEX procedure (Line 4), that are constrained by the structural properties (neighbourhood structure) of  $u_{init}$ . By querying the index  $\mathcal{S}$  for initial query vertex  $u_{init}$ , we obtain the candidate solutions  $CandInit \in V$  that match the structure (multiedge types) associated with  $u_{init}$ . Although some candidates in  $CandInit$  may be invalid, all valid candidates are present in  $CandInit$ , as deduced in Lemma 1. Further, PROCESSVERTEX procedure is invoked to obtain the candidates solutions according to vertex attributes and IRI information, and then only the common candidates are retained.

Before getting into the algorithmic details, we explain how the solutions are handled and how we process each query vertex. We define  $M$  as a set of tuples, whose  $i^{th}$  tuple is represented as  $M_i = [m_c, M_s]$ , where  $m_c$  is a solution pair for a core vertex, and  $M_s$  is a set of solution pairs for the set of satellite vertices that are connected to the core vertex. Formally  $m_c = (u_c, v_c)$ , where  $u_c$  is the core vertex and  $v_c$  is the corresponding matched vertex;  $M_s$  is a set of solution pairs, whose  $j^{th}$  element is a solution pair  $(u_s, V_s)$ , where  $u_s$  is a satellite vertex and  $V_s$  is a set of matched vertices. In addition, we maintain a set  $M_c$  whose elements are the solution pairs for all the core vertices. Thus during each recursion branch, the size of  $M$  grows until it reaches the query size  $|U|$ ; once  $|M| = |U|$ , homomorphic matches are obtained.

For all the candidate solutions of initial vertex  $CandInit$ ,



we perform recursion to obtain homomorphic embeddings (lines 8-19). Before getting into recursion, for each initial match  $v_{init} \in CandInit$ , if it has satellite vertices connected to it, we invoke the MATCHSATVERTICES procedure (Lines 10-11). This step not only finds solution matches for satellite vertices, if there are, but also checks if  $v_{init}$  is a valid candidate vertex. If the returned solution set  $M_{sat}$  is empty, then  $v_{init}$  is not a valid candidate and hence we continue with the next  $v_{init} \in CandInit$ ; else, we update the set of solution pairs  $M_s$  for satellite vertices and the solution pair  $M_c$  for the core vertex (Lines 12-15) and invoke HOMOMORPHICMATCH procedure (Lines 17). On the other hand, if there are no satellite vertices connected to  $u_{init}$ , we update the core vertex solution set  $M_c$  and invoke HOMOMORPHICMATCH procedure (Lines 18-19).

---

**Algorithm 4:** HOMOMORPHICMATCH( $M, \mathcal{I}, Q, U_c^{ord}$ )

---

```

1 if  $|M|=|U|$  then
2   return GENEMB( $M$ )
3  $Emb = \emptyset$ 
4 FETCH:  $u_{nxt} = u|u \in U_c^{ord}$ 
5  $N_g = \{u_c|u_c \in M_c\} \cap adj(u_{nxt})$ 
6  $N_g = \{v_c|v_c \in M_c \wedge (u_c, v_c) \in M_c\}$ , where  $u_c \in N_g$ 
7  $Cand_{u_{nxt}} = \bigcap_{n=1}^{|N_g|} (QueryNeighIndex(\mathcal{N}, L_E^Q(u_n, u_{nxt}), v_n))$ 
8  $Cand_{u_{nxt}} = Cand_{u_{nxt}} \cap PROCESSVERTEX(u_{nxt}, Q, \mathcal{A}, \mathcal{N})$ 
9 for each  $v_{nxt} \in Cand_{u_{nxt}}$  do
10  FETCH:  $U_{nxt}^{sat} = \{u|u \in V_s \wedge (u_{nxt}, u) \in E(Q)\}$ 
11  if  $U_{nxt}^{sat} \neq \emptyset$  then
12     $M_{sat} = MATCHSATVERTICES(\mathcal{A}, \mathcal{N}, Q, U_{nxt}^{sat}, v_{nxt})$ 
13    if  $M_{sat} \neq \emptyset$  then
14      for every  $[u^s, V^s] \in M_{sat}$  do
15        UPDATE:  $M_s = M_s \cup [u^s, V^s]$ 
16        UPDATE:  $M_c = M_c \cup (u_{nxt}, v_{nxt})$ 
17         $Emb = Emb \cup HOMOMORPHICMATCH(M, \mathcal{I}, Q, U_c^{ord})$ 
18  else
19    UPDATE:  $M_c = M_c \cup (u_{nxt}, v_{nxt})$ 
20     $Emb = Emb \cup HOMOMORPHICMATCH(M, \mathcal{I}, Q, U_c^{ord})$ 
21 return  $Emb$ 

```

---

In the HOMOMORPHICMATCH procedure (Algorithm 4), we fetch the next query vertex from the set of ordered core vertices  $U_c^{ord}$  (Line 4). Then we collect the neighbourhood vertices of already matched core query vertices and the corresponding matched data vertices (Lines 5-6). As we recall, the set  $M_c$  maintains the solution pair  $m_c = (u_c, v_c)$  of each matched core query vertex. The set  $N_g$  collects the already matched core vertices  $u_c \in M_c$  that are also in the neighbourhood of  $u_{nxt}$ , whose matches have to be found. Further,  $N_g$  contains the corresponding matched query vertices  $v_c \in M_c$ . As the recursion proceeds, we find those matchable data vertices of  $u_{nxt}$  that are in the neighbourhood of all the matched vertices  $v \in N_g$ , so that the query structure is maintained. In Line 7, for each  $u_n \in N_g$  and the corresponding  $v_n \in N_g$ , we query the neighbourhood index  $\mathcal{N}$ , to obtain the candidate solutions  $Cand_{u_{nxt}}$ , that are in the neighbourhood of already matched data vertex  $v_n$  and have the multiedge  $L_E^Q(u_n, u_{nxt})$ , obtained from the query multigraph  $Q$ . Finally (line 7), the set of candidates solutions that are common for every  $u_n \in N_g$  are retained in  $Cand_{u_{nxt}}$ .

Further, the candidate solutions are refined with the help of PROCESSVERTEX procedure (Line 8). Now, for each of the valid candidate solution  $v_{nxt} \in Cand_{u_{nxt}}$ , we recursively call the HOMOMORPHICMATCH procedure. When the next query vertex  $u_{nxt}$  has no satellite vertices attached to it, we update

the core vertex solution set  $M_c$  and call the recursion procedure (Lines 19-20). But when  $u_{nxt}$  has satellite vertices attached to it, we obtain the candidate matches for all the satellite vertices by invoking the MATCHSATVERTICES procedure (Lines 11-12); if there are matches, we update both the satellite vertex solution  $M_s$  and the core vertex solution  $M_c$ , and invoke the recursion procedure (Line 17).

Once all the query vertices have been matched for the current recursion step, the solution set  $M$  contains the solutions for both core and satellite vertices. Thus when all the query vertices have been matched, we invoke the GENEMB function (Line 2) which returns the set of embeddings, that are updated in  $Emb$ . The GENEMB function treats the solution vertex  $v_c$  of each core vertex as a singleton and performs Cartesian product among all the core vertex singletons and satellite vertex sets. Formally,  $Emb_{part} = \{v_c^1\} \times \dots \times \{v_c^{|U_c^1|}\} \times V_s^1 \times \dots \times V_c^{|U_s^1|}$ . Thus, the partial set of embeddings  $Emb_{part}$  is added to the final result  $Emb$ .

## 6. RELATED WORK

The proliferation of semantic web technologies has influenced the popularity of RDF as a standard to represent and share knowledge bases. In order to efficiently answer SPARQL queries, many stores and API inspired by relational model were proposed [7, 3, 13, 5]. x-RDF-3X [13], inspired by modern RDBMS, represent RDF triples as a big three-attribute table. The RDF query processing is boosted using an exhaustive indexing schema coupled with statistics over the data. Also Virtuoso[7] heavily exploits RDBMS mechanism in order to answer SPARQL queries. Virtuoso is a column-store based systems that employs sorted multi-column column-wise compressed projections. Also these systems build table indexing using standard B-trees. Jena [5] supplies API for manipulating RDF graphs. Jena exploits multiple-property tables that permit multiple views of graphs and vertices which can be used simultaneously.

Recently, the database community has started to investigate RDF stores based on graph data management techniques [6, 16, 11]. The work in [6] addresses the problem of supporting property graphs as RDF, since majority of the graph databases are based on property graph model. The authors introduce a property graph to RDF transformation scheme and propose three models to address the challenge of representing the key/value properties of property graph edges in RDF. gStore [16] applies graph pattern matching techniques using filter-and-refinement strategy to answer SPARQL queries. It employs an indexing schema, named VS\*-tree, to concisely represent the RDF graph. Once the index is built, it is used to find promising subgraphs that match the query. Finally, exact subgraphs are enumerated in the refinement step. Turbo\_Hom++ [11] is an adaptation of a state of the art subgraph isomorphism algorithm (TurboISO[9]) to the problem of SPARQL queries. Exploiting the standard graph isomorphism problem, the authors relax the injectivity constraint to handle the graph homomorphism, which is the RDF pattern matching semantics.

Unlike our approach, TurboHom++ does not index the RDF graph, while gStore concisely represents RDF data through VS\*-tree. Another difference between AMBER and the other graph stores is that our approach explicitly manages the multigraph induced by the SPARQL queries while no clear discussion is supplied for the other tools.

## 7. EXPERIMENTAL ANALYSIS

In this section, we perform extensive experiments on the three RDF benchmarks. We evaluate the time performance and the robustness of AMBER w.r.t. state-of-the-art competitors by varying the size, and the structure of the SPARQL queries. Experiments are carried out on a 64-bit Intel Core i7-4900MQ @ 2.80GHz, with 32GB memory, running Linux OS - Ubuntu 14.04 LTS. AMBER is implemented in C++.

### 7.1 Experimental Setup

We compare AMBER with the four standard RDF engines: *Virtuoso-7.1* [7], *x-RDF-3X* [13], *Apache Jena* [5] and *gStore* [16]. For all the competitors we employ the source code available on the web site or obtained by the authors. Another recent work *TurboHOM++* [11] has been excluded since it is not publicly available.

For the experimental analysis we use three RDF datasets - *DBPEDIA*, *YAGO* and *LUBM*. *DBPEDIA* constitutes the most important knowledge base for the Semantic Web community. Most of the data available in this dataset comes from the Wikipedia Infobox. *YAGO* is a real world dataset built from factual information coming from *Wikipedia* and *WordNet* semantic network. *LUBM* provides a standard RDF benchmark to test the overall behaviour of engines. Using the data generator we create *LUBM100* where the number represents the scaling factor.

Dataset	# Triples	# Vertices	# Edges	# Edge types
<i>DBPEDIA</i>	33 071 359	4 983 349	14 992 982	676
<i>YAGO</i>	35 543 536	3 160 832	10 683 425	44
<i>LUBM100</i>	13 824 437	2 179 780	8 952 366	13

Table 4: Benchmark Statistics

The data characteristics are summarized in Table 4. We can observe that the benchmarks have different characteristics in terms of number of vertices, number of edges, and number of distinct predicates. For instance, *DBPEDIA* has more diversity in terms of predicates ( $\sim 700$ ) while *LUBM100* contains only 13 different predicates.

The time required to build the multigraph database as well as to construct the indexes are reported in Table 5. We can note that the database building time and the corresponding size are proportional to the number of triples. Regarding the indexing structures, we can underline that both building time and size are proportional to the number of edges. For instance, *DBPEDIA* has the biggest number of edges ( $\sim 15M$ ) and, consequently, AMBER employs more time and space to build and store its data structure.

Dataset	Database		Index $\mathcal{I}$	
	Building Time	Size	Building Time	Size
<i>DBPEDIA</i>	307	1300	45.18	1573
<i>YAGO</i>	379	2400	29.1	1322
<i>LUBM100</i>	67	497	18.4	1057

Table 5: Offline stage: Database and Index Construction time (in seconds) and memory usage (in Mbytes)

### 7.2 Workload Generation

In order to test the scalability and the robustness of the different RDF engines, we generate the query workloads considering a similar setting as in [8, 1, 9]. We generate the query workload from the respective RDF datasets, which are available as RDF triplesets. In specific, we generate two types of query sets: a star-shaped and a complex-shaped query set; further, both query sets are generated for varying sizes (say  $k$ ) ranging from 10 to 50 triplets, in steps of 10.

To generate star-shaped or complex-shaped queries of size  $k$ , we pick an initial-entity at random from the RDF data. Now to generate star queries, we check if the initial-entity is present in at least  $k$  triples in the entire benchmark, to verify if the initial-entity has  $k$  neighbours. If so, we choose those  $k$  triples at random; thus the initial entity forms the central vertex of the star structure and the rest of the entities form the remaining star structure, connected by the respective predicates. To generate complex-shaped queries of size  $k$ , we navigate in the neighbourhood of the initial-entity through the predicate links until we reach size  $k$ . In both query types, we inject some object literals as well as constant *IRIs*; rest of the *IRIs* (subjects or objects) are treated as variables. However, this strategy could choose some very unselective queries [8]. In order to address this issue, we set a maximum time constraint of 60 seconds for each query. If the query is not answered in time, it is not considered for the final average (similar procedure is usually employed for graph query matching [9] and RDF workload evaluation [1]). We report the average query time and, also, the percentage of unanswered queries (considering the given time constraint) to study the robustness of the approaches.

### 7.3 Comparison with RDF Engines

In this section we report and discuss the results obtained by the different RDF engines. For each combination of query type and benchmark we report two plots by varying the query size: the average time and the corresponding percentage of unanswered queries for the given time constraint. We remind that the average time per approach is computed only on the set of queries that were answered.

The experimental results for *DBPEDIA* are depicted in Figure 6 and Figure 7. The time performance (averaged over 200 queries) for *Star-Shaped* queries (Fig. 6a), affirm that AMBER clearly outperforms all the competitors. Further the robustness of each approach, evaluated in terms of percentage of unanswered queries within the stipulated time, is shown in Figure 6b. For the given time constraint, *x-RDF-3X* and *Jena* are unable to output results for size 20 and 30 onwards respectively. Although *Virtuoso* and *gStore* output results until query size 50, their time performance is still poor. However, as the query size increases, the percentage of unanswered queries for both *Virtuoso* and *gStore* keeps on increasing from  $\sim 0\%$  to 65% and  $\sim 45\%$  to 95% respectively. On the other hand AMBER answers  $>98\%$  of the queries, even for queries of size 50, establishing its robustness.

Analyzing the results for *Complex-Shaped* queries (Fig. 7), we underline that AMBER still outperforms all the competitors for all sizes. In Figure 7a, we observe that *x-RDF-3X* and *Jena* are the slowest engines; *Virtuoso* and *gStore* perform better than them but nowhere close to AMBER. We further observe that *x-RDF-3X* and *Jena* are the least robust as they don't output results for size 30 onwards (Fig. 7b);

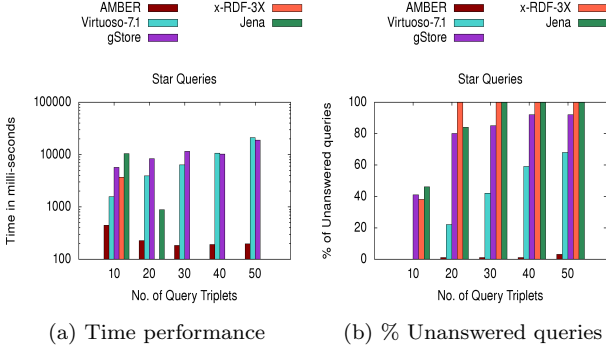


Figure 6: Evaluation of (a) time performance and (b) robustness, for *Star-Shaped* queries on *DBPEDIA*.

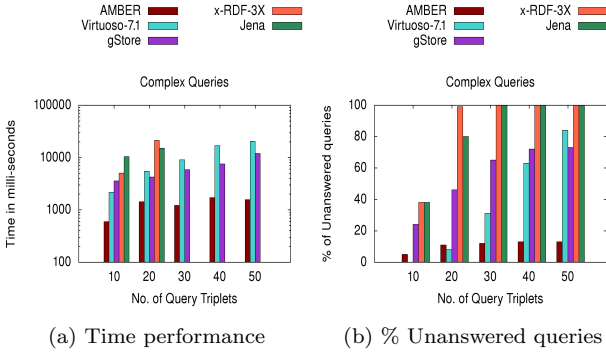


Figure 7: Evaluation of (a) time performance and (b) robustness, for *Complex-Shaped* queries on *DBPEDIA*.

on the other hand AMBER is the most robust engine as it answers  $>85\%$  of the queries even for size 50. The percentage of unanswered queries for *Virtuoso* and *gStore* increase from 0% to  $\sim 80\%$  and 25% to  $\sim 70\%$  respectively, as we increase the size from 10 to 50.

The results for *YAGO* are reported in Figure 8 and Figure 9. For the *Star-Shaped* queries (Fig. 8), we observe that AMBER outperforms all the other competitors for any size. Further, the time performance of AMBER is 1-2 order of magnitude better than its nearest competitor *Virtuoso* (Fig. 8a), and the performance remains stable even with increasing query size (Fig. 8b). *x-RDF-3X*, *Jena* are not able to output results for size 20 onwards. As observed for *DBPEDIA*, *Virtuoso* seems to become less robust with the increasing query size. For size 20-40, time performance of *gStore* seems better than *Virtuoso*; the reason seems to be the fewer queries that are being considered. Conversely, AMBER is able to supply answers most of the time ( $>98\%$ ).

Coming to the results for *Complex-Shaped* queries (Fig. 9), we observe that AMBER is still the best in time performance; *Virtuoso* and *gStore* are the closest competitors. Only for size 10 and 20, *Virtuoso* seems robust than AMBER. *Jena*, *x-RDF-3X* do not answer queries for size 20 onwards, as seen in Figure 9b.

The results for *LUBM100* are reported in Figure 10 and Figure 11. For the *Star-Shaped* queries (Fig. 10), AMBER always outperforms all the other competitors for any size (Fig. 10a). Further, the time performance of AMBER is

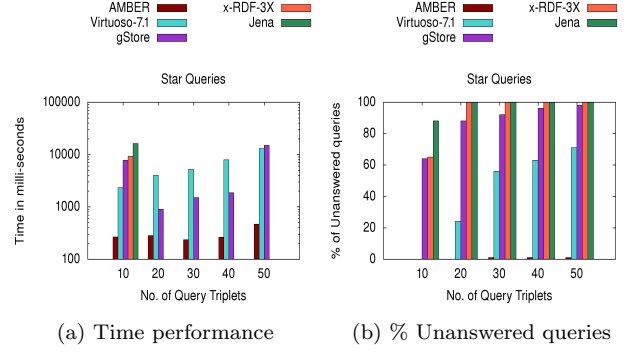


Figure 8: Evaluation of (a) time performance and (b) robustness, for *Star-Shaped* queries on *YAGO*.

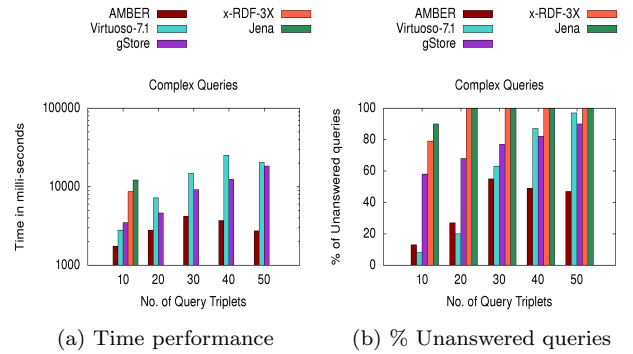


Figure 9: Evaluation of (a) time performance and (b) robustness, for *Complex-Shaped* queries on *YAGO*.

2-3 orders of magnitude better than its closest competitor *Virtuoso*. Similar to the *YAGO* experiments, *x-RDF-3X*, *Jena* are not able to manage queries from size 20 onwards; the same trend is observed for *gStore* too. Further, *Virtuoso* always loses its robustness as the query size increases. On the other hand, AMBER answers queries for all sizes.

Considering the results for *Complex-Shaped* queries (Fig. 11), we underline that AMBER has better time performance as seen in Figure 11a. *x-RDF-3X*, *Jena* and *gStore* did not supply answer for size 30 onwards (Fig. 11b). Further, *Virtuoso* seems to be a tough competitor for AMBER in terms of robustness for size 10 and 20. However, for size 30 onwards AMBER is more robust.

To summarise, we observe that *Virtuoso* is enough robust for *Complex-Shaped* smaller queries (10-20), but fails for bigger ( $>20$ ) queries. *x-RDF-3X* fails for queries with size bigger than 10. *Jena* has reasonable behavior until size 20, but fails to deliver from size 30 onwards. *gStore* has a reasonable behavior for size 10, but its robustness deteriorates from size 20 onwards. To summarize, AMBER clearly outperforms, in terms of time and robustness, the state-of-the-art RDF engines on the evaluated benchmarks and query configuration. Our proposal also scales up better than all the competitors as the size of the queries increases.

## 8. CONCLUSION

In this paper, a multigraph based engine AMBER has

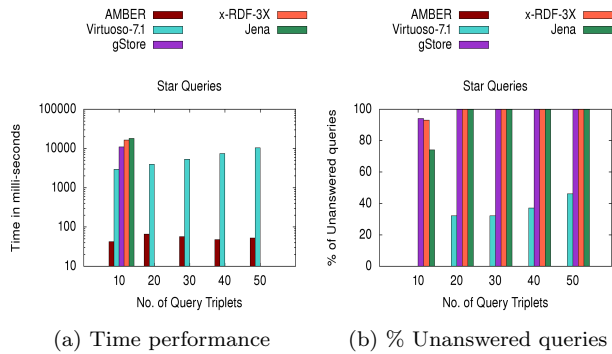


Figure 10: Evaluation of (a) time performance and (b) robustness, for *Star-Shaped* queries on *LUBM100*.

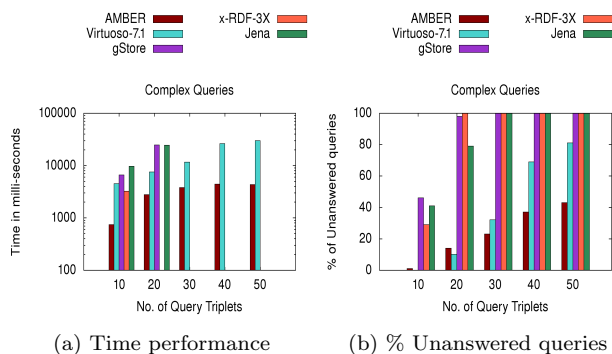


Figure 11: Evaluation of (a) time performance and (b) robustness, for *Complex-Shaped* queries on *LUBM100*.

been proposed in order to answer complex SPARQL queries over RDF data. The multigraph representation has bestowed us with two advantages: on one hand, it enables us to construct efficient indexing structures, that ameliorate the time performance of AMBER; on the other hand, the graph representation itself motivates us to exploit the valuable work done until now in the graph data management field. Thus, AMBER meticulously exploits the indexing structures to address the problem of sub-multigraph homomorphism, which in turn yields the solutions for SPARQL queries. The proposed engine AMBER has been extensively tested on three well established RDF benchmarks. As a result, AMBER stands out w.r.t. the state-of-the-art RDF management systems considering both the robustness regarding the percentage of answered queries and the time performance. As a future work, we plan to extend AMBER by incorporating other SPARQL operations and, successively, study and develop a parallel processing version of our proposal to scale up over huge RDF data.

## 9. ACKNOWLEDGMENTS

This work has been funded by Labex NUMEV (NUMEV, ANR-10-LABX-20).

## 10. REFERENCES

- [1] G. Alu, O. Hartig, M. T. zsu, and K. Daudjee. Diversified stress testing of RDF data management

- systems. In *ISWC*, pages 197–212, 2014.
- [2] G. Alu, M. T. zsu, and K. Daudjee. Workload matters: Why RDF databases need a new design. *PVLDB*, 7(10):837–840, 2014.
- [3] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *ISWC*, pages 54–68, 2002.
- [4] E. Cabrio, J. Cojan, A. P. Aprosio, B. Magnini, A. Lavelli, and F. Gandon. Qakis: an open domain QA system based on relational patterns. In *ISWC*, 2012.
- [5] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *WWW*, pages 74–83, 2004.
- [6] Souripriya Das, Jagannathan Srinivasan, Matthew Perry, Eugene Inseok Chong, and Jayanta Banerjee. A tale of two graphs: Property graphs as rdf in oracle. In *EDBT*, pages 762–773, 2014.
- [7] O. Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.
- [8] A. Gubichev and T. Neumann. Exploiting the query structure for efficient join ordering in sparql queries. In *EDBT*, pages 439–450, 2014.
- [9] W.-S. Han, J. Lee, and J.-H. Lee. Turbo<sub>iso</sub>: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, pages 337–348, 2013.
- [10] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.
- [11] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi. Taming subgraph isomorphism for RDF query processing. *PVLDB*, 8(11):1238–1249, 2015.
- [12] M. Morsey, J. Lehmann, S. Auer, and A.C.N. Ngomo. Dbpedia sparql benchmark performance assessment with real queries on real data. In *ISWC*, pages 454–469, 2011.
- [13] T. Neumann and G. Weikum. x-rdf-3x: Fast querying, high update rates, and consistency for RDF databases. *PVLDB*, 3(1):256–263, 2010.
- [14] M. Terrovitis, S. Passas, P. Vassiliadis, and T. Sellis. A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *CIKM*, pages 728–737. ACM, 2006.
- [15] L. Zou, R. Huang, H. Wang, J. Xu Yu, W. He, and D. Zhao. Natural language question answering over RDF: a graph data driven approach. In *SIGMOD Conference*, pages 313–324, 2014.
- [16] L. Zou, M. T. zsu, L. Chen, X. Shen, R. Huang, and D. Zhao. gstore: a graph-based SPARQL query engine. *VLDB J.*, 23(4):565–590, 2014.

# Optimization of Complex SPARQL Analytical Queries

Padmashree Ravindra\*  
 Microsoft Corporation, Redmond, USA  
 paravin@microsoft.com

HyeongSik Kim, Kemafor Anyanwu  
 North Carolina State University, Raleigh, USA  
 {hkim22, kogam}@ncsu.edu

## ABSTRACT

Analytical queries are crucial for many emerging Semantic Web applications such as clinical-trial recruiting in Life Sciences that incorporate patient and drug profile data. Such queries compare aggregates over multiple groupings of data which pose challenges in expression and optimization of complex grouping-aggregation constraints. While these challenges have been addressed in relational models, the semi-structured nature of RDF introduces additional challenges that need further investigation. Each grouping required in an RDF analytical query maps to a graph pattern subquery with related groups leading to overlapping graph patterns within the same query. The resulting algebraic expressions for such queries contain large numbers of joins, groupings and aggregations, posing significant challenges for present-day optimizers.

In this paper, we propose an approach for supporting efficient and scalable RDF analytics that follows the well known technique of simplifying algebraic expressions of RDF analytical queries in a way that enables better optimization. Specifically, the approach is based on a refactoring of analytical queries expressed in the relational-like SPARQL algebra based on a new set of logical operators. This refactoring achieves shared execution of common subexpressions that enables parallel evaluation of groupings as well as aggregations, leading to reduced I/O and processing costs, particularly beneficial for scale-out processing on distributed Cloud systems. Experiments on real-world and synthetic benchmarks confirm that such a rewriting can achieve up to 10X speedup over relational-style SPARQL query plans executed on popular Cloud systems.

## 1. INTRODUCTION

Growing amount of linked open data is enabling interesting applications that combine data from different domains for analysis. For example, the ReDD-Observatory [38] discusses a study reporting the total number of deaths and the number of clinical trials for Tuberculosis and HIV/AIDS in all countries, to analyze the dispar-

\*Majority of the work was done when the first author was a student at the Department of Computer Science, North Carolina State University

```

SELECT ?country ?feature ((?sumF * (?cntT - ?cntF)) /
                          (?cntF * (?sumT - ?sumF))) As ?priceRatio
{{
  SELECT ?country (count(?price) As ?cntT) (sum(?price) As ?sumT)
  {
    ?product  rdf:type          PT18.
    ?offer    bsbm:product     ?product ;
              bsbm:price       ?price ;
              bsbm:vend        ?vend .
    ?vend     bsbm:country     ?country .
  }
  GROUP BY ?country
}
}

SELECT ?country ?feature
      (count(?price2) As ?cntF) (sum(?price2) As ?sumF)
{
  ?product2  rdf:type          PT18 ;
              bsbm:productFeature ?feature .
  ?offer2    bsbm:product     ?product2 ;
              bsbm:price       ?price2 ;
              bsbm:vend        ?vend2 .
  ?vend2     bsbm:country     ?country .
}
  GROUP BY ?country ?feature
}}
  
```

GP1

GP2

Figure 1: (AQ1): An example SPARQL analytical query, For each country, retrieve product features with the highest ratio between price with that feature and price without that feature

ity between biomedical research and the disease burden in developing countries. This study involved information about clinical trials and effectiveness of treatment options from *ClinicalTrials.gov*, statistics about mortality for different countries from the *Global Health Observatory* (GHO), published by the World Health Organization and biomedical research (MEDLINE publications and other life science journals) available in the *PubMed*. The results need to be grouped based on both country and disease, followed by aggregations on the number of clinical trials and deaths due to the concerned disease in each country, using the grouping-aggregation constructs in SPARQL 1.1 [22]. Another Semantic Web application, AlzPharm [26], queries several semantically-linked neuroscience datasets to find information relevant to neurodegenerative diseases, e.g., identify the different groups of drugs used for Alzheimer's Disease when grouped by their molecular targets and clinical usage.

Non-trivial analytical queries require multiple aggregations over different groupings of data, some of which may be related, resulting

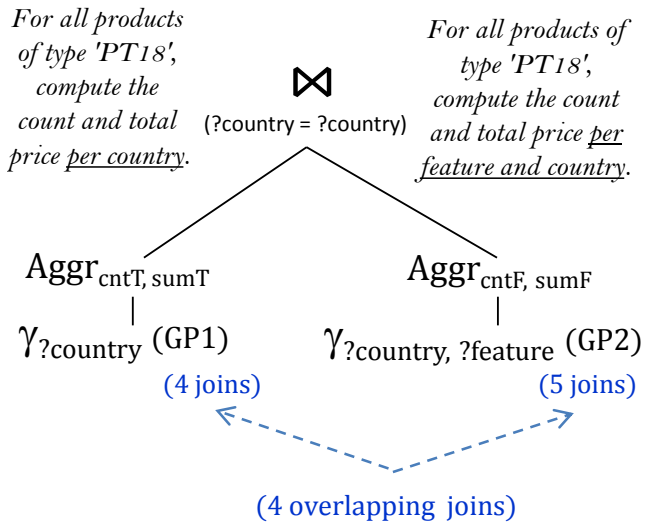


Figure 2: A relational-algebra based query plan for AQ1

in redundant scans and joins over large relations. Consider an example SPARQL analytical query *AQ1* shown in Figure 1, adopted from the Berlin SPARQL BI benchmark [1]. The query involves two descriptions GP1 and GP2 for products of type ‘PT18’ with grouping constraints on *country* and (*country*, *feature*) combinations, respectively. Each grouping constraint is defined over a graph pattern (a combination of one or more triple patterns<sup>1</sup> that specifies constraints to retrieve relevant subgraphs). Queries with multiple groupings involve multiple graph patterns. Further, if the groupings are related then there is a significant amount of overlap in the graph pattern subqueries. Figure 2 shows a summarized query plan with two major subqueries using the traditional evaluation technique: a subquery for GP1 with four joins that matches subgraphs about offers for products of type PT18, their price and vendor information, followed by a grouping on vendor’s *country*. The second subquery contains a similar graph pattern GP2 with five joins (an extra join due to the addition of product feature) followed by a grouping on *country-feature*. Answers from the two subqueries are then joined to compute the final price ratio, resulting in a total of 10 joins and 2 grouping operations.

In contrast, in the relational model, such OLAP queries are evaluated over suitably organized (star or snowflake) schemas consisting of *n*-ary relations. Different optimization strategies ranging from specialized query constructs [20, 9, 10], efficient indexing [31, 37], materialized views [21, 12], and efficient evaluation in distributed data warehouses [7, 6] have been proposed. In the absence of such schema organizations for RDF, a naive approach is to decompose the evaluation into two distinct phases: a graph pattern evaluation phase that constructs a suitable set of *n*-ary relations, followed by relational-style optimizations. However, such an approach prevents the possibility of optimizations across the two phases, e.g., early projections, partial aggregations, etc. Therefore, a holistic optimization strategy is likely to be more advantageous.

A promising direction is based on the observation made in [10] that relational expressions tightly couple grouping and aggregation specifications, often resulting in complex algebraic expressions that confound query optimizers. The approach in this paper has a similar spirit, i.e., grouping-aggregation specifications in RDF analyt-

<sup>1</sup>RDF data is modeled as a set of triples = (*Subject*, *Property*, *Object*). A triple pattern is a triple with at least one variable denoted by a leading ‘?’

ical queries are decoupled to optimize subqueries. Further, with a focus to support large scale RDF analytics, the paper overviews how such a query reformulation can be evaluated on Cloud platforms such as MapReduce [17]. The challenges with evaluating complex queries with many join operations have been addressed in several papers [4, 23, 33], and can be summarized as long, expensive execution workflows with multiple I/O and network data transfer phases. Many techniques have been proposed to mitigate these costs by sharing scans and computations [30, 28, 33] during MapReduce-based processing. In this paper, we present a holistic optimization that integrates the work on algebraic optimization of graph pattern queries with algebraic optimization of OLAP queries. Specifically, we make the following contributions:

- An algebraic rewriting of overlapping graph patterns (in a SPARQL analytical query) using a *composite graph pattern* based on common substructures. A decoupled reformulation of the grouping-aggregation definitions in a SPARQL analytical query expressed using a composite graph pattern.
- A set of logical and physical operators for efficient evaluation of a composite graph pattern, as well as parallel evaluation of independent aggregations on a composite graph pattern. The suite of operators and optimizations are integrated into *RAPIDAnalytics*, an extension of Apache Pig.
- A comprehensive evaluation of *RAPIDAnalytics* using basic and multi-aggregation SPARQL analytical queries on real-world as well as synthetic benchmark datasets.

The rest of the paper is organized as follows: Section 2 provides a background on complex OLAP queries and specific challenges in processing such queries over the RDF data model. Section 3 introduces an algebraic rewriting of SPARQL analytical queries based on a non-relational data model and algebra, followed by formal definitions of newly introduced logical operators. Section 4 describes the physical operators and optimizations to execute such a query plan on MapReduce-based platforms. Section 5 presents the comparative evaluation results between *RAPIDAnalytics* and other popular approaches, and Section 6 presents concluding remarks.

## 2. BACKGROUND AND CHALLENGES

### 2.1 Optimization of Complex OLAP Queries

There has been a body of work to enable better expression and evaluation [20, 19, 10, 6, 11] of complex OLAP queries including introduction of constructs such as the *CUBE BY* [20], grouping sets [9], etc., that allow the user to have a finer control over the grouping and aggregation specifications. An earlier work on *MD-Join* [10] showed that decoupling of the grouping definition and aggregation computations not only allows more succinct expression of complex OLAP queries, but can also eliminate redundant scans and joins over large fact tables.

**Parallel/Distributed Evaluation of Relational OLAP Queries.** An earlier work on parallel evaluation of aggregates proposed adaptive algorithms [35] to handle a range of grouping selectivities (ratio of result size to input size) across queries. Subsequent research [6] on distributed evaluation of OLAP queries identified optimizations that exploit knowledge about data distributions to reduce the amount of data transfer between the local sites and the centralized coordinator. In the context of MapReduce, an overlapping redistribution scheme [14] was proposed to enable parallel evaluation of correlated aggregations with sliding windows. MR-Cube [29] distributes the cube computation of partially algebraic measures on MapReduce. It also introduced a value-partitioning scheme to deal with

reducer-unfriendly (cube) groups that tend to increase the load on a reducer. The work on MR-Cube was integrated into Apache Pig<sup>2</sup> and Apache Hive<sup>3</sup> (GROUPING SETS, CUBE and ROLLUP clauses). Such operations assume the existence of a fact relation on which CUBE and other operations can be applied, which does not hold true in the case of the RDF data model where triples are commonly represented as binary or ternary relations.

**Expression and Evaluation of RDF Analytical Queries.** The RDF Data Cube vocabulary (QB) [16] was provided as a recommendation to enable publication of statistical data in RDF adhering to Linked data principles. The work on Open Cubes vocabulary [18] enables representation of multidimensional data using RDF Schema (RDFS). Other extensions [24] propose a multi-dimensional model based on QB to support OLAP queries, mapping them to SPARQL. Recent work on RDF analytics [15] proposed a way to define an analytical schema on RDF graphs and formalize analytical queries over such an analytical schema, by separating the grouping-aggregation definitions, similar to the relational *MD-Join* [10] operator. An earlier work [36] extended Pig’s query primitives to support MapReduce based execution of the *MD-Join* operator.

**Discussion.** The *MD-Join* approach to eliminate redundant scans and joins involving large fact relations, translates to reduction in I/O and network transfer costs in MapReduce-based processing of complex analytical queries. However, specifics of RDF analytics make it challenging to adopt such an approach. Unlike traditional OLAP systems where the fact and dimension tables are available and suitably organized into star or snowflake schema, the fine-grained data model in RDF necessitates several join operations to reassemble the relevant fact and dimension information, e.g., fact relation described by GP1 requires four join operations. A relational-style query plan that computes the detail relations described by GP1 and GP2, compiles into a lengthy MapReduce execution workflow with 9 map-reduce cycles (one per star-join). Such a sequential execution limits opportunities to share input scans in general. Furthermore, RDF analytical queries often involve (slightly) different join expressions for detail relations (refer to GP1 and GP2). Thus, in order to fully exploit the benefit of a decoupled reformulation using the *MD-Join* approach, we require additional optimizations that enable shared execution of the graph patterns in an RDF analytical query.

## 2.2 Shared Execution of Graph Pattern Queries

A commonly occurring pattern in OLAP queries involves comparing subtotals across multiple dimensions, which results in subqueries that compute groupings over an overlapping subset of dimensions, e.g., GP2 computes groupings on country-feature, while GP1 is a roll-up on ALL features. In the context of RDF, related groupings result in subqueries with common subexpressions (graph patterns with overlapping structure) enabling opportunities for shared execution. For example, if two graph patterns in a query have the same structure (same join expression), then the graph pattern can be evaluated only once. In cases where graph patterns have subsumption relationship in join expressions, there may be opportunities to rewrite the query in a way that allows shared execution of common substructures. Even with structurally different graph patterns, there may be sharing opportunities within a MapReduce cycle.

Several techniques have been proposed to enable sharing of scans and computations across a MapReduce workload in order to reduce the associated I/O and network transfer costs, e.g., MRShare [30]

proposes sharing of input scans, sharing map functions and map output, while executing a batch of grouping queries on a common input table. YSmart [28] groups correlated operations in complex queries, e.g., JOINS and GROUP BYS accessing the same table, into a single MapReduce job to reduce redundant scans, computations, and network transfers (integrated into Hive 0.12.0).

A previous work [27] on multi-query optimization (MQO) of SPARQL queries, rewrites the input graph pattern queries into a set of queries  $Q_{OPT}$  using the SPARQL OPTIONAL<sup>4</sup> clause. Given a set of graph pattern queries  $Q$  with common substructures, the basic idea of SPARQL MQO is to (i) rewrite the input queries into a set of queries  $Q_{OPT}$  with OPTIONAL clauses (representing non-overlapping structures), (ii) evaluate queries  $Q_{OPT}$  over the RDF graph, and (iii) distribute the results of  $Q_{OPT}$  to input queries in  $Q$ . For example, two queries with the following set of triple patterns:

$Q_1: (?s \text{ p1 } ?o)$   
 $Q_2: (?s1 \text{ p1 } ?o1) (?s1 \text{ p2 } ?o2)$

can be expressed using the OPTIONAL clause as follows:

$Q_{OPT}: (?s \text{ p1 } ?o) \text{ OPTIONAL } (?s \text{ p2 } ?o2)$

where the non-overlapping triple pattern in  $Q_2$  is specified as optional, i.e., resulting tuples may have NULL values for bindings of second triple pattern. Results matching original queries are extracted from results of  $Q_{OPT}$ . Note that multi-valued properties in the optional component may introduce duplicity and require special handling.

**Discussion.** A possible strategy to optimize RDF analytical queries is to rewrite and evaluate the individual graph patterns using the SPARQL MQO approach, extract answers to original graph patterns, and compute groupings over extracted subquery results. While such a rewriting seems beneficial when compared to sequential evaluation of individual graph patterns on MapReduce, our experiments on Hive showed that evaluating  $Q_{OPT}$  ahead of time prevents optimizations such as early projection and partial aggregations. This is because  $Q_{OPT}$  would need to be evaluated and stored as an intermediate table, since Hive neither supports logical views involving complex queries with multiple joins, nor does it support materialized views.

## 2.3 Rationale of Our Approach

We argue that it is necessary to approach the problem of optimizing RDF analytics *holistically*, rather than a two-step approach of independently optimizing the graph pattern matching phase and the grouping-aggregation phases. Given that RDF analytical queries often involve repeated computations over slightly different graph patterns, query plans that enable shared execution of common subpatterns are likely to compile into efficient execution plans. An important factor in this regard is the choice of algebra, the associated data model and the set of operators. One may use a relational-like algebra or alternatives such as the Nested TripleGroup Data Model and Algebra (NTGA) [33, 25]. We chose to use NTGA due to its underlying “groups of triples” or *triplegroup* model that enables concurrent computation of star-shaped join subpatterns (star-joins) in a query. The NTGA query plans not only enable sharing of scans and computations across multiple star subpatterns (resulting in shortened map-reduce execution workflows), but also concisely represent intermediate results in a denormalized form. In the next section, we build on the foundations of sharing that is already inherent in the NTGA approach and enhance its benefits by optimizing complex grouping-aggregation constraints.

<sup>2</sup><https://pig.apache.org/>

<sup>3</sup><https://hive.apache.org/>

<sup>4</sup>The OPTIONAL clause is used in SPARQL to allow querying of predicates that may not exist, i.e., answer is returned if there is a subgraph matching the OPTIONAL graph pattern, else it is ignored.

	GP1	GP2	Does GP1 Overlap GP2?	Composite GP'
AQ2	<pre>SELECT ?s1... WHERE {   Stp<sub>a</sub> { ?s1 ty PT18 . (jtp<sub>a</sub>)         ?s2 pr ?s1 . (jtp<sub>b</sub>)   Stp<sub>b</sub> { ?s2 pc ?o1 .         ?s2 ve ?o2 . }</pre>	<pre>SELECT ?s1... WHERE {   Stp<sub>α</sub> { ?s1 ty PT18 . (jtp<sub>α</sub>)         ?s1 pf ?o3 .   Stp<sub>β</sub> { ?s2 pr ?s1 . (jtp<sub>β</sub>)         ?s2 pc ?o4 . }</pre>	<ul style="list-style-type: none"> <li>• { ty } in overlap of Stp<sub>a</sub> and Stp<sub>α</sub></li> <li>• { pr, pc } in overlap of Stp<sub>b</sub> and Stp<sub>β</sub></li> <li>• Property of jtp<sub>a</sub> and jtp<sub>α</sub> match</li> <li>• Property of jtp<sub>b</sub> and jtp<sub>β</sub> match</li> <li>• Role of ?s1 ∈ jtp<sub>a</sub> (subject) is same as role of ?s1 ∈ jtp<sub>α</sub> (subject)</li> <li>• Role of ?s1 ∈ jtp<sub>b</sub> (object) is same as role of ?s1 ∈ jtp<sub>β</sub> (object)</li> </ul> <p>Hence, <b>GP1 overlaps GP2</b></p>	<pre>SELECT ?s1... WHERE {   Stp'<sub>a</sub> { ?s1 ty PT18 .         ?s1 pf ?o6 .   Stp'<sub>b</sub> { ?s2 pr ?s1 .         ?s2 pc ?o7 .         ?s2 ve ?s3 . }</pre>
AQ3	<pre>SELECT ?s3... WHERE {   Stp<sub>c</sub> { ?s3 pr ?s1 .         ?s3 pc ?o5 .   Stp<sub>d</sub> { ?s3 ve ?s4 . (jtp<sub>c</sub>)         ?s4 cn ?o6 . (jtp<sub>d</sub>) }</pre>	<pre>SELECT ?s3... WHERE {   Stp<sub>γ</sub> { ?s3 pr ?s1 .         ?s3 pc ?o5 .   Stp<sub>δ</sub> { ?s3 ve ?o6 . (jtp<sub>γ</sub>)         ?s4 cn ?o6 . (jtp<sub>δ</sub>) }</pre>	<ul style="list-style-type: none"> <li>• { pr, pc, ve } in overlap of Stp<sub>c</sub> and Stp<sub>γ</sub></li> <li>• { cn } in overlap of Stp<sub>d</sub> and Stp<sub>δ</sub></li> <li>• Property of jtp<sub>c</sub> and jtp<sub>γ</sub> match</li> <li>• Property of jtp<sub>d</sub> and jtp<sub>δ</sub> match</li> <li>• Role of ?s4 ∈ jtp<sub>c</sub> (object) is same as role of ?o6 ∈ jtp<sub>γ</sub> (object)</li> <li>• Role of ?s4 ∈ jtp<sub>d</sub> (subject) is NOT same as role of ?o6 ∈ jtp<sub>δ</sub> (object)</li> </ul> <p>Hence, <b>GP1 does NOT overlap GP2</b></p>	Not Applicable

Figure 3: Structural overlap in graph patterns

Table 1: Quick Reference

Symbol	Description
$tp$	Triple pattern
$jtp_i$	Joining triple pattern in $Stp_i$
$jv_{ij}$	Variable joining $tp_i$ and $tp_j$
$GP$	Graph pattern
$Stp$	Subject-rooted star subpattern
$Stp_{abc}$	Star pattern with property-set { a, b, c }
$Stp_{ab\bar{c}}$	Star pattern with primary properties a and b, and secondary (optional) property c
$P_{prim}$	Set of primary properties
$P_{sec}$	Set of secondary properties
$tg$	Triplegroup
$TG$	Set of triplegroups
$TG_{abc}$	Set of triplegroups with property-set { a, b, c }

Function	Returns
$\text{var}(tp)$	Set of variables in triple pattern $tp$
$\text{role}(?v)$	Role of variable $?v$ (subject, property, or object)
$\text{prop}(tp)$	Property of triple pattern $tp$
$\text{props}(Stp_i)$	Set of properties in $Stp_i$
$\delta(?v)$	Variable substitution in a triple matching $tp$

### 3. ALGEBRAIC REWRITING OF SPARQL ANALYTICAL QUERIES

We reformulate SPARQL analytical queries with multiple grouping-aggregation constraints by, (i) identifying *overlaps* between graph patterns in a query based on structural constraints, (ii) evaluating a *composite graph pattern* that retrieves answers for original graph patterns, and (iii) computing required groupings and aggregations based on the composite graph pattern. Common notations and convenience functions used in this paper are summarized in Table 1.

**Definition 3.1 (Overlapping Star Patterns)** Let  $Stp_1$  and  $Stp_2$  be two subject-rooted star subpatterns and let  $L$  be the intersection of their property sets, i.e.,  $L = \text{props}(Stp_1) \cap \text{props}(Stp_2)$ . Then,  $Stp_1$  and  $Stp_2$  are considered to overlap if the following holds:

- Intersection of their property sets is non-empty, i.e.,  $L \neq \emptyset$ .

- For any triple pattern  $tp_1 = (s1, \text{rdf:type}, o1) \in Stp_1$ , there exists some  $tp_2 = (s2, \text{rdf:type}, o2) \in Stp_2$ , with the same object component, i.e.,  $o1 = o2$ .

Figure 3 represents two analytical queries AQ2 and AQ3, each consisting of two graph patterns GP1 and GP2 (properties abbreviated). In the case of query AQ2, star pattern Stp<sub>a</sub> ∈ GP1 overlaps with Stp<sub>α</sub> ∈ GP2 since both match on the object of `rdf : type` triple. Similarly, star patterns Stp<sub>b</sub> and Stp<sub>β</sub> overlap. The graph patterns in AQ4 also have two overlapping star patterns, i.e., Stp<sub>c</sub> structurally overlaps with Stp<sub>γ</sub>, and Stp<sub>d</sub> overlaps with Stp<sub>δ</sub>.

Additionally, analytical queries may contain `FILTER` clauses that need to be considered while determining overlap between star patterns. For example, consider a filter on GP1 to retrieve a subset of products with price (property abbreviated as `pc`) > 5000, i.e., `FILTER(?o5 > 5000)`. A possible strategy is to compute generalized composite star patterns (without filter) and apply restrictions prior to the aggregation phase. Pushing the filter to a later phase in the workflow may have implications on I/O and network transfer costs associated with materialization of some irrelevant intermediate results. Another interesting case is that of unbound-property star patterns containing triple patterns such as `(?s1 ?p o1)`, used to query unknown or don't care relationships. Such queries need special handling, specifically if the unbound-property triple pattern participates in a join with other star patterns. Advanced optimizations for both these cases are out of scope of this paper. For the rest of this paper, we consider optimization of multi-graph-pattern queries involving bound-property star patterns with same filter constraints or filter constraints on a non-intersecting property.

Next, we generalize the notion of overlap to graph patterns by capturing similarity of join structures between star patterns. In order to do so, we introduce the concept of *role-equivalence* of join variables. Given two triple patterns  $tp_1$  and  $tp_2$ , a join variable  $jv_1$  is a variable in  $\text{var}(tp_1) \cap \text{var}(tp_2)$ . A join variable  $jv_1 \in tp_1$  is said to be *role-equivalent* to join variable  $jv_3 \in tp_3$  if, (i) the corresponding triple patterns agree on the property component, i.e.,  $\text{prop}(tp_1) = \text{prop}(tp_3)$ , and (ii) the join variables play the same role (subject, property, or object), i.e.,  $\text{role}(jv_1)$  in  $tp_1$  is the same as  $\text{role}(jv_3)$  in  $tp_3$ .

**Definition 3.2 (Overlapping Graph Patterns)** Let graph pattern GP1 involve star subpatterns  $Stp_a, Stp_b, \dots$ , such that  $jv_{ab}$  de-



### (a) Optional Group Filter:

$$\sigma_{(\{product, price\}, \{validFrom, validTo\})}^{\gamma opt}(TG) = \left[ \begin{array}{l} \left. \begin{array}{l} \mathbf{tg}_1 = \left[ \begin{array}{l} (offer1, \mathbf{product}, \mathbf{prod1}), \\ (offer1, \mathbf{price}, 108), \\ (offer1, \mathbf{validTo}, "08/08/2014") \end{array} \right] \\ \mathbf{tg}_2 = \left[ \begin{array}{l} (offer2, \mathbf{product}, \mathbf{prod3}), \\ (offer2, \mathbf{price}, 121) \end{array} \right] \\ \mathbf{tg}_3 = \left[ \begin{array}{l} (offer3, \mathbf{product}, \mathbf{prod1}), \\ (offer3, \mathbf{validFrom}, "02/08/2014"), \\ (offer3, \mathbf{validTo}, "08/08/2014") \end{array} \right] \\ \mathbf{tg}_4 = \left[ \begin{array}{l} (offer8, \mathbf{product}, \mathbf{prod3}), \\ (offer8, \mathbf{price}, 360), \\ (offer8, \mathbf{validFrom}, "01/01/2014"), \\ (offer8, \mathbf{validTo}, "11/01/2014") \end{array} \right] \end{array} \right\} \mathbf{tg}_{all} \end{array} \right] = TG'$$

### (b) n-split: Example1

$$\chi_{(\{product, price\}, \{\{validFrom\}, \{validTo\}\})}(TG') = \left[ \begin{array}{l} \left. \begin{array}{l} \mathbf{tg}_{12} = \left[ \begin{array}{l} (offer1, \mathbf{product}, \mathbf{prod1}), \\ (offer1, \mathbf{price}, 108), \\ (offer1, \mathbf{validTo}, "08/..") \end{array} \right] \\ \mathbf{tg}_{41} = \left[ \begin{array}{l} (offer8, \mathbf{product}, \mathbf{prod3}), \\ (offer8, \mathbf{price}, 360), \\ (offer8, \mathbf{validFrom}, "01/..") \end{array} \right] \\ \mathbf{tg}_{42} = \left[ \begin{array}{l} (offer8, \mathbf{product}, \mathbf{prod3}), \\ (offer8, \mathbf{price}, 360), \\ (offer8, \mathbf{validTo}, "11/..") \end{array} \right] \end{array} \right\} \end{array} \right]$$

### (c) n-split: Example2

$$\chi_{(\{product, price\}, \{\{\}, \{validTo\}\})}(TG') = \left[ \begin{array}{l} \left. \begin{array}{l} \mathbf{tg}_{11} = \left[ \begin{array}{l} (offer1, \mathbf{product}, \mathbf{prod1}), \\ (offer1, \mathbf{price}, 108) \end{array} \right] \\ \mathbf{tg}_{12} = \left[ \begin{array}{l} (offer1, \mathbf{product}, \mathbf{prod1}), \\ (offer1, \mathbf{price}, 108), \\ (offer1, \mathbf{validTo}, "08/..") \end{array} \right] \\ \mathbf{tg}_{21} = \left[ \begin{array}{l} (offer2, \mathbf{product}, \mathbf{prod3}), \\ (offer2, \mathbf{price}, 121) \end{array} \right] \\ \mathbf{tg}_{41} = \left[ \begin{array}{l} (offer8, \mathbf{product}, \mathbf{prod3}), \\ (offer8, \mathbf{price}, 360) \end{array} \right] \\ \mathbf{tg}_{42} = \left[ \begin{array}{l} (offer8, \mathbf{product}, \mathbf{prod3}), \\ (offer8, \mathbf{price}, 360), \\ (offer8, \mathbf{validTo}, "11/..") \end{array} \right] \end{array} \right\} \end{array} \right]$$

Figure 4: NTGA logical operators to evaluate composite graph patterns

notes the variable that joins a triple pattern  $jtp_a \in Stp_a$  with  $jtp_b \in Stp_b$ . Let graph pattern GP2 involve star subpatterns  $Stp_\alpha, Stp_\beta, \dots$  such that  $jv_{\alpha\beta}$  denotes the variable that joins a triple pattern  $jtp_\alpha \in Stp_\alpha$  with  $jtp_\beta \in Stp_\beta$ . Then, the graph patterns GP1 and GP2 are said to overlap if the following conditions hold:

- Each star pattern  $Stp_a \in GP1$  overlaps with some star pattern  $Stp_\alpha \in GP2$
- Given a pair of overlapping star patterns  $Stp_a$  and  $Stp_\alpha$ , their join variables  $jv_{ab}$  and  $jv_{\alpha\beta}$  are role-equivalent.

In the case of AQ2, graph patterns GP1 and GP2 overlap since both star patterns overlap and have the same join structure, e.g., subject-object join between  $Stp_a$  and  $Stp_b$  in GP1 matches the join structure between  $Stp_\alpha$  and  $Stp_\beta$  in GP2. In the case of AQ3, both star patterns overlap. However,  $Stp_c$  joins  $Stp_d$  using an object-subject join, where as  $Stp_\gamma$  joins  $Stp_\delta$  using an object-object join. Since the join structures are not similar, we consider GP1 and GP2 to be non-overlapping. Though there may be possibilities to share some scans and computations across non-overlapping graph patterns, for the rest of the paper we consider optimization of overlapping graph patterns.

**Construction of a Composite Graph Pattern.** Overlapping graph patterns GP1 and GP2 can be re-written as a *composite graph pattern* GP' that captures the (non) overlapping substructures. For a pair of overlapping star patterns  $Stp_a \in GP1$  and  $Stp_\alpha \in GP2$ , we define a composite star pattern  $Stp'_i$  such that:

- $\text{props}(Stp'_i) = P_{prim} \cup P_{sec}$
- $P_{prim} = \text{props}(Stp_a) \cap \text{props}(Stp_\alpha)$ , set of *primary* properties defining common substructures across star patterns.
- $P_{sec} = \{ p_i \mid p_i \in \text{props}(Stp_a) \cup \text{props}(Stp_\alpha), p_i \notin P_{prim} \}$ , set of *secondary* properties defining non-overlapping structures.

For example,  $Stp_a \in GP1$  and  $Stp_\alpha \in GP2$  can be rewritten as  $Stp'_a$  such that  $\text{props}(Stp'_a) = \{ \mathbf{ty18}, \mathbf{pf} \}$ , where  $\mathbf{ty18}$  (short for  $\mathbf{rdf : type PT18}$ ) is the primary property and  $\mathbf{pf}$  is the secondary property (underlined). Similarly,  $Stp_b \in GP1$  and  $Stp_\beta \in GP2$  can be expressed as  $Stp'_b$  with set of properties  $\{ \mathbf{pr}, \mathbf{pc}, \mathbf{ve} \}$ . Query AQ1 can be re-written using a composite graph pattern:

$$GP' = (Stp'_1 \bowtie Stp'_2 \bowtie Stp'_3)$$

where  $\text{props}(Stp'_1) = \{ \mathbf{ty18}, \mathbf{pf} \}$ ,  $\text{props}(Stp'_2) = \{ \mathbf{pr}, \mathbf{pc}, \mathbf{ve} \}$ , and  $\text{props}(Stp'_3) = \{ \mathbf{cn} \}$ .

Answers matching a composite graph pattern may contain superfluous subtuples that do not match either of the original patterns, resulting in wrong aggregates. Hence, we need a way to validate join combinations.

An NTGA-based rewriting of a SPARQL analytical query requires support to compute and manipulate triplegroups that match composite star patterns and composite graph patterns. Specifically, we need support for the following operations – (i) A specialized triplegroup-filter operator that validates secondary (optional) properties in a composite star pattern; (ii) An operator to extract subsets of a triplegroup that match  $n$  original star patterns; (iii) A special join operator that restricts joins on valid combinations of composite star patterns; (iv) An operator in the spirit of *MD-Join* to compute grouping-aggregations on triplegroups. Next, we formally define the triplegroup-based logical operators. We assume our input to be a set of subject triplegroups (triples grouped on subject column).

## 3.1 Logical Operators

**Definition 3.3 (Optional Group Filter)** Given a set of subject triplegroups  $TG$  and a star pattern  $Stp$  containing a set of primary properties  $P_{prim}$ , and a set of optional properties  $P_{opt}$ , the **optional group-filter** operator  $\sigma^{\gamma opt}$  returns the subset of triplegroups in  $TG$  that contains a non-empty subset of triples matching all properties in  $P_{prim}$  and may contain triples matching properties in  $P_{opt}$ . Specifically,

$$\sigma_{(P_{prim}, P_{opt})}^{\gamma opt}(TG) := \{ tg_i \in TG \mid P_{prim} \subseteq \text{props}(tg_i) \subseteq (P_{prim} \cup P_{opt}) \}$$

where  $\text{props}(tg_i)$  returns the set of properties in a triplegroup  $tg_i$ . Essentially,  $\sigma^{\gamma opt}$  ensures that triplegroups contain a matching triple for each of the primary properties and may contain matches for properties in  $P_{opt}$ . For example, given  $P_{prim} = \{ \mathbf{product}, \mathbf{price} \}$ , triplegroup  $\mathbf{tg}_1$ ,  $\mathbf{tg}_2$ , and  $\mathbf{tg}_4$  are valid results for the  $\sigma^{\gamma opt}$  expression in Figure 4(a). However,  $\mathbf{tg}_3$  does not contain a matching triple for the primary property  $\mathbf{price}$ , and hence gets filtered out. Note that valid triplegroups may have triples matching zero or more of the two optional properties  $P_{opt} = \{ \mathbf{validFrom}, \mathbf{validTo} \}$ .

Table 2: Evaluating composite graph patterns using  $\alpha$ -Join

GP1	GP2	GP'	$\bowtie_{(\alpha_1 \vee \alpha_2)}^{\gamma}(\dots)$
$Stp_1:Stp_2$	$Stp_1:Stp_2$	$Stp'_1:Stp'_2$	$\alpha_1 \quad \alpha_2$
ab:de	ab:de	ab:de	—
ab:de	ab:def	ab:def	$f = \emptyset \quad f \neq \emptyset$
ab:de	abc:def	abc:def	$c = \emptyset \wedge f = \emptyset \quad c \neq \emptyset \wedge f \neq \emptyset$
abc:de	ab:def	abc:def	$c \neq \emptyset \wedge f = \emptyset \quad c = \emptyset \wedge f \neq \emptyset$
abc:de	ab:defg	abc:defg	$c \neq \emptyset \wedge f = \emptyset \quad c = \emptyset \wedge f \neq \emptyset$
			$\wedge g = \emptyset \quad \wedge g \neq \emptyset$

**Definition 3.4 (n-split)** Given a set of triplegroups  $TG$ , a set of primary properties  $P_{prim}$ , and  $n$  sets of secondary properties  $\{P_{sec_1}, P_{sec_2}, \dots, P_{sec_n}\}$ , the **n-split** operator  $\chi$  creates a set of  $n$  triplegroups as follows:

$$\chi_{(P_{prim}, \{P_{sec_1}, P_{sec_2}, \dots, P_{sec_n}\})}(TG) := \{tg'_i, i \in [1, n]\}$$

such that:

- $tg'_i = tg_{prim} \cup tg_{sec_i}$ , where  $tg_{prim}, tg_{sec_i} \subseteq tg, tg \in TG$
- $props(tg_{prim}) = P_{prim}$  and  $props(tg_{sec_i}) = P_{sec_i}$

The n-split operator extracts  $n$  subsets of a triplegroup based on  $n$  sets of secondary properties, one for each of the original star patterns. Figure 4(b) shows triplegroups resulting from an n-split operation on  $TG'$  ( $n=2$ ), with  $P_{prim} = \{\text{product, price}\}$ , and two sets of secondary properties –  $P_{sec_1} = \{\text{validFrom}\}$ , and  $P_{sec_2} = \{\text{validTo}\}$ . While triplegroup  $tg_{41}$  conforms to the first pattern combination with properties  $\{\text{product, price, validFrom}\}$ , triplegroups  $tg_{12}$  and  $tg_{42}$  match the second combination  $\{\text{product, price, validTo}\}$ . Figure 4(c) shows another example of the n-split operation with  $P_{sec_1} = \{\}$  and  $P_{sec_2} = \{\text{validTo}\}$ , i.e., the first combination contains only primary (no secondary) properties.

Let  $GP_{abcde}$  and  $GP_{abdef}$  be original graph patterns in a query and let  $Stp_{abc}$  and  $Stp_{def}$  be composite star patterns. The join ( $Stp_{abc} \bowtie Stp_{def}$ ) may result in pattern combinations such as  $abde$  that do not match either of the original patterns and should be avoided. We encode valid pattern combinations using  $\alpha$  conditions, a set of structural constraints on a TG equivalence class based on its secondary properties. For example, to ensure pattern combinations  $abcde$ , triplegroups in  $TG_{abc}$  must contain at least one triple with property  $c$ , represented as a constraint  $\alpha: c \neq \emptyset$ , for brevity.

**Definition 3.5 ( $\alpha$ -Join)** Let  $TG_x$  and  $TG_y$  be two triplegroup equivalence classes that join on variables  $ju_x$  and  $ju_y$  belonging to joining triple patterns  $tp_x$  and  $tp_y$ , resp. Let  $\alpha_1, \alpha_2, \dots, \alpha_m$  be  $m$  conditions involving secondary properties in the equivalence classes. Then the  **$\alpha$ -Join** operator  $\bowtie_{\{\alpha_1 \vee \dots \vee \alpha_m\}}^{\gamma}$  creates a joined triplegroup involving  $tg_x \in TG_x$  and  $tg_y \in TG_y$  if the following holds:

- Triplegroup  $tg_x$  contains a matching triple for  $tp_x$ , and triplegroup  $tg_y$  contains a matching triple for  $tp_y$ , such that their variable substitutions match.
- $tg_x$  and  $tg_y$  satisfy at least one of the  $\alpha$  conditions.

Table 2 shows examples of graph patterns GP1 and GP2, their composite graph pattern GP', and  $\alpha$  constraints for the  $\alpha$ -Join operator. For example, conditions  $\alpha_1$  and  $\alpha_2$  in row (5) correspond to the original graph patterns  $abcde$  and  $abdefg$  respectively, hence avoiding materialization of triplegroups matching irrelevant patterns such as  $abde$ ,  $abdef$ ,  $abdeg$ ,  $abcdef$ ,  $abcdefg$ , etc.

$$\gamma^{AgJ}(TG_{base}, TG_{\{ty18, pf, pr, pc, ve, cn\}}, l, \theta, \alpha) = TG_{\{sumF, countF\}}$$

where  $l = \{\text{SUM}(\text{price}), \text{COUNT}(\text{price})\}$  and  $\alpha = \{pf \neq \emptyset\}$

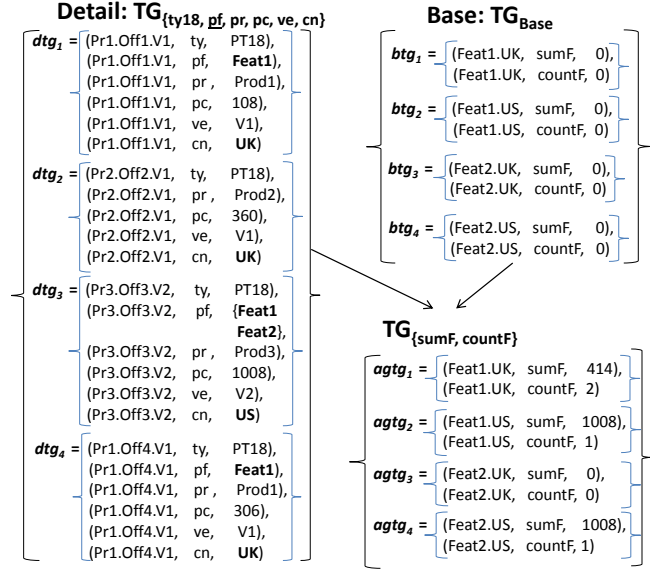


Figure 5: Example Triplegroup Agg-Join operation that computes groupings based on feature-country combination

**Definition 3.6 (TG Agg-Join)** Let  $TG_{base}$  and  $TG_{detail}$  be two triplegroup equivalence classes,  $\theta$  be a condition involving variable substitutions in  $TG_{base}$  and  $TG_{detail}$ , and let  $l$  be a list of aggregation functions ( $f_1, f_2, \dots, f_m$ ) over aggregation variables  $a_1, a_2, \dots, a_m$ , respectively. Let  $\alpha$  be a condition involving one of the secondary properties in  $TG_{detail}$ . Then the triplegroup Agg-Join operator,

$$\gamma^{AgJ}(TG_{base}, TG_{detail}, l, \theta, \alpha)$$

creates a set of aggregated triplegroups  $ATG$ , where any aggregated triplegroup  $agtg_i \in ATG$  satisfies the following conditions:

- Each base triplegroup  $btg_i \in TG_{base}$  is associated with a set of triplegroups in  $TG_{detail}$ , using the following function :

$$RNG(btg_i, TG_{detail}, \theta, \alpha) = \{dtg \in TG_{detail}\}$$

such that triplegroups  $btg_i$  and  $dtg$  satisfy conditions in  $\theta$  and  $\alpha$ .

- Then, for each base triplegroup  $btg_i \in TG_{base}$ , an aggregated triplegroup  $agtg_i \in ATG$  is produced with triples  $t_{ik} \in agtg_i$  that contain values corresponding to some aggregation function  $f_k$  and variable  $a_k$  such that :

$$t_{ik} = (grpKey, createProp(f_k, a_k), f_k\_agtg_i\_a_k)$$

whose values are computed as follows :

- $grpKey$  is the subject of  $btg_i$ ;  $createProp(f_k, a_k)$  returns a unique property based on combination of aggregation function and variable.
- Aggregate  $f_k\_agtg_i\_a_k$  is computed by applying the function  $f_k$  on variable substitutions of  $a_k$  in triplegroups matching  $RNG(btg_i, TG_{detail}, \theta, \alpha)$ .

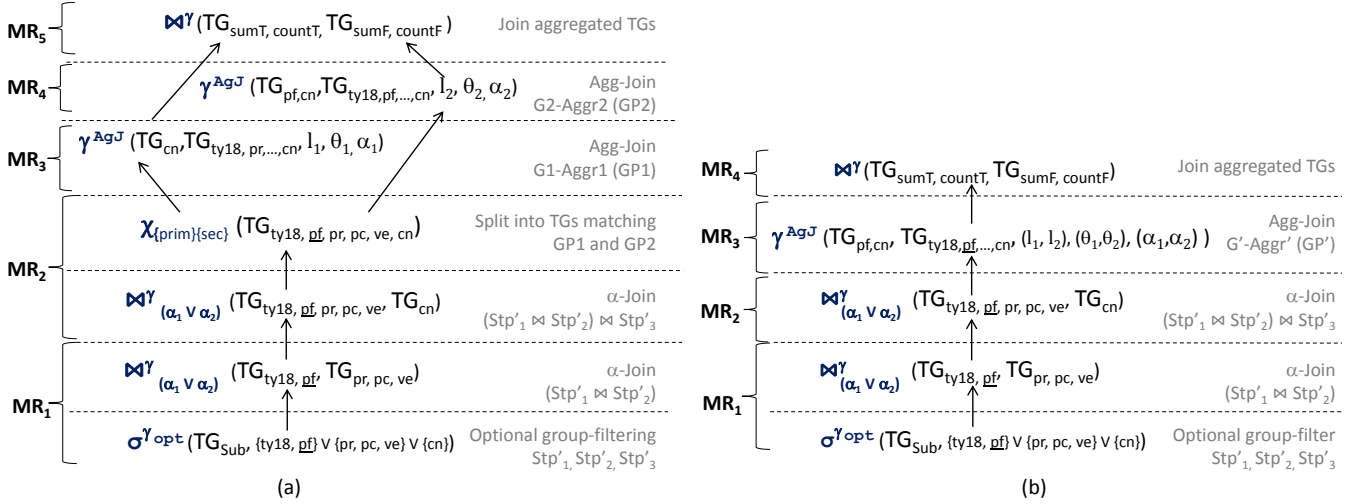


Figure 6: Translation to MapReduce execution workflows: (a) Sequential and (b) Parallel evaluation of aggregations on a composite graph pattern  $GP'$ . Properties:  $ty18$  (rdf : type PT18),  $pf$  (productFeature),  $pr$  (product),  $pc$  (price),  $ve$  (vendor),  $cn$  (country)

A base triplegroup  $btg_i \in TG_{base}$  corresponds to a distinct grouping key and produces an aggregated triplegroup  $agtg_i \in ATG$ . Subset of triplegroups in  $TG_{detail}$  that contribute to an aggregated triplegroup  $agtg_i$  is computed using function  $RNG(btg_i, TG_{detail}, \theta, \alpha)$ , that returns the set of triplegroups in  $TG_{detail}$  that satisfy the join condition  $\theta$  as well as the  $\alpha$  condition with respect to the base triplegroup  $btg_i$ . The  $\alpha$  condition defines restrictions based on secondary properties in  $TG_{detail}$ .

Figure 5 illustrates an example TG *Agg-Join* operation between TG equivalence classes  $TG_{base}$  (base) and  $TG_{\{ty18, pf, pr, pc, ve, cn\}}$  (detail), to compute groupings based on feature and country. The  $RNG$  of a base triplegroup is calculated based on value bindings of the grouping variables  $?feature$  and  $?country$  in detail triplegroups (encoded as join condition  $\theta$ ). For triplegroup  $dtg_1$ , bindings  $\delta_1(?feature)=\{Feat1\}$  and  $\delta_1(?country)=\{UK\}$ . The  $\alpha$  condition  $pf \neq \emptyset$  ensures the presence of the secondary property  $pf$  (product feature). Triplegroup  $dtg_2$  does not satisfy the  $\alpha$  condition and hence does not contribute to any of the aggregated triplegroups. The  $RNG$  of base triplegroups is as follows:

$$\begin{aligned} RNG(btg_1, TG_{\{ty18, pf, pr, pc, ve, cn\}}, \theta, \alpha) &= \{ dtg_1, dtg_4 \} \\ RNG(btg_2, TG_{\{ty18, pf, pr, pc, ve, cn\}}, \theta, \alpha) &= \{ dtg_3 \} \\ RNG(btg_3, TG_{\{ty18, pf, pr, pc, ve, cn\}}, \theta, \alpha) &= \emptyset \\ RNG(btg_4, TG_{\{ty18, pf, pr, pc, ve, cn\}}, \theta, \alpha) &= \{ dtg_3 \} \end{aligned}$$

Given a base triplegroup  $btg_i$ , the aggregated triplegroup is computed by aggregating triplegroups in  $RNG$  of  $btg_i$ . For example,  $agtg_1$  is an aggregation of triplegroups  $dtg_1$  and  $dtg_4$  ( $RNG$  of  $btg_1$ ). Note that  $RNG$  of  $btg_3$  is empty and the aggregated triplegroup  $agtg_3$  retains default values.

## 4. QUERY EXECUTION ON MAPREDUCE

In MapReduce, data processing tasks (or queries) are encoded as a sequence of map-reduce function pairs which are executed in parallel on a cluster of machines. Extended MapReduce systems such as Apache Hive and Pig support high-level query primitives that are automatically compiled into a MapReduce execution workflow. The proposed logical operators were integrated into an NTGA-based extension of Apache Pig, called *RAPID+* [33, 25]. The extended system, called *RAPIDAnalytics*, includes pro-

posed optimizations to evaluate multi-aggregation SPARQL analytical queries. Both systems parse graph pattern queries in SPARQL and support a set of logical and physical operators for both Pig and NTGA. Interested readers can refer to [25] for architectural details of *RAPID+*.

### 4.1 Translation to MapReduce Plans

As with other relational-style Hadoop extensions, query compilation process in *RAPIDAnalytics* begins with a logical plan, which is compiled into a physical plan with physical operators. A physical operator is either a single function or a function pair that corresponds to map and reduce phases of the logical operator. For example, the optional group-filtering operator  $TG_{OptGrpFilter}$  ( $\sigma^{opt}$ ) is a single function and can be pipelined with other operators in either the map or the reduce phases. However, operators such as the triplegroup *Agg-Join*  $TG_{AggJ}$  which require redistribution of input, are defined as map-reduce function pairs. The assignment of the physical operators to MapReduce cycles constitutes a MapReduce plan.

Next, we summarize the execution workflow of our example query  $AQ1$  on MapReduce. As described earlier, overlapping graph patterns  $GP1$  and  $GP2$  are re-written as a composite graph pattern:

$$GP': Stp_{ty18, pf} \bowtie Stp_{pr, pc, ve} \bowtie Stp_{cn}$$

Let  $TG_{Sub}$  be a set of subject triplegroups (set of triples grouped by Subject column). Figure 6(a) shows the query plan with the assignment of operators to map-reduce (MR) cycles. The optional group-filtering operator creates three sets of triplegroup equivalence classes –  $TG_{\{ty18, pf\}}$ ,  $TG_{\{pr, pc, ve\}}$ , and  $TG_{\{cn\}}$ , that match the composite star patterns. The two  $\alpha$ -Join operators compute the  $\alpha$ -join between triplegroups to compute matches to the composite graph pattern. The  $n$ -split operator extracts matches to the original graph patterns  $GP1$  and  $GP2$ . Subsequently, the two TG *Agg-Join* operators ( $\gamma^{AgJ}$ ) compute the aggregations per country and per feature-country, resp. The final ratio is computed by joining the aggregated TG equivalence classes using a map-only phase.

An useful optimization [10, 5] is that a series of aggregations on the same detail relation can be evaluated in parallel if they are independent, i.e., the  $\theta$  conditions of the second *Agg-Join* does not involve values generated by the first *Agg-Join*. Figure 6(b) shows the NTGA query plan and MapReduce execution plan that enables

parallel execution of the TG *Agg-Join* operator by combining them as a generalized operator (executed in MR cycle  $MR_3$ ):

$$\gamma^{AgJ}(TG_{g1}, TG_{\{ty18, pf, pr, pc, ne, cn\}}, (l_1, l_2), (\theta_1, \theta_2), (\alpha_1, \alpha_2))$$

## 4.2 Algorithms for Physical Operators

Algorithm 1 gives an overview of the job flow for key phases in *RAPIDAnalytics* –  $Job_i$ , that computes the join between the triplegroup equivalence classes, and  $Job_k$ , that computes the aggregate join between the triplegroup equivalence classes. If there is a structural overlap in the input graph patterns, the triplegroup equivalence classes are computed based on the *composite* graph pattern. This is achieved by evaluating the optional group-filtering operator, `TG_OptGrpFilter`, based on the required and optional properties in the composite graph pattern. Below are map-reduce algorithms for the physical operators.

---

### Algorithm 1: MR job workflow in *RAPIDAnalytics*

---

```
//Jobi: $\alpha$ -Join between TG equivalence classes
Map:
  TG'  $\leftarrow$  TG_OptGrpFilter(TG, <EC, {Pprim, Popt}>);
  TG_AlphaJoin(TG').Map();
Reduce:
  TG''  $\leftarrow$  TG_AlphaJoin(TG').Reduce();
//Jobk:Agg-Join on TG equivalence classes
Map:
  TG_AgJ(TG'').Map();
Reduce:
  AggTG  $\leftarrow$  TG_AgJ(TG'').Reduce();
//Jobn:Join Aggregated TGs
Map:
  TG_Join(AggTG);
```

---

**TG\_AlphaJoin:** The input to this operator is a set of annotated triplegroups (matching a composite subpattern) whose join is to be computed. In order to eliminate pattern combinations that do not match any of the original graph patterns, all valid combinations are encoded as a list of  $\alpha$  conditions, one for each of the original graph patterns. Algorithm 2 shows the map-reduce functions for the `TG_AlphaJoin` operator that integrates  $\alpha$ -based filtering of irrelevant triplegroups during the join between equivalence classes.

In the map phase, an input triplegroup is tagged either on the *Subject* or *Object* value, based on the type of join. Each `reduce()` receives annotated triplegroups corresponding to the same join key. The algorithm iterates through triplegroups in the left equivalence class (*leftEC*) and right equivalence class (*rightEC*), and computes the join only if at least one of the  $\alpha$  conditions is satisfied. For example, two triplegroups with properties *ab* and *de*, are not joined if the valid pattern combinations are *abcde* and *abdef*.

**TG\_AgJ:** The input to this operator is a set of annotated triplegroups that match the composite graph pattern. The output is a set of aggregated triplegroups that contain the required aggregations. Algorithm 3 shows the map-reduce functions for the `TG_AgJ` operator. In order to reduce the number of intermediate triplegroups that are shuffled to the reducers, we implement a hash-based aggregation per mapper, i.e., instead of generating map output for each map input triplegroup, we partially aggregate the triplegroups at each mapper. The triplegroups are aggregated into a hashmap *multiAggMap* that is accessible across different `map()` invocations at a mapper. This hash-based aggregation resembles a local combiner within each mapper.

Each *Agg-Join* *aggj* (identified by *id*) contains a  $\theta$  condition, from which the grouping key *grp* is extracted. In the map phase,

---

### Algorithm 2: TG\_AlphaJoin (Triplegroup $\alpha$ -Join)

---

```
Map (key:null, val: AnnTG atg)
  if join on Subj then
    emit < atg.Sub, atg >;
  else if join on Obj then
    objList  $\leftarrow$  extract objects corr. to join property from atg;
    foreach obj  $\in$  objList do
      emit < obj, atg >;
Reduce (key:joinKey, val:List of AnnTGs TG');
   $\alpha$ List  $\leftarrow$   $\langle \alpha_1, \dots, \alpha_n \rangle$   $\leftarrow$   $\alpha$  restrictions for current join;
  leftList  $\leftarrow$  extract leftEC AnnTGs from TG';
  rightList  $\leftarrow$  extract rightEC AnnTGs from TG';
  foreach ltg  $\in$  leftList do
    foreach rtg  $\in$  rightList do
      if  $\exists \alpha \in \alpha$ List such that ltg and rtg satisfy  $\alpha$  then
        emit < joinTGs(ltg, rtg) >;
```

---

as each input triplegroup *atg* is processed, aggregations are computed if the  $\alpha$  condition is satisfied. Once all aggregations for *aggj* are computed, triplegroup *currAggTg* is aggregated with existing values in the mapper's global hashmap *multiAggMap*. Once the `map()` functions are complete, pre-aggregated entries in the global hashmap *multiAggMap* are output. Each `reduce()` receives pre-aggregated triplegroups corresponding to the same *id-grp* combination and further aggregates them.

---

### Algorithm 3: TG\_AgJ (Triplegroup Agg-Join)

---

```
Map (k:null, v: AnnTG atg)
  //Initialize multiAggMap for Map()
  //aggregation
  foreach aggj < id, aggList, theta, alpha >  $\in$  aggList do
    if atg satisfies alpha then
      grp  $\leftarrow$  extract aggj.theta from atg;
      curAggTg  $\leftarrow$  Aggregate atg based on aggList;
      Aggregate curAggTg to multiAggMap(k:id#grp);
Map.clean ()
  Emit pre-aggregated entries in multiAggMap;
Reduce (k:id#grp, v:List of AggTGs TG);
  grpAggTg  $\leftarrow$  Aggregate TG based on aggList;
  Emit aggregated triplegroup grpAggTg;
```

---

## 5. EMPIRICAL EVALUATION

This section presents a comprehensive evaluation of the proposed algebraic optimizations for RDF analytical queries. The performance of *RAPIDAnalytics* with two Hive approaches, (i) *Hive (Naive)*, SPARQL query translated into HiveQL, and (ii) *Hive (MQO)*, an MQO-based rewriting [27] of graph patterns using left outer joins, followed by a second HiveQL query to compute associated grouping and aggregations. Evaluation also included *RAPID+ (Naive)* [25], NTGA-based sequential evaluation of multiple graph patterns and grouping-aggregation phases.

### 5.1 Setup

Experiments were conducted on NCSU's VCL [34], where each node in the cluster was a dual core Intel X86 machine with 2.33GHz processor speed, 4GB memory, running Red Hat Linux. 10, 50, and 60-node Hadoop clusters (block size 128MB, 1GB heap-size for child jvms) were used with Hive release 0.12.0 and Hadoop 0.20.2.

**Testbed - Dataset and Queries.** Two synthetic datasets were generated by the Berlin SPARQL Benchmark (BSBM) [1] data

Query	GP1*	Group BY	GP2*	Group BY
MG1:lo, MG2:hi	3:2	{feature}	2:2	ALL
MG3:lo, MG4:hi	3:3:1	{feature, country}	2:3:1	{country}
MG6	4:2:2	{cid, gene}	4:2:2	{cid}
MG7	4:2:2	{cid, drug}	4:2:2	{cid}
MG8	4:2:2	{cid, gene}	4:2:2	ALL
MG9	2:1	{gene}	2:1	ALL
MG10	3:1	{disease, gene}	2:1	{gene}
MG11	2:2	{country}	2:1	ALL
MG12	2:2	{country, pubType}	2:1	{country}
MG13	3:1	{author, pubType}	3:1	{pubType}
MG14	3:1	{author, pubType}	3:1	{pubType}
MG15:lo	3:1	{authorlastname}	3:1	ALL
MG16:hi	3:1	{authorlastname}	3:1	ALL
MG17	3:2	{country}	3:1	ALL
MG18	3:2	{author, country}	2:2	{country}

\* No. of triple patterns in  $Stp_1 : Stp_2 : \dots$

Figure 7: Evaluated RDF Analytical Queries

generator tool – *BSBM-500K* (43GB, 500K Products,  $\sim 175M$  triples) and *BSBM-2M* (172GB, 2M Products,  $\sim 700M$  triples). Evaluation of real-world RDF analytical queries was conducted on a chemogenomics RDF data warehouse, *Chem2Bio2RDF* [13], that is an aggregation of data from multiple chemical, biological, and chemogenomics data sources that link chemical compounds with targets, genes, side-effects, diseases, and publications (60GB,  $\sim 340M$  triples). Additional experiments were conducted on a second real-world dataset, *PubMed* (*Bio2RDF* release 2) [8] (230GB,  $\sim 1.7B$  triples).

The evaluation tested simple ( $G1-G9$ ) as well as multi-grouping queries ( $MG1-MG18$ ) with varying selectivities, varying granularity of groupings (GROUP BY ALL vs. GROUP BY feature), and varying structures of associated graph patterns, as summarized in Figure 7. Queries  $G1-G4$  and  $MG1-MG4$  were adapted from the *BSBM Business Intelligence Use Case 3.1* [1], an e-commerce use case. Queries  $G5-G9$  and  $MG6-MG10$  were adapted based on case studies [13] on the *Chem2Bio2RDF* dataset, with use cases such as disease-specific drug discovery. Queries  $MG11-MG18$  involve *PubMed* records. Additional details about all evaluated queries in SPARQL and Hive scripts are available on the project website [2].

**Pre-processing.** For Hive approaches, triples were vertically partitioned (VP) [3] and loaded into Hive tables with property-object partitions for *rdf:type* triples. All Hive tables were stored as *Optimized Row Columnar* (ORC)<sup>5</sup> file format which aggressively compresses data ( $\sim 80-96\%$  reduction in data size with default compression) and has optimizations such as light-weight indexes to skip row groups for predicate-based filtering, column-level aggregates etc. For *RAPIDAnalytics* and *RAPID+*, triples were grouped on subject column to generate subject triplegroups, stored in text files based on equivalence class (set of properties). Further, *rdf:type* triples with *ProductType* objects were grouped based on prefixes to avoid creation of multiple small files. Additional details about the

<sup>5</sup><https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC>

pre-processing phase is available on the project website [2].

Query	BSBM				Chem2Bio2RDF		
	500K		2M		Hive	R.A.	
	Hive	R.A.	Hive	R.A.			
G1:lo	1023	209	3261	215	G5	144	124
G2:hi	974	182	3002	158	G6	99	102
G3:lo	1632	287	6088	302	G7	105	118
G4:hi	1112	183	5419	170	G8	142	104
					G9	535	91

Table 3: Performance comparison of *Hive* and *RAPIDAnalytics* (R.A.) with varying structures of groupings (in seconds)

## 5.2 Evaluation Results

**Varying Structure of Groupings.** Four single-grouping queries were evaluated with varying selectivity of graph patterns and group granularity ( $G1-G2$  with GROUP BY ALL and  $G3-G4$  with GROUP BY feature). Queries  $G1$  and  $G3$  pertain to *ProductType1* (low selectivity), while  $G2$  and  $G4$  pertain to *ProductType9* (high selectivity). Table 3 shows a performance comparison of *Hive* and *RAPIDAnalytics* for *BSBM-500K* (10-node cluster). *Hive* requires 4 MR cycles for all queries ( $MR_1-MR_2$  for star patterns,  $MR_3$  to join the stars, and  $MR_4$  to compute grouping-aggregation). In cases where  $(n-1)$  of the joining relations are small enough to fit in memory, *Hive* uses a map-join (map-only MR cycle), e.g., all subqueries involving *ProductType1* and *ProductType9*. Also *Hive* enables optimizations such as push down of *PROJECTS* and partial aggregation during preceding join operations. *RAPIDAnalytics* executes all four queries in 2 cycles ( $MR_1$  for graph pattern processing and  $MR_2$  for the *Agg-Join* operation), with a consistent performance gain of  $\sim 80\%$  over *Hive* for all four queries.

**Multiple Grouping-Aggregation Constraints.** Figure 8(a-b) shows a performance comparison of all four approaches for queries  $MG1-MG4$  with lo (low) and hi (high) query selectivity. Queries  $MG1-MG2$  require 3 MR cycles per graph pattern in *Hive*, followed by 2 cycles for the grouping-aggregation (total 9 cycles). MQO-based *Hive* approach executes the composite graph pattern in 3 cycles, followed by 4 MR cycles to extract the distinct combinations matching the original patterns and compute the aggregations (total 7 cycles). *RAPID+* requires 2 MR cycles per subquery (1 MR for graph pattern matching, 1 MR for grouping-aggregation) and a map-only cycle to join the aggregated results (total 5 MR cycles). *RAPIDAnalytics* evaluated  $MG1-MG2$  in 3 cycles ( $MR_1$  to compute the composite graph pattern,  $MR_2$  for parallel evaluation of the two grouping-aggregations and a map-only  $MR_3$  to join the aggregated triplegroups).

Queries  $MG3-MG4$  involve complex graph patterns with 3 star patterns. Sequential graph pattern processing in naive *Hive* results in a total of 11 MR cycles, while MQO-based *Hive* approach takes half the number of cycles for evaluating the composite graph pattern (8 MR cycles). *RAPID+* requires 2 MR cycles per graph pattern (7 MR cycles), while *RAPIDAnalytics* further reduces the number of cycles to 4 by parallel evaluation of the two grouping-aggregations. In general, the algebraic optimization in *RAPIDAnalytics* to group and aggregate on a composite graph pattern showed 30-45% gains over sequential evaluation of the different phases using naive *RAPID+*.

**Scalability Study.** Table 3 and Figure 8(b) show performance comparisons of 8 queries on a larger dataset *BSBM-2M*. The compression of input and intermediate results using the ORC File format, initializes less number of mappers (incur the overhead of decompression). *RAPID+* and *RAPIDAnalytics* initiate more number of mappers for most MR cycles leading to better utilization of re-

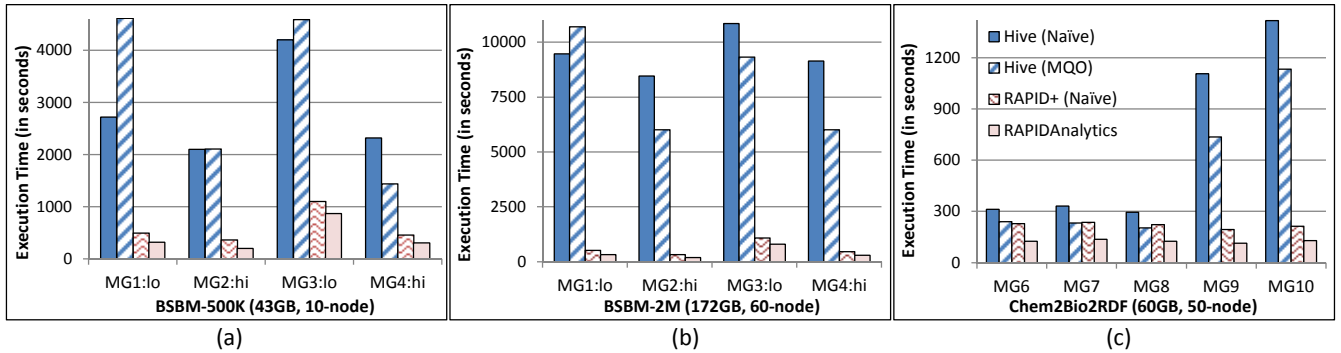


Figure 8: A performance comparison for multi-grouping SPARQL analytical queries

sources. For multi-grouping queries, *Hive (MQO)* did better than *Hive* for most cases with larger dataset due to higher savings in materialization of intermediate results, associated I/Os, and network transfers. *RAPIDAnalytics* showed 90-93% performance gains over *Hive (MQO)* for queries *MG1-MG2* on *BSBM-500K*, which further increased to 97% with *BSBM-2M*. Similar increase was seen for queries *MG3-MG4*, where performance gains of *RAPIDAnalytics* over *Hive (MQO)* increased from 78-81% to 93% with the larger setup.

**Real-world RDF Analytics.** Table 3 shows results for queries *G5-G9* on *Chem2Bio2RDF*. Query *G5* with 6 join operations was evaluated by *Hive* using map-only joins (due to small size VP tables). Similar optimizations were enabled by *Hive* for *G6-G8*, with clear benefits seen in the case of *G7*, where *RAPIDAnalytics* takes 12 additional seconds when compared to *Hive*. Query *G9* involves medline properties with large VP tables, forcing *Hive* to use full map-reduce cycles. *RAPIDAnalytics* shows 83% performance gain over *Hive* for *G9*. Figure 8(c) shows results for multi-aggregation queries, i.e., *MG6-MG8* with high selectivity (small VP relations), while queries *MG9-MG10* involve large VP relations. Naive *Hive* evaluates query *MG6* using 13 MR cycles (11 map-only), while MQO-based *Hive* approach requires 8 MR cycles (6 map-only). *RAPID+* evaluates *MG6* using 7 MR cycles (all map-reduce), with execution times almost comparable with *Hive (MQO)*. *RAPIDAnalytics* requires a total of 4 MR cycles. In general, even though the *Hive*-based approaches evaluate most of the joins in *MG6 - MG8* as map-joins, *RAPIDAnalytics* shows a performance gain of 40-50% over *Hive (MQO)* and 60% gains over naive *Hive* for queries *MG6-MG8*. In case of queries *MG9-MG10*, the findings are similar to *BSBM* datasets, with *RAPIDAnalytics* showing close to 90% performance gain over *Hive* approaches.

Results for the *Pubmed* dataset are summarized in Table 4. Queries *MG11 - MG12* and *MG17 - MG18* compute groupings over *PubMed* records, the associated grants, and the countries where the grants are issued. Queries *MG13-MG16* compute groupings based on publication type and authors of *PubMed* records and aggregate the number of Medical Subject (MeSH) Headings (query *MG13*) or associated chemicals (queries *MG14-MG16*). Further, selectivity of the queries were varied by querying different types of publications, e.g., *MG15* and *MG16* have similar query structure except that *MG15* retrieves *PubMed* records with publication type “Journal Article” while *MG16* concerns publications of type “News” (higher selectivity than journal article). Across all queries, *RAPIDAnalytics* showed improvements of above 93% over both *Hive* approaches. *Hive* performed the worst for queries

Query	PubMed (230GB dataset, 60-node cluster)			
	Hive (Naive)	Hive (MQO)	RAPID+ (Naive)	RAPID Analytics
MG11	2111	1753	229	124
MG12	2771	2898	229	126
MG13	120min*	15060	1102	651
MG14	18713	9124	756	462
MG15	13746	7320	619	338
MG16	10777	5795	464	237
MG17	2210	1851	226	118
MG18	5654	4817	306	202

\* Eventually failed due to insufficient HDFS disk space.

Table 4: Evaluation of real-world queries on *PubMed* dataset (execution time in seconds)

*MG13-MG16* that involve large VP relations (MeSH heading and chemical), due to the initiation of less number of mappers based on compressed (ORC) file sizes. Furthermore, while the *Hive MQO* approach eventually finished execution for query *MG13*, the naive *Hive* approach failed while computing the second graph pattern due to insufficient disk space. This is because one of the star-join cycles produces join output of size 190GB, which is materialized twice in the case of sequential execution of graph patterns, thus increasing the overall demand of required HDFS disk space. On the contrary, *RAPIDAnalytics* benefits from the concise representation of intermediate results using the NTGA approach while representing join results involving the multi-valued property MeSH heading. Further, the shared execution of graph patterns in *RAPIDAnalytics*, results in less number of materialization steps and less demand on required disk space. Overall, *RAPIDAnalytics* resulted in 40-48% performance gains over the sequential execution of graph patterns in *RAPID+*.

**Discussion.** Though *Hive(MQO)* compiles into a shorter execution workflow when compared to naive *Hive*, in some cases the performance is worse than sequential execution of subqueries. This is because of *Hive*’s lack of support for materialized views or views with complex join expressions, forcing the evaluation of the composite graph pattern as a separate *HiveQL* query. A direct implication of this is that optimizations based on the final query such as early projections and partial aggregations, which reduce the I/O and materialization in the intermediate phases, are not applicable. Another observation is that vertical-partitioning coupled with the ORC file format can be beneficial for queries that involve high-selectivity properties. Irrespective of the selectivity of the involved properties,

the algebraic optimization techniques in *RDFAnalytics* were found to be beneficial for multi-grouping queries by enabling shared execution of graph patterns as well as the required aggregations. *RAPIDAnalytics* can further benefit by integration of optimizations such as map-side joins and partial aggregations. While SPARQL analytical queries with unbound properties were not considered in this work, proposed optimizations in this paper can be extended based on NTGA-based optimizations in [32] to support composite graph patterns involving unbound-property triple patterns.

## 6. CONCLUSION AND FUTURE WORK

In this paper, we presented an algebraic optimization of SPARQL analytical queries that enables shared execution of common subexpressions across related groupings. Such a refactoring allows parallel evaluation of independent aggregations with savings in I/O and processing costs, a critical requirement while supporting large scale RDF analytics on Cloud platforms. Experiments on real-world and synthetic benchmark datasets showed promising results for SPARQL queries with multi-aggregation constraints. A natural extension of this work is to support more complex OLAP queries on RDF data models.

## 7. ACKNOWLEDGMENTS

The work presented in this paper is partially funded by NSF grant IIS-1218277.

## 8. REFERENCES

- [1] BSBM Business Intelligence 3.1. <http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/BusinessIntelligenceUseCase/>.
- [2] Project Website: RAPIDAnalytics. <http://research.csc.ncsu.edu/coul/RAPID/RAPIDAnalytics>.
- [3] D.J. Abadi, A. Marcus, S.R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB Endowment*, 2007.
- [4] F.N. Afrati and J.D. Ullman. Optimizing joins in a map-reduce environment. In *ACM EDBT*, 2010.
- [5] Michael O Akinde and Michael H Böhlen. Generalized md-joins: Evaluation and reduction to sql. In *Databases in Telecommunications II*. 2001.
- [6] Michael O Akinde, Michael H Böhlen, Theodore Johnson, Laks VS Lakshmanan, and Divesh Srivastava. Efficient olap query processing in distributed data warehouses. *Information Systems*, 28(1), 2003.
- [7] Jens Albrecht and Wolfgang Lehner. On-line analytical processing in distributed data warehouses. In *IEEE IDEAS*, 1998.
- [8] François Belleau, Marc-Alexandre Nolin, Nicole Tourigny, Philippe Rigault, and Jean Morissette. Bio2rdf: towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics*, 41(5), 2008.
- [9] Don Chamberlin. *Using the new DB2: IBM's object-relational database system*. 1996.
- [10] D. Chatziantoniou, T. Johnson, M. Akinde, and S. Kim. The md-join: An operator for complex olap. In *IEEE ICDE*, 2001.
- [11] Damianos Chatziantoniou and Elias Tzortzakakis. Asset queries: a declarative alternative to mapreduce. *ACM SIGMOD Record*, 38(2), 2009.
- [12] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *ACM Sigmod record*, 26(1), 1997.
- [13] Bin Chen, Xiao Dong, Dazhi Jiao, Huijun Wang, Qian Zhu, Ying Ding, and David J Wild. Chem2bio2rdf: a semantic framework for linking and data mining chemogenomic and systems chemical biology data. *BMC bioinformatics*, 11(1), 2010.
- [14] Lei Chen, Christopher Olston, and Raghu Ramakrishnan. Parallel evaluation of composite aggregate queries. In *IEEE ICDE*, 2008.
- [15] Dario Colazzo, François Goasdoué, Ioana Manolescu, and Alexandra Roatis. RDF Analytics: Lenses over Semantic Graphs. In *Proc. WWW*, 2014.
- [16] Richard Cyganiak, Dave Reynolds, and Jeni Tennison. The rdf data cube vocabulary. *W3C Recomm.*, 2013.
- [17] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.
- [18] Lorena Etcheverry and Alejandro A Vaisman. Enhancing olap analysis with web cubes. In *The Semantic Web: Research and Applications*. 2012.
- [19] Goetz Graefe, Usama M Fayyad, Surajit Chaudhuri, et al. On the efficient gathering of sufficient statistics for classification from large sql databases. In *KDD*, 1998.
- [20] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, 1996.
- [21] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. Implementing data cubes efficiently. *ACM SIGMOD Record*, 25(2), 1996.
- [22] Steve Harris and Andy Seaborne. Sparql 1.1 query language. *W3C Recomm.*, 21, 2013.
- [23] J. Huang, D.J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *VLDB Endowment*, 4(11), 2011.
- [24] Benedikt Kampgen, Sean ORiain, and Andreas Harth. Interacting with statistical linked data via olap operations. In *Interacting with Linked Data*, 2012.
- [25] H.S. Kim, P. Ravindra, and K. Anyanwu. From sparql to mapreduce: The journey using a nested triplegroup algebra. *VLDB Endowment*, 4(12), 2011.
- [26] Hugo YK Lam, Luis Marenco, Tim Clark, et al. Alzpharm: integration of neurodegeneration data using rdf. *BMC bioinformatics*, 8(3), 2007.
- [27] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable multi-query optimization for sparql. In *IEEE ICDE*, 2012.
- [28] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *IEEE ICDCS*, 2011.
- [29] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Distributed cube materialization on holistic measures. In *IEEE ICDE*, 2011.
- [30] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *VLDB Endowment*, 3(1-2), 2010.
- [31] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. *ACM Sigmod Record*, 26(2), 1997.
- [32] P. Ravindra and K. Anyanwu. Scaling unbound-property queries on big RDF data warehouses using mapreduce. In *EDBT*, 2015.
- [33] P. Ravindra, H.S. Kim, and K. Anyanwu. An intermediate algebra for optimizing rdf graph pattern matching on mapreduce. *The Semantic Web: Research and Applications*, 2011.
- [34] H.E. Schaffer, S.F. Averitt, M.I. Hoit, A. Peeler, E.D. Sills, and M.A. Vouk. Ncsu's virtual computing lab: a cloud computing solution. *Computer*, 42(7), 2009.
- [35] Ambuj Shatdal and Jeffrey F. Naughton. Adaptive parallel aggregation algorithms. In *ACM SIGMOD*, pages 104–114, 1995.
- [36] R. Sridhar, P. Ravindra, and K. Anyanwu. Rapid: Enabling scalable ad-hoc analytics on the semantic web. *The Semantic Web-ISWC*, 2009.
- [37] Ming-Chuan Wu and Alejandro P Buchmann. Encoded bitmap indexing for data warehouses. In *IEEE ICDE*, 1998.
- [38] Amrapali Zaveri, Ricardo Pietrobon, Soren Auer, Jens Lehmann, Michael Martin, and Timofey Ermilov. Redd-observatory: Using the web of data for evaluating the research-disease disparity. In *IEEE/WIC/ACM WI-IAT*, 2011.

## APPENDIX

### A. SPARQL ANALYTICAL QUERIES

In this section, we provide a subset of evaluated SPARQL analytical queries with multiple grouping-aggregation constraints. The complete set of evaluated queries and Hive scripts are available on the project website [2].

**G5.** Retrieve drug-like compounds in PubChem that share common targets with Dexamethasone in the DrugBank (count targets per compound).

```
SELECT ?cid (COUNT(?cid) as ?active_assays {
  ?b CID ?cid; outcome ?a; Score ?s1; gi ?gi .
  ?u gi ?gi; geneSymbol ?g .
  ?di gene ?g; DBID ?dr .
  ?dr Generic_Name "Dexamethasone" .
} GROUP BY ?cid
```

**G6.** Retrieve compounds in PubChem that are active towards targets in a given pathway (MAPK signalling pathway) in KEGG pathway dataset.

```
SELECT ?cid (COUNT(?cid) as ?active_assays) {
  ?b CID ?cid; outcome ?a; Score ?s1; gi ?gi .
  ?u gi ?gi .
  ?pathway protein ?u; Pathway_name ?pname .
  FILTER regex(?pname,"MAPK signaling pathway","i")
} GROUP BY ?cid
```

**G7.** Retrieve pathways in the KEGG dataset that contain targets with drugs associated with hepatotoxicity (analyse side-effect hepatomegaly).

```
SELECT ?pid (COUNT(?pid) as ?count) {
  ?sider side_effect ?se; cid ?cid .
  FILTER regex(?se,"hepatomegaly","i")
  ?dr CID ?cid .
  ?target DBID ?dr; SwissProt_ID ?u .
  ?pathway kegg:protein ?u; pathwayid ?pid .
} GROUP BY ?pid
```

**MG1.** Compare the average price of products per feature vs. price across all features (ProductType1).

```
SELECT ?f ?sumF ?cntF ?sumT ?cntT {
  { SELECT ?f ?c (COUNT(?pr2) ?cntF) (SUM(?pr2) ?sumF)
    {?p2 type ProductType1; label ?l2; productFeature ?f.
     ?off2 product ?p2; price ?pr2 .
    } GROUP BY ?f
  }
  { SELECT (COUNT(?pr) As ?cntT) (SUM(?pr) As ?sumT)
    {?p1 type ProductType1; label ?l1 .
     ?off1 product ?p1; price ?pr .
    } } }
```

**MG3.** Compare the average price of products per country-feature vs. price per country across all features (for products of type ProductType1).

```
SELECT ?f ?c ?sumF ?cntF ?sumT ?cntT {
  { SELECT ?f ?c (COUNT(?pr2) ?cntF) (SUM(?pr2) ?sumF)
    {?p2 type ProductType1; label ?l2; productFeature ?f.
     ?off2 product ?p2; price ?pr2; vendor ?v2 .
     ?v2 country ?c .
    } GROUP BY ?f ?c
  }
  { SELECT ?c (COUNT(?pr) As ?cntT) (SUM(?pr) As ?sumT)
    {?p1 type ProductType1; label ?l1 .
     ?off1 product ?p1; price ?pr; vendor ?v1 .
     ?v1 country ?c .
    } GROUP BY ?c
  } } }
```

**MG6.** Compare the count of targets for a chemical compound and gene combination vs. targets per compound (across all genes).

```
SELECT ?cid ?g1 ?aPerCG ?aPerC {
  { SELECT ?cid ?g1 (COUNT(?cid) as ?aPerCD)
    {?b1 CID ?cid; outcome ?a1; Score ?s1; gi ?gil .
     ?u1 gi ?gil; geneSymbol ?g1 .
     ?d1l gene ?g1; DBID ?drl .
    } GROUP BY ?cid ?g1
  }
  { SELECT ?cid (COUNT(?cid) as ?aPerG)
    {?b CID ?cid; outcome ?a; Score ?s; gi ?gi .
     ?u gi ?gi; geneSymbol ?g .
     ?di gene ?g; DBID ?dr .
    } GROUP BY ?cid
  } } }
```

**MG9.** Compare no. of medline publications per gene vs. total count.

```
SELECT ?gs ?pPerGene ?pT {
  { SELECT ?gs (COUNT(?gs) as ?pPerGene)
    {?g geneSymbol ?gs .
     ?pmid gene ?g; side_effect ?se .
    } GROUP BY ?gs
  }
  { SELECT (COUNT(?gs1) as ?pT)
    {?g1 geneSymbol ?gs1 .
     ?pmid1 gene ?g1; side_effect ?se1 .
    } } }
```

**MG11.** Compare the count of journals funded by grant agencies of a country with the total count of journals published.

```
SELECT ?c ?cntC ?cntT {
  { SELECT ? (COUNT(?g) as ?cntC)
    {?pub journal ?j; grant ?g .
     ?g grant_agency ?ga; grant_country ?c .
    } GROUP BY ?c
  }
  { SELECT (COUNT(?g1) as ?cntT)
    {?pub1 journal ?j1; grant ?g1 .
     ?g1 grant_agency ?gal .
    } } }
```

**MG13.** Compare the number of medical subject headings (MeSH) associated per author and publication type with total MeSH per publication type.

```
SELECT ?a ?pty ?perPT ?perAPT {
  { SELECT ?a ?pty (count(?m) as ?perAPT)
    {?pub pub_type ?pty; mesh_heading ?m; author ?a .
     ?a last_name ?ln .
    } GROUP BY ?a ?pty
  }
  { SELECT ?pty (count(?m1) as ?perPT)
    {?p1 pub_type ?pty; mesh_heading ?m1; author ?a1 .
     ?a1 last_name ?ln1 .
    } GROUP BY ?pty
  } } }
```

**MG16.** Compare the number of compounds associated with publications of type "News" (higher selectivity than Journal Articles).

```
SELECT ?ln ?perA ?allA {
  { SELECT ?ln (count(?chem) as ?perA)
    {?pub pub_type "News"; chemical ?ch; author ?a .
     ?a last_name ?ln .
    } GROUP BY ?ln
  }
  { SELECT (count(?chem1) as ?allA)
    {?pub1 pub_type "News"; chemical ?ch1; author ?a1 .
     ?a1 last_name ?ln1 .
    } } }
```

**MG18.** Count journal articles per author and grant-awarding country and compare with total journal articles per country (across authors).

```
SELECT ?c ?a ?perC ?perAC {
  { SELECT ?c ?a (count(?g) as ?perAC)
    {?p pub_type "Journal Article"; author ?a; grant ?g.
     ?g grant_agency ?ga; grant_country ?c .
    } GROUP BY ?c ?a
  }
  { SELECT ?c (count(?g1) as ?perC)
    {?pub1 pub_type "Journal Article"; grant ?g1 .
     ?g1 grant_agency ?gal; grant_country ?c .
    } GROUP BY ?c
  } } }
```



# RDF-TX: A Fast, User-Friendly System for Querying the History of RDF Knowledge Bases

Shi Gao Jiaqi Gu Carlo Zaniolo  
 University of California, Los Angeles  
 {gaoshi, gujiaqi, zaniolo}@cs.ucla.edu

## ABSTRACT

Knowledge bases that summarize web information in RDF triples deliver many benefits, including providing access to encyclopedic knowledge via SPARQL queries and end-user interfaces. As the real world evolves, the knowledge base is updated and the evolution history of entities and their properties becomes of great interest to users. Thus, users need query tools of comparable power and usability to explore such evolution histories or flash-back to the past. An integrated system that supports user-friendly queries and efficient query evaluation on the history of knowledge bases is required. In this paper, we introduce (i) SPARQL<sup>T</sup>, a temporal extension of SPARQL that expresses powerful structured queries on temporal RDF graphs, (ii) an efficient in-memory query engine that takes advantage of compressed multiversion B+ trees to achieve fast evaluation of SPARQL<sup>T</sup> queries, and (iii) a query optimizer that improves selectivity estimation of temporal queries and generates efficient join orders using the statistics of temporal RDF graphs. The performance and scalability of our system are validated by extensive experiments on real world datasets, which shows significant performance improvement comparing with other approaches.

## 1. INTRODUCTION

Knowledge bases that summarize valuable information in the RDF format are rapidly growing in terms of scale and significance and playing a crucial role in many applications such as semantic search and question answering. The extraordinary success of crowdsourcing and text mining for knowledge discovery makes it easy to generate and update the information in the knowledge bases. In fact, large knowledge bases undergo frequent changes. Table 1 lists the statistics of Wikipedia Infobox edit history, which shows that updates are quite common in many properties: e.g., on average each value in the population property of the city pages is updated more than 7 times. This is not specific to Wikipedia, but also happens in other knowledge repositories.

The management of historical information has emerged as a critical issue for knowledge bases. In fact, timestamping is an important part of the provenance information that is associated with each RDF triple in the knowledge base. The evolution history of knowl-

Category	Property	Average Number of Updates
Software	Release	7.27
Player	Club	5.85
Country	GDP(PPP)	11.78
City	Population	7.16

Table 1: Statistics of Wikipedia Infobox Edit History

edge bases captures and describes the change of real world entities and properties, and thus is of great interest to users. However, the size of the history is very large and the schema of knowledge base is also under evolution, which presents challenges in query language, query processing and indexing.

As the RDF model for representing knowledge bases is gaining great popularity, the importance of managing and querying the evolution history of knowledge bases is also recognized. Gutierrez et al. [17] extended the RDF model with time elements and several approaches [16, 29, 30, 32] have been proposed to support the queries on temporal RDF datasets. Most previous works employ relational databases and RDF engines to store temporal RDF triples and rewrite temporal queries into SQL/SPARQL for evaluation. The languages proposed in these works use an interval-based temporal model which leads to complex expressions for temporal queries, e.g., those requiring joins and coalescing [12, 33]. At the physical level, previous approaches exploit indexes such as tGrin [30] to accelerate the processing of simple temporal queries, but they do not explore the use of general temporal indices and query optimization techniques. This limits their scalability and performance on large knowledge bases and for complex queries.

In this paper, we describe a vertically integrated system RDF-TX (RDF Temporal eXpress) that efficiently supports the data management and query evaluation of large temporal RDF datasets while simplifying the temporal queries for SPARQL programmers and consequently, for end-user interfaces facilitating the expression of the same queries. To support the queries over the evolution history of knowledge bases, we propose efficient storage and index schemes for temporal RDF triples using multiversion B+ tree [7] and implement a query engine which achieves fast query evaluation by taking advantage of comprehensive indices. We also build a query optimizer that generates efficient join orders using a cost-based model and the statistics of temporal RDF graphs.

We propose a general and scalable solution for the problem of managing and querying massive temporal RDF data based on three main contributions:

- We propose SPARQL<sup>T</sup>, a temporal extension of the structured query language SPARQL based on a point-based temporal model which simplifies the expression of temporal joins and eliminates the need for temporal coalescing. This approach makes possible end-user interfaces, such as those in [6,

15], where queries are entered via simple by-example conditions in the infoboxes of Wikipedia pages.

- We present an efficient main memory system RDF-TX for managing temporal RDF data and evaluating SPARQL<sup>T</sup> queries. Our system uses multiversion B+ tree (MVBT) to store and index temporal RDF triples. An effective delta encoding scheme is introduced to reduce the storage overhead of indices. The algorithms on MVBT are extended and optimized to exploit the characteristics of the compression scheme and query patterns. Experimental evaluation demonstrates superior performance and scalability of RDF-TX compared with other approaches.
- We implement a query optimizer that generates the efficient join orders of SPARQL<sup>T</sup> query patterns using the statistics of temporal RDF graphs. To manage temporal statistics, we introduce compressed Multi-Version SB Trees (MVSBT) that provide highly accurate estimation of statistics with a small storage overhead.

The rest of this paper is organized as follows. Section 2 provides an overview of RDF-TX system and temporal RDF model. Then we present the SPARQL<sup>T</sup> query language in Section 3. Section 4 describes our storage model and index compression techniques. The query evaluation techniques are discussed in Section 5. Section 6 introduces a query optimizer for join order optimization. We evaluate our system on real world datasets in Section 7, and discuss related work in Section 8. Finally, we conclude in Section 9.

## 2. OVERVIEW AND DATA MODEL

In this section, we discuss the challenges of supporting temporal queries against the history of knowledge bases and provide a general overview of the RDF-TX system. Then we review the temporal RDF model introduced in [17].

### 2.1 Overview

The addition of temporal information to the basic RDF model poses difficult challenges that parallel those encountered by researchers working on extending the relational model with temporal information. A first lesson learned from that experience is that supporting temporal event information is simple, but state-based temporal information presents many challenges. In fact, timestamps can be associated with temporal events via standard RDF predicates (e.g. *birthDate* and *establishedYear*). Then they can be queried as any ordered domain.

Supporting state information is much more complex, as demonstrated by the many temporal representations and constructs proposed [12, 13, 25, 33] and the rich set of interval-based operators required [5]. For instance, answering a simple query such as “Who was the president of University of California on 9/9/2009?” requires determining the time interval that contains 9/9/2009. In a valid time temporal database, the curators are responsible for supplying these timestamps, thus creating a valid-time history. However, in DBpedia and other web repositories, the curators do not update timestamps directly: instead they update web pages and associated Infoboxes to reflect the changes that occurred in the domain they describe. Thus, readers of the web page will notice a change of the president name from *Mark Youdof* to *Janet Napolitano*. The date and time of this change, i.e. the timestamp in which the update was executed by the system, is known as transaction time (or system time) and, as such, it is constructed from the system log.

Whereas tardy curators will eventually enter the correct valid time, any tardiness of their actions adds permanently to the imprecision of a transaction time databases. Nevertheless, when as in

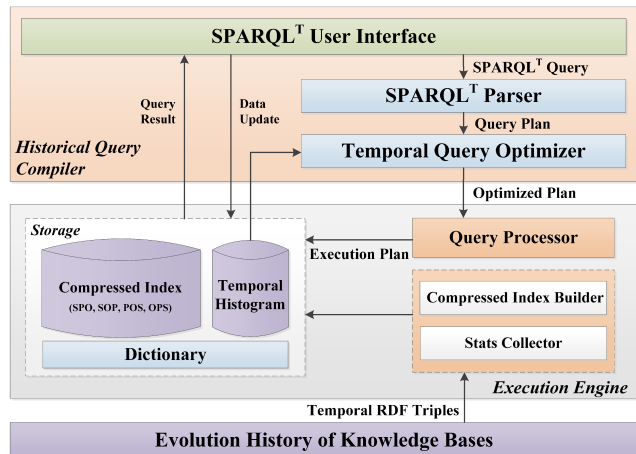


Figure 1: RDF-TX Architecture

the case of Wikipedia, valid time histories are not available, transaction time history will provide a reasonable substitute. Indeed, since the varying tardiness with which Wikipedia Infoboxes are refreshed has not damaged their popularity, it is reasonable to expect that databases describing their system time history will be equally popular, particularly when users are given tools to compensate for temporal imprecision<sup>1</sup>.

Moreover, there are other scenarios in which transaction time histories are needed:

- *History Browsing and Analyzing* [14, 15]. The history of knowledge base captures the revisions of knowledge, which is of great interest for editors and users to understand how the knowledge is evolved and updated.
- *Knowledge Auditing and Verification* [35]. Timestamping, as an important part of provenance, is important for ensuring the quality of facts in many applications. A system with temporal query support also provides administrators with previous states of knowledge bases for auditing purposes.
- *Backup and Recovery*. The history of knowledge bases can be used to backup and recover missing information.

In this paper we present query extensions and an efficient system for managing and querying the transaction time history of knowledge bases. However, the user model and query constructs apply to valid-time histories as well. At the physical level, although the storage structure is designed for the transaction time model, our implementation remains effective for most valid-time histories as discussed in our technical report [2].

Figure 1 shows the high level architecture of our system, which can be divided into two main components as follows.

*Historical Query Compiler.* To express queries against the history of knowledge bases, we introduce a temporal extension of SPARQL called SPARQL<sup>T</sup>. Users can write and submit SPARQL<sup>T</sup> queries through our interface. Then the SPARQL<sup>T</sup> queries are compiled to query plans represented as graphs of query patterns and passed to the temporal query optimizer. The optimized query plans are submitted to the *Execution Engine* for evaluation.

*Execution Engine.* In our engine, the historical information is represented as temporal RDF triples and stored using compressed MVBT indices that support fast query processing. The query processor transforms the query plans from compiler to execution plans expressed in query operators (e.g. temporal selection and join) and executes them on the compressed MVBT indices.

<sup>1</sup>Provenance annotations that record the tardiness of refreshes can go a long way to cure this problem.

Predicate	Object	Timestamp
president	Mark Yudof	06/16/2008 ... 09/29/2013
	Janet Napolitano	09/30/2013 ... <i>now</i>
endowment (billions)	10.3	07/01/2013 ... 06/30/2014
	13.1	07/01/2014 ... <i>now</i>
undergraduate	184,562	05/14/2013 ... 01/29/2015
	188,300	01/30/2015 ... <i>now</i>
staff	18,896	08/29/2013 ... 01/29/2015
	19,700	01/30/2015 ... <i>now</i>
budget (billions)	22.7	01/30/2013 ... 01/29/2015
	25.46	01/30/2015 ... <i>now</i>

**Table 2: Temporal RDF Triples for *University of California***

## 2.2 Data Model

Knowledge bases such as DBpedia [10] and Yago2 [18] can be represented as RDF graphs which consist of RDF triples in the format (*subject, predicate, object*). The subject and predicate of an RDF triple are elements from the set of Uniform Resource Identifiers  $\mathcal{U}$ , while the object is a URI from  $\mathcal{U}$  or a value from the set of literals  $\mathcal{L}$ . For example, the RDF triple for “The president of University of California is Mark Yudof.” is:

- *subject*: [http://www.w3.org/edu/University\\_of\\_California](http://www.w3.org/edu/University_of_California)
- *predicate*: <http://www.w3.org/elements/president>
- *object*: [http://www.w3.org/people/Mark\\_Yudof](http://www.w3.org/people/Mark_Yudof)

For the sake of simplicity, we do not discuss the concept of blank nodes and assume the prefix parts of URI (e.g. <http://www.w3.org/edu/>) are given. Above RDF triple is represented as (*University of California, president, Mark Yudof*).

Since the basic RDF model is designed for static information, we represent the evolution history of knowledge bases using the temporal RDF model proposed in [17] that extends the RDF model with temporal elements. Each RDF triple is annotated with a temporal element to represent the time when this triple is valid. Formally, given a point-based temporal domain  $\mathcal{T}$ , a *Temporal RDF Graph* consists of a set of temporal RDF triples where each temporal RDF triple is a RDF triple ( $s, p, o$ ) annotated with a temporal element  $t \in \mathcal{T}$ . A set of temporal RDF triples with consecutive time points  $\{(s, p, o) [t] \mid t_s \leq t \leq t_e\}$  are encoded using the interval-based expression as:  $(s, p, o) [t_s \dots t_e]$ .

The evolution history of subject *University of California* is represented as a set of temporal RDF triples, as shown in Table 2. All the triples share the same subject *University of California*. We use DAY as the granularity of time and *now* as the current time. Typically, one triple is valid over several days, which we represent with ... between the start day and the end day (start and end included): e.g. [07/01/2013 ... 06/30/2014] represents all the days between 07/01/2013 and 06/30/2014.

## 3. SPARQL<sup>T</sup> QUERY LANGUAGE

To support temporal queries over the history of knowledge bases, we propose a temporal extension of SPARQL called SPARQL<sup>T</sup>. One main difference between SPARQL<sup>T</sup> and previous works is the choice of the temporal model. Many existing works [29, 30, 32] use the interval-based model because of efficiency considerations. However, to express temporal queries, the interval-based representation requires additional operators such as temporal interval overlap, intersect and coalesce, which introduce complications and difficulties [12, 42], particularly for casual users working with friendly wysiwyg interfaces. Therefore, we use a point-based temporal model that resolves these problems at the logical level; however at the physical level we retain the interval representation for efficiency

reasons. Queries on the point-based model can be easily mapped into equivalent ones on the interval-based model for execution.

### 3.1 SPARQL<sup>T</sup> Syntax

SPARQL<sup>T</sup> extends SPARQL with temporal patterns and constructs to query temporal RDF data. To simplify our presentation, we first review the standard syntax of SPARQL [28] and then explain our temporal extension.

**SPARQL graph patterns** are defined as follows:

- A SPARQL graph pattern is a triple  $\{s \ p \ o\}$  from  $(\mathcal{U} \cup \mathcal{L} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{V})$  where  $\mathcal{V}$  is a set of variables.
- If  $S$  and  $S'$  are SPARQL graph patterns and  $F$  is a filter clause,  $(S \text{ AND } S')$ ,  $(S \text{ OPT } S')$ ,  $(S \text{ UNION } S')$  and  $(S \text{ FILTER } F)$  are also graph patterns.

where  $(S \text{ AND } S')$ ,  $(S \text{ OPT } S')$  and  $(S \text{ UNION } S')$  denote the conjunction, optional and union graph patterns.

Temporal queries against the history of knowledge bases are expressed as SPARQL<sup>T</sup> graph patterns.

**SPARQL<sup>T</sup> graph patterns** are defined as follows:

- A SPARQL<sup>T</sup> graph pattern is a tuple  $\{s \ p \ o \ t\}$  from  $(\mathcal{U} \cup \mathcal{L} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{L}) \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{V}) \times (\mathcal{T} \cup \mathcal{V})$ .
- If  $P$  and  $P'$  are two SPARQL<sup>T</sup> graph patterns and  $F'$  is a filter clause,  $(P \text{ AND } P')$  and  $(P \text{ FILTER } F')$  are also SPARQL<sup>T</sup> graph patterns.

Where  $F'$  is constructed using the elements from  $\mathcal{U} \cup \mathcal{L} \cup \mathcal{T} \cup \mathcal{V}$ , comparison symbols, logical connectors and the temporal built-in functions discussed next. In SPARQL, there are 8 types of graph patterns as: S, P, O, SP, SO, PO, SPO, and full scan. For example, SP refers to a query pattern in which subject and predicate are constants and object is a variable. SPARQL<sup>T</sup> supports 16 types of graph patterns, which enable the expression of many interesting queries over temporal RDF graphs.

$(P \text{ UNION } P')$  and  $(P \text{ OPT } P')$  are not supported in current SPARQL<sup>T</sup>, and their implementation is planned for the future. In current state, SPARQL<sup>T</sup> supports efficiently all the queries described in this paper, including the applications discussed in Section 2.1. In passing we observe that they follow the patterns of (i) retrieving information from a previous version of knowledge bases, and (ii) joining the information with similar keys and timestamps. These two scenarios correspond to two operators: single graph pattern matching and temporal join that are supported very efficiently in our system.

**Time Representation and Functions.** In SPARQL<sup>T</sup>, timestamp is from a discrete time domain with a minimum unit as chronon [13]. We define two temporal types: *dateTime* and *period*. *dateTime* corresponds to a single timestamp. *period* corresponds to a set of consecutive timestamps, represented as a pair of two datetime points. For timestamps, SPARQL<sup>T</sup> is equipped with YEAR/MONTH/DAY functions to enable flexible temporal conditions. For periods, we define two built-in functions *TSTART* and *TEND* to return the first and last element in a set of consecutive timestamps.

Many temporal queries involve the reasoning of duration. Thus we define a built-in function *LENGTH* that counts the number of time units (we use DAY as the minimum unit in this paper) within the same consecutive period of time. If one fact is associated with multiple intervals, we return the length of max duration. Another similar function *TOTAL\_LENGTH* is defined to compute the total length of all intervals.

### 3.2 Semantics and Examples

SPARQL<sup>T</sup> is a graph matching language for querying temporal RDF data. The input of a SPARQL<sup>T</sup> query is a temporal RDF

graph and the output is a set of mappings that replace the variables with values from the input temporal RDF graph. The operators of SPARQL are extended to manipulate the temporal element  $t$  of SPARQL<sup>T</sup> graph patterns. For single pattern matching, the query result is the set of temporal RDF triples that match the graph pattern and the filter clause. If there are two or more patterns in the query, the results of single pattern matching are joined. Due to space limitations, we omit the discussion of formal semantics which can be instead found in our technical report [2]. All the syntax discussed in previous section are implemented in RDF-TX. Next we illustrate the usage of SPARQL<sup>T</sup> via several examples.

**Temporal Selection.** We first discuss *temporal selection* queries that have one query pattern (a four-element tuple  $\{s, p, o, t\}$ ). An example of temporal selection query is the “when” query that retrieves the valid timestamps of given facts. Users only need to specify the values for  $(s, p, o)$  and a variable for the temporal element.

EXAMPLE 1. When did Janet Napolitano serve as the president of University of California.

```
SELECT ?t
{University_of_California president Janet_Napolitano ?t}
```

In the query result, the timestamps will be displayed in the compact format  $[t_s \dots t_e]$ . Running Example 1 against the temporal RDF graph in Table 2 returns  $[09/30/2013 \dots now]$ . Another common type of temporal selection queries retrieves information from a previous version of the knowledge base. The temporal constraints (e.g. within a period) can be specified in the FILTER clause.

EXAMPLE 2. Find the budget of University of California in 2013.

```
SELECT ?budget
{University_of_California budget ?budget ?t.
FILTER(YEAR(?t) = 2013)}
```

EXAMPLE 3. Find each person who served as the president of University of California for more than one year before 2010.

```
SELECT ?person ?t
{University_of_California president ?person ?t.
FILTER(YEAR(?t) <= 2010 && LENGTH(?t) > 365 DAY)}
```

**Temporal Join.** More complex queries often use temporal joins which, in SPARQL<sup>T</sup>, are expressed by multiple query patterns that share the same temporal element. General temporal join may involve both key and temporal dimensions.

EXAMPLE 4. Find the name of the university in which Mark Yudof served as the president and the number of undergraduate students when he was in office.

```
SELECT ?university ?number ?t
{?university undergraduate ?number ?t.
?university president Mark_Yudof ?t. }
```

Queries using multiple temporal joins are rather simple to express in SPARQL<sup>T</sup>, whereas in languages based on an interval-based temporal model, such queries tend to be much more complex. For example, if users want to search the number of undergraduate and graduate students when Mark Yudof was in office, we only need to add one more query pattern:  $\{?university graduate ?number2 ?t\}$  to Example 4. If we switch to interval-based model, the query will consist of three query patterns and three temporal conditions:  $?I_1 overlap ?I_2, ?I_1 overlap ?I_3, ?I_2 overlap ?I_3$  where  $?I_1, ?I_2, ?I_3$  are three variables for intervals.

Besides temporal join, the point-based query patterns also support the expression of other temporal operations such as MEET and CONTAIN using the built-in functions TSTART and TEND.

EXAMPLE 5. Find who succeeded Mark Yudof as the president of University of California.

```
SELECT ?successor
{University_of_California president Mark_Yudof ?t1.
University_of_California president ?successor ?t2.
FILTER(TEND(?t1) = TSTART(?t2)). }
```

## 4. STORAGE AND INDEXING

Since the performance of query engines is heavily influenced by its use of indices, it is important to choose an appropriate index structure as well as storage schema for the temporal RDF data.

A natural approach followed by previous works [29] consists in managing temporal RDF triples using existing RDBMS. However, for searching both RDF information and temporal information, two sets of indices are required, and this results in significant costs in storage and retrieval time, which are shown in Section 7. A second natural approach will be using RDF engines, such as Jena and Virtuoso, which have seen recent improvements in performance and functionality. However, this requires the standard RDF reification approach in which a temporal RDF triple is represented as an entity instance with five properties: subject, predicate, object, start time and end time. Thus we need to use five triples for each temporal fact, whereby the space cost increases along with complexity of the queries and time required to optimize and execute them.

Therefore, rather than modifying and extending existing systems we design and build a new system that integrates advanced indexing and data compression techniques into an architecture conceived for efficient support of SPARQL<sup>T</sup> queries

### 4.1 Index Scheme

Many index structures [7, 19, 22, 24] have been proposed for temporal data. Each index has its own strength and they have shared issues such as space overhead and limited support for general temporal queries.<sup>2</sup> In this project, we employ Multiversion B+ Tree (MVBT) to index temporal RDF data for the following reasons. First, MVBT is a bi-dimensional index with asymptotic worst-case guarantee and delivers good performance in real world datasets. Second, we propose an effective approach to compress MVBT which greatly reduces the space cost. The algorithms [8, 41] are extended and optimized on compressed MVBT to enable fast index scan and join. Next we briefly review the structure of MVBT and discuss the index scheme in RDF-TX system.

#### 4.1.1 MVBT

Multiversion B+ Tree [7] is a temporal index structure with optimal worst case guarantees for data insert, update, and delete. The complexity of a temporal query in version  $i$  is asymptotically equal to the complexity of the query on a B+ tree that maintains all the data valid in version  $i$ . Rather than a single tree, an MVBT is actually a forest of trees. It has multiple root nodes and each of them corresponds to a temporal partition of data, as shown in Figure 2(a). An entry in the MVBT node can be represented as  $(key, start\ version, end\ version, data\ value/pointer)$  where  $key$  is unique for a given version and  $start\ version$  and  $end\ version$  together denote the live period of data. An entry that stores data inserted in version  $i$  carries a period of  $(i, now)$ . We denote an entry with end version  $now$  as a live entry. The delete operation modifies the end version of a live entry.

A simple example of MVBT insertion/deletion is shown in Figure 2(b). We first insert five values into an empty MVBT in Version 1, which results in an MVBT tree (i). Then we insert key 14 in Version 2 and delete key 46 in Version 3. The MVBT index becomes

<sup>2</sup>A detailed discussion of temporal index can be found in Section 8.

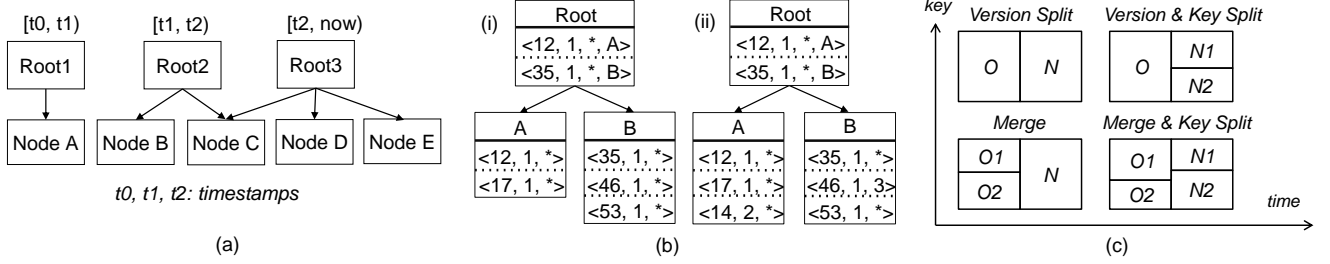


Figure 2: MVBT Example (a) Index Structure; (b) Data Insertion and Deletion; (c) Node Structure Changes.

(ii) with  $\langle 14, 2, * \rangle$  added to Node A and  $\langle 46, 1, * \rangle$  changed to  $\langle 46, 1, 3 \rangle$  in Node B.

To guarantee the performance, MVBT has *weak version condition* that there must be at least  $k$  live entries in a live node. When there are too many entries or not enough live entries (*weak version underflow*) in an MVBT node, node structure changes (version split or merge) are triggered. After structure changes, the number of live entries in a node should be in the range  $[k_l, k_h]$  (*strong version condition*), which prevents long sequences of split and merge operations.

There are four types of node structure changes in MVBT, illustrated in Figure 2(c). We assume that an insertion results in too many entries in a node  $O$ . Then a *Version Split* is performed that copies all the live entries in  $O$  to a new node  $N$ . If node  $N$  has more than  $k_h$  entries, then an additional key split is performed that splits  $N$  into two nodes, as shown in *Version & Key Split*. If  $N$  has less than  $k_l$  entries, *Merge* is triggered.

Assume we need to perform *Merge* operation on the node  $O1$ . MVBT identifies a live sibling node  $O2$ , performs version split and copies the live entries into the new node  $N$ . If node  $N$  has more than  $k_h$  live entries, a key split is performed immediately, as shown in *Merge & Key Split*. Due to the space limitation, we address the readers to [7] for more details of MVBT node structure changes.

#### 4.1.2 Indexing Temporal RDF

In RDF-TX all the data and indices are stored in the main memory. We implement in-memory MVBT to index the temporal RDF triples. The insertion of an interval-encoded RDF triple  $\{(s, p, o) [t_s, t_e]\}$  on MVBT index  $M$  is decomposed into two operations: (i) insert data item  $(s, p, o)$  into  $M$  at time  $t_s$ ; (ii) delete data item  $(s, p, o)$  at time  $t_e$ .

Since the variable may be located in any position of  $(s, p, o)$ , we create four MVBT indices (SPO, SOP, POS, OPS) for different orders of keys  $(s, p, o)$ . These MVBT indices cover all 16 SPARQL<sup>T</sup> graph patterns. For example, the MVBT index for temporal RDF triples in POS order can cover four patterns: P, PT, PO, POT. In query evaluation, the query engine parses the SPARQL<sup>T</sup> prefix patterns to identify the corresponding MVBT index.

We employ dictionary encoding in the index construction, which reduces the index size and avoids the slow comparison between long string literals. Thus RDF-TX replaces the literals with dictionary IDs, and the triples that consist of IDs and timestamps are inserted into our indices. The mapping relations are maintained in our in-memory dictionary for index update and query evaluation. Since the main space cost in our indices is the large number of MVBT entries, dictionary encoding only reduces space cost by 10% - 20%. After dictionary encoding, we exploit delta compression which significantly reduces the space cost of MVBT indices.

## 4.2 Index Compression

For the Wikipedia Infobox History, the size of one standard MVBT index implemented in Java is 1.5–2.2 times of the raw data. More-

over, a temporal RDF graph requires four MVBT indices. If a naive approach is used, comprehensive indexing of temporal RDF data becomes prohibitively expensive. Thus effective compression techniques are needed for large scale datasets.

We observe two characteristics of MVBT. First, the entries in the MVBT node are sorted and neighboring entries often share the same prefix, which could be utilized to reduce space cost. Second, all the node structure operations start from *version split*. This guarantees the query performance but leads to a lot of long intervals. Given these characteristics, we introduce an effective delta encoding method to compress MVBT indices.

#### 4.2.1 Compression Techniques

We design a compression scheme for variable delta encoding of MVBT entry. An MVBT entry for temporal RDF data consists of five values:  $(v_1, v_2, v_3, t_s, t_e)$  where  $v_1, v_2, v_3$  are elements in RDF triples. We store the minimum values for keys and timestamps in each node as base values. Since the data entries are sorted by start version ( $t_s$ ) and key, most entries have very close start versions. Therefore for  $t_s$ , we only keep the minimal value of each node, and compute and store the delta start versions. For  $t_e$ , the compression rules are as follows: (i) if the valid interval  $(t_s, t_e)$  is a short interval,  $t_e$  is stored as the length of intervals; (ii) if the valid interval is long,  $t_e$  is stored as the delta value between  $t_e$  and minimum  $t_e$  in the node; (iii) if the valid interval is a live interval ( $t_e$  is *now*), a special flag is set and  $t_e$  is stored as empty. Other values  $(v_1, v_2, v_3)$  are compressed as the delta values (i) between current value and the value in neighbor entry or (ii) between current value and minimum value in leaf node.

The compressed values are stored in a compact byte array. Figure 3(a) illustrates the format of compressed MVBT entry. Every entry consists of three parts: header, key block ( $v_1, v_2$ , and  $v_3$ ), and time block ( $t_s$  and  $t_e$ ). A normal header (2 bytes) contains a flag (H Flag, 1 bit) for header type (normal/compact), a payload (13 bits in total, 7 bits for key block and 6 bits for time block) that stores the number of bytes for each delta value, and the  $t_e$  flag (2 bits) that records the compression rule for  $t_e$ . For the delta values in key block, we use 1 bit to record how the delta is computed (with neighbor or with node minimum value).

We observe that in large datasets, it is very common that two neighboring MVBT entries (i) share at least one element in key block; (ii) have very close  $t_s$  (delta size  $\leq 4$  bytes) (iii) both  $t_e$  are *now*. Thus for these entries, we propose a compact header which consists of 1 bit header type and 7 bits payload (for two delta values in key block and  $t_s$  delta value).

There is a trade-off between the compression ratio and query performance. Since the number of index nodes is much smaller than the number of leaf nodes and the index nodes are accessed more frequently than leaf nodes, we only compress the leaf nodes of MVBT indices. As shown in evaluation (Section 7), the size of compressed MVBT is about 24% of standard MVBT.

We build MVBT indices for different subsets of Wikipedia dataset

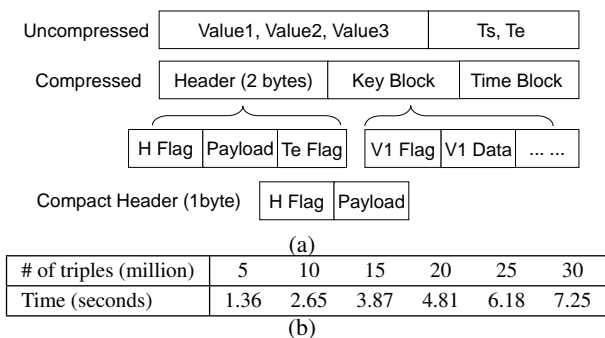


Figure 3: (a) Compressed MVBT Entry (b) Compression Time

and then test the time for compressing MVBT entries, as shown in Figure 3(b). The result shows that it takes very little time to apply the delta encoding technique. Given an MVBT index built from 30 million temporal RDF triples, the time for compressing all the leaf nodes is only 7.25 seconds.

#### 4.2.2 Index Maintenance

An important principle of index compression is to reduce the storage overhead while maintaining the performance of index update and search. For data insertion, we first look up index nodes and identify the leaf node to be updated. In the leaf node, we decompress the start versions ( $t_s$ ) to find the position of input start version  $i$  and compute the delta values of input data. Then we modify the  $(i + 1)$ th entry if its delta values are changed. One issue is that we need to scan from the beginning of all entries. To address this issue, we add a checkpoint in each node that stores the position of MVBT entry with largest  $t_s$ . Then in data insertion, since the  $t_s$  of input data must be larger than existing  $t_s$ , we only decompress the entries after checkpoint. Deletion in MVBT only updates the end version of a live entry. Thus we simply scan all the entries and modify the  $t_e$  of the matched entry. As shown in Section 7, insertion/deletion on compressed MVBT only takes 5% more time than standard MVBT.

## 5. QUERY PROCESSING

In this section, we present the design and implementation of RDF-TX query engine, which makes use of MVBT to process the temporal operations of the language.

### 5.1 Compiling SPARQL<sup>T</sup> Query

The overall evaluation of SPARQL<sup>T</sup> queries consists of four steps:

- Parse the input query and translate point-based query patterns to interval-based query patterns.
- Construct a query plan. The plan is represented as a graph in which each node is an interval-based query pattern.
- When the query contains multiple temporal joins, optimize the query plan to improve the join order.
- Translate the query plan to an execution plan that is evaluated on compressed MVBT indices.

Next we elaborate each step with more details.

**Translating Query Patterns.** Since the temporal RDF graph is stored as interval-based temporal RDF triples, we translate the point-based SPARQL<sup>T</sup> query patterns to the interval-based patterns that can be converted to range queries and executed on MVBT. For key elements, we take the literals as prefix and convert the unknown parts to key ranges. For temporal element ( $t$ ), if there exist temporal constraints in the FILTER clause, we generate time ranges based

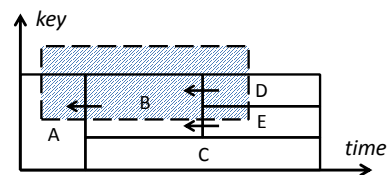


Figure 4: An Example of MVBT Backward Link

on the constraints; otherwise, the default range is  $[0, now]$  where  $0$  refers to the minimum time point. Consider the query pattern  $\{University\ of\ California\ budget\ ?budget\ ?t\} (YEAR(?t) = 2013)$  in Example 2. The interval-based query pattern can be described as a query region with key range and time range as follows:

- key range:  $(University\ of\ California, budget, \_)$  –  $(University\ of\ California, budget, \infty)$
- time range:  $01/01/2013 - 12/31/2013$

Here  $\_$  and  $\infty$  denote the extrema of the string domain.

**Constructing and Optimizing Query Plan.** The query engine generates a query plan that consists of interval-based query patterns from the first step. This query plan can be represented as a graph in which the edges between the nodes are added when two query patterns share the same variable. If there are multiple join operations, the query optimizer (discussed in Section 6) is called to find efficient query plans using the statistics of temporal RDF graphs.

**Executing the query plan on MVBT.** Lastly, the optimized plan is translated to an execution plan which is similar to the query plan in relational databases. Every query pattern is converted to an index scan operator on MVBT indices. Then the join operators are added based on the optimized join order. Finally, appropriate filter operators are added using the FILTER clause of SPARQL.

## 5.2 Executing Query Plan

Next we describe the implementation of index scan and temporal join in RDF-TX. Other operators (e.g. filter) are implemented similar to their counterparts of existing engines thus omitted.

### 5.2.1 Index Scan

We perform an index scan for each interval-based query pattern. For the index scan on MVBT, we employ the link-based range-interval algorithm [8] which introduces *Backward Link* in MVBT to process the range queries. The MVBT leaf nodes are equipped with backward links that point to the temporal predecessors. The index scan is performed as: (i) search all the nodes that intersect the right border of query region; (ii) follow the backward links of the nodes to find all the nodes that intersect query region; (iii) scan the leaf nodes found in the first two steps to retrieve the entries. An example of linked index scan is shown in Figure 4. The shadowed rectangle represents a query. MVBT nodes  $D$  and  $E$  are first visited. Then as the predecessor of  $D$  and  $E$ , node  $B$  is checked. Lastly, node  $A$  is visited.

### 5.2.2 Temporal Join

Temporal join represents one of the most expensive operations in the temporal query language, especially when the size of knowledge base is very large. Therefore we explore three types of joins: *Merge Join*, *Hash Join*, and *Synchronized Join*.

Merge join is very popular and widely used in existing SPARQL engines [27, 38]. These systems build indices for all permutations so that the optimizer leverages the indices to perform order-preserving merge joins. However, this does not work for MVBT index since the entries are sorted by time. We mainly use *Hash Join* in our query engine.

When the size of result is large, the cost of building a hash table may be very expensive. Thus we extend the synchronized join [41]. The basic idea of synchronized join is as follows: (i) synchronously find the set of all MVBT node pairs  $(e_1, e_2)$  that intersect each other and the right border of query region; (ii) join  $e_1$  and  $e_2$ ; (iii) join the predecessors of  $e_1$  and  $e_2$  by following the backward links. This algorithm avoids materializing the intermediate result, but it is much slower than hash-based join since one page and its predecessors are visited many times. So we optimize this algorithm by caching recently visited records; that is, given a page  $e$  from step (i), we cache the records in  $e$  and its predecessors, and perform joins between  $e$  and other pages. This optimized synchronized join is used when the query pattern in the join accesses a large portion of index (e.g. find all the triples valid in a certain period).

## 6. OPTIMIZATION

In RDF-TX, improper join orders may generate large intermediate results and slow down execution. Therefore, a natural step is to optimize complex SPARQL<sup>T</sup> queries by finding efficient join orders. The key of join optimization is to efficiently estimate the costs of different join orders, which is not a trivial task for temporal queries. In this section, we present a query optimizer that uses estimated statistics of temporal RDF graph to optimize the orders of temporal joins in SPARQL<sup>T</sup> queries.

### 6.1 RDF-TX Query Optimizer

For queries that involve multiple temporal joins, we implement a query optimizer that uses the bottom-up dynamic programming strategy [23] to find the cost-optimal query plans. Our optimizer generates multiple query plans and finds the plan with lowest estimated cost. A large query plan is generated by joining two small optimal query plans. The cost is computed based on the cardinalities of query patterns and intermediate results.

The cardinality estimation is a well-studied problem in relational databases and SPARQL engines [26, 27, 31]. To estimate the cardinality of join result, an effective approach is characteristic set [26]. In a RDF graph  $R$ , the characteristic set  $S_C(s)$  of a subject  $s$  is the set of related predicates:  $S_C(s) = \{p | \exists o, (s, p, o) \in R\}$ .

The idea of characteristic set is that semantically similar subjects (e.g. University of California and University of Michigan) usually have the same characteristic set. For every characteristic set, the number of distinct subjects that belong to the characteristic set, and the number of occurrences of the predicates in these subjects are recorded and used to estimate the cardinality. For example, given a characteristic set  $\{president, undergraduate\}$ , there are 100 distinct subjects belong to this characteristic set. The numbers of occurrences for *president* and *undergraduate* are 150 and 110 respectively. Consider a SPARQL query with two query patterns:

```
SELECT ?s ?o1 ?o2 .
{?s president ?o1 .
?s undergraduate ?o2 . }
```

Suppose that only this characteristic set contains both predicates in the query. Then the result cardinality is estimated as:  $100 \times \frac{150}{100} \times \frac{110}{100} = 165$ .

Characteristic sets provide highly accurate estimation of cardinality. But it can not be used to estimate the cardinality of SPARQL<sup>T</sup> queries since the statistics of temporal RDF graph vary on different time points. Consider following SPARQL<sup>T</sup> query:

```
SELECT ?s ?o1 ?o2 ?t
{?s president ?o1 ?t.
?s undergraduate ?o2 ?t. }
```

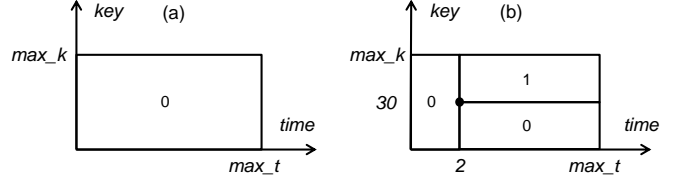


Figure 5: An Example of MVSBT Entry Split

`FILTER(?t ≤ 01/01/2013) . }`

To estimate the cardinality of this SPARQL<sup>T</sup> query, we need to know the number of subjects that (i) belong to the set of temporal RDF triples valid in the period  $[0, 01/01/2013]$  and (ii) share the characteristics set  $\{president, undergraduate\}$ . These statistics change with time and none of existing data structures can provide estimation of these statistics. Thus we introduce a temporal histogram to maintain the statistics of temporal RDF data in next Section. With temporal histogram, the characteristics sets can be easily integrated into our query optimizer.

### 6.2 Temporal Histogram

The problem of estimating the statistics of temporal RDF data is similar to the temporal aggregation that computes the aggregate value in a certain period. Thus we propose Compressed MVSBT that extends the temporal aggregate index Multiversion SB Tree to maintain the statistics of characteristic sets.

#### 6.2.1 MVSBT

MVSBT [39, 40] is a temporal index that combines the features of MVBT and SB Tree [37] and supports the dominance-sum query. Given key  $k$  and time  $t$ , it returns the aggregation value of data records with keys less than  $k$  and timestamps smaller than  $t$ .

Similar to MVBT, MVSBT is a forest of trees with multiple root nodes and each of them points to an SB Tree for a temporal partition of data. Each entry in MVSBT corresponds to a rectangle in the key-time space. The structure of MVSBT entry is as follows:

- Leaf Entry:  $\langle k_s, k_e, t_s, t_e, v \rangle$
- Index Entry:  $\langle k_s, k_e, t_s, t_e, v, ptr \rangle$

A leaf entry has a key range  $(k_s, k_e)$ , an interval  $(t_s, t_e)$ , and a value  $v$ . The key range and the interval represent the rectangle covered by this entry in the key-time space.  $v$  maintains the aggregate value. The index entry has one additional pointer  $ptr$  that points to a child node. The rectangles of the entries are mutually disjoint and the union of all the rectangles is equal to the whole key-time space.

The node structure of MVSBT is similar to MVBT while the insertion algorithm is quite different. First we need to review two concepts introduced in [39]. Given a point  $(k, t)$ , and max key value  $max\_key$ , an entry is referred as *partly covered entry* if its key range intersects the range  $[k, max\_key]$  but not contained in this range; an entry is referred as *fully covered entry* if its key range is contained by the range  $[k, max\_key]$ . When a new point  $p(k, t)$  is inserted into a node  $N$ , the fully covered entries are vertically split at  $t$ . If there exists a partly covered entry, then  $p$  is inserted into the child page. If node  $N$  is in the leaf level, the partly covered entry is split into three entries based on point  $p$ . The node structure operations of MVSBT are similar to the ones of MVBT (Section 4.1.1) thus omitted.

An example of MVSBT for aggregate COUNT is shown in Figure 5. Figure 5(a) shows the initial entry of an empty MVSBT. The aggregate value is 0 since no point is inserted. Then one point with key 30 and timestamp 2 is inserted. The initial entry is split into three entries as shown in Figure 5(b). The entry on the top right corner has aggregate value 1 and other entries has aggregate value

**Algorithm:** leafEntrySplit( $n_f, r, q$ )

**Input:** CMVSBT leaf node  $n_f$ , entry  $r$ , new point  $p(k, t)$

```

1:  $r.c = r.c + 1$ 
2: if  $p.k > r.k_m$  then
3:    $r.k_m = p.k$ 
4:  $r.t_m = p.t$ 
5: if  $r.c = c_m$  then
6:   if  $r.k_m \neq r.k_s$  and  $r.t_m \neq r.t_s$  then
7:     // Split  $r$  to three entries
8:      $v' = r.c/2 + r.v$ 
9:      $r_1 = \text{new entry}(r.k_s, r.k_m, r.t_m, r.t_e, k_s, t_m, v', 0)$ 
10:     $r_2 = \text{new entry}(r.k_m, r.k_e, r.t_m, r.t_e, k_m, t_m, r.c/2, 0)$ 
11:     $n_f.add(r_1); n_f.add(r_2)$ 
12:     $r.t_e = r.t_m$ 
13:   else
14:     // Split  $r$  to two entries, similar to step 7 - 11, omit

```

**Algorithm:** indexEntrySplit( $n_i, r, p$ )

**Input:** CMVSBT index node  $n_i$ , entry  $r$ , new point  $p(k, t)$

```

1:  $r.list.add(p)$ 
2: if  $\text{length}(r.list) == l_m$  then
3:    $r_1 = \text{new entry}(r.k_s, r.k_e, p.t, r.t_e, \text{new list}(), r.ptr, r.c)$ 
4:    $n_i.add(r_1)$ 
5:    $r.t_e = p.t$ 

```

**Figure 6: Algorithms for Entry Split in CMVSBT**

0 since all the points in top right entry are larger than split point (30, 2). Suppose we have two queries (10, 1) and (40, 5). The first query point falls in the left entry and returns 0. The second query points falls in the top right entry and thus gets the result 1.

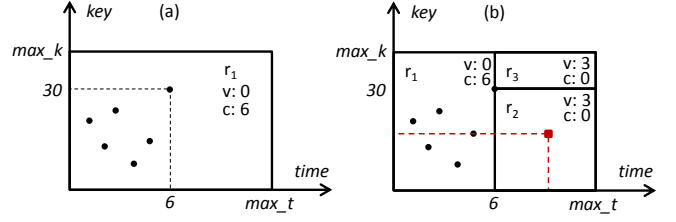
### 6.2.2 Compressed MVSBT

Although MVSBT has good performance on temporal aggregation, it takes too much space for storage. Since query optimization does not require very accurate estimates, we can trade accuracy with efficiency. Compressed MVSBT (CMVSBT) is based on the idea that instead of accurately recording the points and splitting entries for every new point, a CMVSBT entry contains  $m$  ( $m \geq 1$ ) points. Then we can estimate the aggregate value using the ratio of covered space to full space. Instead of storing the exact values of points, we store the statistic values such as total number of points and max value. The structure of CMVSBT entry is as follows:

- Leaf Entry:  $\langle k_s, k_e, t_s, t_e, k_m, t_m, v, c \rangle$
- Index Entry:  $\langle k_s, k_e, t_s, t_e, list, ptr, c \rangle$

where  $k_s, k_e, t_s, t_e$  are for the key range and associated time interval.  $k_m$  and  $t_m$  store the max key value and time value of the points located in the rectangle of this entry;  $list$  is a list of points;  $ptr$  is the pointer to a CMVSBT node;  $v$  and  $c$  are *fixed* and *current statistic values*. *fixed statistic value* refers to the aggregate value, while *current statistic value* refers to the aggregate value computed over the points contained in the current entry. The final statistic value is estimated by combining both values (discussed in Section 6.3).

Since the index nodes are visited more frequently than leaf nodes, we store the exact values of points in a list in the index nodes, while in leaf nodes we only maintain three statistics ( $k_m, t_m, c$ ). The algorithm for data insertion in CMVSBT is similar to the one for MVSBT. Instead of splitting the entry for every input point, CMVSBT entry is split when the number of points in an entry is larger than the threshold. The split point is  $(k_m, t_m)$ . The algorithm for entry splitting in CMVSBT for COUNT is shown in Figure 6. Let  $c_m$  and  $l_m$  denote the thresholds for the number of points in leaf nodes and index nodes. When a new point  $p(k, t)$  is inserted into compressed MVSBT, we look up the index nodes to find a set



**Figure 7: An Example of CMVSBT Entry Split**

of nodes  $N$  whose rectangles cover this point. In a leaf node  $n_f$ , if  $p$  falls in the rectangle,  $c$  is increased by 1, and the max values ( $k_m, t_m$ ) are updated if  $p.k > k_m$  or  $p.t > t_m$ . Then if  $c = c_m$ , we split the entry based on the position  $(k_m, t_m)$ . After split, the statistical values ( $v$ ) of new entries will be equal to (i)  $c/2 + v$  if the new entry has the same  $k_s$  with old one; (ii)  $c/2$  otherwise (based on the logical splitting in MVSBT [40]). We use  $c/2$  since we assume the points are uniformly distributed in the entry. The  $c$  of new entries are initialized to be 0. In an index node  $n_i \in N$ , if  $r_i$  is the lowest entry that fully covers  $p$ ,  $p$  is appended to the end of  $list$ . Like MVSBT, compressed MVSBT also assumes that the data items come in nondecreasing time order. Thus  $list$  is automatically sorted by time. If  $\text{length}(list) = l_m$ , the entry is split on  $p.t$  and  $c$  is copied to new entry. If  $r.k_m = r.k$  or  $r.t_m \neq r.t_s$ , the split point  $(r.k_m, r.t_m)$  falls on the borders of rectangle. Then  $r$  is split into two entries. When we set  $c_m = 1$  and  $l_m = 1$ , the algorithm in Figure 6 is the same with the split algorithm of MVSBT. More details about CMVSBT construction are available in our technical report [2].

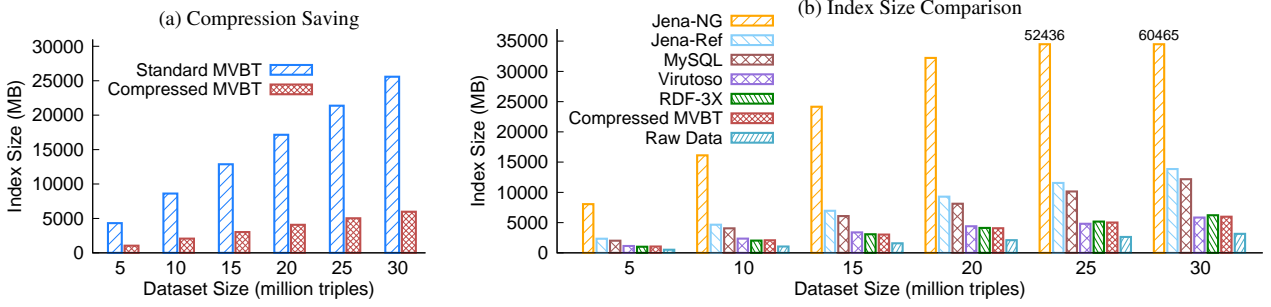
Here we prepare a simple example to illustrate the process of entry split in CMVSBT. We assume that we have inserted six points into a compressed MVSBT, as shown in Figure 7(a). The threshold  $c_m$  is set to be 6. The max key  $r_1.k_m$  is 30 and the max time  $r_1.t_m$  is 6. Although the rectangle of  $r_1$  is the whole space, all the points fall in the effective rectangle  $rec$  (key range:  $[0, k_m]$ , time range:  $[0, t_m]$ ). Since  $r_1.c \geq 6$ , we split it into three rectangles as shown in Figure 7(b). The values of  $r_1$  are not changed since all the points still fall in  $r_1$ .  $r_2.v$  is approximated by the number of points covered by the virtual center of  $r_2$  (the red point). As we can see, the red rectangle covers half of  $rec$ , so  $r_2.v = r_1.c/2 + r_1.v = 3$ ,  $r_3.v = r_1.c/2 = 3$ .

For each characteristic set, we need to maintain: (i) the number of distinct subjects and (ii) the number of predicate occurrences. As discussed in next section, each type of statistic values requires two CMVSBTs: one for start points and one for end points. Thus our temporal histogram consists of four CMVSBTs and the schema of characteristic sets. In RDF-TX, we set the max size of temporal histogram as 10% of raw data. If the size of temporal histogram is larger than the threshold, we increase  $c_m$  and  $l_m$  and merge the neighbor entries until the temporal histogram is small enough.

## 6.3 Statistics Estimation

Given a query  $q(k, t)$ , compressed MVSBT estimates the statistics of data records with keys less than  $k$  and timestamps smaller than  $t$ . The query algorithm consists of two main steps: (i) starting from root node, we look up the CMVSBT nodes whose rectangle covers  $q$ ; (ii) in each node, we find all the rectangles whose time range contains  $t$  and  $k_s \leq k$ , and accumulate the approximate statistic value  $v_a$  of these rectangles. In a CMVSBT entry,  $v_a$  equals to the sum of fixed statistics value  $v$  and current statistic value  $v_e$ .  $v_e$  is approximated by multiplying  $c$  by the proportion of query region in the rectangle  $ratio$ , as  $c \times ratio$  where  $ratio = ratio_k \times ratio_t$ . If  $q.k \geq r.k$ ,  $ratio_k = 1$ ; otherwise,  $ratio_k$





**Figure 8: (a) Compression Saving for MVBT Index; (b) Index Size Comparison. The size of dictionary is included in the results.**

$= (r.k_m - q.k) / (r.k_m - r.k_s)$ . And  $ratio_t$  can be computed in a similar way.

The query pattern in SPARQL<sup>T</sup> is translated to a range query which is not supported by CMVSBT. Thus we use the query reduction approach [40] which reduces one range query into four point queries. In this approach, we need two CMVSBTs for the start points and end points of temporal RDF triples. Then the statistics in the query region (key:  $[k_1, k_2]$ , time:  $[t_1, t_2]$ ) is calculated as:

$$Q_s(k_2, t_2) - Q_e(k_2, t_1) - Q_s(k_1, t_2) + Q_e(k_1, t_1)$$

where  $Q_s(k, t)$  and  $Q_e(k, t)$  refer to the point queries on the CMVSBT of start points and end points respectively.

During query optimization, we cache all the statistics to reduce the time on scanning CMVSBTs. When one statistic value is required, we first search the statistics cache. If it is not cached, then we use the CMVSBTs to estimate the statistic value.

## 7. EXPERIMENTAL EVALUATION

RDF-TX is implemented in Java as a sequential main memory query engine. To evaluate the performance of our system, we conduct experiments on several real world datasets and compare results with alternative approaches.

### 7.1 Experiment Setup

#### 7.1.1 Dataset

**Wikipedia.** Wikipedia [4] is a real world dataset extracted from the edit history of English Wikipedia. We parse the raw file and generate 38 million temporal RDF triples as our test benchmark. This dataset contains the history of 1.8 million subjects and 3500 frequent predicates (used in more than 500 triples).

**GovTrack.** GovTrack [1] is a public dataset that contains the information about congressmen, votes, bills and committees. There are 20 million historical records for 0.4 million subjects and 60 related events (e.g. congressman election or bill voting). We parse the XML source files to temporal RDF triples as our test dataset.

**Yago2.** Yago2 [18] is a knowledge base derived from Wikipedia, WordNet and GeoNames with more than 30 million temporal RDF triples. Due to space limit and the fact that the evaluation results on Yago2 are very similar to Wikipedia and Govtrack, we leave the results on Yago2 in our technical report.

#### 7.1.2 Implementation and Configuration

**RDF Reification.** RDF reification provides a way to store RDF triple and its meta knowledge in standard RDF model by representing annotated RDF triple as an entity with following properties: *subject, predicate, object, meta knowledge*. Similarly, we represent a temporal RDF triple as an entity with five properties: *subject, predicate, object, start time, end time*. Then SPARQL<sup>T</sup> queries are

easily rewritten to SPARQL queries. We evaluate the reification approach in three well known RDF engines: Jena v2.13 [36], Virtuoso v7.20 [3] and RDF-3X v0.3.8 [27].

**RDBMS-based Approach.** Temporal RDF triples can be stored in a relational table with five columns *subject, predicate, object, start time, end time*. We choose MySQL memory engine (v5.5) in our evaluation since it supports in-memory B+ tree index. We build four B+ tree indices on SPO, SOP, PSO, OPS and two additional indices on start/end time for evaluation of temporal constraints.

**Named Graphs.** Named graph [11] is an extension of RDF model that identifies graphs with URLs and allows graph metadata such as provenance and trust. We implement the approach described in [32] that stores temporal information as graph metadata using Jena Named Graph implementation. We also test Ng4j v0.9.3 implementation [9], but it is much slower than Jena and other approaches, so we leave the results on Ng4j in our technical report [2]. In the rest of this paper, we use “Jena Ref” and “Jena NG” to denote Jena Reification and Jena Named Graph respectively.

**RDF-TX.** Our query engine is a single-thread implementation using compressed MVBT as indices. Only the construction of compressed MVBT is paralleled (using at most four threads).

All the experiments are performed on a machine with 4 AMD Opteron 6376 CPUs (64 cores) and 256GB RAM running Ubuntu 12.04 LTS 64-bit. The index decompression time is included in the query execution time. The execution time reported is calculated by taking the average of 5 runs. The datasets and queries are available in our website [2].

### 7.2 Index Space

We first investigate the effectiveness of our delta encoding techniques (Section 4.2). We implement the standard MVBT indices (4 indices: SPO, SOP, POS, OPS) with numeric keys as baseline. Figure 8 (a) shows the space costs of standard MVBT and compressed MVBT in Wikipedia dataset. On average, our delta encoding technique reduces the space cost of MVBT by 76%.

Then we compare the space overhead of compressed MVBT and other types of index in Wikipedia, as shown in Figure 8 (b). Since Wikipedia has a large number of unique timestamps, most named graphs are very small ( $\leq 5$  triples). Thus indexing named graph incurs a lot of overhead and Jena Named Graph takes far more space than other approaches. The space cost of MySQL memory engine and Jena Reification are similar, which are 3-4 times of raw data. The index space of our implementation is almost the same with Virtuoso and RDF-3X, while the query performance is much better as shown in Section 7.3. On average, the space of our comprehensive indices (4 compressed MVBT + dictionary) is about 1.8 times of raw data. The results for GovTrack dataset are similar and thus omitted.

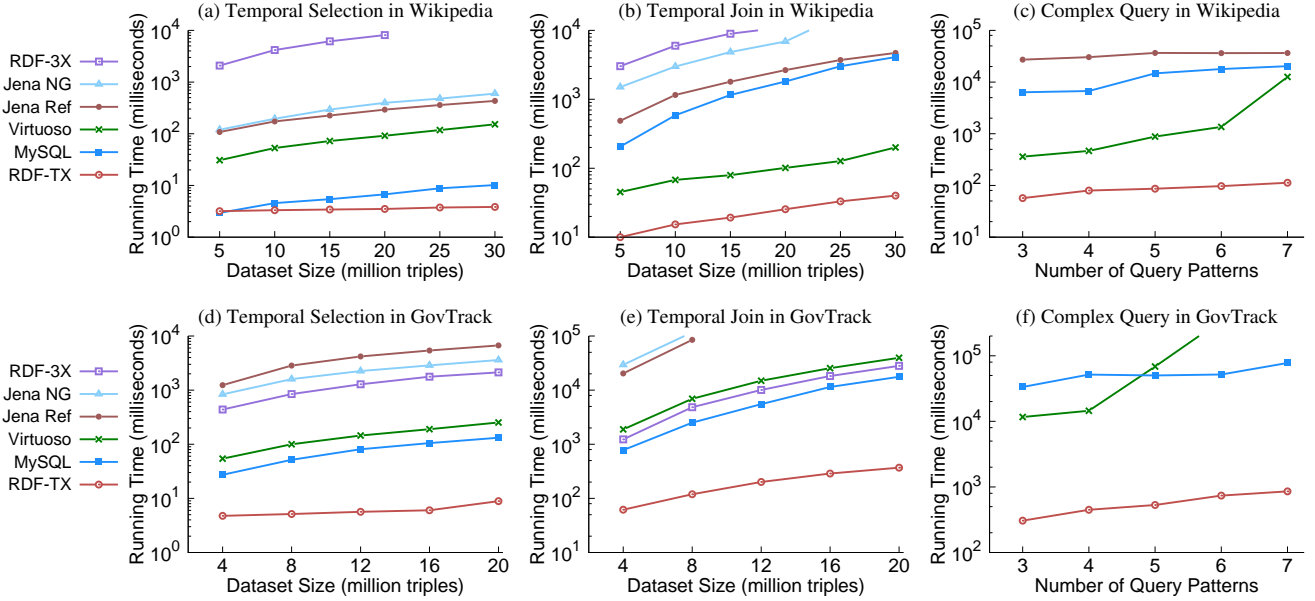


Figure 9: Query Running Time in (a - c) Wikipedia; (d - f) GovTrack.

### 7.3 Query Performance

To evaluate the query performance of our system, we created three sets of queries: (a) Temporal selection queries, as Example 2 in Section 3; (b) Temporal join queries, as Example 4; (c) Complex queries (2 or more temporal joins). We use the first two query sets to evaluate the performance as the dataset size increases, and the third query set to evaluate the performance as the query pattern size increases. For all the implementations, we report the average warm-cache query execution time.

**Temporal Selection and Join.** We create 10 temporal selection and 10 temporal join queries for each dataset and conduct the experiments as the size of dataset  $N$  increases ( $N$ : 5-30 million in Wikipedia and 4-20 million in GovTrack).

Figure 9(a) shows the query execution time for temporal selection in Wikipedia. RDF-TX and MySQL show similar performance in small datasets. As the size of dataset increases, RDF-TX shows better performance than MySQL. In the largest dataset (30 million), RDF-TX is about 3X faster than MySQL and 10X faster than Virtuoso. Jena Named Graph and Reification are 2 orders of magnitude slower than SPARQL engine due to the slow index scan.

RDF-3X is much slower than other systems due to its poor support of constraints. Most historical queries involve temporal constraints. For instance, consider Example 2 in Section 3 that searches the budget of University of California in 2013. This query has one temporal constraint that the valid period of temporal RDF triple should overlap (01/01/2013, 12/31/2013). This constraint can be expressed as:  $?t_s \leq 12/31/2013 \ \&\& \ ?t_e \geq 01/01/2013$ . In RDF-3X, the numbers are encoded as strings. So for temporal constraints, RDF-3X converts strings back to integers at running time to evaluate the constraints, which is inefficient.

The results of temporal join in Wikipedia are shown in Figure 9(b). RDF-TX is about 2 orders of magnitude faster than MySQL and Jena, and 6X faster than Virtuoso. RDF-3X is still slow since the condition of temporal join (e.g. OVERLAP and MEET) is expressed as constraints in FILTER clause.

Figure 9 (d) (e) show the query execution time for temporal selection and join in GovTrack. These approaches take more time to execute since the query patterns (e.g. P and PT) return much more

results in GovTrack due to the reduction of predicates. The RDF-3x performs better than Jena on this dataset since it has a smaller number of distinct time periods ( $\sim 10000$ ) and predicates. MySQL and Virtuoso are about 1 order of magnitude slower than RDF-TX on selection and 2 orders of magnitude slower on Join.

RDF-TX performs 1-2 orders of magnitude faster than most competitors for selection and join. An important reason behind this is that MVBT can process two-dimensional (key and time) range query in one operation, while SPARQL and SQL engines need additional join and index scan.

**Complex Queries.** We generate 25 complex queries for each dataset with increasing query pattern size (3-7). The generation process is as follows: a set of 5 queries is created initially, and each query has 3 query patterns; then we incrementally add query patterns to existing queries until the size of query patterns reaches 7. The experiment is conducted on two datasets (each has 20 million triples) and the optimizers are enabled in all compared approaches.

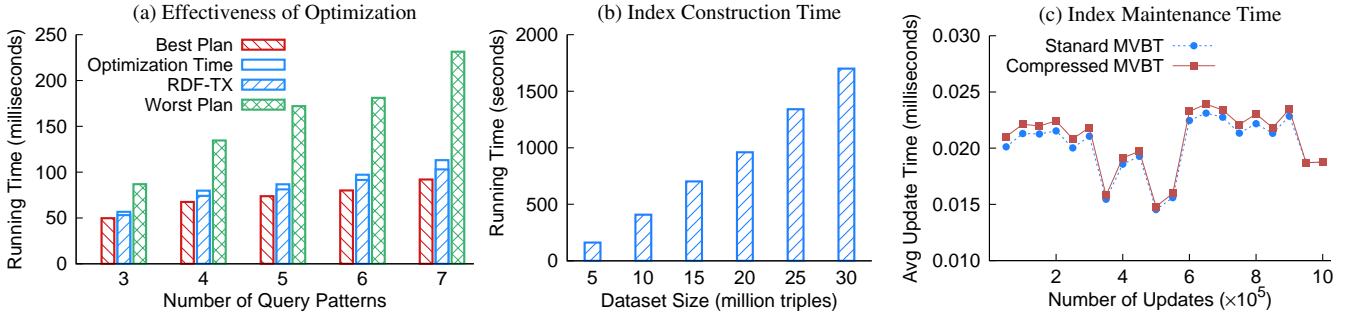
The evaluation results in Wikipedia are shown in Figure 9 (c)<sup>3</sup>. Jena Named Graph and RDF-3X are not reported since they are much slower than other approaches so we omit them. For RDF engine and RDBMS, a query with more patterns is translated to more joins, which increases the complexity of parsing and optimization. On average, RDF-TX is 2 orders of magnitude faster than MySQL and Jena, and 1 order of magnitude faster than Virtuoso.

The evaluation results for GovTrack are shown in Figure 9 (f). A notable change in this graph is that Jena is not reported in this experiment. Jena is too slow compared to other approaches on GovTrack since the query patterns usually cover a large portion of dataset, which leads to slow execution time if an inefficient join order is generated; meanwhile, the column-store traits of Virtuoso excel in this small predicate cardinality case. On average, our system is still about 2 orders of magnitude faster.

### 7.4 Effectiveness of Query Optimizer

In this section, we explore the impact of query optimizer in the

<sup>3</sup>The query running time of Virtuoso on pattern size 6/7 is averaged over four queries since Virtuoso generates a very inefficient join order for one query which takes more than 1 hour to finish.



**Figure 10: (a) Query Execution Time of the best/worst plans, and the plan generated by SPARQL<sup>T</sup> optimizer for complex queries in Wikipedia (b) Index Construction Time (c) Index Maintenance Time**

query evaluation. We enumerate all the possible query plans of the complex queries (Section 7.3) in Wikipedia and find the best and worst execution times. Figure 10 (a) shows the query execution times of best/worst plans and the plan generated by RDF-TX query optimizer (blue bar) in a Wikipedia set with 20 million triples. The result shows that the plan generated by our query optimizer is very close to the best performing execution plan. On average, the execution time of optimized query plan is about half of the time used by worst plan. In the relatively simple queries (3 query patterns), the difference between the best plan and the worst plan is small. As the number of query patterns increases, the difference becomes much larger. Thus, the optimizer is important for scaling up towards complex queries with a lot of query patterns. We also measure the time used for query optimization, which varies from 3.5 to 10 milliseconds as the size of query increases.

Then we measure the storage overhead of temporal histogram. The CMVSBTs for temporal statistics are built using the dictionary IDs. As discussed in Section 6.2, we merge CMVSBT entries and increase  $c_m$  and  $l_m$  until the size is small enough. In this experiment, the size of temporal histogram is 177.5 MB, which is about 8.5% of raw data size.

## 7.5 Index Construction & Maintenance

For large datasets, we first build standard MVBT and then compress the MVBT indices. In RDF-TX, the process of index construction is paralleled using at most 4 threads. We evaluate the index construction time for compressed MVBT time on different sizes of subsets of Wikipedia in Figure 10 (b) (compression time included). The time for index construction is approximately linear with the size of datasets, and it increases slightly faster in the datasets with 25 million and 30 million triples due to degraded performance caused by JVM garbage collection.

RDF-TX also supports the index update on compressed MVBT, which is important for real-time applications. Thus we further measure the average index maintenance time on a compressed MVBT index built from a 25 million subset of Wikipedia. We perform 1 million updates (68% insert, 32% delete) which simulates the changes in real Wikipedia edit history. Figure 10 (c) shows the results by comparing maintenance time of compressed MVBT with the time used on standard MVBT. Our compression technique shows a decent performance. Comparing with the update on MVBT, the update on compressed MVBT only takes 5% more time. This little overhead is negligible w.r.t. 76% space saved using compression.

## 8. RELATED WORK

**Temporal Index.** There has been a large body of research on temporal index in the literature [7, 19, 22, 24, 34]. MAP21 [24] is an index over B+Tree by mapping time ranges to one dimen-

sional points, thus time intervals/points can be used as keys and queried in a B+Tree. OB+tree [34] organizes B+Trees in a versioned way with shared nodes whose contents do not change over versions. However, MAP21 and OB+tree only support single dimension query. BT-tree [19] enables branched versions along with the temporal index, while the time in our system is linear, i.e. no branching. MVBT [7] and TSB-Tree [22] are bi-dimensional indices, which satisfy our requirements exactly. TSB-Tree is a temporal index very similar to MVBT and implemented in Immortal DB [21] on Microsoft SQL Server, with better integration to SQL Server’s existing index structures. The major difference between these two is that TSB-Tree migrates old data to a historical store during node splitting, while MVBT moves new data. Since MVBT is a general approach which is not targeted on specific platforms, we adopt and extend it in RDF-TX.

**Query Languages and Systems for Temporal RDF.** Several query languages [16, 29, 30, 32] have been proposed for temporal RDF triples. T-SPARQL [16] is a temporal extension of SPARQL based on a multi-temporal RDF model. The RDF triple is annotated with a temporal element that represents a set of temporal intervals. Thus a temporal join is expressed using additional functions (e.g. OVERLAP). At the best of our knowledge, no actual implementation of T-SPARQL is available. The  $\tau$ -SPARQL system reported in [32] uses the temporal RDF model [17] and augments SPARQL query patterns with two variables  $?s$  and  $?e$  to bind the start time and end time of temporal RDF triples and express temporal queries. The evaluation is done by rewriting  $\tau$ -SPARQL queries to standard SPARQL queries. Perry et al. [29] propose a framework to support temporal and spatial semantic queries. Simple selection and join queries are expressed using two temporal operators. These operators are implemented in Oracle by extending Oracle Semantic Data Store and SQL functions. These works rely on relational databases/RDF engine to store and query temporal RDF triples, which results in complex SPARQL and SQL queries.

The tRDF system [30] extends the temporal RDF model [17] with indeterminate temporal annotations. The temporal queries are evaluated using tGrin index that clusters the temporal RDF triples based on graphical-temporal distance. However, tRDF only supports a subset of temporal queries discussed in this paper. Most significantly, temporal joins are not supported since tGrin index relies on the temporal distance to filter the triples, while the temporal distance between two temporally joined patterns can not be determined. STUN [20] system supports queries on annotated RDF, but it is not scalable for large temporal datasets.

## 9. CONCLUSION

In this paper, we present SPARQL<sup>T</sup> and its system RDF-TX which supports powerful queries over the history of knowledge bases.

SPARQL<sup>T</sup> enables the expression of a wide variety of temporal queries via simple extension of SPARQL graph patterns and built-in functions. SPARQL<sup>T</sup> queries are efficiently evaluated in the backend query engine that achieves excellent performance by exploiting M-VBT as index and leveraging fast algorithms for range selection and temporal join. RDF-TX also features a query optimizer that uses the statistics of temporal RDF graphs to find the efficient join orders for complex SPARQL<sup>T</sup> queries. Extensive experiments on real world datasets show that RDF-TX outperforms other approaches that use state-of-art RDF engines and relational databases in all kinds of queries and delivers 1 - 2 orders of magnitude performance improvement in complex queries. This confirms the effectiveness and superior performance of RDF-TX .

## 10. ACKNOWLEDGEMENTS

The authors would like to thank Maurizio Atzori, Massimo Mazzeo, Mohan Yang and Alexander Shkapsky for their insightful comments and suggestions on this research. This work was supported in part by NSF Grant IIS 1218471 and IIS 1118107.

## 11. REFERENCES

- [1] GovTrack Dataset. <https://www.govtrack.us/>.
- [2] RDF-TX Technical Report and Datasets. <http://yellowstone.cs.ucla.edu/rdf-tx/>.
- [3] Virtuoso. <https://github.com/openlink/virtuoso-opensource>.
- [4] E. Alfonseca, G. Garrido, et al. WHAD: Wikipedia Historical Attributes Data. *LRE*, pages 1163–1190, 2013.
- [5] J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Commun. ACM*, 26(11):832–843, 1983.
- [6] M. Atzori and C. Zaniolo. SWiPE: Searching Wikipedia by Example. In *WWW*, pages 309–312, 2012.
- [7] B. Becker, S. Gschwind, et al. An Asymptotically Optimal Multiversion B-tree. *VLDB*, 5(4):264–275, 1996.
- [8] J. V. d. Bercken and B. Seeger. Query Processing Techniques for Multiversion Access Methods. In *VLDB*, pages 168–179, 1996.
- [9] C. Bizer, R. Cyganiak, et al. Ng4j-named Graphs Api for Jena. In *ESWC*, 2005.
- [10] C. Bizer, J. Lehmann, et al. DBpedia - A Crystallization Point for the Web of Data. *J. Web Sem.*, 7(3):154–165, 2009.
- [11] J. J. Carroll, C. Bizer, et al. Named graphs, Provenance and Trust. In *WWW*, pages 613–622, 2005.
- [12] C. X. Chen and C. Zaniolo. Universal Temporal Extensions for Database Languages. In *ICDE*, pages 428–437, 1999.
- [13] J. Clifford and A. Rao. A Simple, General Structure for Temporal Domains. In *Temporal Aspects in Information Systems*, pages 17–28, 1987.
- [14] J. D. Fernández, P. Schneider, et al. The DBpedia Wayback Machine. In *SEMANTiCS*, pages 192–195, 2015.
- [15] S. Gao, M. Chen, et al. SPARQLT and its User-Friendly Interface for Managing and Querying the History of RDF Knowledge Bases. In *ISWC (Demo Track)*, 2015.
- [16] F. Grandi. T-SPARQL: a TSQL2-like Temporal Query Language for RDF. *GraphQ*, pages 21–30, 2010.
- [17] C. Gutierrez, C. A. Hurtado, et al. Introducing Time into RDF. *TKDE*, 19(2):207–218, 2007.
- [18] J. Hoffart, F. M. Suchanek, et al. YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. *Artif. Intell.*, 194:28–61, 2013.
- [19] L. Jiang, B. Salzberg, et al. The BT-tree: A Branched and Temporal Access Method. In *VLDB*, pages 451–460, 2000.
- [20] C. Kang, A. Pugliese, et al. STUN: Spatio-Temporal Uncertain (Social) Networks. In *ASONAM*, pages 543–550, 2012.
- [21] D. B. Lomet, R. S. Barga, et al. Immortal DB: Transaction Time Support for SQL server. In *SIGMOD*, pages 939–941, 2005.
- [22] D. B. Lomet, M. Hong, et al. Transaction Time Indexing with Version Compression. *PVLDB*, 1(1):870–881, 2008.
- [23] G. Moerkotte and T. Neumann. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *VLDB*, pages 930–941, 2006.
- [24] M. A. Nascimento and M. H. Dunham. Indexing Valid Time Databases via B<sup>+</sup>-Trees. *TKDE*, 11(6):929–947, 1999.
- [25] S. B. Navathe and R. Ahmed. A Temporal Relational Model and a Query Language. *Inf. Sci.*, 49(1-3):147–175, 1989.
- [26] T. Neumann and G. Moerkotte. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. *ICDE*, pages 984–994, 2011.
- [27] T. Neumann and G. Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *VLDBJ*, 19(1):91–113, 2010.
- [28] J. Pérez, M. Arenas, et al. Semantics and complexity of SPARQL. *TODS*, 34(3), 2009.
- [29] M. Perry, A. P. Sheth, et al. Supporting Complex Thematic, Spatial and Temporal Queries over Semantic Web Data. In *GeoS*, pages 228–246, 2007.
- [30] A. Pugliese, O. Udreă, et al. Scaling RDF with Time. In *WWW*, pages 605–614, 2008.
- [31] M. Stocker, A. Seaborne, et al. SPARQL Basic Graph Pattern Optimization using Selectivity Estimation. In *WWW*, pages 595–604, 2008.
- [32] J. Tappolet and A. Bernstein. Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL. In *ESWC*, pages 308–322, 2009.
- [33] D. Toman. Point vs. Interval-based Query Languages for Temporal Databases. In *PODS*, pages 58–67, 1996.
- [34] T. Tzouramanis, Y. Manolopoulos, et al. Overlapping B<sup>+</sup>-Trees: An Implementation of a Transaction Time Access Method. *DKE*, 29(3):381–404, 1999.
- [35] O. Udreă, D. R. Recupero, et al. Annotated RDF. *TOCL*, 11(2):10:1–10:41, 2010.
- [36] K. Wilkinson, C. Sayers, et al. Efficient RDF Storage and Retrieval in Jena2. In *SWDB*, pages 131–150, 2003.
- [37] J. Yang and J. Widom. Incremental Computation and Maintenance of Temporal Aggregates. In *ICDE*, pages 51–60, 2001.
- [38] P. Yuan, P. Liu, et al. TripleBit: a Fast and Compact System for Large Scale RDF Data. *PVLDB*, 6(7):517–528, 2013.
- [39] D. Zhang, A. Markowetz, et al. Efficient Computation of Temporal Aggregates with Range Predicates. In *PODS*, pages 237–245, 2001.
- [40] D. Zhang, A. Markowetz, et al. On Computing Temporal Aggregates with Range Predicates. *TODS*, 33(2):12:1–12:39, 2008.
- [41] D. Zhang, V. J. Tsotras, et al. Efficient Temporal Join Processing Using Indices. In *ICDE*, pages 103–113, 2002.
- [42] X. Zhou, F. Wang, et al. Efficient Temporal Coalescing Query Support in Relational Database Systems. In *DEXA*, pages 676–686, 2006.

# Efficient Computation of Containment and Complementarity in RDF Data Cubes

Marios Meimaris  
<sup>1</sup>University of Thessaly,  
<sup>2</sup>ATHENA Research Center,  
Greece  
m.meimaris@imis.athena-  
innovation.gr

George Papastefanatos  
ATHENA Research Center,  
Greece  
gpapas@imis.athena-  
innovation.gr

Panos Vassiliadis  
University of Ioannina,  
Greece  
pvassil@cs.uoi.gr

Ioannis Anagnostopoulos  
University of Thessaly,  
Greece  
janag@ucg.gr

## ABSTRACT

Multidimensional data are published in the web of data under common directives, such as the Resource Description Framework (RDF). The increasing volume and diversity of these data pose the challenge of finding relations between them in a most efficient and accurate way, by taking into advantage their overlapping schemes. In this paper we define two types of relationships between multidimensional RDF data, and we propose algorithms for efficient and scalable computation of these relationships. Specifically, we define the notions of *containment* and *complementarity* between points in multidimensional dataspace, as different aspects of relatedness, and we propose a baseline method for computing them, as well as two alternative methods that target speed and scalability. We provide an experimental evaluation over real-world and synthetic datasets and we compare our approach to a SPARQL-based and a rule-based alternative, which prove to be inefficient for increasing input sizes.

## Categories and Subject Descriptors

H.2.8 [Information Systems Applications]: Database Management—Database Applications

## Keywords

RDF, Multidimensional Data, OLAP, Information Extraction, Algorithms, Performance

## 1. INTRODUCTION

Over the past few years, a wide range of public and private bodies such as statistical authorities, academic institutions, financial organizations and pharmaceutical companies adopt RDF and the Linked Data (LD) paradigm [9, 31] to enable

public access to multidimensional data in a variety of domains, including socio-economics, demographics, enterprise OLAP, clinical trials and health data. The published data enables third parties to combine information, perform analytics over different datasets and gain insights to assist data journalism, industry data management, and evidence-based policy making [25, 4, 27].

Multidimensional data are usually treated under the OLAP prism, where they are represented as *observations* that are instantiated over pre-defined *dimensions* and *measures* [7]. The dimensions provide context to the measures and are structured in hierarchies of different granularity *levels*. For example, a dataset that measures population broken down by locations and time periods, will consist of two dimensions, namely *location* and *time-period*, pertaining to granularity levels, such as countries, regions, and cities, or decades, years and quarters, respectively. A combination of fixed dimension levels is referred to as a *cube*; it contains the set of observations that instantiate these levels, such as the *populations of EU countries in the last decade*, or the *unemployment of EU cities in 2014*. All different combinations of dimension levels form a hierarchical cube *lattice*, where cubes are related with ancestry links.

In the context of Linked Data, the modelling recommendation is the Data Cube Vocabulary (QB) [9]. This provides a common meta-schema for mapping multidimensional data to RDF, enabling the representation of datasets, dataset schemas, dimensions, dimension hierarchies (e.g. codelists), measures, observations and slices (parts of datasets). An example observation can be seen in Listing 1, in which observation *obs1* measures the population of Germany in 2001. The values for 2001 (ex:Y2001) and Germany (ex:DE) are URIs from an example code list. The reuse of common URIs for handling multidimensional elements across different sources enables sharing of terms and the use of SPARQL for federating queries over remote datasets, laying the basis for the application of OLAP analytics at web scale.

Currently, there are few works on definitions and techniques for the discovery and classification of relationships between multidimensional observations in RDF settings [18, 32]. Such relationships can provide useful information to the analyst, such as whether an observation contains aggregated data with respect to other observations, and when two ob-

servations measuring different phenomena are complementary and can be combined. Assume that a data journalist is working on the issue of unemployment in different parts of the world and wants to explore whether unemployment rates relate to the population of a city or a country. Assume now that our journalist obtains dozens of datasets on the topic from various sources and wants to see whether and how they can be combined to facilitate his task. Although the journalist has defined his own reference dimension hierarchies<sup>1</sup> as shown in Figure 1, and converted the incoming data values to them via a simple script, it is still unclear to him that among the dozens of datasets he has collected ( $D_1$ ,  $D_2$  and  $D_3$  of Figure 2), coming from different RDF sources, can be combined to support his task.

In our example, observation  $o_{11}$  shares the same dimension values with  $o_{31}$ , but they measure two different facts, which can be complemented. Furthermore, observations  $o_{21}$ ,  $o_{22}$  that measure unemployment for 2011 in Greece and Italy, respectively, contain observations  $o_{32}$ ,  $o_{33}$  that measure unemployment in Athens and Rome for a sub-period of the same year, although  $o_{21}$ ,  $o_{22}$  measure poverty as well. The resulting table can be seen in Figure 3. This knowledge gives insights on how observations are related across datasets, how OLAP operations (roll-ups, drill-downs) can be applied in order to navigate and explore remote cubes, make observations comparable, provide recommendations for on-line browsing and quantify the degree of relatedness between data sources. Furthermore, materialization of these relationships helps speed up online exploration, and computation of k-dominance as defined in [6], and skylines or k-dominant skylines. Especially in the case of skylines, computation of containment between observations provides a means to directly access skyline, or k-dominant skyline points in collections of large web data.

However, placing different data into the same context and finding hidden knowledge is difficult and computationally challenging [4]. The detection of pair-wise relationships between observations is inherently a quadratic task; typically every observation from one dataset must be compared to all others within the same or from different datasets. The use of traditional ways such as SPARQL- or rule-based techniques becomes inefficient as the number of observations and the number of cubes increases. Our preliminary experiments employing recursive SPARQL queries with property paths and negation, indicate that even for 20k observations from 7 datasets it can take more than one hour to complete in commodity hardware, while the same tasks time-out for larger numbers of instances. The same holds when inference-based techniques are used: OWL-based reasoning lacks the expressivity for complex property-based inference, while rule-based reasoning such as SWRL [15] and Jena Rules [5] does not scale adequately due to the transitive nature of the relationships and the universal quantification needed to encode the conditions; the search space expands exponentially [12]. From the above, there is a clear need for establishing new efficient methods for computing pair-wise relationships be-

<sup>1</sup>In realistic settings, schema alignment is often necessary. Two prominent cases where it is used in practice include: (a) traditional BI environments, where all dimensions provide a reconciled *dimension bus*, and (b) user-initiated data collections from the web. In both cases, incoming data have to be translated to a reference vocabulary, *before* being used for further analysis.

```
ex:obs1 a qb:Observation ;
qb:dataSet ex:dataset ;
ex:time ex:Y2001 ;
sdmx-attr:unitMeasure ex:unit ;
ex:geo ex:DE ;
ex:population "82,350,000"^^xmls:integer .
```

Listing 1: Example RDF Data Cube observation

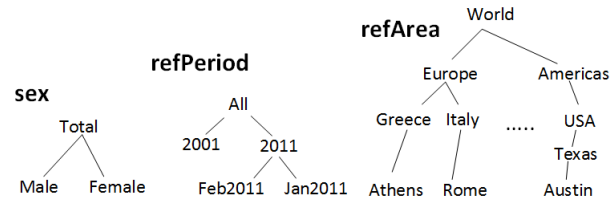


Figure 1: Hierarchical code list for the dimensions in Figure 2.

tween RDF observations that can scale up to the numbers and variety of datasets currently published on the web of data.

**Approach Overview.** In this paper, *we address the problem of efficiently computing containment and complementarity between RDF observations*, extending the preliminary work presented in [22]. Specifically, given an observation  $o_g$ , and a set of observations  $O$  from different sources, we define two relationships, namely *containment* and *complementarity*. A *containment* relationship captures whether an observation contains aggregated information with respect to other observations. It determines if values from the dimensions of  $o_g$  contain fully or partially values of the dimensions of another observation, thus enabling rollup and drilldown operations, directly or indirectly, as well as assigning them a hierarchy-based similarity metric. A *complementarity* relationship captures whether the measures of two observations can be combined together, providing comparable data for the same points in the multidimensional space. More specifically, we extend the notion of schema complement, defined in [10] and apply it at the instance level to discover complementary observations.

We first present a  $O(n^2)$  baseline technique to calculate these properties. Then, we introduce two alternatives that prove to be more efficient and scalable in terms of computation time. The first performs observation *clustering* and limits comparisons between observations in the same clusters. The second uses a *multidimensional lattice* to assign observations to specific permutations of dimensions and levels, and prune the search space by checking for restrictions at the schema-level. We experimentally evaluate these 3 techniques over a set of 7 real-world multidimensional datasets and compare them in terms of efficiency with traditional techniques, namely with a SPARQL-based and a rule-based approach. Finally, we evaluate the scalability of our approach in an artificially generated dataset.

**Contributions.** The contributions of this paper can be summarized as follows:

- we extend the notions of *full* and *partial containment* between two observations defined in [22] as derivatives of the hierarchical relationships between their dimension values, and observation *complementarity* as a

D <sub>1</sub>				
	refArea	refPeriod	sex	ex:Population
o <sub>11</sub>	Athens	2001	Total	5M
o <sub>12</sub>	Austin	2011	Male	445K
o <sub>13</sub>	Austin	2011	Total	885K

D <sub>2</sub>				
	refArea	refPeriod	ex:Unemployment	ex:Poverty
o <sub>21</sub>	Greece	2011	26%	15%
o <sub>22</sub>	Italy	2011	20%	10%

D <sub>3</sub>			
	refArea	refPeriod	ex:Unemployment
o <sub>31</sub>	Athens	2001	10%
o <sub>32</sub>	Athens	Jan2011	30%
o <sub>33</sub>	Rome	Feb2011	7%
o <sub>34</sub>	Ioannina	Jan2011	15%
o <sub>35</sub>	Austin	2011	3%

Figure 2: Candidate relationships between observations.

	refArea	refPeriod	sex	ex:Unemployment	ex:Poverty	ex:Population
o <sub>21</sub>	Greece	2011	Total	26%	15%	-
contains:	contains:	contains:				
--o <sub>32</sub>	--Athens	--Jan2011	Total	30%	-	-
--o <sub>34</sub>	--Ioannina	--Jan2011	Total	15%	-	-
o <sub>22</sub>	Italy	2011	Total	20%	10%	-
contains:	contains:	contains:				
--o <sub>33</sub>	--Rome	--Feb2011	Total	7%	-	-
o <sub>11</sub>	Athens	2001	Total	-	-	5M
complements						
--o <sub>31</sub>	Athens	2001	Total	10%	-	-
o <sub>13</sub>	Austin	2011	Total	-	-	885K
complements						
--o <sub>35</sub>	Austin	2011	Total	3%	-	-

Figure 3: Derived containment and complementarity relationships from datasets  $D_1$ ,  $D_2$  and  $D_3$  of Figure 2.

means of correlation between data points,

- we present a baseline, data-driven technique for computing these properties in memory,
- we introduce two alternative techniques that target efficiency,
- we evaluate our techniques in terms of efficiency and scalability over real-world and synthetic datasets, and we perform comparisons between them, as well as with a SPARQL-based and a rule-based approach.

The remainder of this paper is organized as follows. Section 2 presents preliminaries and formulates the problem of containment and complementarity computation between RDF cubes. Section 3 presents the baseline algorithm and two proposed improvements for computing the defined relationships. Section 4 describes the performed experiments and evaluation. Finally, section 5 discusses related work and section 6 concludes this paper.

## 2. PRELIMINARIES

The problem space is composed of  $n$  datasets, each of which contains a number of observations, structured in one or more cubes. A single dataset is composed of the schema part (i.e. dimensions, measures and attributes), and the data part (i.e. observations or facts). Dimension values are provided by code lists with hierarchical structure, whereas flat code lists pertain to dimensions with exactly one level.

**Definition 1** (Dataset Structure): Let  $D = \{D_1, \dots, D_n\}$  be the set of all input datasets. A dataset  $D_i \in D$  is composed by the set of observations,  $O_i$ , and the set of schema definitions,  $S_i$ , and  $O = \{O_1, \dots, O_n\}$  and  $S = \{S_1, \dots, S_n\}$  are the sets of all observations and schema definitions in  $D$ . Furthermore, a schema  $S_i$  consists of the sets of dimensions  $P_i$  and measures  $M_i$  defined in  $D_i$ , i.e.,  $S_i = \{P_i, M_i\}$ . Let  $P = \bigcup_{i=1}^n P_i = p_1, p_2, \dots, p_k$  and  $M = \bigcup_{i=1}^n M_i = m_1, m_2, \dots, m_l$  be the set of all  $k$  distinct dimensions and  $l$  measure properties in  $D$ . Any  $p_j \in P, m_j \in M$  can belong to more than one

$S_i$ , as dimension and measure properties are reused among sources. In our example,  $S_1$ ,  $S_2$  and  $S_3$  are the schemata of datasets  $D_1$ ,  $D_2$  and  $D_3$ , and dimensions  $refArea$  and  $refPeriod$  belong to all three schemata. Similarly, measure  $ex:unemployment$  belongs to both  $M_2$  and  $M_3$ . An observation  $o \in O_i$  is an entity that instantiates all dimension and measure properties defined in  $S_i$ . The value that observation  $o_i$  has for dimension  $p_j$  is  $h_j^i$ . In the example, the values in the white cells represent dimension values (e.g. "Athens" is a value for dimension  $refArea$ ), while grey cells represent measured values (e.g. 10% unemployment).

**Definition 2** (Hierarchies): Each dimension  $p_j \in P$  takes values from a code list, i.e. a set of fixed values coded by URIs,  $C(p_j) = \{c(p_j)_1, \dots, c(p_j)_m\}, j = 1 \dots k$ , (for simplicity we write  $c_{ji}$  instead of  $c(p_j)_i$ ). Each code list defines a hierarchy such that when  $c_{ji} > c_{jm}$ , then  $c_{ji}$  is an ancestor of  $c_{jm}$ . Furthermore, we define  $c_{jroot}$  as the top concept in the code list of  $p_j$ , i.e., an ancestor of all other terms in the coded list, such that  $\forall c_{ji} : c_{jroot} > c_{ji}$ . This kind of ancestry is reflexive, i.e.  $\forall c_{ji} : c_{ji} > c_{ji}$ . In Figure 2, sample code lists are depicted for the three dimensions shown in Figure 1. In our example,  $Greece > Athens$ ,  $Ioannina$  and  $Italy > Rome$ .

A *complementarity* relationship captures whether two observations measure different facts about the same set of dimension instances. In the motivating example of Figure 1,  $o_{11}$  and  $o_{31}$  are complementary because they measure different facts (population and unemployment respectively) about the city of Athens in 2001. The fact that  $o_{11}$  refers to all values from the sex dimension does not provide any further specialization and is inherently found in  $o_{31}$  as well. This is captured in the following definition.

**Definition 3** (Observation Complement): Given two observations  $o_a$  and  $o_b$  and their dimensions  $P_a$  and  $P_b$ ,  $o_a$  complements  $o_b$  when the following conditions hold:

$$\forall p_i \in P_a \cap P_b : h_a^i = h_b^i \quad (1)$$

$$\forall p_j \in P_b \setminus P_a : h_b^j = c_{jroot} \quad (2)$$

Therefore,  $(1) \wedge (2) \Rightarrow Compl(o_a, o_b)$ . We denote this with  $Compl(o_a, o_b)$ . Definition 1 states that the shared dimensions must have the same values as stated in (1), and all other dimension values of  $o_b$  must be equal to the root of the dimension hierarchy, providing no further specialization, as stated in (2). For example, an observation measuring poverty in Greece is observation complement with an observation measuring the population in Greece for all human genders. In our example, observations  $o_{11}$  and  $o_{31}$  are complementary, in that they measure different things for Athens in 2001. Condition (2) holds for  $o_{31}$  in the sex dimension, where absence of the dimension implies existence of the root value  $c_{jroot}$  (i.e. no specialization).

A *containment* relationship captures whether an observation aggregates the measures of the contained observations. For example, measuring the population of Greece implicitly contains all the populations of Greece’s sub-regions. We distinguish between *full* and *partial* containment. The former denotes that all contained observations must be aggregated (e.g., a roll-up operation) for being observation complement with the containing one, while the latter denotes that both contained and containing observation must be rolled-up on their disjoint dimensions for becoming complementary. These two concepts are defined as follows.

**Definition 4** (Partial and full containment): Given two observations  $o_a$  and  $o_b$ , their dimensions  $P_a$  and  $P_b$  and their measures  $M_a$  and  $M_b$ , partial containment between  $o_a$  and  $o_b$  exists when the following conditions hold:

$$M_a \cap M_b \neq \emptyset \quad (3)$$

$$\exists p_i \in P_a \cap P_b : h_a^i \succ h_b^i \quad (4)$$

Therefore,  $(3) \wedge (4) \Rightarrow Cont_{partial}(o_a, o_b)$ . An observation  $o_a$  partly contains  $o_b$  when there is at least one  $M_i$  shared between  $o_a$  and  $o_b$  as stated in (3), and there exists at least one dimension whose value for  $o_a$  is a hierarchical ancestor of the value of the same dimension in  $o_b$ , as stated in (4). We denote this as  $Cont_{partial}(o_a, o_b)$ . In the example, observation  $o_{21}$  partially contains  $o_{31}$ , because *Greece* contains *Athens* but *2001* does not contain *2011*. By rolling up on the *refPeriod* dimension, the two observations become complementary.

Similarly, full containment between  $o_a$  and  $o_b$  exists when the same preconditions (3)-(4) hold along with a universal restriction on (4) for all dimension values, i.e:

$$\forall p_i \in P_a \cap P_b : h_a^i \succ h_b^i \quad (5)$$

Therefore,  $(3) \wedge (4) \wedge (5) \Rightarrow Cont_{full}(o_a, o_b)$ . An observation  $o_a$  fully contains  $o_b$  when there is one  $M_i$  shared between  $o_a$  and  $o_b$  as stated in (3), and values of all dimensions for  $o_a$  are hierarchical ancestors of the values for the same dimensions in  $o_b$  as stated existentially in (4) and universally in (5). We denote this with  $Cont_{full}(o_a, o_b)$ . Observe that the containment property is not symmetric and that given  $Cont_{full}(o_a, o_b)$ , then  $Cont_{full}(o_b, o_a)$  is not implied. In the example,  $o_{21}$  fully contains  $o_{32}$  and  $o_{34}$ . The notation is summarized in Table 1. Based on the above, our problem is formulated as follows.

**Problem:** Given a set  $D$  of source dataset, and a set  $O$  of observations in  $D$ , for each pair of observations  $o_i, o_j \in O, i \neq j$ , assess whether a)  $Cont_{full}(o_i, o_j)$ , b)  $Cont_{partial}(o_i, o_j)$  and c)  $Compl(o_i, o_j)$ . In the following section, we provide our techniques for computing these properties.

Table 1: Notation

Notation	Description
$o_i$	The i-th observation in a set $O$
$P_i$	The set of dimensions for observation $o_i$
$p_i$	The i-th dimension in a set $P_k$
$M_i$	The set of measures for observation $o_i$
$h_a^i$	Value of dimension $p_i$ for observation $o_a$
$h_a^i \succ h_b^i$	$h_a^i$ is a parent of $h_b^i$
$c_{jroot}$	The root value (ALL) for dimension $p_j$
$Compl(o_a, o_b)$	Observation $o_a$ complements $o_b$
$Cont_{full}(o_a, o_b)$	Observation $o_a$ fully contains $o_b$
$Cont_{partial}(o_a, o_b)$	Observation $o_a$ partially contains $o_b$

### 3. ALGORITHMS

In this section, we present three approaches for computing containment and complementarity relationships. We first present a baseline method of  $O(n^2)$  complexity and then we propose two alternatives in order to achieve scalable and fast solutions.

#### 3.1 Baseline

Our *baseline* algorithm performs pair-wise computations between all pairs of observations in a given data space. To achieve this, we model observations as bit vectors in a multidimensional data space represented by an occurrence matrix  $\mathbf{OM}$ , where rows represent observations and columns are values in their dimensions as well as ancestor values in their dimension hierarchies.  $\mathbf{OM}$  encodes the occurrence of a dimension value in an observation as well as the hierarchy in which this value belongs to, by setting the value of 1 to all columns that are ancestors of this value.

Dimension alignment is often required to take place before this step, in order to have a reconciled dimension bus in the feature space. As discussed later on in the experiments section, we employ a state-of-the-art tool for performing the interlinking of dimension values across different datasets. Note that this procedure is orthogonal to the work presented in this paper. The problem of dimension alignment, and record linkage in general, is a separate research problem and is not underestimated; however, (a) data conversion is feasible and amortized over time (esp., if data collection from "favorite" sources is recurring), and (b) the focus of this paper is on analytics and not data integration.

**Constructing the Occurrence Matrix.** Each observation  $o_i$  defines a bit vector  $\mathbf{o}_i$  and all  $\mathbf{o}_i \in O$  comprise the occurrence matrix  $\mathbf{OM}$  of  $|O| \times |C|$  dimensions, defined by the occurrences of code list values in the respective dimensions. Each value  $c_{ji} \in C_j$  corresponding to dimension  $p_j$  is a feature, i.e., a column in  $\mathbf{OM}$ . Hierarchical containment is encoded into  $\mathbf{OM}$  using a bottom-up algorithm that assigns a value of 1 in column  $c_{ji}$  and all of its parents, if the value  $h_a^i$  of the dimension  $p_j$  of  $o_a$  is equal to the feature  $c_{ji}$ . Finally, we set the  $c_{jroot}$  columns of all observations that do not contain  $p_j$  in their schema. This means that dimensions not appearing in a schema are mapped to the *top* concept (i.e. root term), including all possible values.

Matrix  $\mathbf{OM}$  can be further broken down in separate sub-matrices for each code list, i.e.,  $\mathbf{OM} = [\mathbf{OM}_1, \dots, \mathbf{OM}_{|C|}]$  for all dimensions, where  $\mathbf{OM}_i$  is a sub-matrix that represents occurrences for all values of dimension  $p_i$ . Matrix  $\mathbf{OM}$  for the example of Figure 2, given the hierarchies shown in Figure 1, is shown in Table 2.  $\mathbf{OM}$  is used as an input for our algorithm that computes a containment matrix. The latter is used for assessing both complementarity and containment



properties.

**Computation of containment.** A containment matrix  $\mathbf{CM}_i$  captures pair-wise containment between observations, for dimension  $p_i$ . If a cell  $\mathbf{CM}_i[a, b] > 0$ , then  $o_a$  and  $o_b$  are related via containment for  $p_i$ . To compute  $\mathbf{CM}_i$ , we apply a conditional function between pairs of rows, as bit vectors. Containment exists if the logical AND operation between the bit vectors of the rows returns one of the two bit vectors. We define this check as the following conditional function applied on  $o_a$  and  $o_b$ :

$$sf(o_a, o_b) |_{(p_i)} = \begin{cases} 1, & \text{if } a \wedge b = b \\ 0, & \text{otherwise} \end{cases}$$

Notice that we apply  $sf$  for  $o_a$  and  $o_b$  for dimension  $p_i$  in  $\mathbf{OM}_i$ . Application of this function for each dimension returns a set of  $|P|$  containment matrices,  $\mathbf{CM}_1, \dots, \mathbf{CM}_k$ . By adding these we get the *Overall Containment Matrix*  $\mathbf{OCM}$ :

$$\mathbf{OCM} = \frac{\sum_{i=1}^k u_i \mathbf{CM}_i}{\sum_{i=1}^k u_i}$$

$\mathbf{OCM}$  values are normalized; we derive *full containment* when a cell has a value of 1, and *partial containment* when a cell has a value between 0 and 1 (non-inclusive) To assert which particular dimensions exhibit containment in a partial relationship, we examine the cells in  $\mathbf{CM}_i$  being equal to 1. The occurrence of a 0 value indicates that full containment and complementarity can not hold. Note also that measure overlaps can be easily detected with a simple lookup. The construction of the  $\mathbf{OCM}$  matrix is explained in Algorithm 1 *computeOCM*. We then calculate containment and complementarity using the  $\mathbf{OCM}$ -based Algorithm 2 *baseline*.

**Computation of complementarity.** Following the definition of observation complementarity, and given that the non-appearing values are set to  $c_{jroot}$ , we use  $\mathbf{OCM}$  to assess whether a pair of observations exhibits full containment in both directions, i.e.  $Cont_{full}(o_a, o_b)$  and at the same time  $Cont_{full}(o_b, o_a)$ .

**Analysis.** The baseline algorithm has  $\Theta(n^2)$  time complexity for  $n$  observations, because it visits each pair of observations exactly once, if all three types of relationships are to be retrieved. The occurrence matrix  $\mathbf{OM}$  requires  $\Theta(nk)$  space for  $n$  observations and  $k$  features, given a simple array implementation. However, for large  $k$  the matrix tends to become sparse, therefore a sparse matrix implementation would yield a significant decrease in the required space. In practice, if at least one 0 is found in the  $\mathbf{CM}$  matrices, the pair under comparison is no longer candidate for either full containment or complementarity. We keep this in an index table to avoid meaningless comparisons in the next step, if only full containment or complementarity is to be computed. Finally, if we are not interested in identifying the particular dimensions that exhibit containment in a partial containment relationship, we skip iterating through the  $\mathbf{CM}_i$  matrices and identifying the zero values, and just keep the value as a metric of the degree of partial containment that the pair exhibits.

### 3.2 Computation with Clustering

The baseline algorithm becomes inefficient for large datasets and does not scale due to its quadratic complexity. For this reason, we improve its performance by applying a pre-processing step that prunes the search space by limiting

---

#### Algorithm 1 *computeOCM*

---

**Input:** An occurrence matrix  $\mathbf{OM}$  with  $N$  rows and  $|C|$  columns, a set  $P$  of dimensions and their start indices in  $\mathbf{OM}$

**Output:** A  $N \times N$  overall containment matrix  $\mathbf{OCM}$

```

1: initialize  $\mathbf{OCM}[\ ][\ ]$ 
2: for each  $p_i \in P$  do
3:   initialize  $\mathbf{CM}_{p_i}[\ ][\ ]$   $\triangleright$  one containment matrix per dimension
4:   for each  $o_j \in \mathbf{OM}_{p_i}$  do  $\triangleright p_i$  defines a start index
5:     for each  $o_k \in \mathbf{OM}_{p_i}$  do
6:       if  $o_j$  AND  $o_k == o_j$  then
7:          $\mathbf{CM}_{p_i}[j][k] \leftarrow 1$ 
8:       else
9:          $\mathbf{CM}_{p_i}[j][k] \leftarrow 0$ 
10:     $\mathbf{OCM}[j][k] \leftarrow \mathbf{OCM}[j][k] + (\mathbf{CM}_{p_i}) / |P|$   $\triangleright$ 
    normalize for # of dimensions
return  $\mathbf{OCM}$ 

```

---



---

#### Algorithm 2 *baseline*

---

**Input:** A  $N \times N$  overall containment matrix  $\mathbf{OCM}$ .

**Output:**  $S_F, S_P, S_C$  sets of fully, partial containment and complementarity relationships, and optionally a map of partial containment relationships  $map_P$  with the dimensions they exhibit containment in.

```

1: initialize  $S_F, S_P, S_C$ 
2: for each  $o_i \in \mathbf{OCM}$  do
3:   for each  $o_j \in \mathbf{OCM}$  &&  $o_j \neq o_i$  do
4:     if  $\mathbf{OCM}[i][j] == 1$  then
5:        $S_F = S_F \cup (o_i, o_j)$ 
6:     if  $\mathbf{OCM}[j][i] == 1$  then
7:        $S_C = S_C \cup (o_i, o_j)$ 
8:     else if  $\mathbf{OCM}[i][j] > 0$  then
9:        $S_P = S_P \cup (o_i, o_j)$ 
10:    for each  $p_i \in P$  do
11:      if  $\mathbf{CM}_{p_i}[i][j] == 1$  then
12:         $map_P(o_i, o_j, p_i) = true$ 
13:    else continue
return  $S_F, S_P, S_C, map_P$ 

```

---

comparisons. Algorithm 3 presents our approach. It takes as input the occurrence matrix containing the bit vectors for the observations and applies a clustering algorithm that splits the matrix into smaller occurrence matrices. It then applies the *computeOCM* and *baseline* algorithms to each separate cluster. This way comparisons between pairs of observations are limited within each cluster. This is shown in Algorithm 3.

**Clustering configuration.** In order to compute clusters, state-of-the-art clustering algorithms can be employed. We have experimented with k/x-means [26], bottom-up hierarchical clustering and fast canopy clustering [21] for the purposes of evaluating our algorithms. To avoid introducing extra time overhead in the containment and complementarity computation stage, we approximate the algorithm by clustering a sample of the data and assigning the remaining points to the identified clusters.

**Analysis.** Time and space complexity of the clustering step depends on the complexity of the chosen clustering algorithm, the number of clusters and the distribution of observations in the clusters. The baseline algorithm will run times equal to the number  $k$  of clusters. However, the dis-

Table 2: Matrix OM for the example of Figure 2

	refArea										refPeriod					sex		
	WLD	EUR	AM	GR	IT	Ath	Rom	US	TX	Aus	ALL	2001	2011	Jan11	Feb11	T	F	M
<i>obs11</i>	1	1	0	1	0	1	0	0	0	0	1	1	0	0	0	1	0	0
<i>obs12</i>	1	0	1	0	0	0	0	1	1	1	1	0	1	0	0	1	0	1
<i>obs21</i>	1	1	0	1	0	0	0	0	0	0	1	0	1	0	0	1	0	0
<i>obs22</i>	1	1	0	0	1	0	0	0	0	0	1	0	1	1	0	1	0	0
<i>obs31</i>	1	1	0	1	0	1	0	0	0	0	1	1	0	0	0	1	0	0
<i>obs32</i>	1	1	0	1	0	1	0	0	0	0	1	0	1	1	0	1	0	0
<i>obs33</i>	1	1	0	0	1	0	1	0	0	0	1	0	1	0	1	1	0	0

Table 3: (a) Matrix  $CM_1$  for dimension refArea of the example of Figure 1, (b) Matrix  $OCM$  for the example of Figure 1

(a)								(b)							
	<i>obs11</i>	<i>obs12</i>	<i>obs21</i>	<i>obs22</i>	<i>obs31</i>	<i>obs32</i>	<i>obs33</i>		<i>obs11</i>	<i>obs12</i>	<i>obs21</i>	<i>obs22</i>	<i>obs31</i>	<i>obs32</i>	<i>obs33</i>
<i>obs11</i>	1	0	0	0	1	1	0	<i>obs11</i>	1	0	0.33	0.33	1	0.66	0.33
<i>obs12</i>	0	1	0	0	0	0	0	<i>obs12</i>	0.33	1	0.66	0.66	0.33	0.66	0.66
<i>obs21</i>	1	0	1	0	1	1	0	<i>obs21</i>	0.66	0.33	1	0.66	0.66	1	0.66
<i>obs22</i>	0	0	0	1	0	0	1	<i>obs22</i>	0.33	0	0.33	1	0.33	0.66	0.66
<i>obs31</i>	1	0	0	0	1	1	0	<i>obs31</i>	1	0	0.33	0.33	1	0.66	0.33
<i>obs32</i>	1	0	0	0	1	1	0	<i>obs32</i>	0.66	0	0.33	0.66	0.66	1	0.33
<i>obs33</i>	0	0	0	0	0	0	1	<i>obs33</i>	0.33	0.33	0.66	0.33	0.33	0.33	1

**Algorithm 3** *clustering*

**Input:** An occurrence matrix **OM** with  $N$  rows and  $|C|$  columns.

**Output:**  $S_F, S_P, S_C$  sets for fully, partial containment and complementarity relationships.

- 1:  $clusters[] \leftarrow cluster(OM, \dots)$  ▷ e.g. k-means
  - 2: initialize **OCM**  $\leftarrow 0$
  - 3: **for**  $i = 1$  to  $clusters.size$  **do**
  - 4:   **OCM** $_i \leftarrow computeOCM(clusters[i], P)$
  - 5:    $S_{Fi}, S_{Pi}, S_{Ci} \leftarrow baseline(OCM_i)$
  - 6:    $S_F, S_P, S_C \leftarrow (S_F, S_P, S_C) \cup (S_{Fi}, S_{Pi}, S_{Ci})$
- return**  $S_F, S_P, S_C$

tribution of observations in clusters is not known for a given collection of datasets. In the centroid-based case (canopy, k/x-means), assuming an equal distribution of  $\frac{n}{k}$  observations per cluster, then the time complexity for each cluster is  $\Theta(\frac{n}{k})^2$  thus making the total time complexity  $\Theta(\frac{n^2}{k})$ . Following a rule of thumb where  $k = \sqrt{\frac{n}{2}}$ , this becomes  $\Theta(n^{1.5})$ , at the cost of information loss, as will be shown in the experiments.

### 3.3 Computation with Cube Masking

The efficiency achieved by the clustering approach can result in lower recall levels as observations that are likely to be related might end up in different clusters. In this section, we propose a scalable algorithm that greatly increases speed and at the same time maintains 100% recall. The approach is based on the hierarchical characteristics of the dimension values. We first construct a lattice of all possible level combinations for all the dimension values that appear at least once in the input. Then, we map all observations to their respective lattice nodes (i.e. cubes) and check if the lattice nodes, rather than the observations, meet the containment and complementarity criteria. If so, comparisons need only be performed between observations belonging to these lattice nodes. To demonstrate this further, consider the lattice shown in Figure 3 that corresponds to the dimension hierarchies of Figure 2. Each node corresponds to a combination of the levels of all dimensions. Therefore, node "210" repre-

sents the cube of all observations that pertain to values at level 2 for *refArea*, level 1 for *refPeriod* and level 0 for *sex*. In the cases of full containment and complementarity, we do not need to compare observations that belong to lattice nodes that are not hierarchically related, such as node "121" with node "311". In the case of partial containment we look for at least one dimension inclusion (i.e. path) in the lattice before comparing the contents.

Based on these observations, we propose the *cubeMasking* algorithm (Algorithm 4) that works in the following steps: i) It first identifies cubes in the input datasets and populate the lattice; ii) it maps observations to cubes; iii) it iterates through cubes and does a pair-wise check for the containment criterion and finally iv) it compares observations between pairs of cubes that fulfils this criterion. In order to perform these steps, we use a hash table to ensure that a value's level can be checked in constant time. We then go on to identify the cubes and build the lattice by iterating through all observations and extracting their unique combinations of dimensions and levels. To do so, we apply a hash function on each observation that both identifies and populates its cube at the same step. Finally, we iterate through the identified cubes and by doing a pair-wise check for the containment and complementarity criteria, all meaningful observation comparisons are identified. This can be seen in Algorithm 4.

**Analysis.** Instance-level comparisons are limited between pairs of comparable cubes that are identified by the algorithm. At worst, the maximum number of cubes for a set of input datasets is the number of permutations of dimensions and levels, i.e.  $k^{(|P|)}$ , where  $k$  is the maximum level of all hierarchies and  $|P|$  is the number of dimensions. Checking for potentially comparable pairs of cubes costs  $O(k^{(2|P|)})$  in the worst case, but for an average of  $bk^{(|P|)}$  present cubes in the input, and  $abk^{(|P|)}$  number of comparable cubes, where  $a$  and  $b$  are constants between 0 and 1, the algorithm will require  $ab^2n^2$  comparisons instead of  $n^2$ .

## 4. PERFORMANCE EVALUATION

**Datasets.** We have experimented with seven real-world datasets taken from the statistics domain. Eurostat Linked

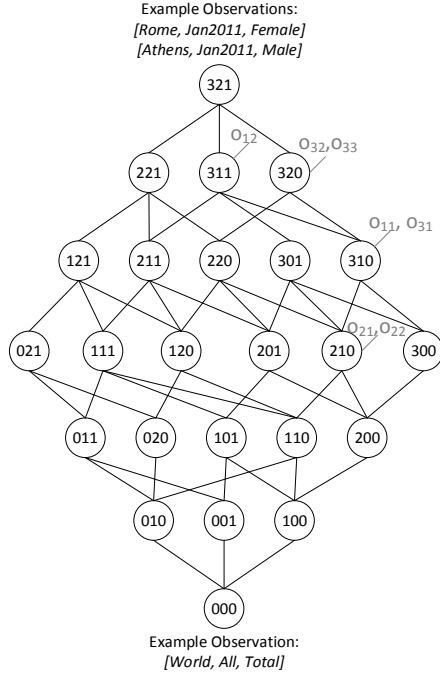


Figure 4: The lattice for the three hierarchies of Figure 2. Observations in Figure 1 are mapped to the appropriate node. The number in each node corresponds to the level of each dimension.

Data Wrapper, the Eurostat database, linked-statistics.gr and World Bank<sup>2</sup> were used as the dataset sources. The datasets were either in RDF form, or converted to RDF. In the case of datasets in CSV format, we follow the approach of [28] for producing RDF QB datasets, although many other popular tools can be used, such as CSV2RDF<sup>3</sup>, OpenCube<sup>4</sup> and Open Refine<sup>5</sup>. We converted CSV column headers to dimension URIs, and rows to observations, by automatically matching cell values to existing code list terms based on their IDs. The datasets contain information of European authorities on unemployment, resident population, number of households, births, deaths and gross domestic product (GDP) on a multitude of dimensions that include locations, dates, citizenship, sex, household size, education level and economic activity. In total, the datasets amount to 250k observations and 2.6k distinct hierarchical values. These exhibit an overlap of 11 dimensions pertaining to common coded lists, and 6 measures.

In our experiments, we consider a separate preprocessing step for the alignment of the schemas and the mapping of the dimension values across the input datasets. In this work we have used LIMES [24], a state-of-the-art link discovery framework commonly used for entity matching tasks in the Web of Data. LIMES is configurable to use SPARQL query restrictions on input data (e.g. *only match nodes of type skos:Concept*), and has a rule language that enables

<sup>2</sup><http://estatwrap.ontologycentral.com/>, <http://data.worldbank.org/>, <http://linked-statistics.gr/>, [http://epp.eurostat.ec.europa.eu/portal/page/portal/statistics/search\\_database](http://epp.eurostat.ec.europa.eu/portal/page/portal/statistics/search_database)

<sup>3</sup><http://www.w3.org/TR/csv2rdf/>

<sup>4</sup><http://opencube-toolkit.eu/>

<sup>5</sup><http://refine.deri.ie/>

---

#### Algorithm 4 *cubeMasking*

---

**Input:** A list  $C$  with all code list terms as they appear in the datasets, a hash table  $levels$  with a mapping of hierarchical values to their levels, and a list  $O$  of  $N$  observations with their descriptions.

**Output:**  $S_F$ ,  $S_P$ ,  $S_C$  sets for full, partial containment and complementarity relationships.

```

1:  $hierarchy \leftarrow createHierarchyTree(C)$  ▷
   iterate through observations once to identify cubes and
   map observations to cubes
2: initialize  $cubeLevels$ ,  $observationsInCubes$ 
3: for each  $o_i \in O$  do
4:   initialize  $cube$ 
5:   for each  $p_j \in P$  do
6:      $cube.p_j \leftarrow levels(o_i, p_j)$ 
7:    $cubeLevels.add(cube)$  ▷ hashing ensures no
   duplicates
8:    $observationsInCubes(o_i) \leftarrow cube$ 
9: for each  $cube_j, cube_k \in cubeLevels$  do
10:  for each  $p_i \in P$  do
11:    if not  $(cube_j.p_i \prec cube_k.p_i)$  then break
12:  for each  $o_i \in cube_j$  do
13:    for each  $o_j \in cube_k$  do
14:       $SF[o_i, o_j] \leftarrow checkFullCont(o_i, o_j)$ 
15:       $SP[o_i, o_j] \leftarrow checkPartialCont(o_i, o_j)$ 
16:       $SC[o_i, o_j] \leftarrow checkCompl(o_i, o_j)$ 
17: return  $S_F, S_P, S_C$ 
18: procedure CREATEHIERARCHYTREE( $C$ )
19:   initialize  $hierarchyTree$ 
20:   for each  $c_i \in C$  do
21:      $hierarchyTree.add(c_i)$ 
22:     for each  $c_i.parent$  do
23:        $hierarchyTree.addParent(c_i, c_i.parent)$ 
24:   return  $hierarchyTree$ 
25: procedure CHECKFULLCONT( $o_i, o_j$ )
26:  for each  $p_i \in P$  do
27:    if not  $hierarchy.isParent(o_i.p_i, o_j.p_i)$  then re-
   turn false
28:  return true
29: procedure CHECKPARTIALCONT( $o_i, o_j$ )
30:  for each  $p_i \in P$  do
31:    if  $hierarchy.isParent(o_i.p_i, o_j.p_i)$  then return
   true
32:  return false
33: procedure CHECKCOMPL( $o_i, o_j$ )
34:  if  $checkFullCont(o_i.p_i, o_j.p_i)$  &&
    $checkFullCont(o_j.p_i, o_i.p_i)$  then return true
35:  return false

```

---

the user to select combinations of distance functions (e.g. the maximum of the *cosine* and *levenshtein* distances). We configured LIMES to match hierarchy nodes by adding their URIs as literal values, and used their cosine distance in order to find close matches based on the identifiers usually found in the suffix part of a URI. The details for each dataset are summarized in Table 4.

**Metrics.** The goal of the experiments was to assess and compare the performance of the proposed algorithms with respect to *execution time* and *recall* of computed relationships. *Execution time* is measured as the time needed for data pre-processing and for computing complementarity and containment properties. *Recall* is calculated as the ratio of computed relationships to actual relationships. However,

Table 4: Dataset dimensions, amount of observations and respective measures

Dataset (# of obs)	refArea	refPeriod	sex	unit	age	economic activi- ties	citizenship	education	household size	measure
$D_1$ (58k)	Y	Y	Y	Y	Y	N	Y	N	N	Population
$D_2$ (4.2k)	Y	Y	N	Y	N	N	N	N	Y	Members
$D_3$ (6.7k)	Y	Y	Y	Y	Y	N	N	Y	N	Population
$D_4$ (15k)	Y	Y	N	Y	N	N	N	N	N	Births
$D_5$ (68k)	Y	Y	Y	Y	Y	N	Y	N	N	Deaths
$D_6$ (73k)	Y	Y	N	Y	N	N	N	N	N	GDP
$D_7$ (21.6k)	Y	Y	N	N	N	Y	N	N	N	Compensation

it is relevant only to the *clustering* algorithm, as the *baseline* and the *cubeMasking* algorithms achieve 100% recall. Note, also, that the precision is 100% for all algorithms as a derivative of the determinism in the relationship definitions. In order to derive recall, we have compared the output of the *clustering* method with the ground truth taken from the *baseline* output. Note that we do not consider any decrease in recall induced by the tool used in the dimension alignment step. Since the recall of our approach is not dependent to the result of the alignment process, we assume that the output of the linking tool (in our case LIMES) achieves 100% recall correctly matching schema elements and dimension values across all datasets.

**Experimental Setting.** Our approach was implemented in Java 1.7, and all experiments were performed on a server with Intel i7 3820 3.6GHz, running Debian with kernel version 3.2.0 and allocated memory of 16GB. We implemented the approaches and performed a series of comparisons between them, starting from an input size of 2k observations, over the original fixed size of dimensions. We continued the experiment with an input size of 20k and then we further increased it with a 20k step. For the *clustering* method, we have experimented with canopy clustering, hierarchical clustering and x-means, all applied on a 10% random sample of the original datasets and empirical studies showed that x-means outperformed the other two methods greatly in terms of recall achieved in comparable time frames. Based on the bit-vector approach, we used the Jaccard coefficient as a similarity metric for our binary feature space. The *cubeMasking* algorithm has been further optimized to limit the number of cube comparisons by storing for each cube, the full set of its children in memory.

**SPARQL-based.** We have experimented with SPARQL queries over a Virtuoso 7.1 instance for computing the relationships. Note that universal quantification as well as recursive querying (i.e. property paths) are necessary to compute full containment and complementarity. Property paths are directly supported by SPARQL 1.1, however, universal quantification must be mimicked by using a negation construct that includes a nested recursion, which makes the queries costly. Furthermore, occurrence of partial containment can be detected by SPARQL queries easily, but it is complicated and costly to quantify it. The SPARQL approach is composed of three SPARQL queries; for simplicity, we are only interested in detecting the underlying *existence* of the containment and complementarity relationships, and we do not quantify it like in the computation of the OCM matrix. In the case of *partial containment*, the query for detecting pairs of observations is as follows:

```
SELECT DISTINCT ?o1, ?o2
```

```
WHERE {
  ?o1 a qb:Observation .
  ?o2 a qb:Observation .
  ?o1 ?d1 ?v1.
  ?o2 ?d1 ?v2.
  ?v1 skos:broaderTransitive/skos:
    broaderTransitive* ?v2
  FILTER(?o1 != ?o2)
}
```

The above query will select pairs of *?o1* and *?o2* that have at least one dimension with ancestral values; *?v1* must be a parent of *?v2*. The above query does not provide the number of dimensions that participate in the *partial containment*; this would make the query more complicated. In the case of *complementarity*, we tested the data against the following SPARQL query:

```
SELECT ?o1, ?o2
WHERE {
  ?o1 a qb:Observation .
  ?o2 a qb:Observation .
  FILTER NOT EXISTS {
    ?o1 ?d1 ?v1.
    ?o2 ?d1 ?v2.
    FILTER(?v1!=?v2)
  }
  FILTER(?o1 != ?o2)
}
```

The query will be matched against pairs of observations whose shared dimensions do not have different values. In both queries, we have relaxed the conditions presented in section 2 regarding the observations' schema.

**Rule-based.** The rule-based approach consists of three forward-chaining rules implemented in Jena, as the Jena generic rule reasoner is simple to use and offers the required expressivity. The rule for computing *full containment* is as follows:

```
observation(o1) ^ observation(o2)
^ (o1 ≠ o2)
^ ∃p.(has_dimension_value(o1,p,v1)
  ^ has_dimension_value(o2,p,v2)
  ^ is_ancestor(v1,v2))
^ ∀p.(has_dimension_value(o1,p,v1)
  ^ has_dimension_value(o2,p,v2)
  ^ is_ancestor(v1,v2))
⇒ full_containment(o1,o2)
```

In essence, we are checking for pairs of different observations that exhibit both existential and universal quantification in having dimension values subsume each other. The existential quantification is needed to ensure that there exists at least one such relationship, while the universal is needed to ensure that all relationships hold true. Similarly, the rule for *partial containment* checks the existential restriction; that

is, we need at least one pair of dimension values to exhibit a containment relationship between  $o_1$  and  $o_2$ . Therefore, the rule is as follows:

```

observation(o1) ∧ observation(o2)
∧ (o1 ≠ o2)
∧ ∃p.(has_dimension_value(o1,p,v)
      ∧ has_dimension_value(o2,p,v))
⇒ partial_containment(o1,o2)

```

The rule for *complementarity* is activated when two different observations have the same values for all of their shared dimensions and is summarized in the following:

```

observation(o1) ∧ observation(o2)
∧ (o1 ≠ o2)
∧ ∃p.(has_dimension_value(o1,p,v)
      ∧ has_dimension_value(o2,p,v))
∧ ∀p.(has_dimension_value(o1,p,v)
      ∧ has_dimension_value(o2,p,v))
⇒ complement(o1,o2)

```

All datasets along with the experiment code are available online at <http://github.com/mmeimaris>.

## 4.1 Experimental Results

Our experiments indicate that the quadratic *baseline* algorithm is improved substantially by our two proposed alternative methods, i.e. *clustering* and *cubeMasking*. Specifically, we have achieved improvement by roughly one order of magnitude for the computation of *full containment* and *complementarity* by using the *cubeMasking* algorithm, while we find that the *clustering* algorithm exhibits a trade-off between execution time and relationship recall. The results can be seen in Figure 5.

**Baseline.** The *baseline* behaves as expected, performing  $n^2$  comparisons for  $n$  observations. The partial containment relationships are quantified with a value between 0 and 1, non-inclusive, which increases monotonically in proportion with the number of dimensions that a pair of observations exhibits containment in. The results for computing the relationships can be seen in Figure 5(a-c). However, as can be seen in the Figure, it is cheaper to compute only full containment and complementarity because the cases where these do not apply can quickly be ruled out in the *computeOCM* step. This approach does not scale with respect to input size, as all pairs of rows in the *OCM* matrix, representing observations, have to be visited in order to detect and quantify containment and complementarity. It should be noted that, when having computed full containment, complementarity can be detected a posteriori by iterating through the fully contained pairs and checking for mutual (i.e. bi-directional) full containment.

**Clustering.** In the case of *clustering*, we have experimented with three different clustering algorithms, one agglomerative and two centroid-based, namely *hierarchical*, *canopy* and *x-means* respectively. Figure 5(d) shows the recall levels of the three algorithms. We have found that *x-means*, even when applied to a random 10% sample of the data, outperforms the other two in the resulting recall. Overall, the clustering method shows promising results and leaves room for improvement as far as the clustering step is concerned. Note that by definition, the *baseline* algorithm is equivalent to *clustering* with exactly one cluster. A large number of clusters  $k$  will limit the total number of comparisons, and consequently make the computations faster, but

the process will be lossy, ultimately achieving lower recall levels.

**Cube Masking.** The *cubeMasking* algorithm is the fastest of all the approaches we experimented with, mainly because of the linear cost of identifying and assigning observations to cubes, and the reduced number of performed comparisons. This is a consequence of the fact that it takes advantage of the distributions of dimension levels and values, and reduces the needed comparisons to a minimum with respect to the baseline and clustering algorithms, while maintaining full recall. Furthermore, the experiments on synthetic data, as well as the observation that the ratio of cubes per input size tends to decrease as input size increases as depicted in Figure 5(f), show that it is scalable for big datasets. In essence, Figure 5(f) shows that the number of cubes in a collection of datasets will increase in a lower rate than the number of input observations. This implies that as the input size increases, the *cubeMasking* algorithm will usually not result in intractable pair-wise comparisons.

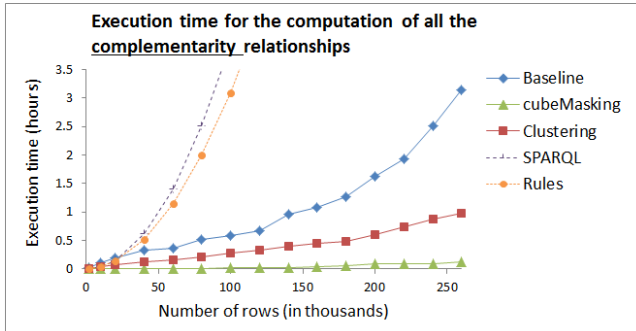
With *cubeMasking*, we need to check for parent-child relationships between cubes. The brute-force way to achieve this is to iterate through all pairs of cubes and check for pair-wise ancestry. This ancestry exists when all dimension levels of the first cube are equal or less than their respective dimension levels of the second cube. By pre-fetching and storing all children of each cube in memory, we introduce a conditional optimization that yields roughly 15-20% faster execution time for any input size as can be seen in Figure 5(g). However, creating this mapping implies either an explicit iteration of all cubes, which is costly, or an unavoidable iteration for one of the relationship types, which can be taken advantage of for the other two. Furthermore, keeping a list with the children of any given cube node adds an extra memory overhead, however in this work we are not interested in the memory footprint of each method.

**SPARQL and Rule-based.** These approaches perform adequately for small inputs, as shown in Figure 5(a), (b) and (c), but either hit the time-out limits or consume all memory resources quickly, which renders them unusable for the computations of such relationships over real world datasets. This is mainly due to the fact that the multi-transitive nature of the containment relationships creates an intractable search space, which is compensated by the dedicated specificities of our approach. From a reasoning-based perspective, it has been argued in the literature that dedicated reasoners tend to out-perform general purpose ones [8]; a fact that is supported by our experiments.

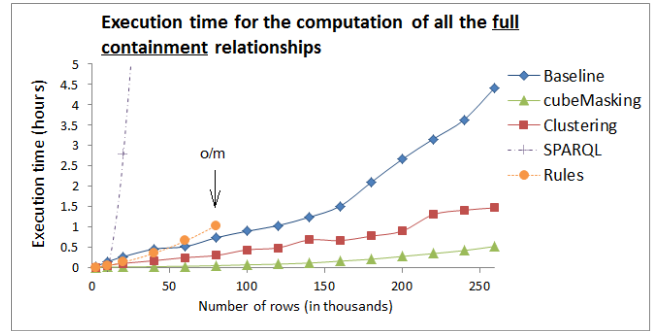
## 4.2 Scalability

In order to test scalability of our methods, we have created a synthetic dataset of 2.5M observations, following a similar approach as in [11]; the creation of synthetic data was based on fixing the number of dimensions and creating observations that follow a projected distribution of the data w.r.t to the real-world datasets. More specifically, we calculated the projected number of lattice nodes to reflect Figure 4(f), where the change in active lattice nodes is shown as the input size increases. Then, we populated the lattice nodes evenly.

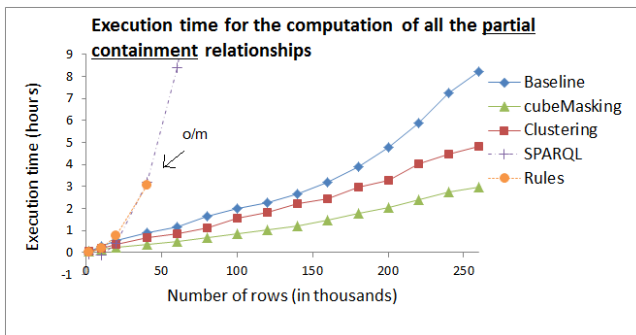
Experiments in the datasets show that the approaches utilizing SPARQL querying and rule inferencing do not scale adequately, even for small input sizes. Figure 5(e) shows how the three proposed methods scale in respect to the in-



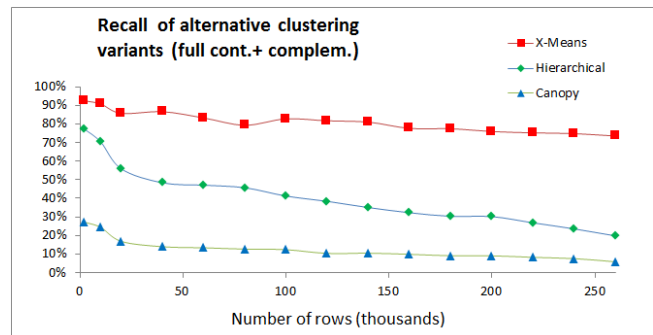
(a) Execution time (hours) for complementarity



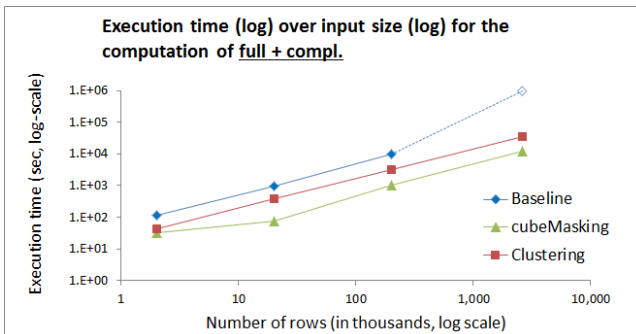
(b) Execution time (hours) for full containment



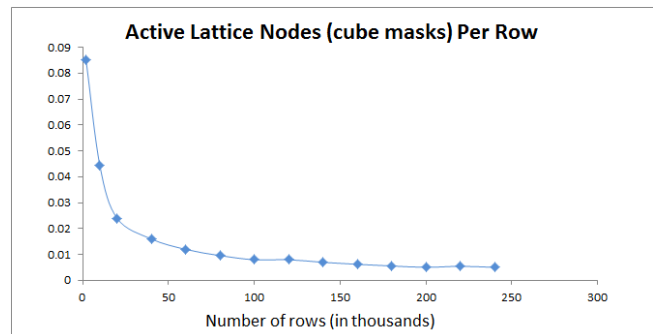
(c) Execution time (hours) for partial containment



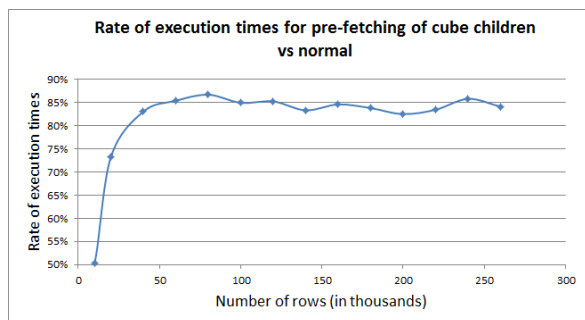
(d) Recall for three clustering algorithms w.r.t. input size



(e) log-log time and input size



(f) ratio of discovered cubes per observation count



(g) Rate of execution time with children pre-fetching vs normal for full containment

Figure 5: (a) Execution time (hours) for complementarity, (b) full containment, (c) partial containment (note that in the SPARQL based approach, partial containment is only detected and not quantified), (d) recall for three clustering algorithms w.r.t. input size, (e) log-log time and input size, (f) ratio of discovered cubes per observation count. o/m=out of memory, (g) rate of execution time with children pre-fetching vs normal for full containment.

put size. While the values for *clustering* and *cubeMasking* are based on measured values on the synthetic data, the value for the *baseline* method for 2.5m records is a projection of the quadratic analysis; it took more than 7 days to complete.

The results indicate that both *clustering* and *cubeMasking* scale better than the baseline, with the latter having a more clear advantage because of the reduced number of needed comparisons in conjunction with the fact that it is lossless. However, in extreme cases where the number of cubes is large and the distribution of observations in these cubes is sparse, the *cubeMasking* method will resemble the baseline. In these cases, the *clustering* method yields a more realistic advantage, especially when efficiency is more important than recall.

## 5. RELATED WORK

In this paper, we extend the work presented in [22], where we introduce preliminary notions of containment and complementarity, we outline a method of computing these notions, and define an RDF vocabulary as an extension of RDF QB for containment and complementarity between multidimensional observations across sources.

The problem of finding related observations in multidimensional spaces has been addressed within several contexts. Online Analytical Mining (OLAM) refers to the application of data mining in OLAP, and addresses OLAP classification [20], outlier detection [1], intelligent querying [28] and recommendation for OLAP exploration, which is based on either query formulation or instance similarity [2, 3]. These approaches enable discovery of latent information, exploratory analysis [13] and efficient querying [20].

In [14] the authors discuss how queries containing grouping and aggregation, which in our case is similar to observation containment, can be facilitated by materialized views. Partial containment is referred to as k-dominance in [6], where it is used to define partial skylines in multidimensional datasets.

Skyline computation in multidimensional datasets uses the notion of dominance, or observation containment, in order to assert whether a point is part of a dataset's skyline [33, 30, 19]. Skyline points are essentially top-level observations, i.e. observations that are not contained by other observations. However, these approaches are concerned with only skyline data points, rather than computation of all containment relationships. Skyline computation is, however, a direct derivative of containment computation.

Aligon et al. [2] address similarities between OLAP sessions by defining distance functions over query features. They experiment with Levenshtein, Dice's coefficient, tf-idf and Smith-Waterman, with the latter being the best for their purposes, whereas in our approach the Jaccard Coefficient addresses the binary nature of our feature space directly. Baikousi et al. [3] define distance functions for dimension hierarchies. However, they address observation similarity in general, rather than computing strict relationships as in our case. Hsu et al. [16] apply multidimensional scaling methods and hierarchical clustering in a hybrid approach in order to measure similarity between reports in the same cubes.

Business model ontologies have been deployed in the context of OLAP cubes in [29], where the authors define notions such as *merge* and *abstraction* of cubes. The *abstraction* notion is of particular interest to our work, as it resembles the

containment relationship, however the setting and motivation behind this work is representational rather than computational.

In the context of RDF, finding related cubes in multidimensional contexts is addressed in [18], where the work is focused in extending the *Drill-Across* operator to address different sources. The authors define conversion and merging correspondences between remote cubes in order to quantify the degree of their overlap and enable meaningful combinations of datasets. However, they do not address specific relationships at the instance level. The need for RDF-specific workflows is addressed in [17], where the authors argue that analytical processing of RDF cubes requires more than the capabilities offered by SPARQL engines for querying, exploration and analysis, and are best complemented with OLAP-to-SPARQL engines that use RDF aggregate views and partial materialization. This is in favour of our approach that tackles efficient materialization of batch relationships.

The more general problem of finding similarity between resources is a main component in entity resolution, record linkage and interlinking [23, 24, 32]. These approaches deal with discovering links between nodes from different datasets by using distance-based techniques. To the best of our knowledge, this is the first work that addresses the definition, representation and computation of relationships between individual multidimensional observations.

## 6. CONCLUSIONS AND FUTURE STEPS

In this paper, we have presented and compared three novel approaches for discovering relationships between observations of multidimensional RDF data. We have defined new relationships, namely full and partial containment, and complementarity between observations as derivatives of the hierarchical relationships between their dimension values, and as a means of comparison and correlation of their measures. We performed an experimental evaluation and comparison between them and with a SPARQL-based and a rule-based technique and we show that our methods outperform the traditional approaches in both execution time and scalability. As this work is based on batch analysis, in the future we plan to study and define efficient incremental techniques, as well as hybrid probabilistic methods that take into advantage the positive points of the *clustering* and *cubeMasking* algorithms. We will also address space efficiency and examine the behaviour of our approaches on settings with memory restrictions. Finally, we intend to examine the performance of our algorithms in distributed and parallel contexts.

**Acknowledgements.** This work is supported by the EU-funded ICT project "DIACHRON" (agreement no 601043).

## 7. REFERENCES

- [1] C. C. Aggarwal and P. S. Yu. Outlier detection for high dimensional data. In *ACM Sigmod Record*, volume 30(2), pages 37–46. ACM, 2001.
- [2] J. Aligon, M. Golfarelli, P. Marcel, S. Rizzi, and E. Turricchia. Similarity measures for olap sessions. *Knowledge and information systems*, 39(2):463–489, 2014.
- [3] E. Baikousi, G. Rogkakos, and P. Vassiliadis. Similarity measures for multidimensional data. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 171–182. IEEE, 2011.

- [4] C. Böhm, F. Naumann, M. Freitag, S. George, N. Höfler, M. Köppelmann, C. Lehmann, A. Mascher, and T. Schmidt. Linking open government data: what journalists wish they had known. In *Proceedings of the 6th International Conference on Semantic Systems*, page 34. ACM, 2010.
- [5] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 74–83. ACM, 2004.
- [6] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. K. Tung, and Z. Zhang. Finding k-dominant skylines in high dimensional space. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 503–514. ACM, 2006.
- [7] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *ACM Sigmod record*, 26(1):65–74, 1997.
- [8] S. Coppens, M. Vander Sande, R. Verborgh, E. Mannens, and R. Van de Walle. Reasoning over sparql. In *LDOW*. Citeseer, 2013.
- [9] R. Cyganiak, D. Reynolds, and J. Tennison. The rdf data cube vocabulary. *W3C Recommendation (January 2014)*, 2013.
- [10] A. Das Sarma, L. Fang, N. Gupta, A. Halevy, H. Lee, F. Wu, R. Xin, and C. Yu. Finding related tables. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 817–828. ACM, 2012.
- [11] B. Ding, M. Winslett, J. Han, and Z. Li. Differentially private data cubes: optimizing noise sources and consistency. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 217–228. ACM, 2011.
- [12] F. M. Donini. Complexity of reasoning. In *The description logic handbook*, pages 96–136. Cambridge University Press, 2003.
- [13] A. Giacometti, P. Marcel, E. Negre, and A. Soulet. Query recommendations for olap discovery driven analysis. In *Proceedings of the ACM twelfth international workshop on Data warehousing and OLAP*, pages 81–88. ACM, 2009.
- [14] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.
- [15] I. Horrocks, P. F. Patel-Schneider, S. Bechhofer, and D. Tsarkov. Owl rules: A proposal and prototype implementation. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(1):23–40, 2005.
- [16] K. C. Hsu and M.-Z. Li. Techniques for finding similarity knowledge in olap reports. *Expert Systems with Applications*, 38(4):3743–3756, 2011.
- [17] B. Kämpgen and A. Harth. No size fits all—running the star schema benchmark with sparql and rdf aggregate views. In *The Semantic Web: Semantics and Big Data*, pages 290–304. Springer, 2013.
- [18] B. Kämpgen, S. Stadtmüller, and A. Harth. Querying the global cube: Integration of multidimensional datasets from the web. In *Knowledge Engineering and Knowledge Management*, pages 250–265. Springer, 2014.
- [19] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 275–286. VLDB Endowment, 2002.
- [20] V. Markl, F. Ramsak, and R. Bayer. Improving olap performance by multidimensional hierarchical clustering. In *Database Engineering and Applications, 1999. IDEAS’99. International Symposium Proceedings*, pages 165–177. IEEE, 1999.
- [21] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–178. ACM, 2000.
- [22] M. Meimaris and G. Papastefanatos. Containment and complementarity relationships in multidimensional linked open data. In *Second International Workshop for Semantic Statistics SemStats*, 2014.
- [23] P. N. Mendes, M. Jakob, A. García-Silva, and C. Bizer. Dbpedia spotlight: shedding light on the web of documents. In *Proceedings of the 7th International Conference on Semantic Systems*, pages 1–8. ACM, 2011.
- [24] A.-C. N. Ngomo and S. Auer. Limes—a time-efficient approach for large-scale link discovery on the web of data. *integration*, 15:3, 2011.
- [25] S. M. Nutley, H. T. Davies, and P. C. Smith. *What works?: Evidence-based policy and practice in public services*. MIT Press, 2000.
- [26] D. Pelleg, A. W. Moore, et al. X-means: Extending k-means with efficient estimation of the number of clusters. In *ICML*, pages 727–734, 2000.
- [27] G. Piatetsky-Shapiro, R. J. Brachman, T. Khabaza, W. Kloesgen, and E. Simoudis. An overview of issues in developing industrial data mining and knowledge discovery applications. In *KDD*, volume 96, pages 89–95, 1996.
- [28] G. Sathe and S. Sarawagi. Intelligent rollups in multidimensional olap data. In *VLDB*, volume 1, pages 531–540, 2001.
- [29] C. Schütz, B. Neumayr, and M. Schrefl. Business model ontologies in olap cubes. In *Advanced Information Systems Engineering*, pages 514–529. Springer, 2013.
- [30] K.-L. Tan, P.-K. Eng, B. C. Ooi, et al. Efficient progressive skyline computation. In *VLDB*, volume 1, pages 301–310, 2001.
- [31] B. Villazón-Terrazas, L. M. Vilches-Blázquez, O. Corcho, and A. Gómez-Pérez. Methodological guidelines for publishing government linked data. In *Linking government data*, pages 27–49. Springer, 2011.
- [32] J. Volz, C. Bizer, M. Gaedke, and G. Kobilarov. Silk—a link discovery framework for the web of data. *LDOW*, 538, 2009.
- [33] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *Proceedings of the 31st international conference on Very large data bases*, pages 241–252. VLDB Endowment, 2005.



# Semi-automatic support for evolving functional dependencies

Mirjana Mazuran  
DEIB – Politecnico di Milano  
mirjana.mazuran@polimi.it

Letizia Tanca  
DEIB – Politecnico di Milano  
letizia.tanca@polimi.it

Elisa Quintarelli  
DEIB – Politecnico di Milano  
elisa.quintarelli@polimi.it

Stefania Ugolini  
Department of Mathematics –  
Milan University  
stefania.ugolini@unimi.it

## ABSTRACT

During the life of a database, systematic and frequent violations of a given constraint may suggest that the represented reality is changing and thus the constraint should evolve with it. In this paper we propose a method and a tool to (i) find the functional dependencies that are violated by the current data, and (ii) support their evolution when it is necessary to update them. The method relies on the use of *confidence*, as a measure that is associated with each dependency and allows us to understand "how far" the dependency is from correctly describing the current data; and of *goodness*, as a measure of *balance* between the data satisfying the antecedent of the dependency and those satisfying its consequent. Our method compares favorably with literature that approaches the same problem in a different way, and performs effectively and efficiently as shown by our tests on both real and synthetic databases.

## 1. INTRODUCTION

The information related to a certain reality of interest is represented in a database by means of data, a vital resource on which decision-making processes are based. The data stored in a database must satisfy the semantic conditions expressed by the schema plus integrity constraints. Specifying and enforcing constraints grants us better data quality, maintenance, query optimization, view updating and database integration and exchange; in particular, *functional dependencies* have been widely applied to these aims and, from the 70s on, their knowledge has been used to support database design and management, reverse engineering and query optimization. The number of application scenarios they are used in has grown over time including, more recently, automated DB analysis such as knowledge discovery and data mining. A functional dependency (FD) is a constraint between two sets X and Y of attributes in a relation

R of a database: R is said to satisfy the functional dependency  $X \rightarrow Y$  if each X value is associated with precisely one Y value.

Nowadays, huge amounts of data are generated daily and may come from different applications and sources where constraints are not equally enforced, thus the original datasets may be inconsistent with each other or even by themselves [1, 2]. Once a database designer or administrator is able to understand that a constraint no longer holds, he/she can decide what to do. Most systems that deal with discrepancies of this kind re-establish consistency by changing the data that violate the constraints. In our work we allow for a different interpretation. Indeed, from our point of view, changes in the data might also mean that their semantics is evolving for some reason, like for instance law or policy changes. Therefore, once the DB administrator has ascertained that this is the case, s/he will be able to re-establish consistency by appropriately modifying the violated constraints. Note that the new constraints capture a succinct representation of the new semantics of the data, thus, this kind of analysis is interesting for knowledge discovery purposes.

This paper's main goal is to propose a method to modify functional dependencies so as to adjust them to the evolutions of the modeled reality that may occur during database life. The method we propose provides a way to understand which FDs are violated and, if needed, to modify them by adding, to their antecedent, a minimal set of attributes that makes them consistent with the data.

**Running example** Consider the relation *Places* in Figure 1 and assume that the following FDs be defined on it:

$$F_1 : [District, Region] \rightarrow [AreaCode]$$

$$F_2 : [Zip] \rightarrow [City, State]$$

$$F_3 : [PhNo, Zip] \rightarrow [Street]$$

All the tuples in *Places* violate  $F_1$ ; tuples  $t_1, t_2$  and  $t_3$  violate  $F_2$  and tuples  $t_{10}$  and  $t_{11}$  violate  $F_3$ . Thus, these three FDs are not satisfied by the data. If the DBMS is able to detect that (e.g. by means of periodic or continuous checks of FDs validity) then it can present it to the designer. Suppose the designer realizes that an FD not being satisfied by the data is not a mistake but a symptom of a real-world situation which is no more reflected by the semantics of the FDs, at this point s/he can decide to integrate some of the candidate

tid	District	Region	Municipal	AreaCode	PhNo	Street	Zip	City	State
t <sub>1</sub>	Brookside	Granville	Glendale	613	974-2345	Boxwood	10211	NY	NY
t <sub>2</sub>	Brookside	Granville	Glendale	613	974-2345	Boxwood	10211	NY	NY
t <sub>3</sub>	Brookside	Granville	Glendale	613	299-1010	Westlane	10211	NY	MA
t <sub>4</sub>	Brookside	Granville	Guildwood	515	220-1200	Squire	02215	Boston	MA
t <sub>5</sub>	Brookside	Granville	Guildwood	515	220-1200	Squire	02215	Boston	MA
t <sub>6</sub>	Alexandria	Moore Park	NapaHill	415	220-1200	Napa	60415	Chicago	IL
t <sub>7</sub>	Alexandria	Moore Park	NapaHill	415	930-2525	Main	60415	Chicago	IL
t <sub>8</sub>	Alexandria	Moore Park	NapaHill	415	555-1234	Tower	60415	Chester	IL
t <sub>9</sub>	Alexandria	Moore Park	QueenAnne	517	888-5152	Main	60415	Chicago	IL
t <sub>10</sub>	Alexandria	Moore Park	QueenAnne	517	888-5152	Main	60601	Chicago	IL
t <sub>11</sub>	Alexandria	Moore Park	QueenAnne	517	888-5152	Bay	60601	Chicago	IL

Figure 1: Running example: relation *Places*

changes into the database. Thus, the idea of the paper is to find a way to change the constraints instead of the data and make them valid again. How can we repair an FD? First of all, without loss of generality we can assume that all FDs are decomposed so that their consequent contains a single attribute. In this way it is easy to see that modifying the consequent is a no-issue. What we can do instead is acting upon the antecedent: of course deleting attributes from it cannot repair the FD but adding attributes might. Thus, our aim is to identify the FDs violated by the data, find possible repairs and present them to the designer to be evaluated. The method can be used periodically on the data in order to keep them consistent with the constraints.

The paper is organized as follows: in Section 2 we review the state of the art, Section 3 introduces the technical notions and the considerations at the basis of our proposal, while Section 4 presents our proposal to evolve violated FDs. In Section 5 we formally compare our approach with a proposal which has the same aim as ours, but is based on the notion of entropy to measure how much an FD is far from being exact, while for the same aim we use the (simpler) notion of *confidence*. In Section 6 we explain the experimental results we have obtained on both real and synthetic datasets and finally in Section 7 we draw the conclusions of our work.

## 2. RELATED WORK

In the last years, research involving FDs has taken a different turn with respect to the past; today, automated data analysis has become fundamental for knowledge discovery and data mining and FDs in particular provide invaluable intensional knowledge on the relation instances. As a consequence, the concept has been used in a wide range of application scenarios, which in turn originated many extensions and variants [3], including: Conditional FDs (CFDs) [4], Approximate FDs (AFDs) [5], Approximate Conditional FDs (ACFDs) [6], Temporal FDs (TFDs) [7], Approximate Temporal FDs (ATFDs) [8], and others.

CFDs [4], have been introduced that specify FDs that do not hold on the whole relation but just on a subset of its data.

Both FDs and CFDs are exact, i.e., they hold for all the instances in the relation (in the case of FDs) or for certain subsets of it (for CFDs). These rules may easily break in the

sense that even small errors or minor changes to the relation instance may cause that the constraint no longer holds. However, as time passes by and the lifecycle of a database goes on, reality may change and so should the constraints defined on the data. It is thus appropriate to monitor data evolution as a mirror of reality in order to get useful insights about the way data are evolving in time. To this aim, allowing some exceptions in the table makes it possible to obtain a better understanding of the data. In fact, a few rows might contain errors due to various noise factors or simply be an exception to the rule, and being able to detect the presence of unexpected exceptions may inform us that something has changed in the data semantics. To deal with this scenario, AFDs have been introduced, that is, FDs that are associated with some degree of approximation. AFDs are FDs that allow some rows to contain “errors” as exceptions to the rules; the more errors they allow, the more approximate they are, that is, their “approximation degree” changes. Thus, AFDs “bend but do not break” and coherently ACFDs have been introduced as an extension of CFDs.

The problem of constraint violation has been faced in the literature in different ways. Works such as [9, 10, 11, 12, 13, 14] propose strategies to query inconsistent databases trying to re-establish consistency by changing the facts that violate the constraints. Thus, the problem of inconsistent databases is considered from a query-answering point of view, that is, the data that can produce inconsistency with respect to the integrity constraints imposed at design time are discarded. By contrast, we aim at modifying the integrity constraints so the semantics of the database will adhere as much as possible to the changing reality. Therefore, the data that violate the constraints are not considered as abnormal facts but are used to update obsolete constraints. A similar approach was developed in [15] whose authors used data mining techniques to repair tuple constraints. In this work we extend their methodology to deal with functional dependencies.

FDs have been used as means to enforce data quality through data profiling and cleaning. In [16] the authors propose an algorithm for *discovering* Denial Constraints (which include functional dependencies) without supposing that any constraint has been specified on the database at design time. In order to apply the proposal in [16] to update the constraints on a given database when they are not up to date, one has: (i) first to discover all the possible constraints from data, then (ii) relax the constraints, considered as if they

have been specified at design time on the database schema, that do not hold on the current instance. This approach is rather impractical when the FDs, though obsolete, have been originally defined by a designer, first because of efficiency reasons, and second because, as we have noticed testing the on-line algorithm of [16] the inferred constraints not always include extensions of the ones specified by the designer.

Not many works have been proposed in the literature that, in the case of inconsistency, try to change the constraints instead of the data. The authors of [17, 18] have for the first time introduced a model that considers functional dependency repair. They illustrate a method to extend, by adding one attribute, the body of a violated FD in order to obtain a new dependency that is not violated anymore. In particular, given a functional dependency  $F : X \rightarrow Y$  over a relation  $R$ , each attribute of  $R$  (other than  $X$  and  $Y$ ) is evaluated as a candidate using the notion of *entropy* for comparing clusterings of tuples. As a result, a ranked list of candidate attributes is given to the designer. Given an FD that is violated by the data, the strategy in [17]: i) first computes a ground truth clustering of the data, based on the attributes in the FD; ii) then, for each attribute  $A$ , not present in the FD, computes a clustering of the data based on  $A$  and finally iii) computes the relative entropy (see Section 3) between this clustering and the ground truth clustering, which means comparing all its clusters with those in the ground truth clustering. The metric used in the work, which is the information variation requires frequent clustering of tuples in order to understand how good a candidate attribute is. We propose a very simple approach based on confidence and a measure of goodness that only require to count tuples. In Section 5 we explain the details of [17] and give a theoretical comparison between this work and ours. An experimental comparison between the two approaches was unfortunately impossible due to the unavailability of the tool presented in [17].

### 3. BASIC NOTIONS

Let  $\mathcal{U}$  be a finite set of attribute names; we denote attributes by capital letters from the beginning of the alphabet (e.g.,  $A, B, C, A_1$ , etc.), while capital letters from the end of the alphabet (e.g.,  $U, X, Y, Z, X_1$ , etc.) are used to denote sets of attributes. Let  $\mathcal{D}$  be a finite set of domains, each containing atomic values for the attributes; the domain  $D_j$  contains the possible values for the attribute  $A_j$ . A *relation schema*  $R(A_1, A_2, \dots, A_n)$  describes the structure of a relation whose name is  $R$  and whose set of attributes is  $A_1, A_2, \dots, A_n$ . A relation instance  $r$  of relation  $R$ , is a finite set of tuples  $t_1, t_2, \dots, t_m$  of the form  $t_h = (v_1, v_2, \dots, v_n)$ , where each value  $v_k$ ,  $1 \leq k \leq n$ , is an element of  $D_k$ .  $t[A_i]$  denotes the value assumed by the attribute  $A_i$  in the tuple  $t$  (i.e.,  $v_i$ ). Given an instance  $r$  of a relation  $R$ , we denote by  $|r|$  the number of tuples in  $r$  and  $|R|$  the number of attributes in  $R$ . Moreover,  $\pi_X(r)$  is the projection of  $r$  on the attributes of  $X$ . The structure of a functional dependency is defined as follows:

**Definition 1 (Syntax)** *Given a relation schema  $R$ , a functional dependency  $F$  over  $R$  has the form  $F : X \rightarrow Y$  where  $X$  and  $Y$  are sets of attributes in  $R$ .*

We use  $XY$  to denote the union of  $X$  and  $Y$ , moreover  $|F| = |XY|$  is the number of attributes in the FD and, given

two FDs  $F_1$  and  $F_2$ ,  $|F_1 \cap F_2|$  is the number of attributes common to  $F_1$  and  $F_2$ . An instance  $r$  of a relation schema  $R$  can either satisfy an FD or not, according to Definition 2.

**Definition 2 (Semantics)** *Given an instance  $r$  of a relation  $R$ ,  $r$  satisfies  $F$  if, for every pair of tuples  $t_1, t_2$  in  $r$ , if  $t_1[X] = t_2[X]$  then  $t_1[Y] = t_2[Y]$ .*

We say that an instance  $r$  is inconsistent with respect to  $F$  if it does not satisfy Definition 2.

Let us introduce a useful characterization of the notion of FD with the introduction of *confidence* and *goodness*.

**Definition 3** *Let  $r$  be an instance of a relation  $R$  defined over a set of attributes  $S$ . Let  $X$  and  $Y$  be subsets of  $S$  and  $F : X \rightarrow Y$  a functional dependency over  $R$ . The confidence of  $F$  w.r.t.  $r$  is:*

$$c_{F,r} = \frac{|\pi_X(r)|}{|\pi_{XY}(r)|}$$

while the goodness of  $F$  w.r.t.  $r$  is:

$$g_{F,r} = |\pi_X(r)| - |\pi_Y(r)|$$

Moreover, based on the value of the confidence of an FD, we have the following definition:

**Definition 4** *Given a relation  $R$ , an instance  $r$  of  $R$ , a functional dependency  $F$  over  $R$  and its confidence  $c_{F,r}$ , we say that  $F$  is an exact functional dependency iff  $c_{F,r} = 1$ , otherwise it is an approximate functional dependency.*

All three FDs from Example 1 are approximate and have the following confidence and goodness values:  $c_{F_1,Places} = 0.5$  and  $g_{F_1,Places} = -2$ ;  $c_{F_2,Places} = 0.667$  and  $g_{F_2,Places} = -1$ ;  $c_{F_3,Places} = 0.889$  and  $g_{F_3,Places} = 1$ . Note that the confidence of an FD reflects the average number of values of  $Y$  that are associated with each value of  $X$  in  $r$ . When the confidence is 1 it means that, for each distinct value of  $X$ , exactly one value of  $Y$  is associated with  $X$ , thus it is easy to see that exact FDs are the classical FDs of Definition 1:

The notion of functional dependency can be also formalized as a function between clusters of tuples.

**Definition 5 (Clustering)** *Given an instance  $r$  of a relation  $R$  and a set  $X$  of attributes of  $R$ , we call  $X$ -clustering a partition  $\mathcal{C}_X = \{C_1, C_2, \dots, C_K\}$  of  $r$  into mutually disjoint subsets  $C_i$ , with  $i \in \{1, \dots, K\}$ , called *classes (or clusters)*, such that each class  $C_i$  contains all the tuples of  $r$  that have the same value for the attributes in  $X$ .*

Given  $F : X \rightarrow Y$ , there are two clusterings naturally generated by  $F$ :  $\mathcal{C}_X$  and  $\mathcal{C}_Y$ . Intuitively, if each cluster in  $\mathcal{C}_X$  is associated with only one cluster in  $\mathcal{C}_Y$  (that is, if there is a function between classes in  $\mathcal{C}_X$  and classes in  $\mathcal{C}_Y$ ) then  $F$  is satisfied, otherwise it is not. Consider as an example

$$F_1 : [District, Region] \rightarrow [AreaCode]$$

The two clusterings  $\mathcal{C}_{District,Region}$  and  $\mathcal{C}_{AreaCode}$  are shown in Figure 2a. As we can see, the relation between the two clusterings is not a function because there are some tuples, having the same value of *District, Region* that are associated with sets having different values of *AreaCode*. In fact,  $F_1$  is violated by the data.

To understand whether  $F$  is satisfied or not, we consider the two clusterings  $\mathcal{C}_X$  and  $\mathcal{C}_{XY}$ . Since  $\mathcal{C}_{XY}$  is finer-grained than  $\mathcal{C}_X$  we always have:  $|\mathcal{C}_{XY}| \geq |\mathcal{C}_X|$ . When  $|\mathcal{C}_{XY}| > |\mathcal{C}_X|$ ,

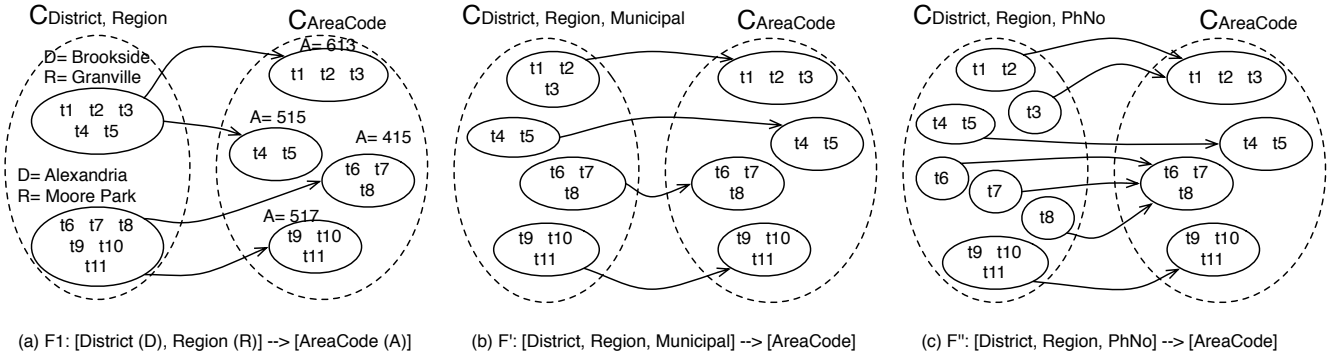


Figure 2: FDs clusterings

it means that there exists at least one class  $C_x$  in  $\mathcal{C}_X$  whose tuples form more than one class in  $\mathcal{C}_{XY}$ , thus, the relation between  $\mathcal{C}_X$  and  $\mathcal{C}_Y$  is not a function. Only when  $|\mathcal{C}_{XY}| = |\mathcal{C}_X|$ , we have that each class in  $\mathcal{C}_X$  forms one class in  $\mathcal{C}_Y$  and therefore  $F$  is satisfied. If there is a function between  $\mathcal{C}_X$  and  $\mathcal{C}_Y$ , such function is surjective, because each tuple in the database contains values for both  $X$  and  $Y$ <sup>1</sup>, thus each value of  $Y$  is necessarily associated with at least one value of  $X$ . Now, the function could be injective (and thus bijective) or not. In particular, we introduce the following definition:

**Definition 6 (Proper association)** *Given two clusterings  $\mathcal{C}_X$  and  $\mathcal{C}_Y$ , we say that a class  $C_x$  in  $\mathcal{C}_X$  is properly associated with a class  $C_y$  in  $\mathcal{C}_Y$ , when  $C_y$  is the unique class in  $\mathcal{C}_Y$  such that  $C_x \subseteq C_y$ .*

The concept of proper association is crucial for defining an FD in terms of a function between two given clusterings. Given  $F : X \rightarrow Y$ , when for every class  $C_x$  in  $\mathcal{C}_X$  there is a class  $C_y$  in  $\mathcal{C}_Y$  which is properly associated with  $C_x$  (i.e.  $C_y$  is the unique class that contains  $C_x$ ), one usually says that the clustering  $\mathcal{C}_X$  is homogeneous with respect to  $\mathcal{C}_Y$ . Accordingly, we say that there is a well-defined function between the classes in  $\mathcal{C}_X$  and those in  $\mathcal{C}_Y$ . Intuitively, a well-defined function is bijective.

From our point of view, FDs that allow us to obtain a well-defined function are to be preferred to other FDs. Consider  $F' : [District, Region, Municipal] \rightarrow [AreaCode]$  and  $F'' : [District, Region, PhNo] \rightarrow [AreaCode]$ ; the clusterings  $\mathcal{C}_X$  and  $\mathcal{C}_Y$  generated by the two FDs are shown in Figure 2b and Figure 2c respectively. As we can see, in both cases there is a function between  $\mathcal{C}_X$  and  $\mathcal{C}_Y$ , however,  $F'$  allows us to obtain a well-defined function while  $F''$  does not. Intuitively speaking, the municipality is “a better choice” than the phone number.

To explain this intuition, consider again that a non-satisfied  $F : X \rightarrow Y$  generates two clusterings  $\mathcal{C}_X$  and  $\mathcal{C}_Y$  where the number of clusters in  $\mathcal{C}_X$  is smaller than the number of clusters in  $\mathcal{C}_Y$ . To obtain a function between the two clusterings, we need to “further fragment”  $\mathcal{C}_X$  so that there is the same number of clusters as in  $\mathcal{C}_Y$ , or more: in our method we do this by adding attributes to  $X$ .

Note that the number of clusters in  $\mathcal{C}_X$  gives us an idea of how “specific” is a set of attributes  $X$ , that is, if the number

<sup>1</sup>Note that attributes involved in FDs do not contain NULL values

of clusters is 1 all the tuples in the relation are put in the same cluster, while, in the extreme case where each tuple is assigned to a distinct cluster,  $X$  is actually a candidate key.

This second case will surely happen if, while repairing  $F$ , we add to its antecedent an attribute that has the property of being UNIQUE in the relation. Adding a UNIQUE attribute allows to repair any FD because this attribute alone determines  $Y$  thus it practically makes  $X$  “useless” in the FD. Something very close to this case happens with the two dependencies  $F'$  and  $F''$  above:  $F'$  is “better” than  $F''$  because adding the municipality to the antecedent allows to generate two clusterings that are associated by a function such that the specificity of the domain is as much as possible similar to the specificity of the codomain, while, due to the high specificity of the phone number, adding it would make the antecedent too specific (also making the other attributes “almost useless”). We will see that our repairing method discourages the addition of such attributes (they are penalized through the goodness coefficient) and instead prefers those attributes that fragment  $\mathcal{C}_X$  just as much as it is necessary to reach the same specificity as  $\mathcal{C}_Y$ .

Notice the relationship between these concepts and the confidence and goodness of an FD  $F$ :

- $F$  is exact when there is a well-defined function from  $\mathcal{C}_X$  to  $\mathcal{C}_Y$ , that is, the confidence coefficient of an FD somehow measures the “degree of being a function” ( $c_{F,r} \leq 1$ ).
- When this pleasant fact happens ( $c_{F,r} = 1$ ), then the goodness coefficient measures how far our function is from being injective. In fact it is injective when  $g_{F,r} = 0$ . In any case, when  $c_{F,r}$  is different from 1, the goodness coefficient measures the distance of our approximate FD from having the domain with the same cardinality of the co-domain.

Therefore the case  $\{c_{F,r} = 1, g_{F,r} = 0\}$  is such that to each class in  $\mathcal{C}_X$  is associated one and only one class in  $\mathcal{C}_Y$ : the corresponding FD allows to obtain a bijective function between the two clusterings and so it is the best function which cannot be further improved. In fact, from a well-known result on functions<sup>2</sup>, since the correspondence  $F$  is surjective and  $|\pi_X(r)| = |\pi_Y(r)|$ , it is therefore bijective. Finally we

<sup>2</sup>Let  $f : X \rightarrow Y$  be a function. If  $|X| = |Y|$ , then  $f$  is injective if and only if  $f$  is surjective.

notice that the *goodness coefficient* can be positive or negative. It is positive when the domain cardinality is higher than the co-domain one and negative instead when the cardinality of the domain is smaller.

As a final remark, notice that, if the DB schema is in a higher normal form, the only non-trivial FDs are those determining candidate keys. However, we believe this to be a strong assumption, especially nowadays, since NoSQL, semi-structured and other types of poorly organized data are widely used. Thus, we do not rely on the assumption that a database is in a higher normal form and obtaining some insight about the data becomes very important in such a scenario.

## 4. EVOLVING FUNCTIONAL DEPENDENCIES

The goal of our method is to: (i) understand which FDs are violated and (ii) repair these FDs by adding attributes to the antecedent of the dependency.

**Objective** *Given an FD  $F : X \rightarrow Y$ , not satisfied by the data, our aim is to find a minimal set of attributes  $U$  such that the new FD  $F^U : XU \rightarrow Y$  is satisfied by the data.*

Given a relation schema  $R$ , an instance  $r$  of  $R$ , and all the FDs defined on it, for each FD,  $F : X \rightarrow Y$ , we compute its confidence. If the result is lower than 1 then the FD is not satisfied and, to repair it, we look for a set of attributes  $U$  in  $R \setminus XY$  such that, if added to the antecedent of  $F$ , generate a new dependency  $F^U : XU \rightarrow Y$  whose confidence is 1.

### 4.1 FD ordering

If an instance  $r$  of a relation  $R$  violates more than one constraint we need to decide the order in which they should be repaired. To do this, similarly to [17], for each FD  $F$  we compute a rank that is the average of two indicators:

1.  $ic_{F,r}$ : the “degree of inconsistency” of  $F$  with respect to the relation  $r$ :  $ic_{F,r} = 1 - c_{F,r}$
2.  $cf_F$ : the “conflict score” of  $F$  with respect to the other FDs defined on  $R$ , which is independent of any specific instance and depends on the number of attributes that  $F$  has in common with these FDs:

$$cf_F = \frac{\sum_{F' \in \mathcal{F}} \frac{|F \cap F'|}{\max(|F|, |F'|)}}{|\mathcal{F}|}$$

where  $\mathcal{F}$  is the set of FDs defined on  $R$ .

The final rank is the average of these two indicators:

$$O_F = \frac{ic_{F,r} + cf_F}{2}$$

Given a set  $\mathcal{F}$  of FDs, we sort  $\mathcal{F}$  according to the rank  $O_F$  of each  $F \in \mathcal{F}$  and then follow this order to repair the FDs. Consider the relation in Figure 1 and the three FDs  $F_1$ ,  $F_2$  and  $F_3$  in Example 1. The constraints should be examined in the following order:  $F_1$  (0.25),  $F_2$  (0.167),  $F_3$  (0.056).

### 4.2 Candidate-repair ordering for an FD

Given an instance  $r$  of a relation  $R$ , an FD  $F : X \rightarrow Y$  and an attribute  $A$  which is a candidate to extend the antecedent

of FD, we first compute the confidence of the candidate FD  $F^A : XA \rightarrow Y$  as:

$$c_{F^A,r} = \frac{|\pi_{XA}(r)|}{|\pi_{XAY}(r)|}$$

and then provide an ordered list of candidate attributes sorted in descending order of  $c_{F^A,r}$ . However, this ranking does not allow us to distinguish which attribute is better when the FDs they produce have the same confidence. Consider  $F_1 : [District, Region] \rightarrow [AreaCode]$  defined on table *Places*. Intuitively, if we add the attribute *Municipal* to the antecedent of  $F_1$  we obtain an exact FD and, similarly, if we instead add the attribute *PhNo* we also obtain an exact FD. Now, is it more reasonable to add the municipality or the phone number as attribute that is part of a functional dependency? Which one is better and why?

To provide a better ranking, following the intuition explained in Section 3 we use the goodness of  $F^A$ , which depends on the number of distinct values  $A$  assumes, as shown in Definition 3:

$$g_{F^A,r} = |\pi_{XA}(r)| - |\pi_Y(r)|$$

Once we have computed both the confidence and the goodness of each attribute in  $R \setminus XY$ , we produce, for each violated FD, a ranked list of candidate attributes, sorted first according to  $c_{F^A,r}$  and then, as secondary sorting key, according to  $g_{F^A,r}$ . Consider  $F_1 : X \rightarrow Y$  with  $X = [District, Region]$  and  $Y = [AreaCode]$ . The confidence and goodness of  $F_1$  are:

$$c_{F_1,Places} = \frac{|\pi_{District,Region}(Places)|}{|\pi_{District,Region,AreaCode}(Places)|} = \frac{2}{4} = 0.5$$

$$g_{F_1,Places} = |\pi_{District,Region}(Places)| - |\pi_{AreaCode}(Places)| = 2 - 4 = -2$$

$F_1$  is not satisfied by the tuples in *Places*, thus, for each candidate attribute  $A$  in relation *Places* we compute the two parameters  $c_{F_1^A,Places}$  and  $g_{F_1^A,Places}$  (Table 1 shows the results). The two candidate attributes  $Z_1 = [Municipal]$

$A$	$c_{F_1^A,Places}$	$g_{F_1^A,Places}$
Municipal	$4/4 = 1$	0
PhNo	$7/7 = 1$	3
Street	$7/8 = 0.875$	3
Zip	$4/5 = 0.8$	0
City	$4/5 = 0.8$	0
State	$3/5 = 0.6$	-1

Table 1: Evolving  $F_1 : [District, Region] \rightarrow [AreaCode]$

and  $Z_2 = [PhNo]$  allow us to obtain new *exact* FDs, while every other attribute does not. Moreover, attribute  $Z_1$  has a better rank because it allows to discriminate the distinct values of *District*, *Region*, *AreaCode* in a homogeneous way.

### 4.3 When more than one attribute is needed

Until now we have assumed that we repair an FD by adding only one attribute to its antecedent. Of course, it might happen that adding only one attribute is not enough to obtain an exact new FD. In this case, we can either stop or try to find a “more specific” FD by adding more attributes to

its antecedent. We handle this scenario as an iterative process where, at each step of iteration, the method presented so far is applied. Thus, at each step we have to choose the next attribute to be added to the antecedent and we do so by adding the attribute that produces the candidate FD with the highest rank. Consider  $F_4 : X \rightarrow Y$  with  $X = [District]$  and  $Y = [PhNo]$ , whose confidence and goodness are:

$$c_{F_4, Places} = \frac{|\pi_{District}(Places)|}{|\pi_{District, PhNo}(Places)|} = \frac{2}{7} = 0.29$$

$$g_{F_4, Places} = |\pi_{District}(Places)| - |\pi_{PhNo}(Places)| = 2 - 6 = -4$$

Thus,  $F_4$  is not satisfied, and Table 2 shows the ranking of the attributes which are candidates to extend it.

$A$	$c_{F_4^A, Places}$	$g_{F_4^A, Places}$
Street	0.875	1
Municipal	0.571	-2
AreaCode	0.571	-2
City	0.571	-2
Zip	0.5	-2
State	0.429	-3
Region	0.286	-4

Table 2: Evolving  $F_4 : [District] \rightarrow [PhNo]$

As we can see, there is no attribute that allows us to obtain an exact new FD thus we proceed by adding the attribute that generates the candidate FD with the highest rank. We obtain a new FD which is still not exact and then apply our method again to look for attributes that can be added to its antecedent. Therefore, we add attribute *Street* to the antecedent of  $F_4$  and obtain  $F_4^{Street} : [District, Street] \rightarrow [PhNo]$ . Table 3 shows the ranking of the attributes which are the candidates to extend  $F_4^{Street}$ .

$B$	$c_{F_4^{Street, B}, Places}$	$g_{F_4^{Street, B}, Places}$
Municipal	1	4
AreaCode	1	4
Zip	0.889	4
City	0.875	4
State	0.875	3

Table 3: Evolving  $F_4^{Street} : [District, Street] \rightarrow [PhNo]$

We have found two attributes, *Municipal* and *AreaCode*, that allow to extend  $F_4$  and obtain an exact new FD. Therefore, the two pairs *Street, Municipal* and *Street, AreaCode* allow to extend the antecedent of  $F_4$  and obtain an exact new FD. They score the same value also for the goodness thus they are actually equivalent w.r.t. our aim. In such a case, it is for the designer to choose which one is more significant w.r.t. the application scenario.

#### 4.4 Algorithm

Our approach is formalized by Algorithm 1 which receives as input a relation  $R$  (both instance and schema) and the set  $\mathcal{F}$  of FDs defined over  $R$ . First of all, the function **OrderFDs** (line 2) orders all FDs according to the rank introduced in Section 4.1. Then, for each functional dependency

$F$ , the algorithm computes its confidence  $c_{F,r}$  (line 4) in order to understand whether the FD is satisfied or not. If it is not satisfied, it calls the **ExtendByOne** function that computes the confidence and goodness (w.r.t.  $F$ ) of all attributes in  $R$  other than those that are already in  $F$  (lines 3 and 4 in Algorithm 2) and returns the set of all candidates sorted according to their rank. Finally, if the considered attribute allows to obtain an exact new FD, it is added to the set of exact new FDs (line 9 in Algorithm 1).

---

#### Algorithm 1 FindFDRepairs (pseudocode).

---

**Input:**  $R$  the schema of a relation

$r$  the instances of  $R$

$\mathcal{F}$  the functional dependencies defined on  $R$

**Output:** *Exact* the set of exact FDs obtained

---

```

1:  $Exact = \langle \rangle; Cand = \langle \rangle$ 
2:  $FDs = \mathbf{OrderFDs}(\mathcal{F})$ 
3: for all  $F : X \rightarrow Y \in \mathcal{F}$  do
4:    $c_{F,r} = \frac{|\pi_X(r)|}{|\pi_{XY}(r)|}$ 
5:   if  $c_{F,r} < 1$  then
6:      $Cand = \mathbf{ExtendByOne}(F, R, r)$ 
7:     for all  $\langle F', c_{F',r}, g_{F',r} \rangle \in Cand$  do
8:       if  $c_{F',r} = 1$  then
9:          $Exact = \mathbf{addInOrder}(Exact, \langle F', c_{F',r}, g_{F',r} \rangle)$ 
10:      end if
11:    end for
12:  end if
13: end for
14: return  $Exact$ 

```

---

Notice that the computation of confidence and goodness can be implemented using SQL queries. In fact, the values  $|\pi_X(r)|$ ,  $|\pi_{XY}(r)|$ ,  $|\pi_{XA}(r)|$ ,  $|\pi_{XAY}(r)|$ ,  $|\pi_{XY}(r)|$ ,  $|\pi_A(r)|$  used in the algorithm are computed by counting the number of distinct tuples over the set of attributes involved in the antecedent and consequent of the FD. For example, the confidence of  $F_1$ , is the ratio between the results of:

$Q_1$ : `select count(distinct District, Region) from Places`

$Q_2$ : `select count(distinct District, Region, AreaCode) from Places`

The computation of these queries heavily depends on the query plan implemented by the DBMS and on the presence of supporting data structure such as indices. What we can say is that, considering the worst case scenario where no optimization techniques are implemented, we have a  $O(n \log n)$  complexity because counting the distinct values corresponds to a sorting ( $O(n \log n)$ ) followed by counting ( $O(n)$ ). Moreover, looking for a single attribute at a time to extend the antecedent of an FD is linear with respect to the number of attributes in the relation.

However, when looking for repairs that contain more than one attribute, things get more complicated because the number of candidate repairs grows exponentially with respect to the number of attributes. To limit this problem we use a greedy algorithm that chooses the FD candidates according first to the number of attributes in their antecedent and then to their rank. To this aim, we modify Algorithm 1 by introducing a queue that contains candidate repairs sorted by increasing cardinality of the antecedent and decreasing

---

**Algorithm 2** ExtendByOne ( $F, R, r$ ) pseudocode.

---

```
1:  $Cand = \langle \rangle$ 
2: for all  $A \in R \setminus XY$  do
3:    $c_{F^A,r} = \frac{|\pi_{XA}(r)|}{|\pi_{XAY}(r)|}$ 
4:    $g_{F^A,r} = |\pi_{XA}(r)| - |\pi_Y(r)|$ 
5:   if  $c_{F^A,r} = 1$  then
6:      $Cand = \text{addInOrder}(Cand, \langle F^A, c_{F^A,r}, g_{F^A,r} \rangle)$ 
7:   end if
8: end for
9: return  $Cand$ 
```

---

rank: given an FD  $F$  that needs to be evolved, Algorithm 3 first generates all the candidates obtained by adding one attribute to the antecedent of  $F$  (line 1) and inserts them into a queue ordered by decreasing rank (line 2). Then, one at the time, it removes the first candidate in the queue (line 4); if its confidence is 1, it adds it to the set of candidates that allow to find an exact FD (line 6), otherwise it generates all the candidates obtained by adding one attribute to the antecedent of the FD (line 8) and inserts them into the queue, again sorted first by increasing cardinality of their antecedent and then by decreasing rank (line 9). The process is repeated while there are still candidates left in the queue and, at the end of the process, the algorithm returns the set of all candidates that allow to obtain an exact FD (line 12). Notice two things: (i) the stop condition of the algorithm can be easily changed to end when the first repair is found (in this way the algorithm does not need to explore the whole search space); (ii) since the candidates in the queue are ordered first according to the number of attributes in their antecedent (and then according to their rank), the first repair found is also a minimal one, that is, it contains the minimum number of attributes that need to be added to the antecedent of the considered FD to repair it.

---

**Algorithm 3** Extend ( $F, R, r$ ) (pseudocode).

---

```
1:  $Cand = \text{ExtendByOne}(F, R, r)$ 
2:  $\text{addInOrder}(\text{Queue}_F, Cand)$ 
3: while  $\text{Queue}_F$  not empty do
4:    $\langle F', c_{F',r}, g_{F',r} \rangle = \text{removeFirst}(\text{Queue}_F)$ 
5:   if  $c_{F',r} = 1$  then
6:      $\text{addInOrder}(\text{Exact}, \langle F', c_{F',r}, g_{F',r} \rangle)$ 
7:   else
8:      $Cand = \text{ExtendByOne}(F', R, r)$ 
9:      $\text{addInOrder}(\text{Queue}_F, Cand)$ 
10:  end if
11: end while
12: return  $\text{Exact}$ 
```

---

The complexity of managing the queue corresponds to the complexity of a sorting algorithm. Of course, the number of candidates inserted into the queue is exponential with respect to the number of attributes in the relation. In order to find all candidates that allow to obtain an exact FD we need to explore the whole search space. However, this does not happen if we decide to stop when we find the first candidate instead. In the latter case, given the ordering we use to explore the search space, we can ensure that we reach our objective, as stated at the beginning of Section 4, that is, that we are able to find a minimal repair for a given FD.

Notice that a minimal repair might not always be the

best choice. Suppose we are trying to repair a given FD  $F : X \rightarrow Y$  and suppose there are two ways to do it: i) we can add attribute  $A$  which has the property of being UNIQUE and ii) we can add the two attributes  $B$  and  $C$ , neither of whom is UNIQUE. Of course, since we privilege shorter repairs, the algorithm will privilege the first repair which unfortunately goes against what we have discussed in Section 3. To address this drawback we are currently investigating the use of a user-specified maximum goodness threshold. The idea is to use it to privilege those repairs whose goodness is lower than the threshold. In particular, we are currently considering combining such a threshold with our confidence and goodness measures in order to provide an objective function that guides our repair strategy.

## 5. THEORETICAL COMPARISON WITH THE ENTROPY-BASED APPROACH

The technique presented in [17] finds attributes that are good candidates to extend the antecedent of an FD, based on its variation of information, which in its turn is based on the entropy measure. In this section we dub this Entropy-Based method EB, and compare it with ours (CB) which is based mainly on the Confidence measure.

Even though the aim of the EB method is the same as ours, there are some differences between them: the first difference is that CB easily supports the evolution of an FD by adding more than one attribute in its antecedent; second, and more important, to understand if an attribute is a good candidate to extend the FD, we only compute its confidence, while with EB more complex computations are needed. To discuss this, we need to introduce formally the notion of *Entropy*.

Given two clusterings  $\mathcal{C}$  and  $\mathcal{C}'$ , we can compute the *Variation of Information* (VI) [19] between them as the sum of the two conditional entropies:

$$VI(\mathcal{C}, \mathcal{C}') = H(\mathcal{C}|\mathcal{C}') + H(\mathcal{C}'|\mathcal{C})$$

where the conditional entropy of  $\mathcal{C}$  given  $\mathcal{C}'$  is defined as:

$$H(\mathcal{C}|\mathcal{C}') = - \sum_{k=1}^K \sum_{k'=1}^{K'} P(k, k') \log P(k|k')$$

where:  $P(k, k') = \frac{|C_k \cap C'_{k'}|}{n}$  is the joint probability distribution associated to the pair  $(\mathcal{C}, \mathcal{C}')$ ,

$$P(k|k') = \frac{P(k, k')}{P(k')} = \frac{|C_k \cap C'_{k'}|}{|C'_{k'}|}$$

is the conditional probability distribution associated to  $\mathcal{C}$  given  $\mathcal{C}'$ , and  $P(k') = \frac{|C'_{k'}|}{n}$  the marginal probability distribution associated to  $\mathcal{C}'$ . Note that  $VI(\mathcal{C}, \mathcal{C}')$  is symmetric with respect to the two clusterings.

Given  $F : X \rightarrow Y$ , the EB method creates a ground truth clustering  $\mathcal{C}_{XY}$  and then looks for an attribute  $A$  that, when added to the antecedent of  $F$ , allows to obtain a clustering  $\mathcal{C}_{XA}$  that is either homogeneous or, preferably, homogeneous and *complete* w.r.t.  $\mathcal{C}_{XY}$  (i.e., with VI equal to zero).

Note that, with EB, for each FD, the algorithm computes a ground truth clustering that is obtained by scanning all tuples and grouping them according to the attributes in the FD; then, for each attribute  $A$  in the relation, the algorithm computes the clustering  $\mathcal{C}_A$  of the relation in the same manner; and finally the two clusterings must be compared by

computing the intersections of all pairs of clusters in order to determine the variance of information. This last action requires, for each cluster in the ground truth clustering, to scan all clusters in  $\mathcal{C}_A$ . Thus, the EB method requires to store the tuples in order to be able to perform the intersections between clusters while with the CB technique we do not keep trace of all tuples in the groups but only of their amount.

We now show that the *confidence* and *goodness* parameters introduced in Section 3 can be successfully used instead of the conditional entropies and that these two simple parameters give rise to a measure which is equivalent to the VI measure.

Given  $F : X \rightarrow Y$ , the EB method chooses  $\mathcal{C}_{XY}$  as the ground truth clustering and, for each attribute  $A$ , considers the clustering  $\mathcal{C}_A$  in order to understand how well it matches  $\mathcal{C}_{XY}$ . This is done by taking advantage of the two conditional entropies involved in the definition of VI, but *not symmetrically*. In fact, a modified version of the VI is introduced that considers first the conditional entropy of  $\mathcal{C}_{XY}$  given  $\mathcal{C}_{XA}$ , i.e.  $H(\mathcal{C}_{XY}|\mathcal{C}_{XA})$ , and then the conditional entropy of  $\mathcal{C}_A$  given  $\mathcal{C}_{XY}$ , i.e.  $H(\mathcal{C}_A|\mathcal{C}_{XY})$ . Then, the EB approach selects the attribute  $A$  that has the lowest value of  $H(\mathcal{C}_{XY}|\mathcal{C}_{XA})$  and, in the case of a tie, the attribute  $A$  with the lowest value of  $H(\mathcal{C}_A|\mathcal{C}_{XY})$ .

The first conditional entropy  $H(\mathcal{C}_{XY}|\mathcal{C}_{XA})$  measures the *non homogeneity* property of  $\mathcal{C}_{XA}$  with respect to  $\mathcal{C}_{XY}$ . In fact, it is easy to see that when a class  $C_{xa} \in \mathcal{C}_{XA}$  is such that  $C_{xa} \subseteq C_{xy}$  for some  $C_{xy} \in \mathcal{C}_{XY}$ , then

$$\log P(C_{xy}|C_{xa}) = \log \frac{P(C_{xy} \cap C_{xa})}{P(C_{xa})} = \log \frac{P(C_{xa})}{P(C_{xa})} = 0$$

Thus, when  $\mathcal{C}_{XA}$  is *homogeneous* with respect to  $\mathcal{C}_{XY}$ , the relative conditional entropy  $H(\mathcal{C}_{XY}|\mathcal{C}_{XA})$  is zero. Similarly, the second conditional entropy  $H(\mathcal{C}_A|\mathcal{C}_{XY})$  is zero when every class  $C_{xy} \in \mathcal{C}_{XY}$  is such that  $C_{xy} \subseteq C_a$  for some  $C_a \in \mathcal{C}_A$ . When this happens, the completeness property for  $\mathcal{C}_A$  versus  $\mathcal{C}_{XY}$  is verified.

The best attribute found by the EB technique is both *homogeneous* and *complete* implying that the VI is zero.

In the following we propose a slight variation of the EB approach of [17], based on the original definition on VI as in [19], i.e.

$$VI(\mathcal{C}_{XY}, \mathcal{C}_{XA}) = H(\mathcal{C}_{XY}|\mathcal{C}_{XA}) + H(\mathcal{C}_{XA}|\mathcal{C}_{XY})$$

This allows us to make the comparison clearer, while not affecting the results. Note that, given  $F : X \rightarrow Y$ , VI can be seen as a measure, denoted by  $\varepsilon_{VI} := VI$ , on all FDs  $F^A : XA \rightarrow Y$ , where  $A$  is, as introduced in Section 4.2, a candidate attribute to repair  $F$ . This measure is equal to zero when there is homogeneity and completeness between the two clusterings  $\mathcal{C}_{XA}$  and  $\mathcal{C}_{XY}$ . Let us also introduce the measure  $\varepsilon_{CB}$ , based on our *confidence* and *goodness* coefficients of Definition 3:

$$\varepsilon_{CB} := i_{C_{FA}} + \hat{g}_{FA}$$

where  $i_{C_{FA}} := 1 - c_{FA}$  is the “degree of inconsistency” introduced in Section 4.1 and  $\hat{g}_{FA} := |g_{FA}|$  is the absolute value of the goodness coefficient. This measure is equal to zero when  $F^A$  allows to generate a bijective function between the classes of  $\mathcal{C}_{XA}$  and those of  $\mathcal{C}_Y$ . We can state that the two measures are equivalent, i.e. they have the same null

sets (the sets where they assume the null value) and, consequently, the same support sets (the sets where the measures assume a strictly positive value):

**Theorem 1** *Let  $R$  be a relation schema and  $F^Z : XZ \rightarrow Y$  a functional dependency defined on  $R$  as above. Then the measures  $\varepsilon_{CB}$  and  $\varepsilon_{VI}$  are equivalent.*

**Proof 1** *First we observe that the CB best case  $\{c_{FZ} = 1, g_{FZ} = 0\}$  corresponds to  $\{\varepsilon_{CB} = 0\}$  and the EB best case  $\{VI = 0\}$  corresponds to  $\{\varepsilon_{VI} = 0\}$ .*

*Let us recall that given two measures  $P$  and  $Q$ , the measure  $P$  is absolutely continuous w.r.t. the measure  $Q$  (or  $P$  is dominated by  $Q$ ) if for all  $A$  such that  $Q(A) = 0$  one has that  $P(A) = 0$ , i.e. when the null sets of  $Q$  are also null sets of  $P$ . Moreover in this case by Radon-Nikodym theorem (see [20]) there exists a (positive) density  $f$  such that in differential form one has that  $dP = fdQ$ . If in addition  $Q$  is also absolutely continuous w.r.t.  $P$ , then the two measures are called equivalent. In particular in this case one can show that  $dQ = f^{-1}dP$  ([20]).*

*We prove that  $\varepsilon_{VI}$  is absolutely continuous w.r.t.  $\varepsilon_{CB}$ . We put  $B = F^Z : XZ \rightarrow Y$  and suppose that  $\varepsilon_{CB}(B) = 0$ . Then  $\{c_{FZ} = 1\}$  and  $\{g_{FZ} = 0\}$ . When the confidence is equal to one we also have the homogeneity property of  $\mathcal{C}_{XZ}$  versus  $\mathcal{C}_{XY}$ . In fact when  $\{c_{FZ} = 1\}$  there is a proper association for every  $C_{xz} \in \mathcal{C}_{XZ}$ , that is:*

$$\forall C_{xz} \in \mathcal{C}_{XZ} \quad \exists! \quad C_y \in \mathcal{C}_Y \quad \text{s.t.} \quad C_{xz} \subseteq C_y$$

*Thus,  $\mathcal{C}_{XZ}$  is homogeneous with respect to  $\mathcal{C}_Y$  and it follows that  $\mathcal{C}_{XZ}$  is homogeneous also with respect to  $\mathcal{C}_{XY}$ , i.e.:*

$$\forall C_{xz} \in \mathcal{C}_{XZ} \quad \exists! \quad C_{xy} \in \mathcal{C}_{XY} \quad \text{s.t.} \quad C_{xz} \subseteq C_{xy}$$

*In fact if there exists a  $C_{xz}$  whose tuples are contained in more than one class  $C_{xy}$ , it would be  $|\mathcal{C}_{XZ}| < |\mathcal{C}_{XY}| \leq |\mathcal{C}_{XZY}|$ . But this cannot happen, since*

$$\{c_{FZ} = 1\} \Leftrightarrow |\mathcal{C}_{XZ}| = |\mathcal{C}_{XZY}|$$

*Therefore, also the confidence coefficient of the CB method is a measure of the homogeneity property of  $\mathcal{C}_{XZ}$  versus  $\mathcal{C}_{XY}$ : there is homogeneity when the confidence coefficient is one.*

*Moreover, when in addition also the goodness coefficient is zero, the clustering  $\mathcal{C}_{XZ}$  has also the completeness property versus  $\mathcal{C}_{XY}$ , in the sense that  $H(\mathcal{C}_{XZ}|\mathcal{C}_{XY}) = 0$ . In fact, we recall that a clustering has the cited completeness property when*

$$\forall C_{xy} \in \mathcal{C}_{XY} \quad \exists! \quad C_{xz} \in \mathcal{C}_{XZ} \quad \text{s.t.} \quad C_{xy} \subseteq C_{xz}$$

*If there exists a class  $C_{xy}$  whose tuples are in part contained in some  $C_{xz}$  and in part in some other  $\hat{C}_{xz}$ , then it would be  $|\mathcal{C}_Y| \leq |\mathcal{C}_{XY}| < |\mathcal{C}_{XZ}|$ . But this cannot happen, since*

$$\{g_{FZ,r} = 0\} \Leftrightarrow |\mathcal{C}_{XZ}| = |\mathcal{C}_Y|$$

*Since the homogeneity plus completeness properties between the two clusterings implies  $\varepsilon_{VI} = 0$  we have proved that the measure  $\varepsilon_{VI}$  is dominated by the measure  $\varepsilon_{CB}$ .*

*We show that  $\varepsilon_{CB}$  is dominated by  $\varepsilon_{VI}$ . Let us suppose that  $\varepsilon_{VI}(B) = 0$ . Then  $VI = 0$ , which implies that the homogeneity and the completeness properties hold. As a consequence the two clusterings are exactly equal, i.e. every single class in correspondence contains the same subset of tuples. In particular this means that: a)  $|\mathcal{C}_{XY}| = |\mathcal{C}_{XZ}|$*

*b)  $\forall y \quad \exists! \quad (x, z)$  and therefore  $|\mathcal{C}_{XZ}| = |\mathcal{C}_Y|$*



$c) \forall(x, z) \exists! y = z$  and therefore  $|\mathcal{C}_{XZY}| = |\mathcal{C}_{XZ}|$   
and this implies  $ic_{FZ} = 0$  and  $g_{FZ} = 0$ , i.e.  $\varepsilon_{CB}(B) = 0$ .

Finally the two measures are equivalent: they have the same null sets and, consequently, the same support sets (that is the sets where the measures assume strictly positive values). Moreover by Radon-Nikodym theorem there exists a positive density  $f$ , such that

$$d\varepsilon_{CB}(B) = f(B)d\varepsilon_{VI}(B), \quad d\varepsilon_{VI}(B) = f^{-1}(B)d\varepsilon_{CB}(B).$$

The proof of Theorem 1 shows that the two measures have the same support sets (i.e. the sets where a measure assumes strictly positive values).

We remark that the measures  $\varepsilon_{VI}$  and  $\varepsilon_{CB}$  can be considered as acting on a general  $F : X \rightarrow Y$ , with  $X$  and  $Y$  sets of attributes in  $R$ , and in this case they assume respectively the form

$$\varepsilon_{VI}(F) = H(\mathcal{C}_{XY}|\mathcal{C}_Y) + H(\mathcal{C}_Y|\mathcal{C}_{XY})$$

and

$$\varepsilon_{CB}(F) := ic_F + \hat{g}_F.$$

Moreover it can be proved, in the same way as for Theorem 1, that they are equivalent measures.

Usually the equivalence relation between measures is rather weak: they have the same support sets but the values they assume can be very different. Since in our case the number of possible FDs is finite and our measures are finite too, the absolute continuity property is a sort of continuity property between the two measures; more precisely, one can prove that, in our finite case, the mutual absolute continuity property of  $\varepsilon_{CB}$  and  $\varepsilon_{VI}$  is equivalent to the following  $\epsilon - \delta$  property:

$\forall \epsilon \exists \delta$  s.t.  $\varepsilon_{CB}(A) < \epsilon$  for each  $A$  with  $\varepsilon_{VI}(A) < \delta$  (and the same relation holds with  $\varepsilon_{CB}$  and  $\varepsilon_{VI}$  interchanged).

As a consequence, our measures are equivalent and assume comparable values in their support sets. Moreover both EB and CB are based on algorithms which are looking for exactly the sets where the two measures assume the value zero.

We want to stress the fact that the CB method is indeed simpler than the EB approach both from the conceptual and computational point of view. First, because CB uses the classical framework of set functions with well-known elementary mathematical concepts. Moreover, with CB we have to perform only a few cardinality computations of the notable clusterings associated to a given FD in order to achieve our ranked list of attributes, i.e. with no need to enter in the detailed structure of the involved clusterings.

We have compared our CB approach only with the EB method in [17]. We think that this comparison is sufficient for the following reasons. In [21] it has been shown that from an axiomatic point of view the best *approximation measure* for FDs is the *information dependency measure*. One can easily prove that the measure introduced in [21] for an arbitrary FD  $F : X \rightarrow Y$  is a normalized version of the first conditional entropy entering in the VI, that is  $H(\mathcal{C}_{XY}|\mathcal{C}_X)$ . We observe that also what we call “degree of inconsistency”  $ic_F = 1 - c_F$  can be seen as an *approximation measure* of how far the given FD is from generating an exact function and that from the proof of Theorem 1 one can deduce that our measure  $ic_F$  is equivalent to the *approximation measure* given by  $H(\mathcal{C}_{XY}|\mathcal{C}_X)$ . Since in [21] there is also an accurate and complete review of the *approximation measures* in the

literature, we finally conclude that the comparison of our CB method with the EB approach proposed in [17] is sufficient. It is now quite clear that the CB method grants results that are fully comparable with those obtained by means of the EB method, with the important difference that the basic concepts and the required computations are much simpler.

## 6. EXPERIMENTAL RESULTS

We implemented our method in a Java prototype tool. Initially, users connect to a MySQL database and visualize its relations and all FDs defined on each relation; then, they are allowed to add other FDs to the ones that are already defined, and finally they can start the process of FD validation.

We tested the tool on both real and synthetic databases and studied the time needed to find FD repairs. Our study was conducted varying both the FDs and the number of attributes and tuples in the relations.

All the experiments were run on a Core i5 2.6 GHz PC with 4 GB of memory and Windows 8 x64 operating system.

### 6.1 Synthetic databases

We used DBGEN to independently generate three synthetic databases of different sizes: Table 4 shows the features of the generated relations in terms of numbers of attributes and tuples. For each instance of the three databases, we defined one FD on each relation and run our algorithm to understand how execution time varies depending on the dimension of the dataset. Notice that, as shown in Table 5, all FDs have one attribute in the antecedent and one in the body and by processing time we mean the time it took for the algorithm to find all possible repairs for the given FD.

Table	arity	100MB	250MB	1GB
		card.	card.	card.
customer	8	15 000	30 043	150 249
lineitem	16	601 045	1 196 929	6 005 428
nation	4	25	25	25
orders	9	149 622	301 174	1 493 724
part	9	20 000	40 098	199 756
partsupp	5	80 533	160 611	779 546
region	3	5	5	5
supplier	7	1 000	2 000	10 000

Table 4: TPC-H Databases Overview

Figure 3 shows, for the 1GB synthetic database, how the processing time varies depending on the number of attributes (Figure 3a), number of tuples (Figure 3b) and overall dimension of the table (Figure 3c).

We report only the plots related to the 1GB database because of space limitations. However, we noticed that the time needed to repair the FDs is higher for bigger datasets, but the trend is the same. In fact, the trends for the 100MB and 250MB databases are very similar to the one for the 1GB database, although on a smaller scale. This happens because the structure of the three datasets is the same, thus what changes is only the number of tuples. Synthetic datasets tend to behave in a somehow “uniform” way, thus we performed a study on real datasets, with the aim of understanding in what way the number of attributes and the number

Table	FD	100MB	250MB	1GB
		processing time	processing time	processing time
customer	$[name] \rightarrow [address]$	1s 276ms	2s 873ms	20s 657ms
lineitem	$[partkey] \rightarrow [suppkey]$	9m 42s 708ms	21m 20s 599ms	1h 59m 19s 884ms
nation	$[name] \rightarrow [regionkey]$	5ms	5ms	6ms
orders	$[custkey] \rightarrow [orderstatus]$	8s 621ms	19s 726ms	1m 57s 103ms
part	$[name] \rightarrow [mfgr]$	1s 3ms	1s 983ms	18s 561ms
partsupp	$[suppkey] \rightarrow [availqty]$	4s 450ms	10s 570ms	1m 3s 909ms
region	$[name] \rightarrow [comment]$	3ms	3ms	3ms
supplier	$[name] \rightarrow [address]$	74ms	141ms	717ms

Table 5: FindFDRepairs processing times

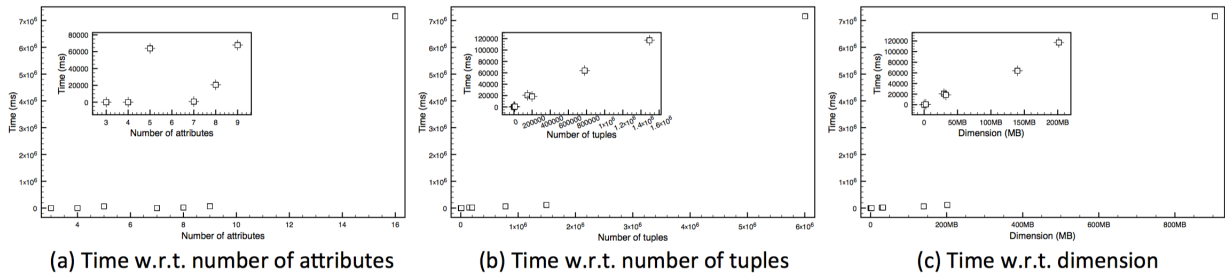


Figure 3: Processing times for the 1GB synthetic database

of tuples in a relation influence the algorithm.

## 6.2 Real-life databases

We conducted our experiments using the relations listed in Table 6, which also summarizes, for each of them, the number of attributes and tuples. On each relation we defined an FD containing one attribute in the antecedent and one in the consequent and then run our algorithm to find one possible repair (that is, the algorithm stops when it finds the first repair). Table 6 shows the results in terms of the time needed to execute the task.

Table	arity	card.	FD process time
Places	9	10	257ms
Country <sup>3</sup>	15	239	32ms
Rental <sup>4</sup>	7	16044	588ms
Image <sup>5</sup>	14	124768	2m 52s
PageLinks <sup>6</sup>	3	842159	4s 678ms
Veterans <sup>7</sup>	481	95412	29m 45s

Table 6: Real Databases Overview and processing times

We can see that relations with a higher number of attributes take longer time to be processed while the same does not hold in general for relations with high amounts of tuples. In fact, as we will see in the next Section, the time grows exponentially with the number of attributes while tuples do not influence it so much.

<sup>3</sup><http://dev.mysql.com/doc/index-other.html>

<sup>4</sup><http://dev.mysql.com/doc/index-other.html>

<sup>5</sup><http://dumps.wikimedia.org/backup-index.html>

<sup>6</sup><http://dumps.wikimedia.org/backup-index.html>

<sup>7</sup><http://kdd.ics.uci.edu/databases/kddcup98/kddcup98.html>

There are many factors that influence execution time: these mainly concern the content of the tables (e.g. number of null values, number of candidates at one level that improve confidence, etc . . . ), whose impact on computation time needs to be studied in details. For example, in the results shown in Table 6 we have that even though Places is a smaller relation than Country (both in terms of attributes and tuples) it took longer to compute the results. This happens because for the first relation, longer repairs were needed, in fact, for table Places, the algorithm added 2 attributes to repair the given FD while for relation Country it added only 1 attribute. Moreover, we can see that even though table PageLinks is the biggest one in terms of tuples, it took less time to repair it than the Image table. This happens because the PageLinks relation has only three attributes, and since the FD defined on it has already two attributes (one in the antecedent and one in the consequent) the algorithm had to consider only the third one in the table. On the other hand, in the Image table, the algorithm had to add 2 attributes to the antecedent of the FD to repair it.

Nr. of tuples	Nr. of attributes		
	10	20	30
10K	26s	4m16s	17m34s
20K	38s	7m56s	35m1s
30K	57s	11m47s	51m48s
40K	2m13s	15m29s	1h28m12s
50K	2m44s	19m34s	1h48s
60K	3m17s	22m51s	1h56m3s
70K	5m13s	36m36s	2h23m8s

Table 7: Processing times for the Veterans relation – find all repairs

### 6.2.1 Case study

To better understand how the number of attributes and the number of tuples in a relation influence our algorithm, we performed a case study using the Veterans relation (see Table 6) which has 481 attributes (323 of which do not have null values) and 95412 tuples containing only non-null values. Notice that, since the attributes occurring in an FD are not allowed to contain NULL values, when generating candidate repairs, we only consider the addition of those attributes that do not contain NULLs. We created several instances of this relation, each containing a different number of attributes and tuples. We defined an FD containing one attribute in the antecedent and one in the consequent and then run our algorithm to find: (i) all possible repairs (Table 7 shows the results) and (ii) the first repair (results in Table 8). From these results we can confirm the evidence of the synthetic dataset study, namely, that execution time grows much quicker with the number of attributes in the relation than with the number of tuples. Computation time grows exponentially with the number of attributes while an increase in the number of tuples implies longer query processing times, but does not affect the algorithm processing time so much. Moreover, we can see that processing times are much smaller if the algorithm stops when it finds the first repair instead of exploring the whole search space. However, it might happen that the two times are very similar (e.g. in the 70K tuples DB with 10 attributes) when the algorithm is not able to find a repair for the given FD.

Nr. of tuples	Nr. of attributes		
	10	20	30
10K	8s76ms	53s96ms	2m23s
20K	18s22ms	1m30s	4m10s
30K	27s64ms	2m15s	6m12s
40K	1m25s	3m4s	8m18s
50K	1m47s	3m46s	10m38s
60K	2m10s	4m44s	12m51s
70K	5m23s	5m57s	16m10s

Table 8: Processing times for the Veterans relation – find the first repair

During the experiments we noticed that there are other parameters, generally application-dependent, that influence our method. Just to name a few: (i) the number of distinct values of an attribute: the more distinct values there are, the more time is needed to compute the queries; (ii) the initial confidence of an FD: as can be expected, the smaller the initial confidence, the greater the probability that a longer repair is needed, that is, the more attributes should be added in the antecedent, thus requiring more time; (iii) the average length of the repairs: if an FD needs repairs that add many attributes to the antecedent it will require more computation time. These parameters are related to each other, depend on the domain of application and are very difficult to control and predict.

### 6.3 Quality of results

We claim that our criterion for choosing the order in which attributes are added to the antecedent of functional dependencies favours the quality of the results we obtain. Indeed,

as we have already discussed, our method privileges the addition of attributes that allow us to obtain functions that are “as well defined as possible”, by approximating the goodness to 0. This is because we try to construct an FD that resembles as much as possible a bijective function, mapping the clusters generated by the antecedent of the FD to the clusters generated by the consequent of the FD. This choice allows us to:

- discourage the addition of a UNIQUE attribute: indeed, that would make the rest of the antecedent useless, because that attribute alone determines the consequent of the FD;
- along the same line of thought, encourage the addition of attributes that make the “specificity” of the antecedent of the FD is as much as possible similar to the “specificity” of the consequent;
- support indexing and query optimization, because, when the method manages to find “invertible” FDs, not only the antecedent determines the consequent but also vice-versa; thus, an index built on the antecedent of an FD can be used to efficiently access the attributes in the consequent (if we know the correspondence between the clusters in the antecedent and the clusters in the consequent).

## 7. CONCLUSION

In this paper we proposed a new method for repairing FD violations that works at the intensional level: rather than changing the data, it repairs the FD by adding one or more attributes to its antecedent. To this aim we have used the notions of confidence and goodness of an FD, as measures to estimate if an FD is violated by the data and to what extent. In future we intend to extend the method to other kinds of constraints and to make a more extensive study on the parameters that influence the processing time and on the impact they have when examining a database.

## 8. ACKNOWLEDGEMENTS

This research has been partially funded by the Italian project SHELL CTN01\_00128\_111357 and by the IT2Rail project, funded by European Union’s Horizon 2020 research and innovation programme under grant agreement No: 636078.

## 9. REFERENCES

- [1] T. Murata and A. Borgida. Handling of irregularities in human centered systems: A unified framework for data and processes. *IEEE Transaction on Software Engineering*, 26(10), 2000.
- [2] G. Cugola, E. Di Nitto, A. Fuggetta, and C. Ghezzi. A framework for formalizing inconsistencies and deviations in human-centered systems. *ACM Trans. Software Eng. and Methodology*, 5(3):191–230, 1996.
- [3] L. Caruccio, V. Deufemia, and G. Polese. Relaxed functional dependencies - A survey of approaches. *IEEE Trans. Knowl. Data Eng.*, 28(1):147–165, 2016.
- [4] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.*, 23(5):683–698, 2011.

- [5] C. Giannella and E. L. Robertson. On approximation measures for functional dependencies. *Inf. Syst.*, 29(6):483–507, 2004.
- [6] H. Nakayama, A. Hoshino, C. Ito, and K. Kanno. Formalization and discovery of approximate conditional functional dependencies. In *DEXA*, pages 118–128, 2013.
- [7] J. Wijsen. Temporal dependencies. In *Encyclopedia of Database Systems*, pages 2960–2966. 2009.
- [8] C. Combi, P. Parise, P. Sala, and G. Pozzi. Mining approximate temporal functional dependencies based on pure temporal grouping. In *ICDM Workshops*, pages 258–265, 2013.
- [9] L. E. Bertossi and J. Chomicki. Query answering in inconsistent databases. In *Logics for Emerging Applications of Databases*, pages 43–83. Springer, 2003.
- [10] S. Flesca, F. Furfaro, S. Greco, and E. Zumpano. Querying and repairing inconsistent xml data. In *Web Information System Engineering*, volume 3806 of *LNCS*, pages 175–188, 2005.
- [11] S. Flesca, F. Furfaro, and F. Parisi. Consistent query answers on numerical databases under aggregate constraints. In *DBPL Workshops*, volume 3774 of *LNCS*, pages 279–294, 2005.
- [12] J. Chomicki. Consistent query answering: Opportunities and limitations. In *DEXA*, pages 527–531. IEEE Computer Society, 2006.
- [13] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 197(1-2):90–121, 2005.
- [14] J. Chomicki. Consistent query answering: Five easy pieces. In D. Suciu T. Schwentick, editor, *Proceeding of ICDT'07*, volume 4353 of *LNCS*, pages 1–17, 2007.
- [15] M. Mazuran, E. Quintarelli, R. Rossato, and L. Tanca. Mining violations to relax relational database constraints. In *DaWaK*, pages 339–353, 2009.
- [16] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.
- [17] F. Chiang and R.J. Miller. A unified model for data and constraint repair. In *ICDE 2011*, pages 446–457, 2011.
- [18] J. Segeren, D. Gairola, and F. Chiang. CONDOR: A system for constraint discovery and repair. In *CIKM*, pages 2087–2089, 2014.
- [19] M. Meilă. Comparing clusterings—an information based distance. *J. Multivar. Anal.*, 98(5):873–895, 2007.
- [20] P. Billingsley. *Ergodic Theory and Information*. Wiley, 1965.
- [21] C. Giannella. An axiomatic approach to defining approximation measures for functional dependencies. In *ADBIS*, pages 37–50, 2002.

# Holistic Data Profiling: Simultaneous Discovery of Various Metadata

Jens Ehrlich<sup>1</sup>  
Jakob Zwiener<sup>1</sup>

Mandy Roick<sup>1</sup>  
Thorsten Papenbrock<sup>2</sup>

Lukas Schulze<sup>1</sup>  
Felix Naumann<sup>2</sup>

Hasso-Plattner-Institute (HPI), Potsdam, Germany

<sup>1</sup> `firstname.lastname@student.hpi.de`

<sup>2</sup> `firstname.lastname@hpi.de`

## ABSTRACT

Data profiling is the discipline of examining an unknown dataset for its structure and statistical information. It is a preprocessing step in a wide range of applications, such as data integration, data cleansing, or query optimization. For this reason, many algorithms have been proposed for the discovery of different kinds of metadata. When analyzing a dataset, these profiling algorithms are often applied in sequence, but they do not support one another, for instance, by sharing I/O cost or pruning information.

We present the holistic algorithm MUDS, which jointly discovers the three most important metadata: inclusion dependencies, unique column combinations, and functional dependencies. By sharing I/O cost and data structures across the different discovery tasks, MUDS can clearly increase the efficiency of traditional sequential data profiling. The algorithm also introduces novel inter-task pruning rules that build upon different types of metadata, e.g., unique column combinations to infer functional dependencies. We evaluate MUDS in detail and compare it against the sequential execution of state-of-the-art algorithms. A comprehensive evaluation shows that our holistic algorithm outperforms the baseline by up to factor 48 on datasets with favorable pruning conditions.

## Categories and Subject Descriptors

H.2.8 [Information Systems]: Database Applications; H.3.3 [Information Systems]: Information Search and Retrieval

## General Terms

Algorithms, Performance, Theory

## Keywords

data profiling, inclusion dependency, functional dependency, unique column combination

## 1. DEPENDENCY DISCOVERY

With the ever growing amount of digitally recorded information, the need to maintain, link, and query these information becomes increasingly hard to fulfill. For many applications, such as data mining, data linkage, query optimization, schema matching, or database reverse engineering, it is crucial to understand the data and, in particular, its structure and dependencies [13]. In biological research, for instance, scientists create large amounts of genome data that grow rapidly every year [2]. Originating from different genome sequencers, the data needs to be analyzed and linked to other datasets. This task requires knowledge of several structural properties of the data.

Usually, the reason why data becomes difficult to access is that metadata about the datasets' structure or their dependencies is missing. Therefore, various profiling algorithms have been proposed for the computationally intensive discovery of metadata, such as inclusion dependencies (INDs) [4, 8], unique column combinations (UCCs) [1, 9, 10, 16], and functional dependencies (FDs) [11, 14]. The algorithms SPIDER [4] for the discovery of INDs, DUCC [10] for UCCs and FUN [14] for FDs are among the most efficient algorithms in their respective problem domain. The idea of all these discovery algorithms is to reduce the tremendous search spaces with so called pruning rules: A pruning rule allows to infer the (in)validity of certain unchecked metadata candidates from already checked ones. Each algorithm, however, computes only one type of metadata. While this might be sufficient for some applications, most applications like data exploration or data integration require different types of metadata at the same time [13]. Therefore, current data profiling processes run several highly complex algorithms in a row. Considering that these algorithms and the metadata they discover have many commonalities, the sequential execution is a waste of time and resources.

Some existing profiling algorithms, such as HCA [1] or FUN [14], already leverage some knowledge about other types of metadata to reduce the discovery time (pruning). However, the combination of different profiling algorithms, i.e., a *holistic algorithm*, can utilize many more interleavings to increase its overall efficiency: First, it can facilitate new pruning rules using all collected information at once. In this way, fewer validity checks on the actual data are required. Second, it can share common costs like those required for I/O operations and iteration cycles. Third, it can combine similar data structures from different profiling tasks into one holistic data structure reducing overall memory consumption and initialization costs.

In this paper, we describe new inter-task pruning capabilities and analyze their impact on the algorithm’s runtime. We also develop and evaluate a novel holistic algorithm called MUDS, which jointly discovers unary INDs, minimal UCCs, and minimal FDs in one execution while facilitating all three opportunities for performance optimization mentioned above. If a dataset has favorable pruning conditions – which is true for most real-world datasets – MUDS improves upon the sequential execution of profiling algorithms by up to a factor of 48. In our evaluation, we investigate dataset characteristics that lead to good and poor runtime behavior.

**Contributions.** We first analyze INDs, UCCs, and FDs for their commonalities and examine state-of-the-art profiling algorithms (Section 2). We then discuss different approaches for holistic data profiling and possible pruning rules across profiling tasks (Section 3). Next, we describe how different types of FDs can be discovered or pruned if minimal UCCs are already known (Section 4). Based on the discovery and pruning techniques, we present the novel algorithm MUDS, which utilizes inter-task pruning rules (Section 5). MUDS derives FDs directly from discovered minimal UCCs and facilitates a new depth-first traversal strategy that is based on minimality pruning and the knowledge about non-dependencies (unlike previous level-wise approaches for FDs that solely rely on minimality pruning). Finally, we compare MUDS with the sequential execution of state-of-the-art algorithms and with a holistic adaption of the algorithms SPIDER and FUN (Section 6). Our evaluation shows that MUDS usually not only considerably outperforms the baseline algorithms but also outperforms common FD discovery algorithms on datasets with more than 10 columns.

## 2. PROFILING TASKS

For our holistic approach, we focus on three common and computationally intensive profiling tasks: The discovery of all unary inclusion dependencies, all unique column combinations, and all functional dependencies in a given dataset. This section defines the three tasks and explains one state-of-the-art algorithm for each of them. The chosen algorithms are among the most efficient algorithms for their specific task and exhibit favorable features to combine them into a holistic algorithm. At the end of this section, we compare the nature of the profiling tasks and their search space complexities.

### 2.1 Inclusion dependencies

An inclusion dependency (IND)  $X \subseteq Y$  between attribute sets  $X$  and  $Y$  describes that the projection of  $Y$  contains all values of the projection of  $X$ , i.e., all values in  $X$  are also contained in  $Y$ . Attribute set  $X$  is called the *dependent* and  $Y$  is called the *referenced*. INDs with only one attribute in the sets  $X$  and  $Y$  are called *unary INDs*. Because only these unary INDs are of interest for the holistic discovery of other metadata types, we only consider them in our holistic algorithm. Without any loss of generality, we could discover  $n$ -ary INDs as well, but these would not contribute to the holistic discovery. We also artificially restrict the IND discovery to a single relation, because the two other metadata types UCCs and FDs are defined on only one relation. The search space for a relation with  $n$  attributes, hence, comprises  $n \cdot (n - 1)$  unary IND candidates.

SPIDER is the currently most efficient algorithm for the detection of unary INDs. The algorithm developed by Bauckmann et al. [4] consists of two phases: A sorting phase and a comparison phase. In the first phase, SPIDER sorts the values of each column, eliminates duplicate values, and stores the sorted values in separate lists (Tables 1.1 and 1.2). In the second phase, SPIDER initially assumes that all attributes are included in one another. Then, it iterates simultaneously over the sorted lists in order to invalidate IND candidates (Tables 1.3 and 1.4). For the invalidation, SPIDER selects the group of attributes that all contain the currently smallest value. In our example  $A$  and  $C$  both contain  $w$ . By set intersection, the algorithm then excludes INDs from the candidates: The attributes in this group can only be included in one another, because they exclusively contain a value. So,  $A$  can still depend on  $C$ , but  $A$  cannot depend on  $B$ , because  $B$  does not contain  $w$ . The algorithm continues with next smallest values until only valid INDs remain.

1.			2.			3.			4.		
A	B	C	A	B	C	A	B	C	A	B	C
w	z	x	w		w	→w		→w	w		w
w	x	x	x	x	x	x	→x	x	→x	→x	→x
x	z	w	y			y			y		
y	z	z		z	z		z	z		z	z

Table 1: Example execution of Spider

### 2.2 Unique column combinations

A set  $X$  of attributes is a unique column combination (UCC) if the projection of  $X$  does not contain duplicates; otherwise, it is a non-unique column combination (non-UCC). UCCs are also called key candidates, because the values of the projection of a UCC uniquely define all records.  $X$  is a *minimal* unique column combination (minimal UCC) if it is a UCC and no proper subset of  $X$  is a UCC. The number of possible candidates for UCCs in a relation  $R$  with  $n$  attributes is  $2^n - 1$ . We can visualize the search space of UCC candidates as a lattice of attribute sets, i.e., a Hasse diagram (Figure 1). Each node represents a set of attributes and each edge a superset/subset relationship.

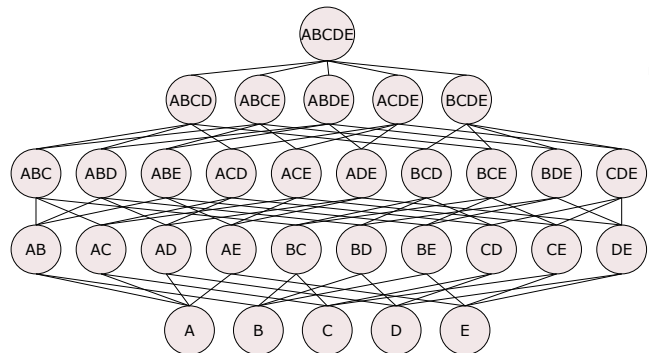


Figure 1: Attribute lattice for five columns

The DUCC algorithm is among the most efficient algorithms for detecting UCCs [10]. It applies a combination of depth-first and random walk strategies to traverse the lattice of UCC candidates. Thereby, DUCC uses the information about both discovered UCCs and non-UCCs for pruning.

The traversal starts from the bottom of the lattice. Every time the algorithm detects a non-UCC, it generates all direct supersets and picks one randomly as its next candidate; if the algorithm detects a UCC, the next candidate is a random direct subset. While traversing, DUCC prunes subsets of non-UCCs and supersets of UCCs from the search space, because subsets of non-UCCs are also non-UCCs, and supersets of UCCs cannot be minimal. It is possible that some column combinations remain unvisited after the random walk. The reason for such “holes” in the lattice is the combination of upwards and downwards pruning. DUCC identifies and fills these holes by comparing the found minimal UCCs with the complement of the found maximal non-UCCs.

During traversal, the algorithm has to check whether a column combination is a UCC or a non-UCC. This check is performed by constructing a position list index (PLI, also called stripped partition) for the column combination. A PLI is a list that contains sets of tuple ids [11]. These tuple ids belong to tuples of the column combination that contain the same value. If a PLI contains no id-set of size two or larger, the column combination contains no duplicate value and is, hence, unique. Because only id-sets of size two or larger are necessary for this test, all id-sets of size one can be removed, i.e., stripped from the PLI. So if the PLI is empty, all values are unique and the column combination is a UCC. To calculate the PLI for an unvisited lattice node  $AB$ , DUCC intersects the PLIs of the nodes  $A$  and  $B$  by pair-wise intersecting their id-sets.

## 2.3 Functional dependencies

Given a relation  $R$ , a functional dependency (FD)  $X \rightarrow A$  between a set of attributes  $X$  and an attribute  $A$  exists if the values of  $X$  uniquely determine the values of  $A$  [6]. We call  $X$  the *left hand side* and  $A$  the *right hand side*. An FD is called trivial if  $A \in X$ . The FD is minimal if no proper subset of  $X$  determines  $A$ . A set of attributes  $X$  can determine multiple other attributes  $Y$ . In the following, we use the short notation  $X \rightarrow Y$  to denote FDs with one or more right hand side attributes. In the lattice shown in Figure 1, every *edge* represents a potential FD. For example, the edge between  $ABC$  and  $ABCD$  represents the FD candidate  $ABC \rightarrow D$ . To count all FD candidates for  $n$  attributes, we count the edges in each level  $k$  with  $k = 1 \dots n$ . Starting with attribute sets of size one, the sets are extended by one attribute in each level until the set in the highest level contains all attributes. In each level  $k$  we find  $\binom{n}{k}$  nodes. Each node in level  $k$  can be connected to  $n - k$  nodes in the next level without generating duplicate connections. The number of FD candidates in a relation with  $n$  attributes is therefore  $\sum_{k=1}^n \binom{n}{k} \cdot (n - k)$ .

Among the many FD discovery algorithms, FUN is one of the fastest algorithms [14]. FUN discovers FDs in a relation  $R$  by traversing the attribute lattice level-wise with a bottom-up strategy. While exploring the attribute lattice, FUN generates PLIs for each traversed attribute set. The algorithm then derives the cardinality of attributes and attribute combinations from their PLIs. This information is used to efficiently detect FDs by *partition refinement* as described in Lemma 1 (from [14]). In a relation  $R$  and a relation instance  $r$ , let  $|X|_r$  denote the cardinality of the projection of  $X$  over  $r$ .

LEMMA 1.  $\forall X \subseteq R, A \in R: X \rightarrow A \Leftrightarrow |X|_r = |X \cup \{A\}|_r$

FUN classifies column combinations into *free sets* and *non-free sets*. A free set  $X$  contains only those attributes that are not functionally dependent on any other attribute in  $X$ . A non-free set contains at least one functionally dependent attribute. Definition 1 describes the set of free sets  $\mathcal{FS}_r$ :

DEFINITION 1. *The set of free sets  $\mathcal{FS}_r$  is defined as*  
 $\forall X \subseteq R: X \in \mathcal{FS}_r \Leftrightarrow \nexists X' \subset X: |X'|_r = |X|_r$ .

The FUN algorithm has a pruning advantage in comparison to other FD algorithms like TANE [11], because it omits certain PLI intersect operations and retrieves missing cardinality information from a node’s child nodes. So instead of performing a PLI intersect, FUN infers the cardinality of a pruned non-free set with a recursive look-up in the non-free set’s subsets.

## 2.4 Comparison of profiling tasks

The discovery of INDs inherently differs from the discovery of UCCs and FDs: For INDs, the attribute values are of interest whereas for FDs and UCCs only the *position* of equal values is relevant. Hence, UCC and FD algorithms use quite different data structures than IND algorithms; these data structures do not allow the reconstruction of values (e.g., PLIs), but improve the algorithms’ efficiency. However, several relationships between FDs and UCCs can be used for pruning (see Sec. 3).

To determine the complexity of a holistic algorithm, we need to inspect the search spaces of the different sub-tasks: A relation with  $n$  attributes contains  $n \cdot (n - 1)$  unary IND candidates. IND discovery is, hence, in  $O(n^2)$ . In the same relation, the number of UCC candidates is  $\sum_{k=1}^n \binom{n}{k}$ , which places the search space of UCCs in  $O(2^n)$ . The number of FD candidates is  $\sum_{k=1}^n \binom{n}{k} \cdot (n - k)$  so that FD discovery is in  $O(n \cdot 2^n)$ . The search space for FDs, therefore, clearly dominates the overall discovery cost. The exponential search spaces of UCCs and FDs in particular dominate the quadratic search space of unary INDs. Our evaluation in Sec. 6.4 shows that the time spent on IND discovery is indeed negligible in comparison to the time spent on UCC and FD discovery. For this reason, INDs can best be calculated as a byproduct in the starting phase of a holistic algorithm.

## 3. HOLISTIC APPROACH

As we described in Section 2.4, IND discovery differs greatly from the discovery of UCCs and FDs. Therefore, INDs are discovered while the input data is read and the values are still accessible. The discovery uses the SPIDER algorithm and mainly profits from sharing its I/O costs with the UCC and FD algorithms, i.e., the data is read only once and then used for the discovery of all three types of metadata. SPIDER additionally profits from the initial PLI-construction that is performed for both the UCC and FD discovery: At construction time, PLIs map values to positions so that SPIDER can retrieve duplicate-free value lists from this mapping, which are more efficient to sort.

If the input dataset contains two identical rows, i.e., duplicate records, then it cannot contain any UCC and, hence, most inter-task pruning rules would not apply. Therefore, we assume that duplicate records, which are forbidden in most database systems anyway, have been removed in a pre-processing step.

We now discuss three basic approaches for the holistic discovery of INDs, UCCs, and FDs.

### 3.1 FDs first

By discovering minimal FDs first, we can derive all minimal UCCs from the discovered FDs [15]. The UCC inference follows Lemma 2. Under the assumption that each row in  $R$  is distinct, every column combination that functionally determines all other attributes of the relation  $R$  is a key in  $R$  and, hence, a UCC:

LEMMA 2.  $\forall U \subseteq R : U \rightarrow R \setminus U \Rightarrow U$  is a UCC

Thus, all UCCs can be inferred from the set of minimal FDs. Without describing an actual algorithm for UCC inference, it is clear that the inference and minimization of UCCs introduces an additional overhead. As several FD discovery algorithms (e.g., FUN and TANE) exist that already find all minimal UCCs while discovering FDs, we do not pursue this FDs-first approach for a holistic algorithm. Instead, we focus on approaches that improve the overall runtime by avoiding this additional overhead and by leveraging pruning.

### 3.2 FDs and UCCs simultaneously

To discover FDs and UCCs simultaneously, we analyzed several FD discovery algorithms and evaluated their extensibility towards UCC discovery. As already described in Sections 2.3 and 2.4, FUN uses an attribute lattice for pruning that is very similar to DUCC’s way of calculating minimal UCCs. Furthermore, FUN traverses all unpruned free sets. The following Lemma 3 shows that all minimal UCCs are “free sets” in FUN’s sense. Therefore, FUN must traverse the minimal UCCs for FD discovery, enabling the discovery of minimal UCCs with little impact on the overall runtime. In a relation instance  $r$ ,  $\mathcal{U}_r$  is the set of all minimal UCCs in  $r$ .

LEMMA 3.  $\forall U \in \mathcal{U}_r : U \in \mathcal{FS}_r$

PROOF. Let  $X \in \mathcal{U}_r$  be a minimal UCC and let  $X \notin \mathcal{FS}_r$  be a non-free set. According to Definition 1,  $X' \subset X$  with  $|X'|_r = |X|_r$  exists.  $X'$  has the same distinct count as  $X$ . Therefore,  $X'$  is a UCC. This contradicts the initial assumption that  $X$  is a *minimal* UCC and, therefore, no subset of the minimal UCC  $X$  can be a UCC.  $\square$

In the original version of FUN, minimal UCCs are detected and used for key-pruning, which is a common pruning rule for FD discovery: The supersets of UCCs can be pruned, because they cannot be the left hand side of a minimal FD. This pruning rule is applied in FD discovery algorithms like TANE and FUN. With small adaptations, it is possible to store the minimal UCCs and return them when the algorithm terminates. This does not impair the runtime of FUN, because no further checks are necessary. We implemented this algorithm and call it HOLISTIC FUN.

### 3.3 UCCs first

If a holistic algorithm first discovers all minimal UCCs, it can leverage this information for the discovery of minimal FDs, due to the key-pruning rule. As we explain in Section 4.1, many left hand sides of minimal FDs are subsets of minimal UCCs. This observation can be used to discover and minimize relevant FDs faster. It can further be used for pruning, because applying this rule reduces the number of traversed column combinations in comparison to level-wise FD discovery algorithms. Several rules can derive the invalidity of certain FDs from the known minimal UCCs. We present these rules in more detail in Section 4 and use them in the implementation of our algorithm MUDS in Section 5.

## 4. DISCOVERING FDS BASED ON UCCS

In this section, we present pruning and inference rules that allow for a fast FD discovery based on known UCCs. These rules lay the foundation for our MUDS algorithm described in Section 5. We show in Section 6 that an algorithm using these rules is usually faster than current state-of-the-art FD discovery algorithms. We first describe the UCC-based pruning rules for FDs. The following three subsections then match the three sub-algorithms of MUDS’ FD discovery: *minimize FDs*, *calculate  $R \setminus Z$* , and *shadowed FDs*.

Figure 2 categorizes the attributes of a relation  $R$  into the set  $Z := \bigcup_{U \in \mathcal{U}} U$ , which is the union of all minimal UCCs, and the set  $R \setminus Z$ , which contains all columns that are not contained in any minimal UCC. In the following, we describe the two situations, shown in Figure 2, where the non-existence of functional dependencies can be inferred from the set of UCCs.

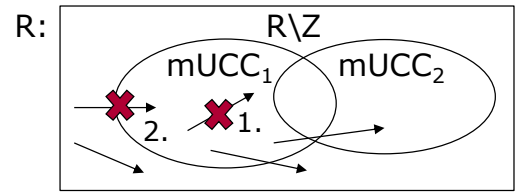


Figure 2: Possible FDs in  $R$  with two minimal UCCs

**1. FDs fully contained in a minimal UCC:** An FD cannot exist if it is fully contained in a minimal UCC (both left and right hand side are subsets of the same minimal UCC). The right hand side of an FD carries only information that could be inferred from the left hand side. Therefore, if a functional dependency existed that is contained in a minimal UCC, the right hand side can be removed from the minimal UCC without changing the uniqueness property. This contradicts the minimality of the UCC.

**2. FDs with a lhs in  $R \setminus Z$  and rhs in  $Z$ :** Consider an FD  $X \rightarrow A$  with a left hand side  $X \subseteq R \setminus Z$  and a right hand side  $A \in U$  with  $U \in \mathcal{U}$ .  $A$  is determined by  $X$ ; hence,  $A$  can be substituted by  $X$  in  $U$ . This yields at least one unique  $U_{subs} = X \cup U \setminus A$ . Thus, a minimal unique  $U_{min}$  must exist that is subset of  $U_{subs}$ .  $U_{min}$  must still contain one or more attributes of  $X$  (otherwise  $U$  could not have been minimal, as  $A$  could have been omitted). It follows that a part of  $X$  is contained in a minimal UCC, which contradicts the proposition that  $X$  is a subset of  $R \setminus Z$ . Thus, no such FD may exist.

In the following, we present rules and operations that enable the discovery of FDs using the two pruning rules described above. In Section 4.1, we first describe how to discover FDs that have left and right hand sides in overlapping minimal UCCs.

### 4.1 FDs between minimal UCCs

A UCC functionally determines all other columns of the same relation. Thus, FDs can be inferred from a discovered UCC. Not all of these FDs are minimal. To find the minimal FDs, the substitution pruning rule can be applied:

**Substitution rule:** For every FD that has an attribute of a minimal UCC as right hand side, we can infer a new UCC, by substituting that attribute in the UCC with the



left hand side of the FD. If the inferred UCC does not exist, the corresponding FD cannot hold and must not be checked. This insight is used to validate FD candidates by checking for corresponding UCCs. For instance, given a relation  $R = \{A, B, C, D, E, F\}$ , the minimal UCCs  $ABC$  and  $DEF$ , and the FD candidate  $BC \rightarrow D$  that we need to check. The UCC  $BCEF$  follows by substituting  $D$  with  $BC$ . Now, a subset of  $BCEF$  that is also a proper superset of  $EF$  must be a minimal UCC. As this minimal UCC does not exist, we know that the FD  $BC \rightarrow D$  cannot exist.

It follows that valid FDs between minimal UCCs must fulfill the following condition: The left hand side and the right hand side of valid FD must be subsets of different and intersecting minimal UCCs. We use this insight in the MUDS algorithm, using an operation that we call *connector lookup*. We describe this operation in detail in Section 5.1.

## 4.2 FDs with right hand sides in $R \setminus Z$

TANE and similar algorithms for FD discovery traverse the attribute lattice bottom-up. In the traversal, these algorithms utilize pruning rules based on the minimality of dependencies. If a left hand side is known to yield only non-minimal FDs (e.g., it already contains an FD), this left hand side is pruned from the lattice and the traversal is continued with a reduced candidate set (upwards pruning). We now propose a traversal strategy that operates similarly to the random walk strategy in the UCC discovery algorithm DUCC. For this traversal, pruning of subsets (downwards pruning) is necessary. This pruning is based on a property of FDs: If  $X \rightarrow A$  does not hold,  $A$  cannot be functionally dependent on any subset of  $X$ .

LEMMA 4.  $\forall X \subseteq R, \forall A \in R, \forall X' \subseteq X :$   
 $X \rightarrow A \Rightarrow X' \rightarrow A$

Lemma 4 can be used to prune downwards. To facilitate this pruning, we traverse a *sub-lattice* for each possible right hand side. A sub-lattice is a lattice created for a specific right hand side attribute, which is omitted from the lattice. The nodes in the sub-lattice contain only the different left hand side candidates. All sub-lattices for an exemplary relation with columns  $A, B, C, D$  are shown in Figure 3. In the sub-lattices, the above mentioned pruning rule applies, because of the fixed right hand side.

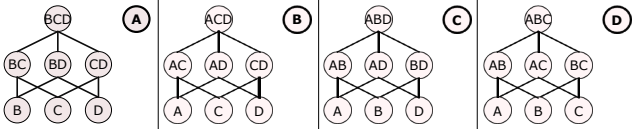


Figure 3: Sub-lattices for the right hand side columns A, B, C, and D

The example in Figure 3 shows that some column combinations are represented in multiple lattices. The column combination  $CD$ , for example, is contained in the first and in the second lattice. Such column combinations are only fully pruned, i.e., we do not need to calculate their PLIs, if the column combination is pruned in all sub-lattices. In the best case, entire sub-lattices can be pruned.

Section 5.2 presents an algorithm that leverages the sub-lattice pruning. The algorithm also assures that only columns of  $R \setminus Z$  are used as right hand sides of FD candidates.

## 4.3 Shadowed FDs

Section 4.1 describes how FDs can be deduced from minimal UCCs. Unfortunately, not all minimal FDs can be found in this way. If a minimal UCC  $U$  and a column combination  $X \subseteq U$  exist and the FD  $X \rightarrow A$  holds, a column  $A$  may be *shadowed* in  $U$ :  $A$  can be part of a left hand side with only other columns of  $U$ . As our algorithm would not consider this left hand side, we call the left hand side and the resulting FD *shadowed*. To still find all minimal FDs, a post-processing step is needed that explicitly checks all left hand sides containing attributes from different minimal UCCs or even  $R \setminus Z$ . We explain the algorithm in Section 5.3 but first give an example of how shadowed attributes arise:

Let  $BCD$ ,  $CDE$ , and  $AD$  be the only minimal UCCs in a dataset for the relation  $R = \{A, B, C, D, E\}$ . Suppose that the minimal UCCs directly contain the minimal FDs, i.e.,  $BCD \rightarrow AE$ ,  $CDE \rightarrow AB$ , and  $AD \rightarrow BCE$ . Now, assume that there is an additional minimal functional dependency  $AC \rightarrow B$ . The inference rule from Section 4.1 cannot find this FD, because the left hand side  $AC$  cannot be deduced from the minimal UCCs  $BCD$ ,  $CDE$ , or  $AD$ . Thus,  $AC \rightarrow B$  is never checked. The existence of the minimal FDs  $BCD \rightarrow A$  indicates that  $A$  is shadowed in every subset of  $BCD$ .  $C$  is shadowed analogously in every subset of  $AD$  by the minimal FD  $AD \rightarrow C$ . Therefore, every attribute that is determined by a subset of  $BCD$  or  $AD$  may also be determined by a subset of  $ABCD$  or  $ACD$  containing  $A$  and  $C$ .

For each minimal FD  $Y \rightarrow W$ , we must find all shadowed attributes  $S$ . The shadowed attributes are added to the left hand side of the functional dependency. This results in the FD  $Y \cup S \rightarrow W$ , which holds, but is not necessarily minimal. The FD must be minimized and we provide pseudo-code for a minimization algorithm in Section 5.3.

## 5. THE MUDS ALGORITHM

In this section, we develop a holistic profiling algorithm called MUDS, which jointly discovers unary INDs, minimal UCCs, and minimal FDs facilitating pruning across the different profiling tasks. The acronym MUDS is a composition of the algorithms key features: “Mu” for Minimizing UCCs, “D” for the Depth-first approach that is used to find FDs with right hand side in  $R \setminus Z$ , and “s” for shadowed FDs.

The algorithm uses the following execution strategy: While reading the input dataset, MUDS directly applies the SPIDER algorithm described in Section 2.1 to calculate INDs (one shared I/O operation). Since this algorithm already requires to read and sort all records, MUDS also builds the PLIs in this step. Afterwards, MUDS runs the DUCC algorithm, described in Section 2.2, using the previously calculated PLIs to identify all minimal UCCs. Finally, it executes a novel FD discovery algorithm that is based on the concepts explained in Section 4. The FD discovery takes advantage of the known minimal UCCs (inter-task pruning) and the already calculated PLIs (shared data structures).

The FD discovering part of MUDS consists of three phases: In the first phase, MUDS discovers the FDs among overlapping minimal UCCs (Section 5.1). In the second phase, the algorithm finds FDs with right hand sides in the set of columns that are non-minimal UCCs ( $R \setminus Z$ ) (Section 5.2). In the third phase, MUDS discovers and minimizes the remaining FDs that have a shadowed left hand side (Sec-

tion 5.3). Because the FD validations in the MUDS algorithm perform many superset look-ups to find minimal UCCs for left hand side column combinations, we introduce a prefix tree of UCCs that ensures fast look-up times (Section 5.4).

## 5.1 FDs in connected minimal UCCs

MUDS is a “Unique Column Combinations First” algorithm. So it first discovers all minimal UCCs to use them for the FD discovery. So when starting the FD discovery, all minimal UCCs are already known. Now, the first part of MUDS’ FD discovery analyzes only those FDs whose left and right hand side columns are included in minimal UCCs. For an FD  $X \rightarrow Y$ , we call the complete set of functionally determined attributes  $Y$  the *closure* of  $X$ . Suppose  $U$  is a minimal UCC in the relation  $R$ ; then, the closure of  $U$  is  $R$ , because  $U$  functionally determines all other attributes. It is possible that some of the attributes in the closure of  $U$  are non-minimally determined, which means that they are also determined by a subset of  $U$ . To minimize the left hand sides of the non-minimal FDs, we check whether the direct subsets of  $U$  functionally determine attributes in the closure. In the following, we present an algorithm that minimizes the left hand sides, starting from the minimal UCCs in a top-down manner.

**Recursive minimization.** In order to minimize the left hand sides of the FDs deduced from minimal UCCs, we start from the UCCs themselves and recursively analyze subsets of the UCCs. Algorithm 1 shows the pseudocode for the recursive minimization. MINIMIZEFDS takes two input parameters: the set  $\mathcal{U}$  of all minimal UCCs discovered in the previous step and the union  $Z$  of all attributes that appear in at least one minimal UCC. The algorithm iterates over all minimal UCCs  $U \in \mathcal{U}$ , generates FD candidates from them, and then creates minimization tasks for the generated FDs (lines 2-3). We use tasks and a queue of tasks to avoid recursive method calls, which uses heap memory instead of stack memory.

For every task, we know the valid right hand sides from the parent closure, but the minimality of these FDs is not yet known. To determine the minimal right hand sides, MINIMIZEFDS iterates over the left hand side’s direct subsets and checks which FDs are also valid in the subsets  $U' \subset U$  (line 8). To this end, the algorithm determines the subset’s connector  $C := U \setminus U'$  and performs the connector look-up (lines 9-10). We describe this connector look-up below. Then, MINIMIZEFDS tests the candidate FDs using partition refinement (line 11). If it finds an FD, MINIMIZEFDS removes the corresponding attribute from the closure of the current left hand side, because the left hand side cannot be minimal (line 12). Finally, we create new tasks for all direct subsets with the valid right hand side (line 15). The valid right hand sides contain all attributes that are still functionally determined by the current subset. After checking the FDs in all subsets, the closure of the current left hand side contains only those right hand sides for which the current left hand side is minimal. The MINIMIZEFDS function then outputs these minimal FDs (line 16).

**Connector look-up.** MUDS generates the right hand side candidates using an operation that we call *connector look-up*. The connector look-up ensures that the left and right hand side are subsets of different minimal UCCs and that the minimal UCCs overlap. We already discussed the princi-

---

### Algorithm 1 Calculate FDs from minimal UCCs

---

**Require:** minimal UCCs  $\mathcal{U}$ , union of all minimal UCCs  $Z$

```

1: function MINIMIZEFDS( $\mathcal{U}$ ,  $Z$ )
2:   for  $U \in \mathcal{U}$  do
3:      $tasks.add((lhs \leftarrow U, rhs \leftarrow Z \setminus U, mUcc \leftarrow U))$ 
4:    $FDs \leftarrow \text{new Map}\langle \text{ColumnComb}, \text{ColumnComb} \rangle()$ 
5:   while  $!tasks.isEmpty()$  do
6:      $task \leftarrow tasks.remove()$ 
7:      $currentRhs \leftarrow task.rhs$ 
8:     for  $lhsSubset \in task.lhs.getDirectSubsets()$  do
9:        $connector \leftarrow task.mUcc \setminus lhsSubset$ 
10:       $potentialRhs \leftarrow connectorLookup(connector \setminus$ 
11:         $getImpossibleColumns(lhsSubset))$ 
12:       $validRhs \leftarrow checkFDs(task.lhs, potentialRhs)$ 
13:       $currentRhs \leftarrow currentRhs \setminus validRhs$ 
14:      if  $validRhs.isEmpty()$  then
15:        continue
16:       $tasks.add((lhs \leftarrow lhsSubset, rhs \leftarrow validRhs,$ 
17:         $mUcc \leftarrow task.mUcc))$ 
18:    $FDs[lhs] \leftarrow FDs[lhs] \cup currentRhs$ 
19:   return  $FDs$ 

```

---

ples used for this operation in Section 4.1. We now describe the connector look-up in an example shown in Table 2. To perform the connector look-up for a column combination, MUDS splits the minimal UCCs into a potential left hand side and a connector. In our example, we split the minimal UCC  $AFG$  into the potential left hand side  $A$  and the connector  $FG$ . Then, we use the connector to perform a look-up on the minimal UCCs: All minimal UCCs that are supersets of the connector, are candidates for the right hand side. Table 2 lists all minimal UCCs of our example in the *mUCCs* column. In the *matched* column, Table 2 depicts the retrieved UCCs. The subset of the matching UCCs that is not the connector is printed in bold font. The result of the look-up is the union of these columns; these columns serve as right hand sides for the next FD candidates.

mUCCs	matched	
<u>AFG</u>	<b>AFG</b>	<b>union: ABCDE</b>
<u>BDFG</u>	<b>BDFG</b>	
<u>DEF</u>	-	
<u>CEFG</u>	<b>CEFG</b>	

**Table 2: Connector look-up with connector  $FG$**

After the connector look-up, the algorithm removes all left hand side columns of the new FD candidate from the right hand side columns, because trivial FDs do not need to be checked. So in the example, we would remove  $A$  from the union  $ABCDE$ . Additionally, the algorithm checks whether the union of the left and right hand side is a subset of a minimal UCC; such dependencies cannot exist, because minimal UCCs cannot contain FDs in themselves.

In the following, we describe the minimization of FDs for the example minimal UCC  $AFG$ . Figure 4 depicts the traversed graph. Note that for conciseness, we describe the recursion in only one branch. At first, MUDS initializes the closure of the minimal UCC  $AFG$  to  $R \setminus \{AFG\}$ , which in this case is  $BCDE$ . Then, the algorithm splits  $AFG$  into its direct subsets, i.e., left hand sides for new FD candidates,

and the connector (the connector is shown in round brackets). At node  $AF$ , MUDS performs the connector look-up, which yields the potential right hand sides  $BD$ . By checking the FDs  $AF \rightarrow B$  and  $AF \rightarrow D$  using partition refinement (see Section 2.3), the algorithm finds that both FDs still hold. Therefore, it can remove the FDs' right hand sides from the superset closure in node  $AFG$ , because these cannot be minimal at  $AFG$ . Then, MUDS computes the direct subsets of  $AF$  and again performs the connector look-up. For the column  $A$ , the look-up yields  $BD$ , whose dependencies are then tested. In the test, the algorithm finds only  $A \rightarrow B$  to be valid, which invalidates the minimality of the parent FD  $AF \rightarrow B$ . After completing an entire level, MUDS outputs the previous level's remaining right hand sides as valid minimal FDs. The algorithm terminates when no potential right hand sides remain or the list of direct subsets is empty. Upon termination, the algorithm has discovered and minimized all FDs among connected minimal UCCs.

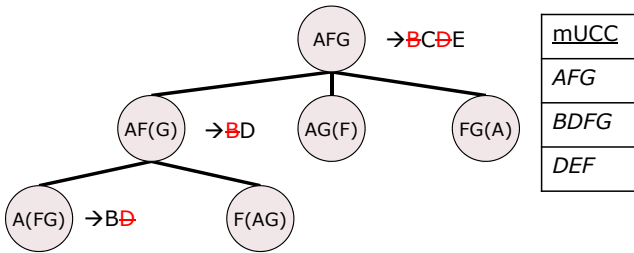


Figure 4: Example for recursive FD minimization

## 5.2 Graph traversing for $R \setminus Z$

This section focusses on those FDs whose right hand side is in  $R \setminus Z$ . The set  $R \setminus Z$  contains all columns that are not element of any minimal UCC. Hence, the previous step did not find these FDs. In contrast to existing FD discovery algorithms, we can restrict possible right hand sides to columns that belong to  $R \setminus Z$ , because the previous step has already found all minimal FDs with right hand side in  $Z$ . Therefore, not all columns in the relation need to be analyzed as potential right hand sides.

As discussed in Section 4.2, MUDS uses one sub-lattice for each potential right hand side to profit from downwards pruning. The algorithm traverses each sub-lattice using a random walk strategy that is based on the traversal strategy of the UCC discovery algorithm DUCC, which we already introduced in Section 2.2: Suppose that we search for FDs with right hand side  $A$ . MUDS first constructs the lattice of left hand side candidates for  $A$  using the attributes  $R \setminus A$ . Then the random walk starts at a random seed node on the first level of this lattice. At each node, we check if the column combination in that node determines  $A$ .

Suppose we are at the node that contains the column combination  $X$ . If  $X$  determines  $A$ , we continue the traversal by choosing a random subset of  $X$ , because the discovered FD  $X \rightarrow A$  might not be minimal. All supersets of  $X$  also determine  $A$ , but are minimal. Therefore, we prune all supersets of  $X$ . If the column combination  $X$  does not determine  $A$ , the next traversal step chooses a random superset of  $X$ . All subsets of  $X$  can then be pruned, because they cannot determine  $A$  (Lemma 4). As noted in Section 2.2, the lattice might contain holes of unvisited nodes, which have to be

discovered in the end. For this task, MUDS uses the same approach as DUCC [10].

The mayor difference of this traversal strategy to DUCC's traversal strategy lies in the check that decides which nodes are traversed next: DUCC checks the uniqueness of the node's column combination by inspecting the cardinality of the node's PLI; MUDS checks whether the node's column combination functionally determines the sub-lattice's right hand side  $A$  by testing for partition refinement as described in Lemma 1 in Section 2.3.

Before MUDS starts the graph traversal for  $R \setminus Z$ , it has already determined several minimal FDs (see Section 5.1). These known FDs are used for additional pruning in the lattice: The combination of a left hand side with its right hand side can never be the left hand side of an already known minimal FD (see Section 2.3). The random walk on a sub-lattice terminates when all candidates are checked or pruned. Then, MUDS continues with the remaining sub-lattices until all FDs with right hand sides in  $R \setminus Z$  are found.

## 5.3 Shadowed FD discovery

The two previously described sub-algorithms of MUDS' FD discovery part do not yet find all FDs. This is due to the fact that some FDs are shadowed (see Section 4.3). In the following, we present an algorithm that finds and minimizes these shadowed FDs by first extending and then minimizing the left hand sides of already discovered FDs.

**Shadowed FD discovery:** Algorithm 2 shows the discovery of the (not necessarily minimal) shadowed FDs. At first, the algorithm calculates the potentially shadowed columns. The columns are shadowed, because they occur in a right hand side of an FD and, thus, are not discovered when the algorithm deduces the FDs from UCCs in a previous step.

To find the missing FDs, the algorithm iterates over all previously discovered FDs in line 2 of Algorithm 2. For each FD, the algorithm iterates over the subsets in line 3 and creates a connector in line 4 that is used for a lookup of potentially shadowed columns in line 5. With these shadowed columns, the algorithm creates a new left hand side from the union of the current left hand side and the shadowed columns in line 6. The FD  $newLhs \rightarrow fd.rhs$  obviously holds, because  $fd.lhs \rightarrow fd.rhs$  holds and  $newLhs = fd.lhs \cup shadowedRhs$ . However, the FD  $newLhs \rightarrow fd.rhs$  is not minimal and FDs that were previously not discovered might be deduced by minimizing the FD  $newLhs \rightarrow fd.rhs$ .  $newLhs$  can contain many columns and be larger than UCCs. Thus, the amount of columns can be reduced prior to the minimization of the FD. This is a powerful pruning step, which is shown in line 7-8 of Algorithm 2. The

---

### Algorithm 2 Discover shadowed FDs

---

**Require:**  $\text{map}(lhs,rhs)$  of minimal  $FDs$ , minimal UCCs  $\mathcal{U}$

- 1: **procedure** DISCOVERSHADOWEDFDs( $FDs, \mathcal{U}$ )
- 2:   **for**  $fd \in FDs$  **do**
- 3:     **for**  $subset \in fd.lhs.getAllSubsets()$  **do**
- 4:        $connector \leftarrow fd.lhs \setminus subset$
- 5:        $shadowedRhs \leftarrow FDs[connector]$
- 6:        $newLhs \leftarrow fd.lhs \cup shadowedRhs$
- 7:       **for**  $reduceLhs \in \text{removeUCCs}(newLhs, \mathcal{U})$  **do**
- 8:          $shadowedTasks.add(reduceLhs, fd.rhs)$
- 9:      $minimizeFDs(shadowedTasks)$

---

---

**Algorithm 3** Remove UCCs from the left hand side

---

**Require:** column combination to minimize  $lhs$ , minimal UCCs  $\mathcal{U}$

```
1: function REMOVEUCCs( $lhs$ ,  $\mathcal{U}$ )
2:    $reducedLhs \leftarrow \{\}$ 
3:    $tasks.add(\langle pos \leftarrow 0, remCol \leftarrow \{\} \rangle)$ 
4:    $subsetUniques \leftarrow \mathcal{U}.getSubsetsOf(lhs)$ 
5:   while  $!tasks.isEmpty()$  do
6:      $task \leftarrow tasks.remove()$ 
7:     if  $task.pos \geq subsetUniques.size()$  then
8:        $reducedLhs.add(lhs \setminus task.remCol)$ 
9:       continue
10:     $unique \leftarrow subsetUniques[task.pos]$ 
11:    if  $(task.remCol \cap unique).isEmpty()$  then
12:       $tasks.add(\langle pos \leftarrow task.pos + 1,$ 
13:                 $remCol \leftarrow task.remCol \rangle)$ 
14:      continue
15:      for  $column \in unique.columns$  do
16:         $tasks.add(\langle pos \leftarrow task.pos + 1,$ 
17:                   $remCol \leftarrow task.remCol \cup column \rangle)$ 
18:    return  $reducedLhs$ 
```

---

method  $removeUCCs()$  compares the FD to the previously discovered UCCs. If  $newLhs$  contains at least a single UCC, a part of the UCC is removed and multiple  $reducedLhs$  are returned that do not contain any UCC. This is possible because no left hand side that contains a UCC can yield a minimal FD. This approach allows the algorithm to skip steps in the minimization of the FD, because the initial left hand sides are already smaller. In line 9, the algorithm minimizes the reduced left hand sides. After this step, all FDs in the dataset are discovered. In the following, we describe the methods  $removeUCCs()$  and  $minimizeFDs()$  in detail.

**Shadowed FD pruning:** As previously described, we are not interested in left hand sides that contain UCCs, because they cannot yield minimal FDs. Algorithm 3 shows the minimization of left hand sides by removing parts of UCCs. The resulting left hand sides do not contain any UCC. For a single non-minimized left hand side multiple reduced left hand sides may be generated.

We use tasks and a queue to avoid recursive method calls that is shown in line 3. Then, the algorithm calculates all UCCs that are contained in the left hand side in line 4. For each task, we save the next UCC that must be removed ( $pos$ ) and the columns that should be removed from the initial left hand side to create a UCC-free left hand side ( $remCol$ ). The algorithm iterates over the tasks in line 5-15. Lines 7-9 show a task that already iterated over all contained UCCs. Thus, the calculation of the reduced left hand side is finished and the reduced left hand side is added to the result in line 8. If at least a single UCC must be removed from the current left hand side, the algorithm obtains this UCC in line 10. Lines 11-13 check if the current UCC is completely removed due to the columns in  $remCol$ . If this is the case, MUDS continues with the next UCC by adding the task with an increased  $pos$  index in line 12. Lines 13-15 show the actual removal of columns. The algorithm iterates over the columns in the UCC in line 14. For each column, a new task is generated in line 15, where the current column is removed. When all tasks are processed, the algorithm returns the reduced left hand sides in line 16.

---

**Algorithm 4** Discover and minimize potential FDs

---

**Require:** shadowed Tasks  $tasks$ ,  $map(lhs, rhs)$  of minimal functional dependencies  $FDs$

```
1: procedure MINIMIZEFDs( $tasks$ ,  $FDs$ )
2:   while  $!tasks.isEmpty()$  do
3:      $task \leftarrow tasks.remove()$ 
4:      $currentRhs = task.rhs$ 
5:     for  $subset \in task.lhs.getDirectSubsets()$  do
6:        $validFD \leftarrow checkFDs(subset, task.rhs)$ 
7:       if  $validFDs.isEmpty()$  then
8:         continue
9:        $currentRhs \leftarrow currentRhs \setminus validFDs$ 
10:       $tasks.add(\langle lhs \leftarrow subset, rhs \leftarrow validFDs \rangle)$ 
11:       $FDs[task.lhs] \leftarrow FDs[task.lhs] \cup currentRhs$ 
```

---

**Shadowed FD minimizing:** MUDS calls Algorithm 4 in Algorithm 2 to minimize FDs. The algorithm also uses a task queue to avoid recursive method calls. For every shadowed FD task containing a potential left and right hand side, MUDS generates all direct left hand side subsets (line 5). These subsets might already determine the right hand side. Afterwards, the algorithm checks, which FDs actually hold on the current subset's left hand side and then removes these dependencies from the superset's right hand side, because they are not minimal (lines 6-9). Then, MUDS creates new tasks for all direct subsets (lines 10). The set of minimal FDs is updated with the known superset's minimal FDs (line 11). When all tasks have been processed, the algorithm terminates. At this stage, all shadowed FDs have been discovered and minimized.

## 5.4 Subset pruning tree

As described in Section 5.3, MUDS also finds shadowed FDs. For each left hand side  $X$ , it needs to look-up all minimal UCCs that are subsets of  $X$ . The naïve implementation of this operation iterates over the list of minimal UCCs and performs a subset check for each minimal UCC. With an increasing number of attributes, the number of minimal UCCs increases as well, which makes this operation increasingly expensive. Therefore, we organize all minimal UCCs in a prefix tree that guarantees an efficient look-up of UCC subsets.

The concept of prefix trees was first presented by De La Briandais [7]. For the construction of our prefix tree, we assume that the columns in all column combinations are sorted (e.g. by their position in the dataset). Figure 5 depicts an exemplary prefix tree for UCCs. Level 1, which is the root node of the prefix tree, stores all columns that occur as the first column in any column combination. For each entry in level 1, a node in level 2 may exist. These nodes store the second columns of column combinations that share the same first column. This pattern continues for the other levels. The last column of a column combination can be identified as an entry that has no child node.

The MUDS algorithm uses the prefix tree to efficiently find subsets of minimal UCCs: Given a column combination  $X$ , we need to find all subsets of  $X$  that are contained in the prefix tree. To find them, we iterate over the sorted columns of  $X$ . For each column in  $X$ , we check whether the column is contained in level 1 of the prefix tree. If the column is not contained, we discard that column and continue with the

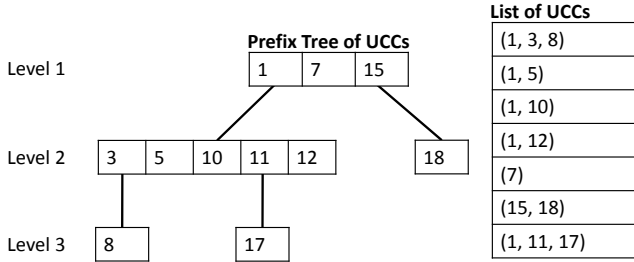


Figure 5: Prefix tree for seven column combinations

next column of  $X$ . If it is contained, we visit the associated node in level 2. There, we perform the same check with the remaining columns of  $X$ . Then, we go to level 3 and so on. Two conditions can stop the search: First, we find a matching node entry that has no associated node in the next level. This means that we found a subset of  $X$ . Second, there are no remaining columns in  $X$ , because we discarded all of them, but we have not reached the end of a path in the prefix tree. Then, the current path does not lead to a valid subset of  $X$ . In both cases, we need to trace back and start over with the remaining columns of  $X$ .

## 6. EVALUATION

We compare MUDS (Sec. 5), HOLISTIC FUN (Sec. 3.2), and the composition of baseline algorithms (Sec. 2) using the Metanome data profiling framework<sup>1</sup>. The framework provides a standardized execution environment for pre-packaged profiling algorithms. In this way, common tasks like file I/O, user interaction, and result handling are decoupled from the algorithms allowing for fair comparisons. Metanome and all algorithms use OpenJDK 1.7 64-Bit. We show that the algorithms have different characteristics and sweet spots, but that MUDS is preferable in general.

**Hardware.** We execute our experiments on a Dell PowerEdge R620 with CentOS 6.4. The machine has 128 GB DDR3 RAM, of which 120 GB are assigned to the Java virtual machine, and two Intel Xeon E5-2650 (2.00 GHz, Octa-Core) CPUs. Because the implementations of our algorithms are all single threaded, only one of the cores is actually used.

**Datasets.** For the row scalability experiment in Section 6.1, we use the Universal Protein Resource<sup>2</sup> (*uniprot*) dataset. It is a public dataset about protein sequences and their functions and contains 539,165 rows and 223 columns. For the column scalability experiment in Section 6.2, we use the *ionosphere* [3] dataset. This dataset contains radar data of the ionosphere and features 351 rows and 34 columns (and many dependencies). Our third dataset, the North Carolina Voter Registration Statistics<sup>3</sup> (*ncvoter*) dataset, contains non-confidential data about 7,503,575 voters from North Carolina and features 94 columns. We use this dataset for the evaluation in Section 6.4. For our experiments described in Section 6.3, we use additional real world datasets from the UCI machine learning repository [3]. The section also compares the performance of MUDS with the

performance of TANE [11], the most popular FD discovery algorithm, to show that MUDS can also compete with non-holistic FD algorithms. Finally, we discuss dataset properties that MUDS’ FD discovery is optimized for in Section 6.5.

### 6.1 Scalability on rows

In the following experiment, we evaluate the scalability of our algorithms MUDS and HOLISTIC FUN with respect to the number of rows in the input dataset and compare them to the baseline algorithm, which is the sequential execution of SPIDER, DUCC, and FUN. Because of its length, the *uniprot* dataset is best suited for row-scalability experiments. Figure 6 depicts the execution times of the different algorithms on *uniprot* with ten columns while varying the dataset’s number of rows.

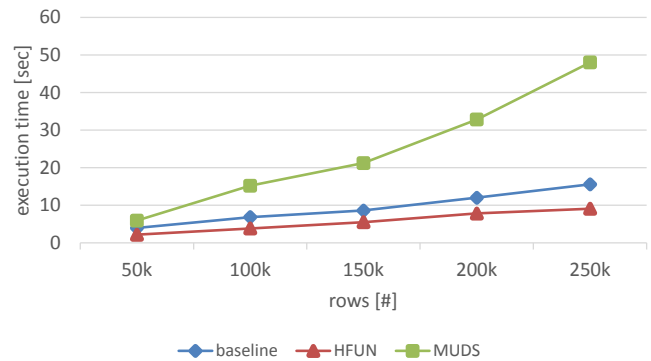


Figure 6: Scalability with regard to the number of rows on the *uniprot* dataset

The measurements show that all three algorithms scale almost linearly with the number of rows. On this particular dataset, HOLISTIC FUN is the fastest algorithm; it is about 1/3 faster than the baseline, because HOLISTIC FUN shares the cost for I/O among the three profiling tasks and discovers UCCs directly with the FDs. MUDS, on the contrary, is the slowest algorithm on *uniprot*, because the discovery of shadowed FDs is particularly expensive on this dataset; if many shadowed FDs are to be discovered, the costs for this step also scale linearly with the number of rows.

### 6.2 Scalability on columns

The following experiment evaluates the algorithms with regard to the number of columns: Using the *ionosphere* dataset, we successively include more and more columns from the original dataset in each run. The *ionosphere* dataset is particularly interesting for column-scalability experiments, because it contains many and large FDs – a challenge for any FD discovery algorithm and a test of its pruning capabilities. The results are shown in Figure 7, which also presents the number of discovered FDs in the dataset.

The experiment shows that the execution times of all three algorithms scale exponentially with the number of columns, which is due to the fact that the search space for UCCs and FDs also grows exponentially. However, MUDS scales clearly better with the number of columns on the *ionosphere* dataset than both HOLISTIC FUN and the baseline, because due to the UCC-first discovery approach MUDS searches a much smaller search space and the number of shadowed FDs is manageable on *ionosphere*. The baseline and HOLISTIC

<sup>1</sup>[www.metanome.de](http://www.metanome.de)

<sup>2</sup>[www.uniprot.org](http://www.uniprot.org) Accessed: 2015-03-03

<sup>3</sup>[ftp://alt.ncsbe.gov/data/](http://ftp://alt.ncsbe.gov/data/) Accessed: 2015-03-03

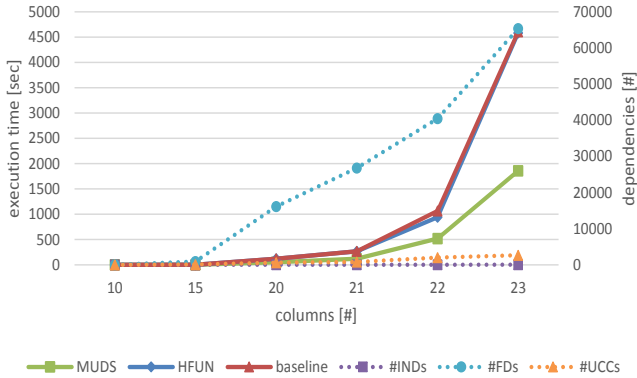


Figure 7: Scalability over the number of columns on the *ionosphere* dataset.

FUN both spend 99% of their runtime on FD discovery for 23 columns, which gives MUDS’ improved FD discovery a significant advantage. For the same reason, HOLISTIC FUN performs only slightly better than the baseline; it optimizes those algorithmic parts that make up only 1% of the overall runtime.

### 6.3 Performance on various Datasets

The scalability experiments evaluated the performance of HOLISTIC FUN and MUDS on the *uniprot* and the *ionosphere* dataset. To verify the observations that we made on these two datasets, we now evaluate the algorithms on various real-world datasets from the UCI machine learning repository [3]. These UCI datasets have been used in most related work benchmarks and thus allow for comparability of results. We also add the TANE algorithm to this comparison in order to investigate how MUDS performs in comparison to state-of-the-art non-holistic FD discovery. Table 3 lists the execution times of the four profiling algorithms.

As in the scalability experiments, HOLISTIC FUN always performs slightly better than the baseline algorithm showing that a holistic approach should always be preferred over the sequential execution of individual profiling algorithms. The experiment also shows that it strongly depends on the properties of the given input dataset whether MUDS or HOLISTIC FUN is the overall best algorithm: For small numbers of columns and higher numbers of rows, HOLISTIC FUN appears to be the faster algorithm and for higher numbers of columns and small numbers of rows, MUDS performs best. The number of columns, thereby, determines the algorithms’ performance much stronger than the number of rows as the *adult* and the *letter* dataset show, on which MUDS is up to factor 48 faster than HOLISTIC FUN.

When analyzing the algorithms’ performance on the different datasets in detail, we find that some minimal FDs in datasets where MUDS is clearly faster than HOLISTIC FUN have very large left hand sides. The HOLISTIC FUN algorithm, then, must traverse many nodes in the lattice, which is expensive. MUDS, in contrast, utilizes the minimal UCCs for pruning and the depth first search strategy in order to traverse a much smaller part of the lattice. This leads to a clear performance advantage. So in general, the performance of HOLISTIC FUN and MUDS depends on the position of the FDs in the lattice. If the minimal FDs have small left hand sides, HOLISTIC FUN is the better algorithm, be-

Dataset	Col	Row	FDs	basel.	HFUN	MUDS	TANE
iris	5	150	4	.1s	.1s	.1s	.6s
balance	5	625	1	.3s	.1s	.1s	.9s
chess	7	28k	1	2.0s	.9s	1.5s	2.0s
abalone	9	4k	137	1.3s	.6s	1.1s	1.0s
nursery	9	12k	1	2.3s	1.9s	3.1s	3.1s
b-cancer	11	699	46	.8s	.6s	.5s	1.4s
bridges	13	108	142	.8s	.7s	.6s	1.3s
echocard	13	132	538	1.0s	.6s	1.6s	.8s
adult	14	48k	78	126s	118s	9.9s	81.2s
letter	17	20k	61	706s	636s	13.2s	326.0s
hepatitis	20	155	8k	462s	450s	88.1s	10.9s

Table 3: Runtime comparison on 11 real world datasets

cause the overhead of finding shadowed FDs in MUDS is unproportionately large. But if there are FDs with large left hand sides in the dataset, MUDS is more efficient. Because the average size of minimal FDs correlates with number of columns, we can choose MUDS or HOLISTIC FUN based on the number of columns. Section 6.5 discusses the difference between HOLISTIC FUN and MUDS in more detail.

When comparing the runtimes of MUDS with the runtimes of TANE, we observe up to 8 times slower runtimes on the *hepatitis* dataset, where many shadowed FDs exist. However, we also see that the holistic algorithm can outperform the non-holistic algorithm by orders of magnitude: It is 8 times faster on the *adult* dataset and 24 times faster on the *letter* dataset. This is possible if MUDS finds favorable pruning conditions, i.e., FDs with large left hand sides. The algorithm can, then, prune much more efficiently than TANE, which makes MUDS also the better FD algorithm here.

### 6.4 Analysis of MUDS’ phases

Our previous experiments identified MUDS’ last phase, namely the discovery of shadowed FDs, to be its most computationally expensive phase. We now analyze the different phases in more detail. Figure 8 depicts the execution time for each phase of the algorithm. We ran this experiment on 20 columns and 10,000 rows of the *ncvoter* dataset.

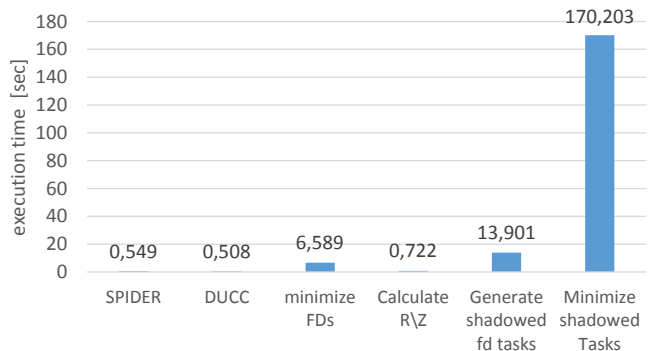


Figure 8: Runtime of Muds’ different phases on the *ncvoter* dataset with 10,000 rows and 20 columns

As the measurement show, the computational effort spent on SPIDER and DUCC is almost negligible. The last two phases that detect shadowed FDs, however, are the performance bottleneck of MUDS in this experiment. We observe a 22 times higher execution time than in the previous phases. The *generate shadowed tasks* phase iterates over the already

found FDs and creates tasks for possibly shadowed FDs. Each task immediately checks if the FDs holds. These FD checks consume 78% of the time in this phase. Because the found FDs are not necessarily minimal, MUDS then minimizes the valid FDs in the *minimize shadowed tasks* phase. The FD checks in this phase take 90% of the phase’s time. If the number of discovered shadowed FDs is high, the minimization is very expensive, because minimizing FDs corresponds to a top-down FD discovery that scales exponentially with the number of columns. In both phases, the primary time-consuming operation is the PLI intersect, which is necessary to check whether an FD holds. However, the execution time distribution between the phases of the MUDS algorithm is dataset specific and especially disadvantageous in this experiment. Nevertheless, we observed that the shadowed FD finding part is the most expensive phase of MUDS on all datasets.

## 6.5 Favorable dataset properties for MUDS

From our previous experiments, we learned that MUDS usually performs best on datasets with ten or more columns. But the number of columns is only one possible indicator for the performance of MUDS over HOLISTIC FUN. In both algorithms, the discovery of FDs is the most expensive part. So to understand the difference between these two approaches better, we have to compare the two FD search strategies in more depth. Depending on the size of these search spaces on the given dataset, one or the other algorithm is superior.

HOLISTIC FUN searches for FDs by starting at the bottom of the lattice and, then, traverses the nodes level-wise until all minimal FDs are found. Thereby, it classifies visited elements into *free sets* and *non-free sets* to prune non-minimal FD candidates. The size of HOLISTIC FUN’s search space is, hence, defined by the height of the lattice levels that hold the minimal FDs; the higher the algorithm has to search, the longer it takes

MUDS first computes all minimal UCCs and, then, searches for FDs in two different search spaces: The first search starts from the minimal UCCs and traverses the lattice in a top-down strategy until all minimal FDs are found (see Section 5.1). All nodes that MUDS traverses in this way are subsets of minimal UCCs and *non-free sets* in the classification of FUN. Because this search space approaches the minimal FDs from the top down, it differs greatly from the search space of HOLISTIC FUN. The second search space of MUDS uses only the columns in  $R \setminus A$  for the construction of sub-lattices that are then traversed bottom-up, depth-first (see Section 5.2). This search space overlaps with HOLISTIC FUN’s search space, but it is much smaller depending on the size of  $R \setminus A$ . Furthermore, MUDS’ depth-first search reaches minimal FDs on high lattice levels much faster than HOLISTIC FUN’s breath-first search.

So MUDS also performs best, if the minimal FDs and minimal UCCs are located on low lattice levels. In comparison to HOLISTIC FUN and any bottom-up, breath-first FD discovery algorithm MUDS performs best when:

1. The minimal UCCs lie close to the minimal FDs in the lattice.
2. The minimal UCCs lie on high lattice levels so that the minimal FDs lie on high lattice levels as well.
3. Many columns participate in at least one minimal UCC, i.e., the column set  $R \setminus A$  is small.

Criterion 1 is intuitively clear, because the size of MUDS’ first search space shrinks with the distance of minimal UCCs and FDs. Criterion 2 seems to be a disadvantage for MUDS, because it increases the size of the second search space; but MUDS’ second search space is (a) much smaller than HOLISTIC FUN’s search space and (b) the depth-first search outperforms the breath-first search if this search depth is large. Criterion 3 is a big advantage for MUDS, because its bottom-down search is much faster than the bottom-up search (see Section 6.4).

So now it becomes clear, why MUDS scales better with the number of columns than HOLISTIC FUN: The average distance between minimal UCCs and minimal FDs grows only slightly with the number of columns so that Criterion 1 always holds; the average height and number of both minimal UCCs and minimal FDs grows constantly so that Criterion 2 becomes increasingly advantageous for MUDS; the chances for columns being part of some minimal UCC with some other columns also increases with the number of columns so that the size of column set  $R \setminus A$  increases only slightly making Criterion 3 hold.

The number of columns is, however, only an indirect indicator for the performance of MUDS. If the Criteria 1 to 3 do not hold in a particular dataset or if the dataset comprises many rows (see Section 6.1), a lot more columns would be needed to make MUDS faster than HOLISTIC FUN. So we could, instead, use the number and size of minimal UCCs to choose the FD discovery strategy: Because MUDS calculates the minimal UCCs before it starts the FD discovery, one could choose MUDS’ FD discovery part if many, large UCCs have been found or the FUN algorithm if few, small UCCs are found. In practice, however, making the decision based on the number of columns is easier and similarly precise, because the properties “many” and “large” depend on the actual number of rows and columns so that they are non-trivial to define.

## 7. RELATED WORK

Many data profiling applications, such as data integration and data exploration, need to calculate various metadata at once, but almost all existing algorithms focus on only one task. We already introduced the most important related algorithms for holistic data profiling in Section 2, so we now give a more comprehensive overview on existing IND, UCC, and FD discovery algorithms:

De Marchi et al. [8] presented one of the first algorithms for IND detection. The algorithm constructs an inverted index upon the values of all attributes to check them for inclusions. This technique has been outperformed by the SPIDER algorithm [4] that discards attributes early on. As SPIDER is the currently most efficient algorithm for IND detection, we integrated it in our holistic approaches.

The algorithms for UCC discovery can be separated in two groups: row-based and column-based techniques. Giannella et al. [9] presented a column-based algorithm that generates relevant column combinations to check their uniqueness. An improved version of this algorithm, HCA [1], uses an optimized candidate generation and additional statistical pruning to find UCCs. In both algorithms the check for uniqueness is costly. GORDIAN [16], in contrast, is a row-based UCC discovery algorithm that organizes the data in a prefix tree to check for UCCs. It traverses the tree to determine maximal non-UCCs, which are then used to cal-

culate the minimal UCCs. This is also costly if the number of maximal non-UCCs is large. The currently most efficient UCC algorithm DUCC [10] implements a combination of row-based and column-based techniques: It builds upon efficient data structures and uses UCCs and non-UCCs simultaneously for pruning. Due to the efficiency of DUCC's random walk strategy, we used it in our holistic algorithms.

While TANE [11], proposed by Huhtala et al., is the most popular FD algorithm, it is often outperformed by FUN [14]. TANE and FUN both use partition refinement to identify FDs and apriori-gen to traverse the search space. As FUN additionally incorporates a cardinality inference method that reduces the necessary partition intersect operations, it needs to traverse an overall smaller number of candidates. The CORDS algorithm by Ilyas et al. [12] is capable of identifying various correlations and *soft* FDs. As the algorithm's identification process builds upon sampling techniques, it only approximates the real result.

Although current profiling algorithms focus on one specific profiling task, there are some algorithms that already leverage the knowledge about one type of metadata to draw conclusions about another. The first work in this context was contributed by Beeri and Bernstein [5] who used the defined FDs in a relational database to find additional keys. The algorithms FUN and HCA both use dependencies between FDs and UCCs for pruning. However, they use discovered FDs to derive UCCs, while our algorithm MUDS uses discovered UCCs to derive FDs.

## 8. CONCLUSION

We investigated the problem of simultaneously discovering three types of metadata: INDs, UCCs, and FDs. By interleaving their calculation, we demonstrated how to share common costs for I/O operations and the traversal of data structures in order to reduce the overall profiling time. Furthermore, we introduced new inter-task pruning and inference rules, in particular rules that derive possible FDs from discovered UCCs. For the analysis of our holistic techniques, we proposed the two algorithms HOLISTIC FUN and MUDS, which jointly discover unary INDs, minimal UCCs, and minimal FDs. HOLISTIC FUN always outperforms the sequential execution of the SPIDER, DUCC, and FUN, the fastest of algorithms for the respective tasks, by avoiding duplicate work. The MUDS algorithm, on the other hand, also facilitates the additional inter-task pruning and inference rules giving it a significant performance advantage on different real world datasets: Our evaluation shows that MUDS is up to 48 times faster than HOLISTIC FUN and the baseline if favorable (and common) pruning conditions (many FDs and UCCs with long left hand sides) are given. On such datasets, MUDS performs even faster than the fastest pure FD algorithm. The number of columns and the size of discovered UCCs can both be used to decide whether MUDS or HOLISTIC FUN should be used.

## 9. REFERENCES

- [1] Z. Abedjan and F. Naumann. Advancing the discovery of unique column combinations. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1565–1570, 2011.
- [2] A. Andreeva, D. Howorth, J.-M. Chandonia, S. E. Brenner, T. J. P. Hubbard, C. Chothia, and A. G. Murzin. Data growth and its impact on the SCOP database: new developments. *Nucleic Acids Research*, 36(1):419–425, 2008.
- [3] K. Bache and M. Lichman. UCI machine learning repository. <http://archive.ics.uci.edu/ml>, 2013. Accessed: 2015-03-03.
- [4] J. Bauckmann, U. Leser, and F. Naumann. Efficiently computing inclusion dependencies for schema discovery. In *ICDE Workshops*, page 2, 2006.
- [5] C. Beeri and P. A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems (TODS)*, 4(1):30–59, 1979.
- [6] E. F. Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971.
- [7] R. De La Briandais. File searching using variable length keys. In *Proceedings of the ACM Western Joint Computer Conference*, pages 295–298, 1959.
- [8] F. De Marchi, S. Lopes, and J.-M. Petit. Efficient algorithms for mining inclusion dependencies. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 464–476, 2002.
- [9] C. Giannella and C. Wyss. Finding minimal keys in a relation instance. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.7086&rep=rep1&type=pdf>, 1999. Accessed: 2015-03-03.
- [10] A. Heise, J.-A. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 301–312, 2013.
- [11] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.
- [12] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 647–658, 2004.
- [13] F. Naumann. Data profiling revisited. *SIGMOD Record*, 32(4):40–49, 2013.
- [14] N. Novelli and R. Cicchetti. FUN: An efficient algorithm for mining functional and embedded dependencies. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 189–203, 2001.
- [15] H. Saiedian and T. Spencer. An efficient algorithm to compute the candidate keys of a relational database schema. *The Computer Journal*, 39(2):124–132, 1996.
- [16] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. Gordian: efficient and scalable discovery of composite keys. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 691–702, 2006.



# Monitoring MaxRS in Spatial Data Streams

Daichi Amagata

Department of Multimedia Engineering Graduate  
School of Information Science and Technology  
Osaka University

amagata.daichi@ist.osaka-u.ac.jp

Takahiro Hara

Department of Multimedia Engineering Graduate  
School of Information Science and Technology  
Osaka University

hara@ist.osaka-u.ac.jp

## ABSTRACT

Due to the increase of GPS enabled devices and a lot of location-based services, spatial objects are continuously generated. This paper addresses a problem of monitoring MaxRS (Maximizing Range Sum) in spatial data streams. Given a set of weighted spatial (2-dimensional) objects, this problem is to monitor a location of a given user-specified sized rectangle where the sum of the weights of the objects covered by the rectangle is maximized. Many real life applications obtain a benefit from monitoring MaxRS, e.g., traffic analysis and event detection in urban sensing, but this problem has not yet been addressed so far. Although some algorithms for static objects have been proposed, executing such an algorithm whenever new objects are generated is computationally expensive. These motivate us to develop an efficient algorithm that can monitor MaxRS efficiently. In this paper, we first design a basic algorithm that is based on an index framework and incrementally updates the result. We then enhance the algorithm and show that the enhanced algorithm can deal with error-guaranteed approximation and monitoring top-k MaxRS. Our experimental results confirm the efficiency of our approach.

## 1. INTRODUCTION

Due to the increase of GPS enabled devices such as smartphones and tablet machines, a lot of location-based services are used in many real life applications. From this fact, spatio-temporal databases have been receiving significant attention recently. Supporting spatial queries is therefore becoming more important, and many studies developed techniques for efficient spatial query processing [13, 15]. These studies can be classified into spatial objects retrieval problems [3, 24] and location finding problems [10, 32, 33]. For example,  $k$ NN query processing, which retrieves the  $k$  nearest neighbor objects w.r.t. a given query point, is the representative of the spatial objects retrieval problems. In the location finding problems, on the other hand, some queries have been proposed, for example, optimal location queries [10, 33], bichromatic reverse nearest neighbor queries [14, 35], and MaxRS (Maximizing Range Sum) queries [6, 8, 9, 25]. This paper focuses on a kind of MaxRS problem.

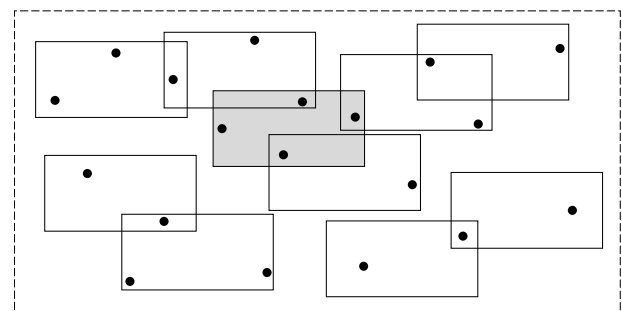


Figure 1: An example of a MaxRS query

**Motivation.** Given a set of weighted spatial (2-dimensional) objects and a user-specified sized rectangle, a MaxRS query finds a location of the rectangle where the sum of the weights of the objects covered by the rectangle is maximized.

EXAMPLE 1.1. *Figure 1 shows an example of a MaxRS query. In Figure 1, the rectangle with dashed line, the black points, and the rectangles with solid line show a general monitoring space, spatial objects with weight 1, and user-specified sized rectangles, respectively. The MaxRS query identifies the location of the shaded rectangle as one of the optimal results because it covers the largest number of objects, i.e., the sum of the weights of objects covered by the rectangle is the maximum.*

This query is useful because it can automatically find an important place without specifying any query points [8, 9]. In this paper, we address a novel problem of monitoring MaxRS in spatial data streams. In other words, we address a problem of continuous MaxRS query processing over sliding-window. Since a large amount of spatio-temporal objects are *continuously generated* [2, 5], e.g., in the context of location-based service usages, one-time finding an important location does not make sense but continuously monitoring an important location is required. A continuous MaxRS query can achieve this and has the following real life applications.

EXAMPLE 1.2. *Consider urban sensing. A system, e.g., base-station, continuously collects spatio-temporal objects, e.g., generated by devices with GPS, in an urban city. Such objects can be represented as  $\langle x, y, w \rangle$  where  $x$ ,  $y$ , and  $w$  are latitude, longitude, and weight, respectively. If  $w$  is communication traffic, a continuous MaxRS query can monitor an area where traffic is concentrated. In this case, the system can notify the users holding mobile devices in the area of warning about communication delay.*

Example 1.2 shows that continuous MaxRS queries can help to analyze communication errors and also support decision making, e.g., where to place Wi-Fi access points. We next consider location-

based games in which continuous MaxRS queries are useful for decision making.

EXAMPLE 1.3. *In BotFighters, players try not to be attacked by other players [4], and in Ingress, players try to occupy places. Let  $w$  in Example 1.2 represent the strength or level of a given player, and players keep checking the area monitored by a continuous MaxRS query. This can detect an event, e.g., a competition by many and/or high-level players, and support to plan a strategy.*

Other examples include mobile sensor networks [27] that a base-station continuously collects sensor readings. By employing continuous MaxRS queries, the base-station may be able to detect sensor failures and over-concentration of mobile sensor nodes.

**Technical challenge and overview.** Some techniques for exact MaxRS query processing on static objects have been developed in the past. [12, 18] proposed in-memory algorithms while [8, 9] proposed an external-memory algorithm. The computation (I/O) complexity of the in-memory (external-memory) algorithm is shown to be optimal. These algorithms however focus on *one-time* computation, so they are not efficient for continuous MaxRS queries. This is because computing the result from scratch whenever new objects are generated is obviously computationally expensive, meaning that an approach which can incrementally update the result is required.

In this paper, we first propose a basic algorithm that exploits a graph in grid index, namely G2. G2 integrates graph and grid structures, thus its storage cost is  $\mathcal{O}(|V| + |E|)$ , where  $V$  corresponds to a set of objects on a given sliding-window and  $E$  is a set of edges. (The details are described in Section 4). One of the properties of G2 is that no overhead incurs when objects expire. We then enhance both the basic algorithm and G2, and propose aggregate G2, aG2, and a branch-and-bound algorithm that exploits aG2. This algorithm eliminates unnecessary update computation as much as possible and accelerates the query processing efficiency.

Interestingly, the branch-and-bound algorithm can deal with error-guaranteed approximation. Let  $w^*$  and  $\epsilon$  respectively be the maximum range sum and a user-tolerance error rate. Also, let  $w$  be the weight of the area monitored by the algorithm, and we can guarantee that  $w \geq (1 - \epsilon)w^*$ . We show that the relationship between query processing efficiency and  $\epsilon$  is trade-off but the practical error rate is much less than  $\epsilon$ . In addition to the approximation, we consider a problem of monitoring top-k MaxRS. For example, Examples 1.2 and 1.3 may require not only a single area but  $k$  (e.g., 5) areas with the largest range sum. This requirement is satisfied by a simple modification of the branch-and-bound algorithm.

**Contributions and organization.** We summarize our contributions as follows.

- We address a novel problem of continuous MaxRS query processing in spatial data streams (Section 2). To the best of our knowledge, we are the first to investigate this problem.
- We design a basic algorithm for a continuous MaxRS query (Section 4). This algorithm incrementally updates the result by exploiting an efficient index framework.
- We enhance the basic algorithm and propose a more efficient index and branch-and-bound algorithm (Section 5). This algorithm prunes unnecessary computation and accelerates the computation efficiency.
- We show that the branch-and-bound algorithm can deal with error-guaranteed approximation and efficient continuous top-k MaxRS query processing (Section 6).

- We conduct extensive experiments using both synthetic and real datasets that confirm the efficiency of our approach (Section 7).

In addition to the above contents, we review some related literatures in Section 3, and Section 8 concludes this paper.

## 2. PRELIMINARY

We are given a set of spatial stream objects in a general monitoring space, as shown in Figure 1. A spatial object  $o_i$  is represented by  $o_i = \langle x, y, w \rangle$  where  $i$  is its identifier,  $\langle x, y \rangle$  shows the location that  $o_i$  is generated, and  $w$  is a non-negative value (weight), i.e.,  $o_i.w \in \mathbb{R}^+$ . We assume a spatial stream environment, thus consider a sliding-window model [1] because it is usual that many applications are interested only in recent objects. Count- and time-based sliding-window models are widely accepted. The count-based sliding-window considers the most recent  $n$  objects, and in this model,  $m$  new objects generations lead to the expirations of the  $m$  oldest objects. The time-based sliding-window, on the other hand, considers the objects generated within the last  $T$  time-units. Selection of a suitable sliding-window model depends on applications, and our algorithms can deal with both the models. So, without loss of generality, we assume the count-based sliding-window in this paper. Let  $O$  be the set of the most recent  $n$  objects, and  $O$  residents in-memory because real-time continuous query processing generally requires main memory computation [4, 17].

Let  $P$  be an infinite set of points in the general monitoring space. Given a user-specified sized rectangle  $r$ , the weight of a point  $p \in P$ ,  $p.w$ , is defined by

$$p.w = \sum o_i.w,$$

where  $o_i \in O$  is covered by  $r$  centered at  $p$ . We are now ready to formally define the monitoring MaxRS problem.

DEFINITION 1 (MONITORING MAXRS PROBLEM). *Given a set of objects on a sliding-window  $O$ , an infinite set of points in the general monitoring space, and a user-specified sized rectangle  $r$ , the goal of the monitoring MaxRS problem is to continuously monitor a location  $p^* \in P$  that satisfies.*

$$p^* = \operatorname{argmax}_{p \in P} p.w.$$

It is infeasible to find and monitor  $p^*$  from such infinite points, but as the literatures [8, 9, 25] introduce, MaxRS problems can be solved by transformation to an alternative problem. Given a user-specified sized rectangle  $r$ , let  $r_i$  be the weighted rectangle of the same size of  $r$  centered at  $\langle o_i.x, o_i.y \rangle$  ( $o_i \in O$  and  $r_i.w = o_i.w$ ). Consider  $o_i, o_j \in O$ , and if  $r_i$  overlaps with  $r_j$ , it is not difficult to see that the weight of the overlapped space  $s$  is  $s.w = r_i.w + r_j.w$ . We formally define this *space weight*.

DEFINITION 2 (SPACE WEIGHT [8]). *Given  $O$ , we have a set of rectangles each of which is centered at the location of the corresponding object. The weight of a space  $s$  is the sum of the weights of the rectangles covering  $s$ .*

Then the alternative problem is to find  $s$  with the maximum weight denoted by  $s^*$ . Interestingly but not surprisingly,  $p^*$  exists in  $s^*$ , that is, any points in  $s^*$  can be  $p^*$ .

EXAMPLE 2.1. *Figure 2 shows the alternative problem of Figure 1. The center of each rectangle is at the corresponding object. Note that the size of each rectangle is the user-specified size. Recall that the weight of each object is 1, and the shaded overlapped space has*

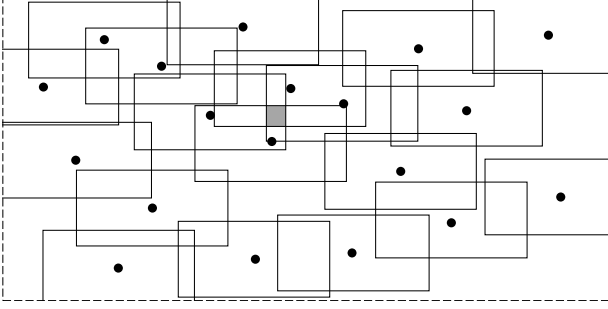


Figure 2: An example of the alternative problem of Figure 1

the maximum weight. We know that the weights of other overlapped spaces are less than 4. The center of the shaded rectangle in Figure 1 exists in the shaded space.

Therefore, monitoring  $s^*$  is equivalent to monitoring  $p^*$ . We here define continuous MaxRS queries that solve the monitoring MaxRS problem.

**DEFINITION 3 (CONTINUOUS MAXRS QUERY).** Given a set of objects on a sliding-window  $O$  and a user-specified sized rectangle  $r$ , each weighted object is converted to a weighted rectangle centered at the location of the object. A continuous MaxRS query monitors  $s^*$  that satisfies

$$s^* = \operatorname{argmax}_{s \in S} s.w, \quad (1)$$

where  $S$  is the set of overlapped spaces.

We also consider a problem of monitoring top-k MaxRS to support the applications that require not only a single space but also  $k$  spaces with the maximum weight. The definition of monitoring top-k MaxRS problem is given by extending Definition 1, thus we formally define continuous top-k MaxRS queries below.

**DEFINITION 4 (CONTINUOUS TOP-K MAXRS QUERY).** Given a set of objects on a sliding-window  $O$  and a user-specified sized rectangle  $r$ , each weighted object is converted to a weighted rectangle centered at the location of the object. A continuous top-k MaxRS query monitors the set  $S^*$  that satisfies  $|S^*| = k$  and  $\forall s \in S^*$  and  $\exists s' \in S \setminus S^*$ ,  $s.w \geq s'.w$  where  $S$  is the set of overlapped spaces, and ties are arbitrarily broken.

Since applications that execute continuous queries basically require real-time monitoring [3, 4, 17], algorithms that process such queries have to update query results efficiently. Our objective of this paper is therefore to minimize computation time to update  $s^*$ , which is incurred by generations and expirations of objects. Table 1 summarizes the symbols frequently used in this paper.

### 3. RELATED WORK

As introduced in Section 1, many spatial queries have been developed. This section reviews related works of the monitoring MaxRS problem. In particular, we introduce spatial preference queries and facility location queries in which MaxRS queries are categorized. We then review the existing studies of MaxRS query processing.

Before introducing the above queries, we should make it clear that MaxRS queries are different from range aggregation queries. The goal of range aggregation queries is to return the aggregate result (e.g., values and points) from (i) the set of points in a given rectangle with fixed location [19, 21, 23] or (ii) the set of values in a given interval [26]. On the other hand, the goal of MaxRS queries is

Table 1: Overview of symbols

Symbol	Description
$o_i$	The weighted spatial object with identifier $i$
$m$	The number of objects generated at the same time
$s^*$	The space with the maximum weight
$r_i$	The weighted rectangle centered at the location of $o_i$
$r_i.E$	The set of edges held by $r_i$
$N(r_i)$	The set of neighboring vertices (rectangles) of $r_i$
$s_i$	The space with the maximum weight covered by $r_i$
$c_{i,j}$	The cell with identifier $(i, j)$
$G_{i,j}$	The graph maintained by $c_{i,j}$
$V_{i,j}$	The set of vertices (rectangles) of $G_{i,j}$
$c_{i,j}.w$	The upper-bound weight of $c_{i,j}$

to find a location from an infinite set of points. In the same sense, continuous spatial queries [15, 16, 17] that monitor not locations but objects are different from our problem.

**Spatial preference queries.** Spatial preference query processing problem is one of location selection problems. Given a 2-dimensional point  $p$  and a distance constraint  $d$ , a top-k spatial preference query [22, 29] determines the score of  $p$  by the sum of the weights of the feature objects existing within  $d$  from  $p$ . [28] studies a problem of finding top- $t$  most influential sites. The influential score of a given site is defined as the sum of weights of its reverse nearest neighbor objects. In the above queries, the scores of the target points are defined by a kind of range sum function. However, the locations of the target points are already known, which is different from our problem.

**Facility location queries.** The objective of this problem is to find an optimal location w.r.t. a given condition. It is often the case that a set of customers (or clients) with weights and a set of facilities are given [11]. Then a facility location query retrieves a facility that maximizes the total weight of its reverse ( $k$ ) nearest neighbor customers [35]. [10] proposed optimal location queries, and the extension version of the optimal location queries, min-dist optimal location queries, has been studied in [20, 32]. Such optimal location queries have also been studied in road networks [7].

The monitoring MaxRS problem is also one of the facility location problems. We are interested in continuously monitoring a location that maximizes the total weight of objects covered by a user-specified sized rectangle. As introduced in Section 1, this is useful in monitoring applications in spatial data streams. The main difference between MaxRS and the above queries is their categories: *monotonic* (e.g., MaxRS queries) and *bichromatic* (e.g., optimal location queries).

**MaxRS queries.** To the best of our knowledge, the existing works of MaxRS problems and the variants are [6, 8, 9, 12, 18, 25]. An external-memory algorithm for exact MaxRS queries has been proposed in [8, 9], while [25] proposed a randomized sampling algorithm that bounds the error with high probability. That is, given a tolerance error  $\epsilon$ , the approximate algorithm returns a space with the weight  $w$  that satisfies  $w \geq (1 - \epsilon)w^*$  with probability  $1 - \frac{1}{n}$ . ( $w^*$  is the weight of the optimal result and  $n$  is the number of objects.) Rotating MaxRS queries have been proposed in [6]. This literature assumes that a given rectangle is rotatable, but we do not assume this case and assume the usual case, as well as [8, 9, 12, 18, 25].

We now focus on the in-memory algorithms [12, 18] because we also consider an in-memory algorithm for continuous MaxRS queries. It is notable that the algorithm in [18] solves a max-

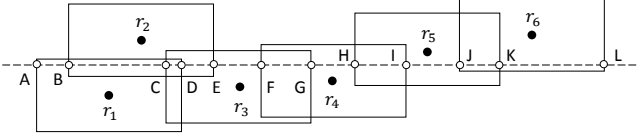


Figure 3: An example of processing a plane-sweep algorithm

enclosing rectangle problem, to which the MaxRS problem can be converted, based on well-known *plane-sweep* strategy, and actually the external-memory and the approximate algorithms [8, 25] also employ this strategy. (The algorithm proposed in [12] also employs the same strategy and the computation cost is the same as [18].) The plane-sweep algorithm [18] employs a horizontal line, and given a set of rectangles, this line is swept from bottom to top of the rectangles while counting the weights of intersecting intervals on the sweeping line. Figure 3 shows an instance when sweeping the dashed line. Let the weight of each rectangle in Figure 3 be 1, and we can see that the weights of intervals AB, BC, and CD are 1, 2, and 3, respectively. In the plane-sweep algorithm, when a sweeping line reaches the bottom edge of a rectangle, a newly generated interval is inserted into a binary-tree, while when reaching the top of the rectangle, the expired interval is deleted from the binary-tree. During these procedures, the counts of intervals are also updated. This algorithm returns the interval with the maximum weight, and the complexity of this algorithm is  $\mathcal{O}(n \log n)$  [18], where  $n$  is the number of objects (rectangles). This is because the algorithm executes  $2n$  binary-tree updates ( $n$  insertions and  $n$  deletions) that take  $\mathcal{O}(\log n)$  time. (This algorithm also can return a set of  $k$  intervals with the maximum weight without sacrificing its computation efficiency.)

The above algorithm is shown to be an optimal solution for the MaxRS problem on *static objects*. However, it is inefficient to compute  $s^*$  from scratch by this algorithm even in the case where the number of newly generated objects is not large, which is shown by our experimental results. From the next section, we describe our solutions that efficiently update and monitor the result over stream data.

## 4. BASIC SOLUTION

The generations and the expirations of objects lead to additions and eliminations of overlapped spaces. This suggests that  $S$  in Equation (1) is dynamic, in other words, the ranking of range sum is dynamic, thereby efficient  $s^*$  monitoring is not trivial. In spite of this nature, our index framework enables to design a simple but efficient algorithm.

### 4.1 G2: Graph in Grid Index

We first present our index framework Graph in Grid index (or G2). Recall that our objective is to achieve real-time monitoring of MaxRS in a stream environment, meaning that unnecessary computations have to be avoided. A dynamic graph, where a vertex is a rectangle and an edge shows the overlap between given two vertices, realizes this. If the weight of each vertex (rectangle) is 1, we can see that  $s^*$  is a part of the vertex with the largest number of edges. To monitor  $s^*$  with using the dynamic graph, the basic operation that we have to do is just to check the updated parts of the dynamic graph. This is because only the updated parts may affect  $s^*$ . Motivated by this observation, we develop G2.

As described, the first idea is to maintain  $n$  rectangles on a sliding-window by a graph. Vertices of the graph are the rectangles, and if two rectangles overlap each other, there is an edge between them.

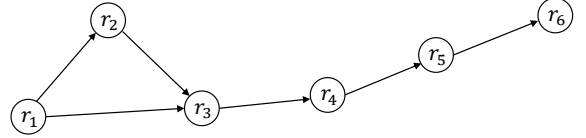


Figure 4: The graph constructed from the rectangles in Figure 3

Table 2: Edge and neighbor sets of the vertices of the graph in Figure 4

Vertex $r_i$	Edge $r_i.E$	Neighbor set $N(r_i)$
$r_1$	$(r_1, r_2), (r_1, r_3)$	$\{r_2, r_3\}$
$r_2$	$(r_2, r_3)$	$\{r_3\}$
$r_3$	$(r_3, r_4)$	$\{r_4\}$
$r_4$	$(r_4, r_5)$	$\{r_5\}$
$r_5$	$(r_5, r_6)$	$\{r_6\}$
$r_6$	$\emptyset$	$\emptyset$

We define our graph and provide a concrete example below.

**DEFINITION 5 (GRAPH  $G$ ).**  $G = (V, E)$  is a dynamic graph.  $V$  is a set of rectangles of a given size on a sliding-window, and  $E$  is a set of direct edges. Given two vertices  $r_i, r_j \in V$  overlapping each other, where  $r_i$  was generated earlier than  $r_j$ , there is a direct edge between them,  $(r_i, r_j)$ , and this edge is held by the vertex (rectangle) that was generated earlier than the other, i.e.,  $r_i$ . (Ties are broken by identifiers.)

**EXAMPLE 4.1.** We construct a graph  $G$  from the set of rectangles in Figure 3, and Figure 4 shows  $G$ . Assume that the rectangles are generated in order of identifiers. Since  $r_2$  overlaps with  $r_1$ , there is  $(r_1, r_2)$  that represents a direct edge from  $r_1$  to  $r_2$ .  $r_1$  maintains the edges  $(r_1, r_2)$  and  $(r_1, r_3)$ .

When context is clear, we use vertices and rectangles interchangeably since they are the same, as Definition 5 describes. We denote  $r_i.E$  the set of edges held by the vertex  $r_i \in V$ . The set of neighbors of a vertex  $r_i \in V$  is also denoted by  $N(r_i) = \{\forall r_j \mid \exists (r_i, r_j) \in r_i.E\}$ . Table 2 shows an example that uses Figure 4 in terms of  $r.E$  and  $N(r)$ . Then we obtain the spaces on  $r_i$  covered by the rectangles in  $N(r_i)$ .  $r_i$  maintains  $s_i$ , the space with the maximum weight among the spaces. That is,  $s_i$  is definitely the subspace on  $r_i$ , which provides the following property.

**PROPERTY 1.** Given  $r_i, r_j \in V$ , we have that  $s_i \neq s_j$ .

The proof is straightforward since  $\nexists r_i \in N(r_j)$  if  $\exists r_j \in N(r_i)$ . How to obtain  $s_i$  is explained later, but it is not difficult to see the following property.

**PROPERTY 2.** Let  $S_G$  be the set of  $s_i$  maintained by  $r_i$  on a sliding-window, we have that

$$s^* = \operatorname{argmax}_{s_i \in S_G} s_i.w. \quad (2)$$

Furthermore, even if some vertices (rectangles) expire, other vertices need no maintenances, because the corresponding edges are held by older vertices.

**PROPERTY 3.** Given  $r_i \in V$ , and when the older vertices than  $r_i$  expire,  $s_i$  does not change.

We next have to consider the case where  $m$  new rectangles (objects) are generated. Due to the graph structure, we need to check whether the  $m$  rectangles overlap with the existing rectangles. It is obvious that simple computation takes  $\mathcal{O}(mn)$  time. To alleviate this, we employ a grid structure like the one shown in Figure 5,

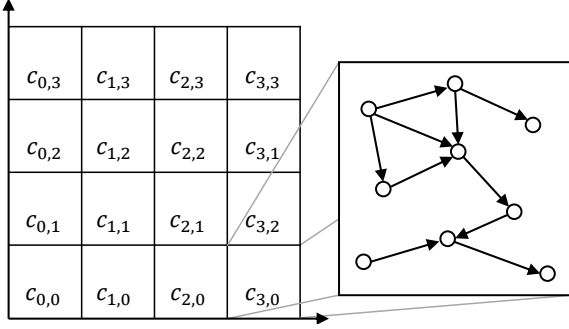


Figure 5: An example of G2

which is the second idea. When dataset updates frequently occur, grid structure is more suitable than complex structures like R-tree and Quad-tree [4]. Each cell in the grid is assigned an identifier and its size is fixed, as shown in Figure 5. Note that each cell  $c_{i,j}$  maintains the graph constructed by the mapped rectangles, denoted by  $G_{i,j} = (V_{i,j}, E_{i,j})$ . Therefore, Definition 5 is rewritten as follows.

**DEFINITION 6 (GRAPH  $G_{i,j}$ ).**  $G_{i,j} = (V_{i,j}, E_{i,j})$  is a dynamic graph maintained by  $c_{i,j}$  of a grid.  $V_{i,j}$  is a set of given sized rectangles that are mapped to  $c_{i,j}$  and on a sliding-window, and  $E_{i,j}$  is a set of direct edges. The condition of direct edges follows Definition 5.

For example, Figure 5 illustrates an example of a G2 and  $G_{3,0}$ . When  $m$  new rectangles are generated, we map them to the cells with which they overlap. (So, a new rectangle may be mapped to multiple cells.) We then update the graphs in the cells where new objects are mapped.

**Complexities.** The time complexity of the rectangle mapping is  $\mathcal{O}(m)$ . Let  $c'$  and  $m'$  respectively be the number of cells where new objects are mapped and the average number of new objects mapped to the cells. Also, let  $n'$  be the average number of vertices in the cells where new objects are mapped, and the time complexity of the graph update is  $\mathcal{O}(c'm'n')$ . In practice, we have that  $n' \ll n$ . We next consider storage cost. Let  $V$  be the set of all vertices in the grid, then we know that  $|V| = \sum |V_{i,j}|$ . Similarly,  $|E| = \sum |E_{i,j}|$ , where  $E$  is the set of all edges in the grid. Recall that each vertex  $r_i$  maintains  $s_i$  and its storage cost is  $\mathcal{O}(|V|)$ . We can therefore conclude that the storage cost of G2 is  $\mathcal{O}(|V| + |E|)$ .

## 4.2 Monitoring Algorithm with G2

We design an online monitoring algorithm using G2, and Algorithm 1 illustrates the high level algorithm.

**Algorithm description.** Consider an instance when  $m$  new rectangles are generated and the  $m$  oldest rectangles expire. As described in Section 4.1, we first update G2 (lines 1–3). Lines 4–6 are designed based on the following idea. Given the set  $S$  of the spaces each of which (i.e.,  $s_i$ ) is maintained by  $r_i \in V$ , we know that  $s_i$  has to be correct due to Equation (2). In addition, if a new edge is inserted to  $r_i.E$ ,  $s_i$  may change because  $N(r_i)$  varies. We therefore need to compute  $s_i$  if  $r_i.E$  is updated. The plane-sweep algorithm is an optimal solution to find the space with the maximum weight covered by the given rectangles [12]. Hence, we employ the plane-sweep algorithm to compute  $s_i$  (line 6). Note that the input of this plane-sweep algorithm is *only*  $N(r_i) \cup \{r_i\}$ . To summarize, for  $\forall r_i \in V$ , where  $r_i.E$  has new edges and  $V$  is the vertex set maintained by a given cell  $c$  of G2, Algorithm 1 executes the plane-sweep algorithm locally, denoted by Local-Plane-Sweep( $\cdot$ ) (lines 4–6). After that, we can correctly monitor  $s^*$ .

---

### Algorithm 1: Monitoring algorithm using G2

---

```

1 Mapping( $R$ ) //  $R$  is the set of new rectangles
2  $C' \leftarrow$  the set of the cells where new objects are mapped
3 G2-Update( $C'$ ) // rectangle overlap computation
4 for  $\forall c \in C'$  do
5   for  $\forall r_i \in c.V$  where  $r$  has new edges do
6      $s_i \leftarrow$  Local-Plane-Sweep( $N(r_i) \cup \{r_i\}$ )
7  $s^* \leftarrow \operatorname{argmax}_{s_i \in S} s_i.w$ 
8 return  $s^*$ 

```

---

Recall that given a set of rectangles, the plane-sweep algorithm sweeps a horizontal line from the bottom to the top among the rectangles. To obtain  $s_i$ , however, we only need to sweep the horizontal line from the bottom to the top of  $r_i$ . Local-Plane-Sweep( $\cdot$ ) in Algorithm 1 (line 6) is optimized to do so. Although the time complexity does not vary, the practical execution time is reduced.

The following simple example highlights the efficiency of our incremental approach.

**EXAMPLE 4.2.** Assume that the graph in Figure 4 is the graph maintained by one of the cells in Figure 5. Assume further that  $m = 1$  and  $r_6$  is the new rectangle mapped to the cell. Algorithm 1 checks the vertices overlapping with  $r_6$ , and in this case,  $(r_5, r_6)$  is inserted to  $r_5.E$ . Algorithm 1 next executes Local-Plane-Sweep( $\{r_5, r_6\}$ ) and obtains  $s_5$ . If  $s_{5.w} > s^*.w$ ,  $s^*$  is replaced by  $s_5$ .

**Time complexity.** As discussed before, lines 1–3 take  $\mathcal{O}(c'm'n')$  time. Let  $v$  be the number of vertices to which new edges are inserted. Also, let  $e$  be the average number of edges of the above vertices, then lines 4–6 take  $\mathcal{O}(ve \log e)$  time since Local-Plane-Sweep( $\cdot$ ) for a vertex takes  $\mathcal{O}(e \log e)$  time. Therefore, when  $m$  new objects are generated, Algorithm 1 takes  $\mathcal{O}(c'm'n' + ve \log e)$  time.

## 5. ENHANCED SOLUTION

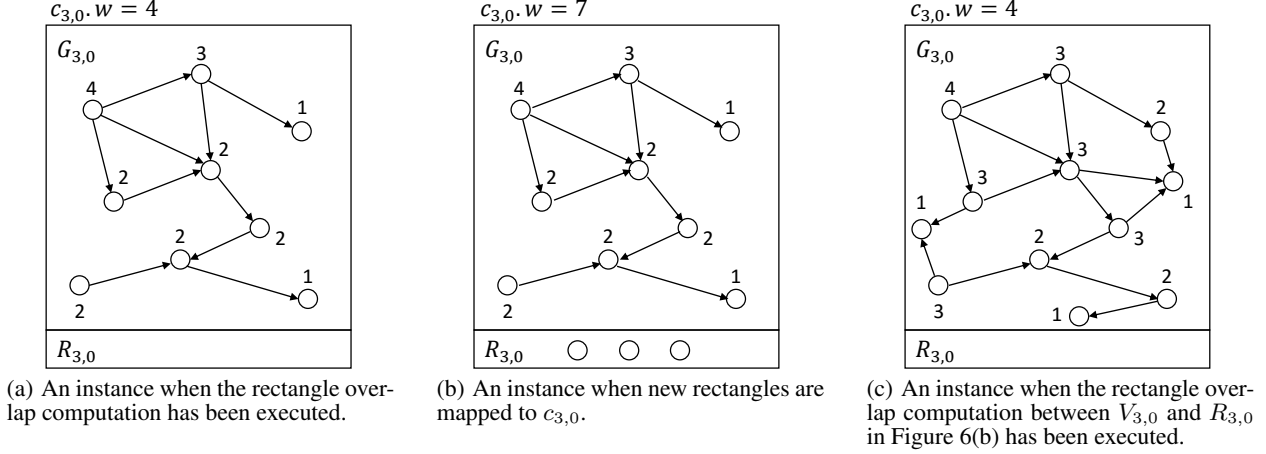
Algorithm 1 can identify *where to update* and compute  $s_i$  maintained by  $r_i$  efficiently since  $N(r_i)$  is easily obtained by the graph representation. In fact, however, the most time consuming operation in Algorithm 1 is Local-Plane-Sweep( $\cdot$ ) (line 6). Algorithm 1 executes Local-Plane-Sweep( $\cdot$ ) whenever  $r_i.E$  is updated, thus it is intuitively seen that the approach degrades the performance in the case where the number of  $r.E$  updates is large. We observe the following usual cases that motivate to enhance G2.

1. The  $r_i.E$  update increases  $s_i.w$  but it is less than  $s^*.w$ .
2. The  $r_i.E$  update does not increase  $s_i.w$ .

To consider a more concrete situation, we give Example 5.1, and Table 3 shows the weights of each vertex  $r_i$  and  $s_i$  of the graph in Figure 4.

Table 3: Weights of vertex  $r_i$  and  $s_i$  of the graph in Figure 4

Vertex $r_i$	$r_i.w$	$s_i.w$
$r_1$	10	55
$r_2$	30	45
$r_3$	15	40
$r_4$	25	45
$r_5$	20	25
$r_6$	5	5



**Figure 6: An example of  $G_{3,0}$  (followed by Figure 5) and  $R_{3,0}$  of  $c_{3,0}$  in an aG2 and its dynamic update where the weight of each vertex is 1.**

EXAMPLE 5.1. Assume the same situation as Example 4.2 and  $s^* = s_1$ . Before  $(r_5, r_6)$  is inserted to  $r_5.E$ ,  $s_5.w = 20$ , and after the insertion, we obtain  $s_5.w = 25$ . However, it is obvious that the edge  $(r_5, r_6)$  insertion to  $r_5.E$  does not affect  $s^*$ , which corresponds to the case 1. In addition, even if  $r_6$  overlaps with  $r_2$  and does not overlap with  $r_3$ ,  $s_2$  keeps the same and does not become  $s^*$ . This corresponds to the case 2.

The observation in Example 5.1 suggests that we may not have to compute  $s_i$  even when  $r_i.E$  is updated. Therefore, by enhancing G2, we aim at eliminating such unnecessary computation and improving query processing efficiency. The enhanced solution achieves this by an upper-bounding technique.

## 5.1 Aggregate G2

It has been shown in the past that data structures considering aggregate results work well for query processing which deal with aggregate functions such as `sum` and `count` [30]. The monitoring MaxRS problem considers `sum` function, then we know that G2 can be extended to deal with aggregate values like aR-tree [19].

We propose aG2 (aggregate G2), which is essentially G2. The main difference between G2 and aG2 is that aG2 employs *upper-bound weights*. Given a graph in an aG2, each vertex  $r_i$  of the graph maintains  $s_i.\bar{w}$ , which is the upper-bound weight of  $s_i$ . How to compute  $s_i.\bar{w}$  depends on algorithms, so we briefly introduce our approach to compute  $s_i.\bar{w}$  here (the detail is described in Section 5.2). Given the graph  $G_{i,j} = (V_{i,j}, E_{i,j})$  maintained by a cell  $c_{i,j}$  in an aG2, we have a vertex  $r_{i'} \in V_{i,j}$  and  $s_{i'}$ . Consider rectangles  $r_{j'}$  where  $(r_{i'}, r_{j'})$  is newly inserted to  $r_{i'}.E$ ,  $s_{i'}.\bar{w}$  is computed by the following Equation.

$$s_{i'}.\bar{w} = s_{i'}.w + \sum r_{j'}.w \quad (3)$$

Not only the vertices but also the cells in aG2 maintain upper-bound weights. Before we introduce the upper-bound weight maintained by  $c_{i,j}$ , we have to note that in aG2,  $c_{i,j}$  maintains

- $G_{i,j}$ : the graph defined by Definition 6, and
- $R_{i,j}$ : a set of rectangles that have been mapped to  $c_{i,j}$  but have not yet been checked whether they overlap with vertices in  $V_{i,j}$  or not (denoted by rectangle overlap computation).

The new rectangles mapped to  $c_{i,j}$  are initially maintained in  $R_{i,j}$ . When we execute the rectangle overlap computation, the rectangles in  $R_{i,j}$  are moved to  $V_{i,j}$ . Now we introduce how to compute

$c_{i,j}.w$ , i.e., the upper-bound weight maintained by  $c_{i,j}$ . Basically,  $c_{i,j}.w$  is set as follows.

$$c_{i,j}.w = \max_{s_{i'} \in S_{i,j}} s_{i'}.\bar{w}, \quad (4)$$

where  $S_{i,j}$  is the set of  $s_{i'}$  maintained by  $r_{i'} \in V_{i,j}$ . Given a set of new rectangles  $r'$  that are mapped to  $c_{i,j}$ ,  $c_{i,j}.w$  is updated by the following equation.

$$c_{i,j}.w \leftarrow c_{i,j}.w + \sum r'.w \quad (5)$$

We give a simple example of how to dynamically update upper-bound weights below.

EXAMPLE 5.2. Figure 6 shows an example of  $G_{3,0}$  and  $R_{3,0}$  where  $c_{3,0}$  follows Figure 5. We first assume the situation of Figure 6(a), and note that the value shown next to each vertex is the upper-bound weight. Then we see that  $c_{3,0}.w = 4$ , which is the maximum value among the upper-bound weights of  $V_{3,0}$ . Next we assume that three new rectangles are mapped to  $c_{3,0}$ , which are maintained in  $R_{3,0}$ , as shown in Figure 6(b). From Equation (5),  $c_{3,0}.w$  is updated to 7. We finally assume Figure 6(c), where the three rectangles in  $R_{3,0}$  have been checked whether they overlap with the vertices (rectangles) in  $V_{3,0}$  or not. Note that the upper-bound weight maintained by each vertex is updated and  $c_{3,0}.w$  is also updated to 4 as the result of the rectangle overlap computation.

From Equations (3)–(5), we have the following property.

PROPERTY 4.  $c_{i,j}.w \geq s_{i'}.\bar{w} \geq s_{i'}.w$  for  $\forall r_{i'} \in V_{i,j}$ .

Because of generations and expirations of rectangles, those upper-bound weights may vary dynamically, but our branch-and-bound algorithm (introduced later) dynamically updates the upper-bound weights so that Property 4 is kept. At the same time, this property provides the correctness of our branch-and-bound algorithm.

As well as G2, we discuss about the storage cost of an aG2 below.

PROPERTY 5. An aG2 has the same storage cost as a G2, i.e.,  $\mathcal{O}(|V| + |E|)$  where  $V$  and  $E$  are respectively the sets of all the vertices and the edges in the (a)G2.

PROOF. To prove Property 5, we have to discuss about the storage costs of vertices, edges, and upper-bound weights. Let  $n_{i,j}$  and  $n'_{i,j}$  be the size of  $V_{i,j}$  in G2 and aG2, respectively. Because  $V_{i,j} \cap R_{i,j} = \emptyset$  in aG2, we have that  $n_{i,j} = n'_{i,j} + |R_{i,j}|$ , thereby  $|V| = \sum (n'_{i,j} + |R_{i,j}|)$ . The number of edges in aG2 may be less than that of G2 but it takes  $\mathcal{O}(E)$  cost. Let  $V'$  be the set of  $V_{i,j}$  of

aG2, and  $|V| > |V'| = \sum n'_{i,j}$ . The storage cost of upper-bound weights maintained by vertices is  $\mathcal{O}(|V'|)$ . Let  $C$  be the set of cells of an aG2, and the storage cost of upper-bound weights maintained by the cells is  $\mathcal{O}(|C|)$ . We know that  $|C| \ll |V'|$  in practice. Therefore we can complete the proof.  $\square$

## 5.2 Branch-and-Bound Algorithm

As described in Section 5.1, each cell and vertex in an aG2 maintain the upper-bound weights. This enables us to process a continuous MaxRS query while pruning unnecessary computations. Specifically, we can obtain two pruning rules, which can reduce the number of executions of Local-Plane-Sweep( $\cdot$ ). Assume an instance when  $m$  new rectangles are generated, the upper-bound weight of each cell is obtained by Equation (5). Given  $s^*$ , we first obtain the following pruning rule.

**PRUNING RULE 1.** *Given a cell  $c_{i,j}$  in the aG2, and if  $c_{i,j}.w < s^*.w$ , all the vertices in  $V_{i,j}$  do not have  $s^*$ . Therefore we do not need to compute the exact  $s_{i'}$  of  $r_{i'} \in V_{i,j}$ .*

The above case efficiently prunes the computation of the exact  $s_{i'}$ . However, we are likely to hold the case where  $c_{i,j}.w \geq s^*.w$ , since  $c_{i,j}.w$  is the maximum value among the set of the upper-bounds maintained by vertices in  $G_{i,j}$  or more (See Equation(5)). In this case, we focus on each vertex in  $V_{i,j}$ , and then apply the next pruning rule.

**PRUNING RULE 2.** *Given a cell  $c_{i,j}$  in the aG2, and we assume that  $c_{i,j}.w \geq s^*.w$ . Given a vertex  $r_{i'} \in V_{i,j}$ , if  $s_{i'}.w < s^*.w$ ,  $r_{i'}$  does not have  $s^*$ , thus we do not need to compute the exact  $s_{i'}$ .*

From the above pruning rules, we can focus only on cells and vertices with non-zero probability to have  $s^*$ . Note that the above pruning rules assume that  $s^*$  is given, although  $s^*$  might expire. If  $s^*$  expires, we first obtain a temporal  $s^*$ , and the temporal  $s^*$  is retrieved from the cell  $c$  that satisfies

$$c = \operatorname{argmax}_{c_{i,j} \in C} c_{i,j}.w, \quad (6)$$

where  $C$  is the set of cells in the aG2. To keep the efficiency of the pruning rules, the weight maintained by the temporal  $s^*$  should be large as much as possible. It is intuitive that a cell with large upper-bound weight probably has the space with large weight, thereby we employ this heuristic. From the above discussion, we design a branch-and-bound algorithm, which efficiently updates  $s^*$  and is illustrated in Algorithm 2.

**Algorithm description.** Consider an instance when  $m$  new rectangles are generated and the  $m$  oldest rectangles expire. We first map the newly generated rectangles to the corresponding cells  $c_{i,j}$  while updating the upper-bound weight  $c_{i,j}.w$  and  $R_{i,j}$  (lines 1–5). Next, we update (or find a temporal)  $s^*$  to enhance the pruning efficiency (line 6–10). Let  $c$  is the cell holding  $s^*$  (if  $s^*$  expires, we retrieve  $c$  that satisfies Equation (6)), we execute  $\text{OverlapComputation}(c)$  (line 9), which is illustrated in Algorithm 3.  $\text{OverlapComputation}(c)$  updates the graph in  $c$  and  $c.w$ . More specifically, we check whether rectangles in  $R$  of the cell  $c$  overlap with the rectangles (vertices) in  $V$ , and if overlap, new edges are inserted while updating the upper-bound weights maintained by the vertices and  $c$  (lines 3–8 in Algorithm 3). After that,  $\text{ExactWeightComputation}(s^*, c)$  is executed (line 10 in Algorithm 2), which is illustrated in Algorithm 4. In  $\text{ExactWeightComputation}(s^*, c)$ , we apply Pruning rule 2 to each vertex  $r_i$ . If  $s_i.w > s^*.w$ , we execute  $\text{Local-Plane-Sweep}(\cdot)$  for  $r_i$ , and update  $s^*$  if necessary (lines 6–10 in Algorithm 4). Also,  $c.w$  is kept so that it satisfies Equation (4). From the above procedures, we obtain the updated (or temporal)  $s^*$ , and then a branch-

---

### Algorithm 2: Branch-and-bound algorithm using aG2

---

```

1  $R_{new} \leftarrow$  the set of newly generated rectangles
2 for  $\forall r \in R_{new}$  do
3   if  $r$  is mapped to  $c_{i,j}$  then
4      $c_{i,j}.w \leftarrow c_{i,j}.w + r.w$ 
5      $R_{i,j} \leftarrow R_{i,j} \cup \{r\}$ 
6  $c \leftarrow \{c_{i,j} \mid s^* \text{ is in } c_{i,j}\}$ 
7 if  $s^*$  expired ( $c = \emptyset$ ) then
8    $c \leftarrow \operatorname{argmax}_{c_{i,j} \in C} c_{i,j}.w$  //  $C$  is the set of cells in aG2
9  $\text{OverlapComputation}(c)$ 
10  $s^* \leftarrow \text{ExactWeightComputation}(s^*, c)$ 
11 for  $\forall c_{i,j} \in C \setminus \{c\}$  do
12   if  $c_{i,j}.w > s^*.w$  then
13      $\text{OverlapComputation}(c_{i,j})$ 
14   if  $c_{i,j}.w > s^*.w$  then
15      $s^* \leftarrow \text{ExactWeightComputation}(s^*, c_{i,j})$ 
16 return  $s^*$ 

```

---



---

### Algorithm 3: $\text{OverlapComputation}(c_{i,j})$

---

```

Input:  $c_{i,j}$  // a cell in aG2
1  $c_{i,j}.w \leftarrow 0$ 
2 for  $\forall r' \in R_{i,j}$  do
3   for  $\forall r \in V_{i,j}$  do
4     if  $r'$  overlaps with  $r$  then
5        $r.E \leftarrow r.E \cup \{(r, r')\}$ 
6        $s.w \leftarrow s.w + r'.w$ 
7     if  $c_{i,j}.w < s.w$  then
8        $c_{i,j}.w \leftarrow s.w$ 
9   if  $c_{i,j}.w < r'.w$  then
10     $c_{i,j}.w \leftarrow r'.w$ 
11     $s'.w \leftarrow r'.w$ 
12     $V_{i,j} \leftarrow V_{i,j} \cup \{r'\}$ 
13     $R_{i,j} \leftarrow R_{i,j} \setminus \{r'\}$ 

```

---

and-bound approach is employed (lines 11–15 in Algorithm 2) to guarantee the correct  $s^*$ .

Given a cell  $c_{i,j}$ , we first apply Pruning rule 1, and if  $c_{i,j}.w > s^*.w$ , we update (i.e., decrease)  $c_{i,j}.w$  by  $\text{OverlapComputation}(c_{i,j})$  (lines 12–13). Again we apply Pruning rule 1 to  $c_{i,j}$ , and if  $c_{i,j}.w > s^*.w$  again,  $\text{ExactWeightComputation}(s^*, c_{i,j})$ , which we explained above, is executed. In the case where we compute the exact  $s_{i'}$  maintained by  $r_{i'} \in V_{i,j}$ , in  $\text{ExactWeightComputation}(s^*, c_{i,j})$ , and  $s_{i'}.w > s^*.w$ ,  $s^*$  is updated (line 10 in Algorithm 4). These operations are executed for  $\forall c_{i,j} \in C \setminus \{c\}$ , and after that, we can keep monitoring the correct  $s^*$ .

**Time complexity.**  $\text{OverlapComputation}(c_{i,j})$  takes  $\mathcal{O}(|V_{i,j}| |R_{i,j}|)$ . Let  $C'$  be the set of the cells that  $\text{OverlapComputation}(\cdot)$  is executed, and the amortized time to compute the rectangle overlapping is  $\mathcal{O}(|C'| |V_{i,j}| |R_{i,j}|)$ . Let  $v'$  be the number of the vertices that  $\text{Local-Plane-Sweep}(\cdot)$  is executed. Also let  $e'$  be the average number of edges of the above vertices, and the total cost of  $\text{ExactWeightComputation}(\cdot, \cdot)$  is  $\mathcal{O}(v' e' \log e')$ . When  $m$  new objects are generated, Algorithm 2 takes  $\mathcal{O}(|C'| |V_{i,j}| |R_{i,j}| + v' e' \log e')$  (amortized) time. Recall the time complexity of Algorithm 1, and note that  $|C'| |V_{i,j}| |R_{i,j}| \leq c' m' n'$  and  $v' e' \log e' < v e \log e$ .

**Correctness.** The correctness of Algorithm 2 is proven by Property 4. In Algorithms 2–4, we define the condition that we cannot prune  $\text{OverlapComputation}(\cdot)$  and  $\text{ExactWeightComputation}(\cdot, \cdot)$  as “>” instead of “ $\geq$ ,” e.g., line 12 of Algorithm 2. This is because

---

**Algorithm 4:** ExactWeightComputation( $s^*, c_{i,j}$ )

---

**Input:**  $s^*, c_{i,j}$  //  $c_{i,j}$  is a cell in aG2

```
1  $c_{i,j}.w \leftarrow 0$ 
2 for  $\forall r_{i'} \in V_{i,j}$  do
3    $\rho \leftarrow 0$ 
4   if  $s^* \neq \emptyset$  then
5      $\rho \leftarrow s^*.w$ 
6   if  $s_{i'}.w > \rho$  then
7      $s_{i'}.w \leftarrow \text{Local-Plane-Sweep}(N(r_{i'} \cup \{r_{i'}\}))$ 
8      $s_{i'}.w \leftarrow s_{i'}.w$ 
9     if  $s_{i'}.w > s^*.w$  then
10       $s^* \leftarrow s_{i'}$ 
11   if  $c_{i,j}.w < s_{i'}.w$  then
12      $c_{i,j}.w \leftarrow s_{i'}.w$ 
13 return  $s^*$ 
```

---

we monitor one of the spaces with the maximum weight, thus if  $s^*.w = c_{i,j}.w$  for example, we keep monitoring  $s^*$ , and this does not sacrifice the correctness. If applications require all spaces with the maximum weight like the AllMaxRS problem [9], we just need to define the condition as “ $\geq$ .”

### 5.3 Discussion

To demonstrate the efficiency and practicality of Algorithm 2, we review the following conceivable approaches that can tight the upper-bound weights maintained by rectangles more than Algorithm 2.

1. An approach that all rectangles  $r_i$  maintain *all* the overlapped spaces on  $r_i$ .
2. An additional approach of Algorithm 2 that computes the maximum space weight among the common spaces on  $r_i$  and  $r_j$  when  $(r_j, r_i)$  is inserted to  $r_i.E$ .
3. An additional approach of Algorithm 2 that tries to decrease  $s_i.w$  when  $s_i.w > s^*.w$ .

We show that these approaches do not guarantee the reduction of time complexity, rather, the worst-case time complexity becomes worse than Algorithm 2.

**Approach 1.** Given a rectangle  $r_i$ , we assume that  $S_i$  is the set of the overlapped spaces on  $r_i$ . Since  $r_i$  maintains all the overlapped spaces on  $r_i$ , we can compute the tightest  $s_i.w$ . However, given  $m'$  new rectangles overlapping with  $r_i$ , we need at least  $\mathcal{O}(m' \log |S_i|)$  time in practice<sup>1</sup> to compute the tightest  $s_i.w$ . Because  $|S_i| \geq |r_i.E|$ , we may have that  $\mathcal{O}(m' \log |S_i|)$  is larger than the time cost of the plane-sweep algorithm, i.e.,  $\mathcal{O}(|r_i.E| \log |r_i.E|)$ . To bound the worst time complexity, we execute Local-Plane-Sweep( $\cdot$ ), if  $\mathcal{O}(m' \log |S_i|) > \mathcal{O}(|r_i.E| \log |r_i.E|)$  (if we can estimate this situation). Then the worst-case time complexity is the same as Algorithm 1, thus is worse than Algorithm 2. Note that this approach is impractical because we cannot bound the number of spaces maintained by rectangles.

**Approach 2.** Recall that an optimal way to compute the maximum space weight is the plane-sweep algorithm, thereby this approach is equivalent to the exact  $s_i$  computation. That is, this approach does not make sense.

**Approach 3.** Given a vertex  $r_i$  in an aG2, and when  $s_i.w > s^*.w$ , this approach tries to decrease  $s_i.w$ . This approach is illustrated

<sup>1</sup>In the case where  $S_i$  is indexed by an R-tree or a Quad-tree.

---

**Algorithm 5:** UpperboundUpdate( $r_i$ )

---

**Input:**  $r_i$  // a vertex of a given cell.

```
1  $R(r_i) \leftarrow$  the set of vertices that are included in  $N(r_i)$  but have not
   been executed Plane-Sweep( $\cdot$ )
2  $\tau \leftarrow s_i.w$ 
3 for  $\forall r \in R(r_i)$  do
4   if  $r$  overlaps with  $s_i$  then
5      $\tau \leftarrow \tau + r.w$ 
6     if  $\tau > s^*.w$  then
7        $s_i.w \leftarrow \tau$ 
8       break
9   else
10     $\rho \leftarrow r.w + r_i.w$ 
11    for  $r' \in N(r_i) \setminus \{r\}$  do
12      if  $r$  overlaps with  $r'$  then
13         $\rho \leftarrow \rho + r'.w$ 
14      if  $\tau < \rho$  then
15         $\tau \leftarrow \min(\tau + r.w, \rho)$ 
16        if  $\tau > s^*.w$  then
17           $s_i.w \leftarrow \tau$ 
18          break
```

---

by Algorithm 5, which may be executed after line 6 of Algorithm 4. Assume an instance when  $s_i$  is computed, and let  $R(r_i)$  be the set of vertices that are included in  $N(r_i)$  after the instance. In other words,  $s_i$  has been computed based on  $N(r_i) \setminus R(r_i)$ . In a nut shell, this approach computes  $s_i.w$  by checking whether a given  $r \in R(r_i)$  overlaps with  $s_i$  (line 4) or the rectangle  $r' \in N(r) \setminus \{r\}$  (line 12). Due to the latter case, this approach does not necessarily add  $r.w$  to  $s_i.w$ , which may result in less  $s_i.w$  than the original Algorithm 4. This approach however needs an extra  $\mathcal{O}(|R(r_i)| |N(r_i)|)$  time for each  $r_i$  where  $s_i.w > s^*.w$ . Recall that Local-Plane-Sweep( $N(r_i) \cup \{r_i\}$ ) takes  $\mathcal{O}(|N(r_i)| \log |N(r_i)|)$  time, and we may have that  $\mathcal{O}(|R(r_i)| |N(r_i)|) > \mathcal{O}(|N(r_i)| \log |N(r_i)|)$ . It would be better to execute Algorithm 5 only in the case where  $|R(r_i)| |N(r_i)| < 2|N(r_i)| \log |N(r_i)|$  (since the plane-sweep algorithm needs  $2n \log n$  operations). We can therefore see that this approach does not work well if  $|R(r_i)|$  is large. Also, this approach increases the worst-case time complexity, which means no guarantee to reduce the time complexity of Algorithm 2. Our experimental results also show that this approach does not guarantee the acceleration of query processing.

From the above discussion, we see that more storage and non-reasonable computational costs are required to tight the upper-bound weights. The upper-bounding cost of Algorithm 2 is reasonable and the worst-case time complexity is better than the above approaches.

## 6. APPLICATION TO RELATED PROBLEMS

In this section, we address two problems of approximate monitoring MaxRS and monitoring top-k MaxRS. We solve these problems efficiently by employing the branch-and-bound algorithm using aG2 with simple extensions.

### 6.1 Approximate Monitoring MaxRS

To improve query processing efficiency, some applications require not the exact but approximate results [25]. In this case, it is important to bound the error rate, thus, given a user-tolerance error  $\epsilon$  ( $0 \leq \epsilon < 1$ ), the objective of this problem is to continuously monitor a space  $s$  with the weight  $s.w$  that satisfies

$$s.w \geq (1 - \epsilon)s^*.w.$$



Algorithm 2 can deal with this problem by respectively replacing Pruning rules 1 and 2 with Pruning rules 3 and 4, which are shown below.

**PRUNING RULE 3.** *Given a cell  $c_{i,j}$  in the aG2 and the space  $s$  monitored by our approximate algorithm, if  $(1 - \epsilon)c_{i,j}.w < s.w$ , we do not compute the exact  $s_{i'}$  of  $r_{i'} \in V_{i,j}$ .*

**PRUNING RULE 4.** *Given a cell  $c_{i,j}$  in the aG2 and the space  $s$  monitored by our approximate algorithm, if  $(1 - \epsilon)c_{i,j}.w \geq s.w$ , we cannot prune  $\text{OverlapComputation}(c_{i,j})$  by Pruning rule 3. Given a vertex  $r_{i'} \in V_{i,j}$ , if  $(1 - \epsilon)s_{i'}.w < s.w$ , we do not compute the exact  $s_{i'}$ .*

We demonstrate that our approximate algorithm (which is the approximate version of Algorithm 2) guarantees the error bound.

**THEOREM 1.** *Our approximate algorithm always guarantees that  $s.w \geq (1 - \epsilon)s^*.w$ .*

To prove Theorem 2, we need to introduce the following lemmas.

**LEMMA 1.** *Assume that we are monitoring  $s$  that satisfies  $s.w \geq (1 - \epsilon)s^*.w$  before applying Pruning rule 3 to a cell  $c_{i,j}$ . It is guaranteed that Pruning rule 3 does not lose that  $s.w \geq (1 - \epsilon)s^*.w$ .*

**PROOF.** If  $c_{i,j}.w < s^*.w$  and  $(1 - \epsilon)c_{i,j}.w < s.w$ , it is trivial that  $(1 - \epsilon)s^*.w \leq s.w$  due to the assumption. On the other hand, if  $c_{i,j}.w \geq s^*.w$ , we have that  $(1 - \epsilon)c_{i,j}.w \geq (1 - \epsilon)s^*.w$ . Therefore if  $s.w > (1 - \epsilon)c_{i,j}.w$ ,  $s.w > (1 - \epsilon)s^*.w$ .  $\square$

**LEMMA 2.** *Assume that we are monitoring  $s$  that satisfies  $s.w \geq (1 - \epsilon)s^*.w$  before applying Pruning rule 4 to a vertex  $r_{i'}$ . It is guaranteed that Pruning rule 4 does not lose that  $s.w \geq (1 - \epsilon)s^*.w$ .*

**PROOF.** Essentially the same as the proof of Lemma 1.  $\square$

Now, we are ready to prove Theorem 1.

**PROOF.** As long as we are monitoring  $s$  that satisfies  $s.w \geq (1 - \epsilon)s^*.w$ , Pruning rules 3 and 4 do not lose the bound, which can be seen from Lemmas 1 and 2. We here assume that we are monitoring  $s$  that satisfies  $s.w < (1 - \epsilon)s^*.w$ . Let  $c^*$  be the cell holding  $s^*$ , and we have that  $s^*.w \leq s^*.w \leq c^*.w$ . Therefore, in this case, we definitely cannot prune  $\text{OverlapComputation}(c^*)$  and  $\text{ExactWeightComputation}(s^*, c^*)$  by Pruning rules 3 and 4, and  $s$  is replaced by  $s^*$ . This means that  $s$  cannot be the result, thus we conclude that Theorem 2 is true from the contradiction.  $\square$

## 6.2 Monitoring Top-k MaxRS

If the requirement is to monitor not only a single space but multiple spaces with the largest weight, a continuous top-k MaxRS query is a promising solution. This query achieves the requirement while controlling the result size as Definition 4 describes.

We know that Pruning rules 1 and 2 are based on a threshold, and the threshold is  $s^*.w$ , i.e., the (temporal) top-1 weight. It is intuitively known that the threshold is set as the  $k$ th largest weight in continuous top-k MaxRS queries.

Algorithm 6 illustrates the high level algorithm. Although the algorithm for continuous top-k MaxRS queries is essentially the same as Algorithm 2, the main modification is to deal with the set  $S^*$  of the  $k$  spaces with the maximum weight. We do not show  $\text{OverlapComputation}(C')$  and  $\text{ExactWeightComputation}(S^*, C')$  because they are also essentially the same as Algorithms 3 and 4, respectively.

## 7. EXPERIMENTS

**Algorithm 6:** Branch-and-bound algorithm using aG2 for continuous top-k MaxRS queries

```

1 Execute lines 1–5 in Algorithm 2
2  $C' \leftarrow \{\forall c_{i,j} \mid \exists s \in S^* \text{ is in } c_{i,j}\}$ 
3 if  $C' = \emptyset$  (all spaces in  $S^*$  expire) then
4    $C' \leftarrow \underset{c_{i,j} \in C}{\text{argmax}} c_{i,j}.w$  //  $C$  is the set of cells in aG2
5  $\text{OverlapComputation}(C')$ 
6  $S^* \leftarrow \text{ExactWeightComputation}(S^*, C')$ 
7 for  $\forall c_{i,j} \in C \setminus C'$  do
8   Execute lines 12–15 in Algorithm 2
9 return  $S^*$ 

```

**Table 4: Configuration of parameters**

Parameter	Values
Window-size, $n$ [ $\times 1000$ ]	100, 250, <b>500</b> , 750, 1000
Generation rate, $m$	50, <b>100</b> , 200, 500, 1000
Side length of a rectangle, $l$	100, 500, <b>1000</b> , 1500, 2000
Error rate, $\epsilon$	0, 0.1, 0.2, 0.3, 0.4, 0.5
$k$	1, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50

This section provides our experimental results on the performances of our algorithms. Recall that this is the first work of monitoring MaxRS problem, so there is no existing algorithm that can deal with this problem. Therefore, to show the efficiency of the incremental approach of our algorithms, we use an algorithm with non-incremental approach as comparison. That is, we evaluated naive plane-sweep [12, 18], the algorithm using G2 (Section 4), and the branch-and-bound algorithms using aG2 (Sections 5 and 6). For easy recognition, our algorithms are represented by G2 and aG2. Recall that naive plane-sweep is an optimal in-memory algorithm for computing  $s^*$  from scratch, and the algorithm proposed in [8, 9] also uses naive plane-sweep in the case where all objects fit in the main memory. All algorithms were implemented in C++, and all experiments were conducted on a PC with 3.4GHz Intel Core i7 processor and 32GB RAM.

### 7.1 Setting

**Datasets.** We used a synthetic dataset and three real datasets. In the synthetic dataset, we generated objects under uniform distribution. The cardinality of this synthetic dataset is 10,000,000, and the range of each coordinate is  $[0, 1000000]$ . The three real datasets are T-Drive [31], Geolife [34], and Roma<sup>2</sup>, which are the sets of continuously generated GPS data. The objects in the real datasets exist over a very wide range, thus we selected the objects existing around respective main areas. The cardinalities of T-Drive, Geolife, and Roma are 5,037,794, 3,662,876, and 8,368,858, respectively. The objects in the datasets are sorted in order of generation time, and we normalized the range of each coordinate to  $[0, 1000000]$ . In the above four datasets, the weight of a given object is a real-value randomly chosen from  $[0, 1000]$ .

**Parameters.** Table 4 summarizes the parameters used in the experiments and bold values are default values. Note that a given rectangle is a square in the experiments, thus the size of a rectangle is  $l \times l$ , i.e.,  $1000 \times 1000$  by default.

**Evaluation.** In Section 7.2, we investigate the impact of Algorithm 5. In Section 7.3, to investigate the performances of the algorithms w.r.t. monitoring MaxRS, we varied three parameters,  $n$ ,  $m$ , and  $l$ ,

<sup>2</sup><http://crawdad.org/index.html>

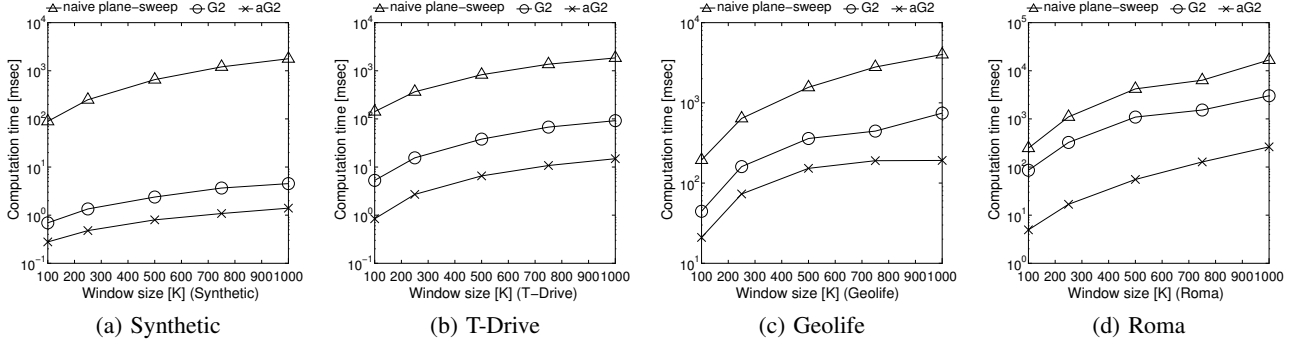


Figure 7: Impact of  $n$

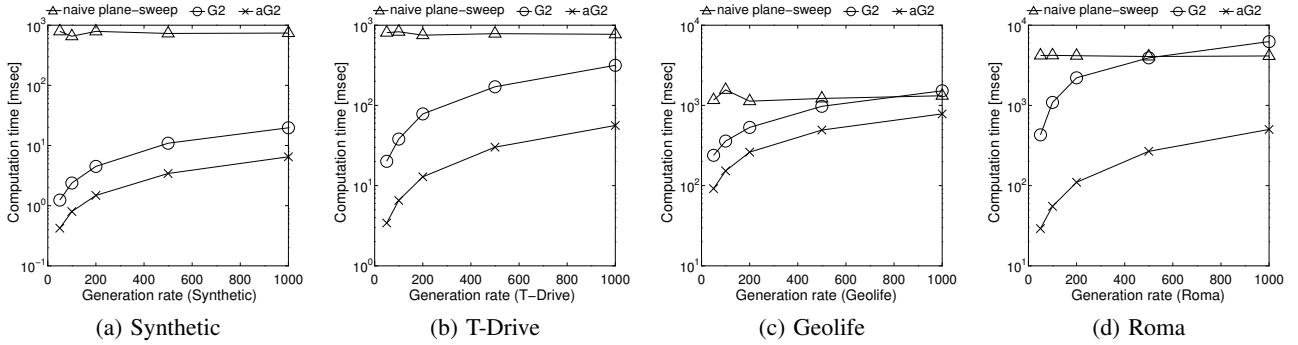


Figure 8: Impact of  $m$

Table 5: Computation time [msec]

Algorithm	Synthetic	T-Drive	Geolife	Roma
Algo. 2	0.799	6.547	152.644	55.196
Algo. 5 with cond.	0.796	5.952	202.127	49.245
Algo. 5	0.812	6.229	217.541	54.837

and measured the average computation time to update  $s^*$ . We also measured the practical error rate and computation time of aG2 by varying  $\epsilon$  to investigate the performance of our approximate algorithm in Section 7.4. Let  $s$  be the space monitored by our approximate algorithm, and the practical error is defined by  $1 - s.w/s^*.w$ . Finally, we measured the average computation time to update  $S^*$  by varying  $k$  in Section 7.5.

## 7.2 Impact of Algorithm 5

We first study the impact of Algorithm 5 in our default setting. Table 5 shows the result. In Table 5, ‘‘Algo. 5 with cond.’’ is denoted by Algorithm 2 with Algorithm 5 in the case where the upper-bounding cost is less than the plane-sweep algorithm. We can see that Algorithm 5 provides a trivial impact for all the four datasets and does not outperform Algorithm 2. We observed that  $|R(r_i)|$  in Algorithm 5 is large in Geolife, thus Algorithm 5 provides a negative impact. Therefore we do not employ Algorithm 5 since it is not scalable.

## 7.3 Results on monitoring MaxRS

**Impact of  $n$ .** We study the impact of the number of objects on a sliding-window, and Figure 7 shows the results. All the algorithms need longer computation time as  $n$  increases. In terms of naive plane-sweep, this is intuitive since its cost is  $\mathcal{O}(n \log n)$ . It is also intuitive that the increase of the number of overlapped spaces due to the increase of  $n$  is likely to lose the chances which avoid  $\text{OverlapComputation}(\cdot)$  and  $\text{ExactWeightComputation}(\cdot, \cdot)$ ,

thus our algorithms also need longer computation time. However, we can see that our algorithms are more efficient than naive plane-sweep in the four datasets, as expected. Naive plane-sweep is not scalable, which is shown in the case of large  $n$ . Moreover, aG2 scales better than G2 (the computation time is shown in log-scale). aG2 updates the result more than 2 times faster than G2 in the four datasets.

**Impact of  $m$ .** Next, we study the impact of the generation rate, i.e., the number of objects generated at the same time. Although the practical average generation rates of the three real datasets are less than 50 objects (per second), we employ larger generation rates to investigate the scalability of our algorithms. Figure 8 shows the results. Because naive plane-sweep computes  $s^*$  from scratch, it is not affected by  $m$  basically. The computation time of our algorithms increases as  $m$  increases. When  $m$  is large, the upper-bound weights maintained by cells and vertices are likely to become large, which results in the same observation as large  $n$ . Note that even a case of large  $m$ , e.g.,  $m = 1000$ , we can observe that aG2 still updates the result faster than naive plane-sweep.

**Impact of  $l$ .** User-specified rectangle size also has impacts on the performances of continuous MaxRS query processing, because large rectangles tend to overlap with others. Figure 9 shows the results. We can see that aG2 keeps outperforming naive plane-sweep, but the tendencies are different between the datasets. In the uniform distribution, i.e., Figure 9(a), G2 and aG2 are not much affected by  $l$ , but in the real datasets, the computation time of our algorithms (and naive plane-sweep) increase as  $l$  increases. We observed that the distributions of the real datasets are *skewed*. Therefore many rectangles overlap with each other, then G2 and aG2 are likely to encounter the case that  $\text{LocalPlaneSweep}(\cdot)$  is not avoided.

## 7.4 Results on monitoring Approximate MaxRS

We evaluated our approximate branch-and-bound algorithm us-

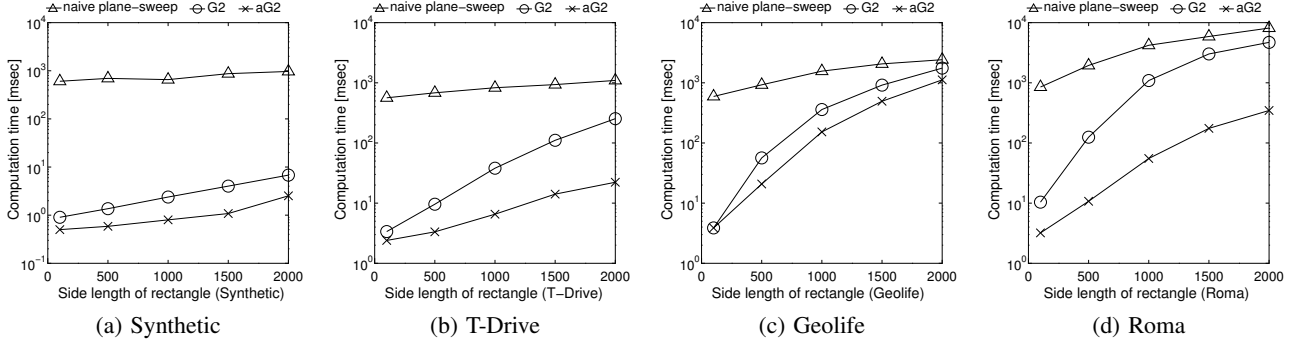


Figure 9: Impact of  $l$

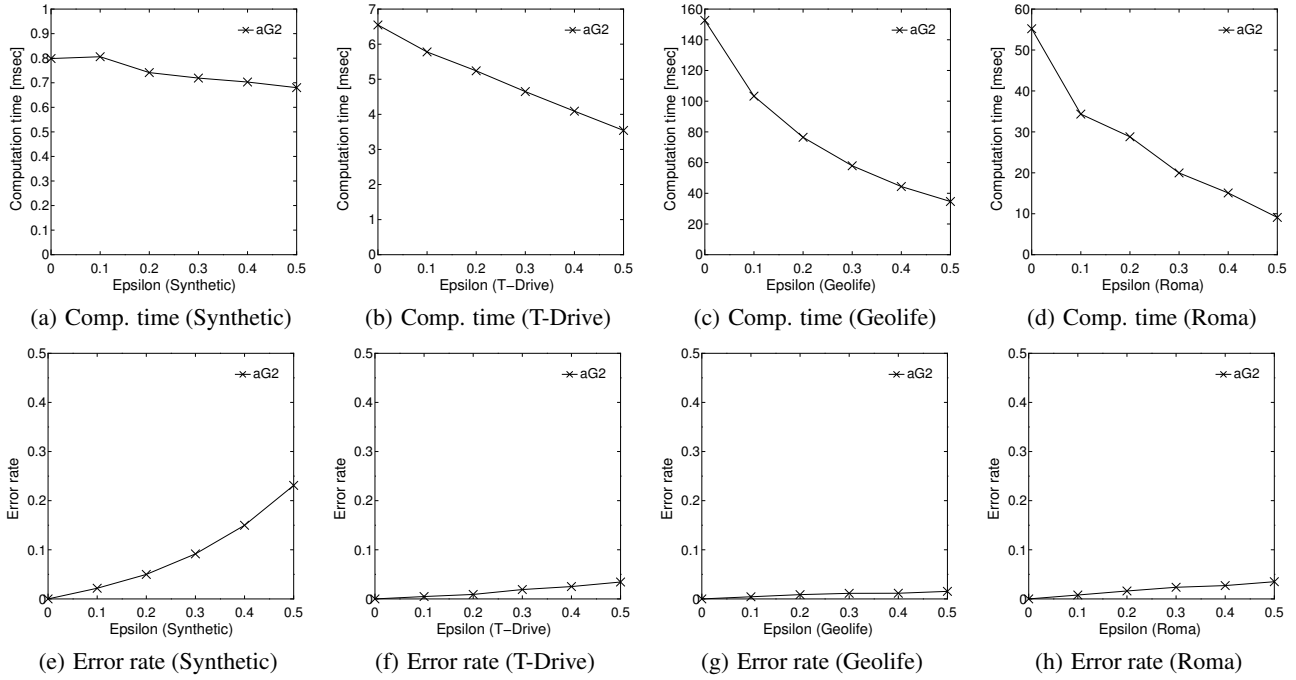


Figure 10: Impact of  $\epsilon$

ing aG2 by varying  $\epsilon$ . Although an approximate algorithm for *one-time computation* of MaxRS queries has been proposed in [25], this algorithm cannot be used for comparison due to three reasons. First, this algorithm is based on randomized sampling, thus the answers returned by this algorithm and our algorithm are different. Second, the answer returned by this algorithm varies every time, which is not suitable as monitoring algorithm. Last, repeating such one-time computation is shown to be inefficient in Section 7.3. We therefore focus on our algorithm, and the experimental results are shown in Figure 10.

From Figures 10(a)–10(d), although its impact is different between the datasets, we can see that the computation time decreases as  $\epsilon$  increases. This result satisfies the requirement that needs faster computation with an approximate answer. In the synthetic dataset, it seems that the computation time does not decrease so much (Figure 10(a)), but even when  $\epsilon = 0$ , the computation time is about 0.8 [msec], which is fast enough. From Figures 10(e)–10(h), we can see that as  $\epsilon$  increases, the practical error also increases but is less than  $\epsilon$ . The results show that the relationship between the query processing efficiency and the quality of the result is trade-off, but an interesting observation is that the practical error rates in the cases of the real datasets are very small.

## 7.5 Results on monitoring Top-k MaxRS

To evaluate our branch-and-bound algorithm for continuous top-k MaxRS queries, we conducted an experiment by varying  $k$ . We compare our algorithm with naive plane-sweep. Although naive plane-sweep is for one-time computation, this algorithm can deal with top-k MaxRS queries without sacrificing the computation cost. We do not evaluate G2 since its performance is not better than aG2.

Figure 11 shows the results. Again, naive plane-sweep is not affected by  $k$  since it scans all rectangles on a sliding-window. As  $k$  increases, the computation time of aG2 increases, but we can see that the increase of the computation time of aG2 is slight for all the four datasets. These results confirm that the pruning rules keep efficient and avoid unnecessary computation.

## 8. CONCLUSION

In this paper, we addressed a novel problem of monitoring MaxRS and its variants, i.e., monitoring approximate MaxRS and top-k MaxRS. In the environments where spatio-temporal objects are generated frequently, monitoring and analysis of objects are often required, and a continuous MaxRS query is useful to support such requirements. Unfortunately, the existing solutions [8, 25] focus on static objects, thus are not efficient in our problem. Motivated

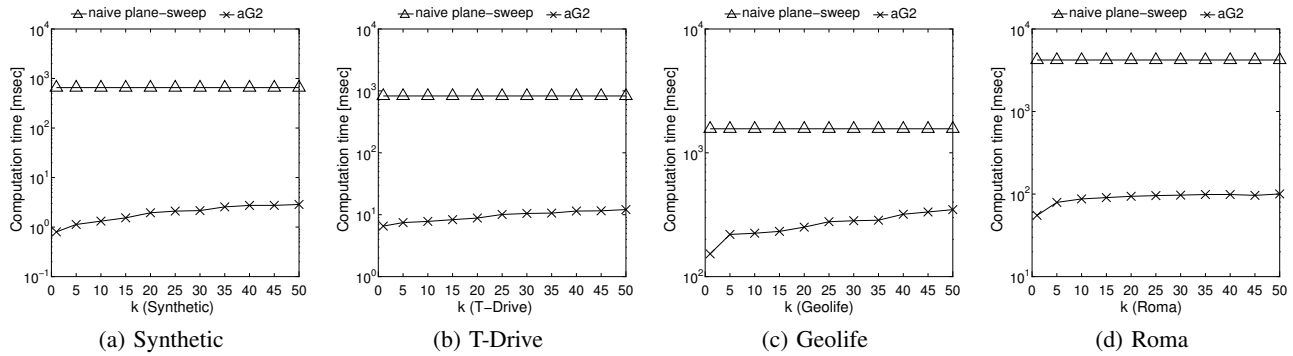


Figure 11: Impact of  $k$

by this, we proposed the first algorithm that can incrementally update the exact result. This algorithm incorporates our index framework G2 (Graph in Grid index) which supports efficient query processing. We extended G2 and proposed aG2 (aggregate G2) and a branch-and-bound algorithm using aG2. The branch-and-bound algorithm accelerates the query processing efficiency. Moreover, we showed that the branch-and-bound algorithm can deal with the monitoring approximate MaxRS and the top- $k$  MaxRS problems with simple modifications. To demonstrate the efficiency of our algorithms, we conducted experiments using synthetic and real datasets. The results show that the branch-and-bound algorithm using aG2 is superior to the one-time computation approach and the algorithm using G2.

As shown (but not theoretically) in Section 5.3, we need extra computation and storage costs to tight the upper-bound weights more. Hence, it is interesting to theoretically explore (or clarify the impossibility of) an approach that can tight the upper-bound the most without sacrificing the computational cost and storage cost. It is also interesting to develop an efficient algorithm that can deal with multiple continuous MaxRS queries at the same time. These are the works that need to be considered in the future.

**Acknowledgment.** This research is partially supported by the Grant-in-Aid for Scientific Research (A)(26240013) of MEXT, Japan, and JST, Strategic International Collaborative Research Program, SICORP.

## 9. REFERENCES

- [1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.
- [2] J. Bao, M. F. Mokbe, and C.-Y. Chow. Geofeed: A location aware news feed system. In *ICDE*, pages 54–65, 2012.
- [3] M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang. Continuous monitoring of distance-based range queries. *IEEE TKDE*, 23(8):1182–1199, 2011.
- [4] M. A. Cheema, W. Zhang, X. Lin, Y. Zhang, and X. Li. Continuous reverse  $k$  nearest neighbors queries in euclidean space and in spatial networks. *The VLDB Journal*, 21(1):69–95, 2012.
- [5] L. Chen, G. Cong, X. Cao, and K.-L. Tan. Temporal spatial-keyword top- $k$  publish/subscribe. In *ICDE*, pages 255–266, 2015.
- [6] Z. Chen, Y. Liu, R. C.-W. Wong, J. Xiong, X. Cheng, and P. Chen. Rotating maxrs queries. *Information Sciences, Elsevier*, 305:110–129, 2015.
- [7] Z. Chen, Y. Liu, R. C.-W. Wong, J. Xiong, G. Mai, and C. Long. Efficient algorithms for optimal location queries in road networks. In *SIGMOD*, pages 123–134, 2014.
- [8] D.-W. Choi, C.-W. Chung, and Y. Tao. A scalable algorithm for maximizing range sum in spatial databases. *PVLDB*, 5(11):1088–1099, 2012.
- [9] D.-W. Choi, C.-W. Chung, and Y. Tao. Maximizing range sum in external memory. *ACM TODS*, 39(3):21, 2014.
- [10] Y. Du, D. Zhang, and T. Xia. The optimal-location query. In *SSTD*, pages 163–180, 2005.
- [11] J. Huang, Z. Wen, J. Qi, R. Zhang, J. Chen, and Z. He. Top- $k$  most influential locations selection. In *CIKM*, pages 2377–2380, 2011.
- [12] H. Imai and T. Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *Journal of algorithms, Elsevier*, 4(4):310–323, 1983.
- [13] C. Li, Y. Gu, J. Qi, G. Yu, R. Zhang, and W. Yi. Processing moving  $knn$  queries using influential neighbor sets. *PVLDB*, 8(2):113–124, 2014.
- [14] Y. Liu, R. C.-W. Wong, K. Wang, Z. Li, C. Chen, and Z. Chen. A new approach for maximizing bichromatic reverse nearest neighbor search. *Knowledge and Information Systems, Springer*, 36(1):23–58, 2013.
- [15] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of spatial queries in wireless broadcast environments. *IEEE TMC*, 8(10):1297–1311, 2009.
- [16] K. Mouratidis and D. Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE TKDE*, 19(6):789–803, 2007.
- [17] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, pages 634–645, 2005.
- [18] S. C. Nandy and B. B. Bhattacharya. A unified algorithm for finding maximum and minimum object enclosing rectangles and cuboids. *Computers & Mathematics with Applications, Elsevier*, 29(8):45–61, 1995.
- [19] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *SSTD*, pages 443–459, 2001.
- [20] J. Qi, R. Zhang, L. Kulik, D. Lin, and Y. Xue. The min-dist location selection query. In *ICDE*, pages 366–377, 2012.
- [21] S. Rahul and Y. Tao. On top- $k$  range reporting in 2d space. In *PODS*, pages 265–275, 2015.
- [22] J. B. Rocha-Junior, A. Vlachou, C. Doukeridis, and K. Nørvgå. Efficient processing of top- $k$  spatial preference queries. *PVLDB*, 4(2):93–104, 2010.
- [23] C. Sheng and Y. Tao. New results on two-dimensional orthogonal range aggregation in external memory. In *PODS*, pages 129–139, 2011.
- [24] Y. Sun, J. Qi, Y. Zheng, and R. Zhang.  $K$ -nearest neighbor temporal aggregate queries. In *EDBT*, pages 493–504, 2015.
- [25] Y. Tao, X. Hu, D.-W. Choi, and C.-W. Chung. Approximate maxrs in spatial databases. *PVLDB*, 6(13):1546–1557, 2013.
- [26] Y. Tao, C. Sheng, C.-W. Chung, and J.-R. Lee. Range aggregation with set selection. *IEEE TKDE*, 26(5):1240–1252, 2014.
- [27] S.-H. Wu, K.-T. Chuang, C.-M. Chen, and M.-S. Chen. Diknn: an itinerary-based  $knn$  query processing algorithm for mobile sensor networks. In *ICDE*, pages 456–465, 2007.
- [28] T. Xia, D. Zhang, E. Kanoulas, and Y. Du. On computing top- $t$  most influential spatial sites. In *VLDB*, pages 946–957, 2005.
- [29] M. L. Yiu, X. Dai, N. Mamoulis, and M. Vaitis. Top- $k$  spatial preference queries. In *ICDE*, pages 1076–1085, 2007.
- [30] M. L. Yiu and N. Mamoulis. Multi-dimensional top- $k$  dominating queries. *The VLDB Journal*, 18(3):695–718, 2009.
- [31] J. Yuan, Y. Zheng, X. Xie, and G. Sun. Driving with knowledge from the physical world. In *SIGKDD*, pages 316–324, 2011.
- [32] D. Zhang, Y. Du, T. Xia, and Y. Tao. Progressive computation of the min-dist optimal-location query. In *VLDB*, pages 643–654, t, 2006.
- [33] J. Zhang, W.-S. Ku, M.-T. Sun, X. Qin, and H. Lu. Multi-criteria optimal location query with overlapping voronoi diagrams. In *EDBT*, pages 391–402, 2014.
- [34] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from gps trajectories. In *WWW*, pages 791–800, 2009.
- [35] Z. Zhou, W. Wu, X. Li, M. L. Lee, and W. Hsu. Maxfirst for maxbrknn. In *ICDE*, pages 828–839, 2011.

# Similarity Search on Spatio-Textual Point Sets

Christodoulos Efstathiades  
European University Cyprus  
Cyprus  
C.Efstathiades@euc.ac.cy

Alexandros Belesiotis  
IMIS, R.C. Athena  
Greece  
abelesiotis@imis.athena-  
innovation.gr

Dimitrios Skoutas  
IMIS, R.C. Athena  
Greece  
dskoutas@imis.athena-  
innovation.gr

Dieter Pfoser  
George Mason University  
USA  
dpfoser@gmu.edu

## ABSTRACT

User-generated content on the Web increasingly has a geospatial dimension, opening new opportunities and challenges in location-based services and location-based social networks for mining and analyzing user behaviors and patterns. The applications of such analysis range from recommendation systems to geo-marketing. Motivated by these needs, querying and analyzing spatio-textual data has received a lot of attention over the last years. In this paper, we address the problem of matching point sets based on the spatio-textual objects they contain. This is highly relevant for users associated with geolocated photos and tweets. We formally define this problem as a Spatio-Textual Point-Set Join query, and we introduce its top- $k$  variant. For the efficient treatment of such queries, we extend state-of-the-art algorithms for spatio-textual joins of individual points to the case of point sets. Finally, we extensively evaluate the proposed methods using large scale, real-world datasets from Flickr and Twitter.

## Categories and Subject Descriptors

H.2.8 [Database Management]: Database Applications—*Spatial databases and GIS*

## General Terms

Algorithms

## Keywords

spatio-textual join, spatio-textual point sets, similarity search

## 1. INTRODUCTION

Social media platforms such as Twitter, Flickr, Facebook and Foursquare have attracted billions of active users. In the case of Twitter 500 million tweets are exchanged every

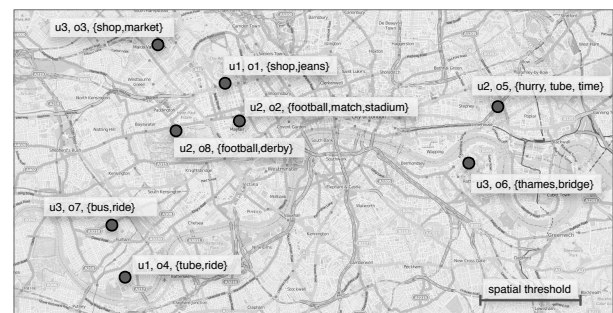


Figure 1: STPSJoin query scenario. Multiple objects are spatially or textually similar, but only users  $u_1$  and  $u_3$  have objects which are mutually similar.

day from 100 million active users. User activities in these platforms generate content that has *textual component*, e.g., status updates, short messages, or tags, and, following the widespread adoption of GPS in mobile devices, a *geospatial component*, e.g., geotagged tweets, photos, and user check-ins. Thus, the actions of users are documented by their messages in social networks and as such generate “traces”, which consist of spatio-textual objects.

Efficient indexing and querying of spatio-textual data has received a lot of attention over the past years, due to the high importance of such content in location-based services, such as nearby search and recommendations. In particular, multiple types of spatio-textual queries have been extensively studied, including boolean range queries, top- $k$  queries,  $k$ -nearest neighbor queries, and more recently, spatio-textual similarity joins [11, 7]. Nevertheless, in existing works, *spatio-textual entities are typically treated as isolated observations*. A typical example query is to find nearby restaurants or hotels matching certain criteria.

The work in [7] deals with finding pairs of entities that are both spatially close and textually similar. Example use cases are de-duplicating Points-of-Interest across datasets, or finding matching photos taken at roughly the same location and having similar tags.

Now considering looking for similar users in social networks. Here, a user is characterized by the messages they generate and, if available, respective location information. As such, each message can be considered a spatio-textual object, e.g., a geotagged photo or tweet. With each user being character-

ized by a set of spatio-textual objects, to find similar users, one needs to examine the similarity of these respective sets. Effectively, this characterizes users by what and where they tweet. An example of such a scenario is depicted in Figure 1. To that effect, this work addresses the problem of similarity search for *spatio-textual entities*, with an entity being characterized by a *set of spatio-textual objects*. We introduce the *Spatio-Textual Point-Set Similarity Join (STPSJoin)* query. Given sets of spatio-textual objects, each one belonging to a specific entity, this query seeks pairs of entities that have similar spatio-textual objects.

The *STPSJoin* can naturally model the search for entities exhibiting similar behavior according to the spatio-textual objects they generate. With social media users posting messages at various locations, the *STPSJoin* allows us to discover users that exhibit similar “geo-textual” behavior. This holds especially true for location-based social media sites, e.g., Foursquare, where users report on the places they visit. Following the general observation that spatio-textual object sets can be used to define the context of a user, e.g. has family because of frequent toy store visits, the *STPSJoin* can be used to discover such groups of similar users.

To efficiently process (top- $k$ ) *STPSJoin*, we adapt and extend the state-of-the-art algorithms for processing similarity joins for single points [7]. The proposed algorithms make use of spatio-textual indexes in conjunction with an early termination and a filter-and-refinement strategy to effectively prune the search space, thus reducing the execution time by orders of magnitude. More specifically, the contributions of our work are as follows.

- We formally define the spatio-textual point set similarity join (*STPSJoin*) query, which extends and generalizes the spatio-textual similarity join for the case of point sets, and its top- $k$  variant.
- We first derive a baseline algorithm for the *STPSJoin* query by adapting the state-of-the-art *PPJ-C* algorithm [7] to work for point sets. Then, we propose two optimized algorithms, *S-PPJ-B* and *S-PPJ-F*, which apply an early termination and a filter-and-refinement strategy, respectively, to drastically prune the search space. This significantly reduces the number of comparisons required, both in terms of pairs of entities and in terms of individual points for each candidate pair.
- In addition, we present an alternative version of *S-PPJ-F*, denoted as *S-PPJ-D*, which relies on an R-tree instead of a grid for the spatial indexing.
- We adapt our methods to efficiently treat the top- $k$  *STPSJoin* query. We provide a direct adaptation of our best performing algorithm, and extended it in order to allow additional pruning of the search space.
- Finally, we perform an extensive experimental evaluation using three large, real-world datasets. The results of the experimental evaluation demonstrate that the proposed algorithms achieve an order of magnitude and above improvement in terms of execution time when compared to the baseline method.

The remainder of this work is structured as follows. Section 2 reviews related work. The *STPSJoin* query and its top- $k$  variant are formally introduced in Section 3. The

algorithms for the efficient evaluation of (top- $k$ ) *STPSJoin* queries are presented in Section 4. Section 5 presents an experimental evaluation of the proposed approaches. Finally, Section 6 gives conclusions and directions for future work.

## 2. RELATED WORK

First, we review recent advances on spatio-textual search, which exploit spatial and textual characteristics in order to efficiently prune the search space while searching for similar objects. Then, we detail the state-of-the-art in similarity joins, in order to establish the basis for our work on point set joins. Finally, we present related literature on user recommendations using location histories.

### 2.1 Spatio-Textual Search

A large amount of web documents nowadays contain both spatial and textual information, characteristics which are exploited by modern applications to provide enhanced location-based services. Such applications rely on spatio-textual indexing for efficient computation.

**Spatio-textual indexes.** Current research enables the combination of spatial and textual indexes into hybrid spatio-textual indexes that explicitly support geographically aware search. Established spatial indexes, such as R-trees [22], regular grids, and space-filling curves, are integrated with textual indexes, such as inverted files or signature files. For example, SPIRIT [36] uses regular grids as spatial indexes and inverted files for the indexing of the documents. [43] proposed different approaches to hybrid indexing that employ R\*-trees [6] for spatial indexing and inverted files for textual indexing. [12] follows a similar approach, but utilizes space-filling curves for spatial indexing. [15] combines R-trees with signature files, which are stored internally in the nodes of the tree. The IR-tree index [13, 26], leverages inverted files for each node of the tree, in order to keep aggregate information of the textual characteristics of the relevant objects, while [33] uses aR-Trees [30] in combination with inverted files. Spatio-textual search is employed in order to answer a range of queries. An overview of the performance of the most commonly used spatio-textual indexes in such queries can be found in [11].

**Spatial group keyword queries.** Another type of spatio-textual queries is spatial group keyword queries [40, 41, 9, 29]. These aim at finding groups of spatio-textual objects that collectively satisfy a number of given keywords, while minimizing the collective distances between points in the group and the given query point.

Although the aforementioned queries involve searching for groups of objects, they differ from the problem addressed in this paper. The *STPSJoin* query is not constrained by an input query point nor a given textual description. In addition, groups of objects are predefined according to the user they are associated with. *STPSJoin* considers spatial and textual distances of objects across groups, rather than within groups. Finally, *STPSJoin* deals with the problem of spatio-textual join, which is fundamentally different from range and  $k$ NN queries.

### 2.2 Similarity Joins

Similarity joins seek to identify pairs of objects from given sets that satisfy a predefined similarity threshold.

**Set similarity joins.** The set similarity join task is computationally challenging; a naive approach requires the consideration of the similarity between every possible pair of objects across sets. Set similarity joins have been extensively studied, especially with respect to textual characteristics, and multiple optimizations have been proposed. [34] uses an inverted index based probing method to reduce the number of potential candidates. [10] observes that the prefixes of potential candidates must satisfy a minimal overlap. The ALL-PAIRS algorithm proposed by [5] further optimizes the size of the inverted index. [37] presents *Adapt-Join* and [38] proposes PPJOIN+ which are the state-of-the-art algorithms for set similarity joins. PPJOIN+ builds on ALL-PAIRS and introduces a *positional filtering principle* which exploits the ordering of tokens, and operates both on the prefix and the suffix of the tokens of objects. PPJOIN+ is internally used as the final step of our algorithms in order to efficiently compute textual similarity joins. An experimental analysis and evaluation on string similarity joins can be found in [24].

**Spatial joins.** Data structures and algorithms for spatial joins have been widely studied in the literature. A relevant survey can be found in [23]. Spatial joins have been used in combination both with space partitioning as well as with data partitioning structures. The state of the art algorithm for spatial joins has been proposed in [8]. We utilize this algorithm to prune the search space when searching for spatially relevant users using an R-Tree (see Section 4.1.4).

The problem of spatial joins over point sets has not received much attention. Adelfio et al. [2, 1] focus on similarity search for a collection of spatial point set objects based on the Hausdorff distance. The motivation behind their work is highly relevant to the STPSJoin query. However, there are important differences. We consider web objects with spatio-textual characteristics and measure the distance among point sets using a different similarity measure. The Hausdorff distance measures the maximum discrepancy between two point sets, whereas in our work we use a measure inspired by the Jaccard coefficient which focuses on the amount of objects from different point sets that are similar.

**Spatio-textual joins.** Spatio-textual joins have attracted some attention recently with a specific focus on joins for spatio-textual points. This process is primarily executed for the purpose of duplicate detection. The work in [3] is one of the first examples of spatio-textual join methods. They propose the SpSJoin query that follows the MapReduce paradigm for scalable computation of spatio-textual join queries. The spatio-textual join query has been also studied in the form of spatial regions associated with textual descriptions ([27, 28, 20]). Pruning strategies, based on spatial and textual signatures of objects, are employed to filter the number of candidates. [32] presents grid and quad tree based indexes in order to efficiently partition the database either in a local or global fashion. They also explore different dimensions of the problem, including the use of PPJOIN+ and All-Pairs for text similarity joins, as well as single and multi-threaded approaches.

Bouros et al. [7] propose the state of the art spatio-textual join algorithms. Their work builds on top of PPJ, a baseline method that extends PPJOIN+ to account for objects with spatio-textual characteristics and a given spatial distance threshold. The algorithms PPJ-C and PPJ-R extend PPJ by leveraging a grid and an R-Tree based index respec-

tively. These methods provide the basis for our work; thus, we revisit them in more detail in Section 4.1.1.

Work on spatio-textual joins is highly relevant to our approach. However, the focus is different. To the best of our knowledge, current research in the field has focused on spatio-textual similarity joins among points. On the contrary, our work introduces spatio-textual similarity joins among point sets. Point sets are relevant when objects are grouped with respect to a common characteristic. In this case, the focus is on identifying similarities among groups, rather than single elements. For instance, in the case of web objects, a group consists of objects associated with the same user. In this case, point-set joins identify user similarity instead of object similarity.

## 2.3 User Recommendation Systems

Matching users based on their location history is one of the main tasks of recommendation engines in location-based social networks [4]. User location histories have been used for identifying local experts, recommending friends, and extracting local communities. It has been revealed by several studies that location information plays a vital role in determining such relationships [14, 16].

Typically, these approaches take into consideration additional information, such as location ratings, semantics of location descriptions and tags, sequence of visit or duration of stay. [25] identifies users with similar traveling patterns based on matching sequences of locations visited by the users. Similar works ([25, 42, 39]) deal with finding users with similar patterns in behavior. [21] study several features to identify users that are similar to a given user. Their methods are based on a logistic regression model. According to their work, a single and in many cases imprecise user location feature (such as city or country) is not effective.

In this paper, we focus on multiple geo-tagged objects for each user, which may provide a better insight into location-based user similarity.

## 3. PROBLEM DEFINITION

We assume a database  $\mathcal{D}$  of spatio-textual objects created by different users  $U$ . A spatio-textual object  $o \in \mathcal{D}$  is a triple  $o = \langle u, loc, doc \rangle$ , where  $u \in U$  is the user associated with this object,  $loc = \langle x, y \rangle$  is a spatial point and  $doc$  is a set of keywords. We refer to the user, location and keywords associated with an object  $o$  using the notation  $o.u$ ,  $o.loc$  and  $o.doc$  respectively. In addition, we use  $D_u$  to denote the set of objects belonging to user  $u$ .

The spatial distance  $\delta(o, o')$  between two objects is calculated as the Euclidean distance between their spatial locations. Moreover, the textual similarity  $\tau(o, o')$  is measured according to the Jaccard similarity of their keywords:

$$\tau(o, o') = \frac{|o.doc \cap o'.doc|}{|o.doc \cup o'.doc|}.$$

Given a spatial threshold  $\epsilon_{loc}$  and a textual threshold  $\epsilon_{doc}$ , we say that two objects  $o, o' \in \mathcal{D}$  match if their spatial distance is below  $\epsilon_{loc}$  and their textual similarity is above  $\epsilon_{doc}$ . Matching between objects is defined using the predicate  $\mu$ :

$$\mu(o, o') = \begin{cases} True & \text{if } \delta(o, o') \leq \epsilon_{loc} \text{ and } \tau(o, o') \geq \epsilon_{doc} \\ False & \text{otherwise.} \end{cases}$$

For brevity, we overload  $\mu$  to account for matching an object

with a set of objects  $D \subseteq \mathcal{D}$ :

$$\mu(o, D) = \begin{cases} \text{True} & \text{if there exists } o' \in D \text{ such that } \mu(o, o') \\ \text{False} & \text{otherwise.} \end{cases}$$

Furthermore, let two spatio-textual point-sets  $D$  and  $D'$ . Function  $M(D, D')$  returns the set of objects in  $D$  that match with at least one object in  $D'$ :

$$M(D, D') = \{o \in D \text{ such that } \mu(o, D')\}.$$

We then use  $M$  to define the similarity of point-sets  $D$  and  $D'$ . In particular, this is measured as the fraction of the matched points from one set to the other divided by the total number of points in the two sets. Formally:

$$\sigma(D, D') = \frac{|M(D, D')| + |M(D', D)|}{|D| + |D'|}.$$

The employed measure is inspired by the Jaccard similarity, which is not directly applicable since it does not support partial similarity between elements. More elaborate similarity metrics over point sets can be found in [19, 31].

We can now define the *Spatio-Textual Point Set Join* query (STPSJoin). STPSJoin identifies all pairs of users  $U$  which are associated with sets of spatio-textual objects that have a match higher than a specified threshold  $\epsilon_u$ . We assume a total ordering over  $U$  (i.e.  $\prec_U$ ) to avoid returning duplicate pairs. Formally, the STPSJoin query is defined as follows.

**DEFINITION 1.** *Given a database  $\mathcal{D}$  of spatio-textual objects belonging to a set of users  $U$ , the STPSJoin query is a tuple  $Q = \langle \epsilon_{loc}, \epsilon_{doc}, \epsilon_u \rangle$  which returns a set  $R$  containing all pairs of users  $(u, u')$  such that  $u, u' \in U$ ,  $u \prec u'$ , and  $\sigma(D_u, D_{u'}) \geq \epsilon_u$  with respect to the spatial and textual thresholds  $\epsilon_{loc}$  and  $\epsilon_{doc}$ .*

An extension of the STPSJoin query in which we seek only the  $k$  best pairs of users, in terms of spatial and textual similarity of their objects, is the top- $k$  STPSJoin query. Formally, the top- $k$  STPSJoin query is defined as follows.

**DEFINITION 2.** *Given a database  $\mathcal{D}$  of spatio-textual objects belonging to a set of users  $U$ , the top- $k$  STPSJoin query is a tuple  $Q = \langle \epsilon_{loc}, \epsilon_{doc}, k \rangle$  which returns a set  $R$  containing  $k$  pairs of users  $(u, u')$  such that  $u, u' \in U$ ,  $u \prec u'$ , and for any pair of users  $(v, v') \notin R$  it holds that  $\sigma(D_u, D_{u'}) \geq \sigma(D_v, D_{v'})$  for each  $(u, u') \in R$  with respect to the spatial and textual thresholds  $\epsilon_{loc}$  and  $\epsilon_{doc}$ .*

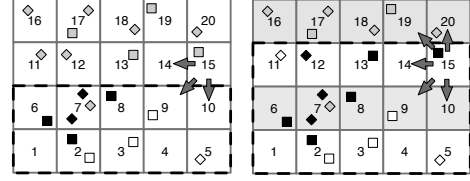
## 4. ALGORITHMS

This section presents algorithms for the evaluation of the STPSJoin query and the top- $k$  STPSJoin query. First, we present a baseline algorithm, and then we introduce methods that exploit a filter and refine strategy in combination with spatio-textual indexes in order to direct the search. Then, we explain how our methods can be adapted to account for the top- $k$  STPSJoin query.

### 4.1 Algorithms for STPSJoin

#### 4.1.1 Baseline Approach

**Preliminaries.** The straightforward method for evaluating an STPSJoin query is to find, for every pair of users, the set of matching objects, and then to check whether the resulting similarity score  $\sigma$  exceeds the specified threshold  $\epsilon_u$ . Thus,



(a) PPJ-C traversal. (b) PPJ-B traversal.

Figure 2: PPJ-C and PPJ-B grid traversal strategies examples. Objects associated with user  $u$  ( $u'$ ) are depicted by squares (diamonds). Matched objects are painted black, objects that do not match are painted white, while objects whose state has not been determined are painted grey. PPJ-B has determined the state of every object in cells 1 to 15, while PPJ-C only the objects until cell 10.

for a pair of users  $(u, u')$ , the problem can be cast as a spatio-textual similarity join query, ST-SJOIN( $D, \epsilon_{loc}, \epsilon_{doc}$ ), which has been studied in [7]. This query returns all pairs of objects  $(o, o')$  in  $D$  such that  $o, o' \in D$ ,  $\delta(o, o') \leq \epsilon_{loc}$  and  $\tau(o, o') \geq \epsilon_{doc}$ . Based on this, we can find the objects of  $u$  that match with those of  $u'$ , and vice versa, and then proceed with computing the score  $\sigma$  for this pair of users.

For this purpose, we adapt the PPJ-C algorithm from [7] for the purposes of ST-SJOINS. PPJ-C uses a grid to partition the space, in order to limit the search to those candidates that can satisfy the spatial predicate of the join. The grid is constructed dynamically at query time, using cells that have an extent in each dimension that equals the spatial distance threshold  $\epsilon_{loc}$ . The cells are assigned ids in a row-wise order from bottom to top (see Figure 2a).

PPJ-C visits the cells in ascending order of their ids, taking advantage of the spatial filtering, since the objects in each visited cell  $c$  need to be joined only with those in  $c$  and in the cells adjacent to  $c$ . In fact, to avoid duplicates, only the adjacent cells with ids lower than  $c$  need to be examined. Thus, for each cell, one self-join operation and at most four non-self join operations need to be performed. These are performed using the PPJ algorithm, that in turn extends the set similarity join algorithm PPJOIN [38] by including an additional check on the spatial distance of two objects.

**The S-PPJ-C algorithm.** Using PPJ-C as basis, we can derive a baseline algorithm, denoted as S-PPJ-C (Set-PPJ-C), for the STPSJoin query. S-PPJ-C is presented in Algorithm 1. During the construction of the grid, we maintain the following additional information: (a) for each cell  $c$ , we maintain the contained objects in separate lists according to the user they belong to; we denote by  $D_c^u$  the set of objects of user  $u$  that are contained in  $c$ ; (b) for every user  $u$ , we maintain a list of cells  $C_u$  that contain objects belonging to  $u$ ;  $C_u$  is sorted according to cell ids in ascending order.

The S-PPJ-C algorithm loops through all pairs of users, taking into consideration the total ordering  $\prec_U$  of the user set  $U$ . For each pair of users  $(u, u')$ , S-PPJ-C executes a non-self join version of the PPJ-C algorithm from [7] presented above. The difference with the standard PPJ-C, lies in the fact that in this case pairs with objects from both users are returned. To do so, first the lists  $C_u$  and  $C_{u'}$  containing the cells for  $u$  and  $u'$  respectively are gathered. Next, the algorithm iteratively selects from either list the cell  $c$  with the lowest id that has not been selected yet. Assume that



---

**Algorithm 1: S-PPJ-C Algorithm**


---

**Input:**  $D, U, \epsilon_{doc}, \epsilon_{loc}, \epsilon_u$   
**Output:** Pairs of matched users  $R$

- 1  $R \leftarrow \emptyset$
- 2  $selectedUsers \leftarrow \emptyset$
- 3  $G \leftarrow createGridIndex(D, U, \epsilon_{loc})$
- 4 **foreach**  $u_1 \in U$  **do**
- 5     **foreach**  $u_2 \in selectedUsers$  **do**
- 6          $r \leftarrow PPJ-C(u_1, u_2, \epsilon_{doc}, \epsilon_{loc})$
- 7          $\sigma \leftarrow \frac{|r|}{|D_{u_1}| + |D_{u_2}|}$
- 8         **if**  $\sigma \geq \epsilon_u$  **then**
- 9              $R.add(\langle u_2, u_1 \rangle)$
- 10          $selectedUsers.add(u_1)$
- 11 **return**  $R$

---

the next selected cell  $c$  is from the list of user  $u$ . For every cell  $c'$  in  $C_{u'}$  with  $c'.id \geq c.id$ , a non-self join version of PPJ is executed with input the spatio-textual point sets  $D_u^c$  and  $D_{u'}^{c'}$ . Since  $C_u$  and  $C_{u'}$  may both contain the cell  $c$ , we avoid the duplicate execution of PPJ for  $c$ .

The results of PPJ-C are used to compute the user similarity score  $\sigma$  (line 6-7). Pairs of users that achieve a similarity score above the threshold  $\epsilon_u$  are collected in the result set.

#### 4.1.2 The S-PPJ-B Algorithm

The drawback of the S-PPJ-C algorithm is that for each pair of users it finds all their matching points and computes the exact value of their similarity score  $\sigma$  before checking whether this exceeds the given threshold. Instead, since we are only interested in finding those pairs with a similarity that exceeds  $\epsilon_u$ , we can reduce the execution time of the algorithm by terminating the computation for a pair of users as soon as it can be decided that their similarity is below  $\epsilon_u$ . Following this observation, we derive a more efficient algorithm, denoted as S-PPJ-B (where B stands for bound).

S-PPJ-B operates in the same manner as S-PPJ-C, with the only difference that it replaces the execution of PPJ-C with a modified process, denoted as PPJ-B. PPJ-B leverages the use of an upper bound on the number of unmatched objects for a pair of users to effectively prune the search on the spatial grid. More specifically, the intuition behind PPJ-B is the following. While examining two users, PPJ-B leverages the user similarity threshold  $\epsilon_u$  and the number of objects belonging to each user in order to compute an upper bound on the number of unmatched objects between the two users, above which the user similarity cannot exceed  $\epsilon_u$ . In the following, we first derive this upper bound, and then we explain the process followed by PPJ-B in order to allow for early termination during the examination of two users.

For a pair of users  $(u, u')$ , let  $\beta_{u,u'}$  denote the number of objects from user  $u$  and user  $u'$  that do not match with the other user, i.e.:

$$\beta_{u,u'} = |D_u| + |D_{u'}| - |M(D_u, D_{u'})| - |M(D_{u'}, D_u)|$$

An upper bound for  $\beta_{u,u'}$  is derived as follows.

**LEMMA 1.** *For a pair of users  $(u, u')$ , if  $\beta_{u,u'} > (1 - \epsilon_u) \cdot (|D_u| + |D_{u'}|)$  then  $\sigma(D_u, D_{u'}) < \epsilon_u$ .*

**PROOF.** The proof is derived from the definition of the

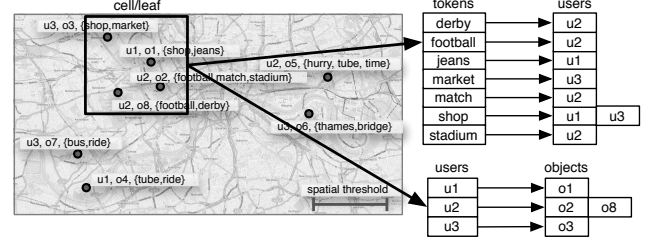


Figure 3: Spatio-textual structure for S-PPJ-F and S-PPJ-D.

similarity score between two users, as follows:

$$\begin{aligned} \sigma(D_u, D_{u'}) \geq \epsilon_u &\Rightarrow \frac{|M(D_u, D_{u'})| + |M(D_{u'}, D_u)|}{|D_u| + |D_{u'}|} \geq \epsilon_u \Rightarrow \\ \frac{|D_u| + |D_{u'}| - \beta_{u,u'}}{|D_u| + |D_{u'}|} &\geq \epsilon_u \Rightarrow 1 - \frac{\beta_{u,u'}}{|D_u| + |D_{u'}|} \geq \epsilon_u \Rightarrow \\ \beta_{u,u'} &\leq (1 - \epsilon_u) \cdot (|D_u| + |D_{u'}|) \end{aligned}$$

□

Upon traversing a cell  $c$ , PPJ-C checks for potential matches in cells with ids lower than  $c.id$ . Therefore, we cannot be certain that objects that have not been matched so far will also not match with objects in cells with higher ids (i.e. in the next cell or row). Therefore, the bound may be used within PPJ-C, but only with respect to the objects discovered from the beginning of the grid until the previous row. The objects that were traversed in the current row have to be excluded from calculation.

To that end, PPJ-B devises a different grid traversal strategy that allows the pruning mechanism to utilize every object appearing in cells traversed when the bound evaluation is executed. Specifically, this strategy traverses the rows from bottom to top (considering the id of the bottom row as 1), and depending on whether the id of a row is odd or even, different treatment is followed. If a cell  $c_{i,j}$  belongs to a row with odd id, i.e.  $j$  is odd, then the objects contained in it are matched with objects from all surrounding cells, except the cell directly on the right, i.e.  $c_{i+1,j}$ . Matching is done by executing PPJoin. Otherwise, if the cell belongs to an even row, then we match its objects only with objects from the other user from the cell that is directly on the left, i.e.  $c_{i-1,j}$ . This process is illustrated in Figure 2b.

Following this traversal strategy, PPJ-B allows for early termination using the bound  $\beta$ , while still maintaining the property of PPJ-C to avoid duplicate examination of the same pair of cells. Indeed, when PPJ-B traverses the last cell of an odd row, it has considered every potential match for any object it has encountered up to that point. Thus, it checks whether the number of objects that have not been matched exceeds the calculated bound  $\beta$ . If so, the search stops, since it is impossible to result in a user similarity score that exceeds  $\epsilon_u$ . Note that, in practice, since the grid may be rather sparse, some rows may be empty. In that case, when the next visited cell belongs to a row that is not directly above the previous one, the same check can be performed, even if the last examined row was even, since previously encountered objects cannot have any future matches.

#### 4.1.3 The S-PPJ-F Algorithm

The S-PPJ-B algorithm presented above exploits an upper bound on the number of unmatched objects between two

users in order to allow for early termination when comparing each pair of users. In the following, we present the S-PPJ-F algorithm that further increases efficiency by following a filter and refine strategy that concentrates the search on those pairs of users that are promising candidates, while pruning others that can not exceed the similarity threshold  $\epsilon_u$ .

---

**Algorithm 2:** *S-PPJ-F* Algorithm

---

**Input:**  $D, U, \epsilon_{doc}, \epsilon_{loc}, \epsilon_u$   
**Output:** Pairs of matched users  $R$

```

1  $R \leftarrow \emptyset$ 
2  $G \leftarrow \text{initialiseSTGridIndex}(D, \epsilon_{loc})$ 
3 foreach  $u \in U$  do
4   foreach  $c \in C_u$  do
5      $T \leftarrow \text{calculateTokens}(u, c)$ 
6     foreach  $c' \in G.\text{getRelevantCells}(c)$  do
7       foreach  $t \in T$  do
8         foreach  $u' \in G.\text{getTokenUsers}(c', t)$  do
9            $M_u^u.add(c), M_{u'}^{u'}.add(c')$ 
10     $G.addUser(u)$ 
11    foreach  $u' \in M.\text{keys}()$  do
12       $m \leftarrow \sum_{c \in M_u^u} |D_u^c| + \sum_{c' \in M_{u'}^{u'}} |D_{u'}^{c'}|$ 
13       $\bar{\sigma} \leftarrow \frac{m}{|D_u| + |D_{u'}|}$ 
14      if  $\bar{\sigma} \geq \epsilon_u$  then
15         $\sigma \leftarrow \text{PPJ-B}(D_u, D_{u'}, G, \epsilon_{doc}, \epsilon_{loc}, \epsilon_u)$ 
16        if  $\sigma \geq \epsilon_u$  then
17           $R.add(\langle u', u \rangle)$ 
18 return  $R$ 

```

---

S-PPJ-F is outlined in Algorithm 2. It operates on top of a spatio-textual index structure that is constructed at runtime. In every iteration, the algorithm selects a new user  $u$ , searches for potential matches with the users that have been selected in previous steps, and updates the spatio-textual index with the objects in  $D_u$ .

The spatio-textual index is a dynamic grid enhanced with an inverted index for every cell. This list maintains for every token that appears in objects in a cell, the users that are associated with these objects. An example is depicted in Figure 3. The grid structure additionally maintains the objects associated with every user within a cell.

The search for matches follows the filter and refine principle. After a user  $u$  is selected, the algorithm traverses through every cell  $c \in C_u$  which contains objects associated with  $u$ , and calculates the set of tokens  $T$  that appear in any one of these objects. This set is then utilized to identify candidate users in  $c$  and its surrounding cells (lines 6-9). Every user  $u'$  with objects that appear in one of these cells that at least one keyword from  $T$  is considered to be a candidate.  $M_u^u$  maintains cells that contain objects from  $u$  that potentially match (both spatially and textually) with objects from  $u'$ . Respectively,  $M_{u'}^{u'}$  maintains the relevant for  $u'$ .

For every user  $u$  and candidate user  $u'$ , the algorithm calculates an upper bound  $\bar{\sigma}$  of their user similarity score (lines 12-13). This is performed by assuming that all of their objects which are contained in the same or adjacent cells match. Formally,  $\bar{\sigma}$  is computed as follows:

$$\bar{\sigma} = \frac{\sum_{l \in M_u^u} |D_u^l| + \sum_{l' \in M_{u'}^{u'}} |D_{u'}^{l'}|}{|D_u| + |D_{u'}|}.$$

If  $\bar{\sigma} < \epsilon_u$ , then this pair can be safely pruned. Otherwise, a refinement step follows, during which the PPJ-B algorithm is executed to identify whether the exact similarity score for the pair exceeds the user similarity threshold.

#### 4.1.4 The S-PPJ-D Algorithm

In the following, we consider databases that are already partitioned by a data partitioning scheme. In particular, we consider data partitioning schemes induced by an R-tree structure combined with a textual index similar in fashion to the index outlined with respect to S-PPJ-F. The main difference is that instead of indexing grid cells, in this case, we index the leaf nodes of the R-tree.

S-PPJ-D implements a filter and refinement strategy similar to S-PPJ-F, based on a given data partitioning and a spatio-textual index  $I$  that is constructed at runtime.  $I$  maintains an entry for every leaf node  $l$  in the tree. This entry holds an inverted list that maps a token  $t$   $U_t^l$  (i.e. users with objects in  $l$  that contain  $t$ ). In addition, every leaf node  $l$  maintains a mapping between users and their objects within  $l$ , denoted by  $D_u^l$ . Finally, the intersections among the extended MBRs of the leaf nodes in the tree are precomputed by performing a spatial join using the process described in [8].

The filter step iterates over the leaf nodes  $L_u$  of a user  $u$ . For every leaf node  $l$ , it calculates the set of tokens  $T$  that appear in objects within  $l$  that are associated with  $u$  (i.e.  $D_u^l$ ). These tokens are then used to probe the spatio-textual index and identify the candidate users that are associated with objects containing tokens from  $T$ . This is performed for each leaf node that intersects with the  $\epsilon_{loc}$ -extended bounding box of  $l$ . To avoid duplicates, we only search for candidate users which are higher in the user ordering.  $M$  maintains for every candidate  $u'$  the leaf nodes of  $u'$  containing objects that can potentially match objects associated with user  $u$   $M_{u'}^{u'}$ , as well as the leaf nodes of the relevant objects from  $u$   $M_u^u$ . S-PPJ-D calculates for every candidate  $u'$  a bound on the similarity score between  $u$  and  $u'$ . This is calculated by considering the extreme case in which all objects from  $M_{u'}^{u'}$  and  $M_u^u$  match. The refinement step uses PPJ-D in order to calculate the exact similarity between candidate users.

Algorithm 3 outlines PPJ-D. PPJ-D leverages the spatio-textual index in combination with an appropriate leaf node traversal strategy in order to return the similarity score between two users. PPJ-D functions similar to PPJ-B for the context of a data-driven partitioning scheme. Given two users  $u_1$  and  $u_2$ , two lists  $L_1$  and  $L_2$  are maintained for their leaf nodes ordered with respect to a predefined ordering (e.g. in ascending order of their ids). The algorithm proceeds iteratively, and selects the lowest (with respect to the ordering) unvisited leaf node  $l$  from  $L_1$  and  $L_2$ .

Let user  $u$  be the user from which the element was selected, and  $u'$  the other user. The index is used to identify every leaf node  $l'$  that is spatially relevant to  $l$ , and contains objects from  $u'$ . Spatially relevant leaf nodes are nodes with intersecting  $\epsilon_{loc}$ -extended MBRs. For every  $l'$  we execute PPJoin to identify the exact similarity between the objects  $D_u^l$  and  $D_{u'}^{l'}$ . This is performed by focusing only on objects that belong within the intersection  $A$  of the  $\epsilon_{loc}$ -extended MBRs of  $l$  and  $l'$  (lines 11-12, 18-19). This optimisation is based on the observation that objects which are not contained in  $A$  do not satisfy the spatial threshold  $\epsilon_{loc}$ .

PPJ-D follows a similar pruning strategy with PPJ-B. The

---

**Algorithm 3: PPJ-D Algorithm**

---

**Input:**  $D_{u_1}, D_{u_2}, I, \epsilon_{doc}, \epsilon_{loc}, \epsilon_u$   
**Output:** Similarity score for users  $u_1, u_2$

```
1  $\beta \leftarrow (1 - \epsilon_u) \cdot (|D_{u_1}| + |D_{u_2}|)$ 
2  $J \leftarrow \emptyset$  // joined objects
3  $L_1 \leftarrow I.getLeafs(u_1)$  // sorted
4  $L_2 \leftarrow I.getLeafs(u_2)$ 
5  $i_1 \leftarrow 0, i_2 \leftarrow 0$ 
6  $t \leftarrow 0$ 
7 while  $i_i < |L_1|$  or  $i_2 < |L_2|$  do
8   if  $L_1[i_1] \leq L_2[i_2]$  then
9     foreach  $l_2 \in I.getRelevantLeafs(l_1)$  do
10      if  $l_2 \geq l_1$  and  $l_2 \in L_2$  then
11         $A \leftarrow I.extend(l_1, \epsilon_{loc}) \cap I.extend(l_2, \epsilon_{loc})$ 
12         $PPJoin(D_{u_1}^{l_1} \cap A, D_{u_2}^{l_2} \cap A, J)$ 
13         $t \leftarrow t + |D_{u_1}^{l_1}|$ 
14      else if  $L_2[i_2] \leq L_1[i_1]$  then
15         $l_2 \leftarrow L_2[i_2]$ 
16        foreach  $l_1 \in I.getRelevantLeafs(l_2)$  do
17          if  $l_1 > l_2$  and  $l_1 \in L_1$  then
18             $A \leftarrow I.extend(l_1, \epsilon_{loc}) \cap I.extend(l_2, \epsilon_{loc})$ 
19             $PPJoin(D_{u_1}^{l_1} \cap A, D_{u_2}^{l_2} \cap A, J)$ 
20             $t \leftarrow t + |D_{u_2}^{l_2}|$ 
21        if  $t - |J| > \beta$  then
22          return 0
23        if  $L_1[i_1] \leq L_2[i_2]$  then  $i_1 \leftarrow i_1 + 1$ 
24        if  $L_2[i_2] \leq L_1[i_1]$  then  $i_2 \leftarrow i_2 + 1$ 
25       $\sigma \leftarrow |J| / (|D_{u_1}| + |D_{u_2}|)$ 
26      if  $\sigma \geq \epsilon_u$  then return  $\sigma$ 
27      else return 0
```

---

objects of every leaf node for user  $u$  are evaluated against every potential candidate from  $D_{u'}$  that falls within a leaf node that is higher in the given ordering. Therefore, after an iteration that visits an object, candidate matches from leaf nodes, both higher and lower in the ordering, are considered. This observation is the basis of a pruning step (lines 21-22) that calculates the number of objects  $t - |J|$  that are already found to fail to satisfy the thresholds. If this number is lower than a computed bound, the search is pruned since the users fail to satisfy the user similarity threshold.

## 4.2 Algorithms for top-k STPSJoin

Next, we extend our methods to support the top- $k$  STPSJoin query. The main intuition behind our approach is that the algorithm must keep track of the top- $k$  pairs identified thus far, and utilise the exact user similarity score of the  $k$ th best pair to update the user similarity threshold.

### 4.2.1 TOPK-S-PPJ-F

Algorithm 4 outlines TOPK-S-PPJ-F, which modifies S-PPJ-F for the purposes of the top- $k$  STPSJoin query. The main modifications with respect to S-PPJ-F relate to the maintenance of intermediate results and the update of the user similarity threshold. Results are stored in a fixed capacity priority queue of size  $k$ , which is updated whenever a pair that is better than the  $k$ th pair in the queue is identified. The user similarity threshold  $\epsilon_u$  is set as the similarity score of the  $k$ th best pair in the queue. Accordingly, the

---

**Algorithm 4: TOPK-S-PPJ-F Algorithm**

---

**Input:**  $D, U, \epsilon_{doc}, \epsilon_{loc}, k$   
**Output:** Top- $k$  Pairs of matched users  $R$

```
1  $R \leftarrow \emptyset$ 
2  $\epsilon_u \leftarrow -1$ 
3  $G \leftarrow initialiseSTGridIndex(D, \epsilon_{loc})$ 
4 foreach  $u \in sorted(U)$  do
5   foreach  $c \in C_u$  do
6      $T \leftarrow calculateTokens(u, c)$ 
7     foreach  $c' \in G.getRelevantCells(c)$  do
8       foreach  $t \in T$  do
9         foreach  $u' \in G.getTokenUsers(c', t)$  do
10           $M_{u'}^u.add(c), M_{u'}^{u'}.add(c')$ 
11         $G.addUser(u)$ 
12      foreach  $u' \in M.keys()$  do
13         $m \leftarrow \sum_{c \in M_{u'}^u} |D_u^c| + \sum_{c' \in M_{u'}^{u'}} |D_{u'}^{c'}|$ 
14         $\bar{\sigma} \leftarrow \frac{m}{|D_u| + |D_{u'}|}$ 
15        if  $\bar{\sigma} > \epsilon_u$  then
16           $\sigma \leftarrow PPJ-B(D_u, D_{u'}, G, \epsilon_{doc}, \epsilon_{loc}, \epsilon_u)$ 
17          if  $\sigma > \epsilon_u$  then
18             $R.update(\langle u', u \rangle)$ 
19          if  $|R| = k$  then
20             $\epsilon_u \leftarrow R.getTail()$ 
21 return  $R$ 
```

---

threshold  $\epsilon_u$  is updated whenever a new pair is introduced in the results queue (lines 18-20). The user similarity threshold is used in the filtering phase of the algorithm in a similar manner with S-PPJ-F. The same principle can be straightforwardly applied to S-PPJ-D. Pseudocode for the resulting algorithm is omitted due to lack of space.

TOPK-S-PPJ-F orders users in an ascending order of the size of their object-sets. This strategy is based on the observation that the treatment of users with larger object-sets requires more computations than the evaluation of users with fewer objects. By the time the algorithm reaches the most computationally demanding users, the user similarity threshold has been updated to reflect the best pairs identified so far, increasing the possibility that pairs that do not belong to the top- $k$  result set are filtered out.

### 4.2.2 TOPK-S-PPJ-S

TOPK-S-PPJ-S operates similarly with TOPK-S-PPJ-F. However, it uses a heuristic strategy in order to decide the order by which users are evaluated. User objects are placed in a spatial grid and each cell in the grid  $c$  is given a score by counting the amount of users whose object-sets belong to  $c$  or its adjacent cells. Users are then assigned a score by summing, for every object  $o$  associated with them, the score of the cell that  $o$  is contained in. Formally, cell scores  $s_c$  are calculated as follows:

$$s_c = |\cup_{c' \in G.getRelevantCells(c)} G.getUsers(c')|$$

where  $G$  is the spatial grid,  $c$  is a cell in the grid,  $G.getUsers$  returns the users with objects in  $c$  and  $G.getRelevantCells(c)$  returns the cells that are adjacent to  $c$  (including  $c$ ).

Accordingly, users are assigned scores  $s_u$  according to the

following formula:

$$s_u = \sum_{o \in D_u} s_{o_c}$$

$D_u$  denotes the object-set for user  $u$  and  $o_c$  describes the cell that object  $o$  is located in.

Therefore the rationale behind TOPK-S-PPJ-S is to start the search with users whose objects are placed in popular areas. This strategy aims at quickly identifying high scoring pairs, in order to increase the user similarity threshold quickly, and improve the efficiency of the filtering step.

### 4.2.3 TOPK-S-PPJ-P

The TOPK-S-PPJ-P algorithm introduces an additional filtering step. Users are selected in ascending order of the size of their object-sets. For every user  $u$ , we calculate an upper bound on the similarity score between  $u$  and any user  $u'$  that was selected in a previous iteration. To do so, we identify the objects from  $D_u$  that match with any object from  $D_{U'}$ , i.e. the union of the objects of every user that was selected in previous iterations. This is denoted as  $M(D_u, D_{U'})$ . This allows the calculation of an upper bound on  $\sigma(u, u')$  for every  $u'$  that was selected prior to  $u$ . Formally, this bound is calculated as follows:

$$\bar{\sigma}_u = \frac{|\bigcup_{u' \in U'} M(D_u, D_{u'})| + \max_{u' \in U'} |D_{u'}|}{|D_u| + \max_{u' \in U'} |D_{u'}|}$$

If the users are selected in an ascending order of the size of their object-sets, we show that  $\bar{\sigma}_u$  is an upper bound on the similarity score between  $u$  and a user  $u'$  with fewer or equal objects.

LEMMA 2. *Let a user  $u$  and a set of users  $U'$ . If for every user  $u' \in U'$   $|D_{u'}| \leq |D_u|$ , then it holds that  $\sigma(u, u') \leq \bar{\sigma}_u$ .*

PROOF. Let for any user  $u' \in U'$ ,  $m_u = |\bigcup_{u'' \in U'} M(D_u, D_{u''})|$ ,  $m_{u'} = |M(D_u, D_{u'})|$ ,  $m_{u''} = |M(D_{u'}, D_u)|$ ,  $d_u = |D_u|$  and  $d_{U'} = \max_{u'' \in U'} |D_{u''}|$ . Then, since  $m_u \geq m_{u'}$  and  $d_{u'} \geq m_{u'}$  it holds that

$$\frac{m_u + d_{u'}}{d_u + d_{u'}} \geq \sigma(u', u).$$

We show that:

$$\bar{\sigma}_u = \frac{m_u + d_{u''}}{d_u + d_{u''}} \geq \frac{m_u + d_{u'}}{d_u + d_{u'}}.$$

$$\begin{aligned} \frac{m_u + d_{u''}}{d_u + d_{u''}} &\geq \frac{m_u + d_{u'}}{d_u + d_{u'}} \Rightarrow \\ m_u \cdot d_u + m_u \cdot d_{u'} + d_{u''} \cdot d_u + d_{u''} \cdot d_{u'} &\geq \\ m_u \cdot d_u + m_u \cdot d_{u''} + d_{u'} \cdot d_u + d_{u'} \cdot d_{u''} &\Rightarrow \\ (d_{u''} - d_{u'}) \cdot d_u &\geq (d_{u''} - d_{u'}) \cdot m_u. \end{aligned}$$

This holds since  $d_{u''} \geq d_{u'}$  and  $d_u \geq m_u$ .  $\square$

In order to avoid the computation of exact similarity scores among user objects, and speed up the bound calculation process, we follow the same principle with the filtering step from the S-PPJ-F algorithm. We utilise the spatio-textual index described in Figure 3 and place in  $M(D_u, D_{U'})$  every object with a token that appears (due to a previously selected user) in the same or adjacent cell. This process provides a fast estimation of the  $\bar{\sigma}_u$  bound. Since this process overestimates, the resulting score is still an upper bound on the actual user

similarity score and can be used to prune the search space. This process yields relaxed bounds, which are irrelevant for a fixed user similarity threshold (as in the case of STPSJoin). However, it is useful with respect to the top- $k$  algorithms that quickly increase the user similarity threshold.

## 5. EXPERIMENTAL EVALUATION

Next, we present our experimental evaluation of the proposed algorithms. We first describe the datasets used and the parameters involved, and then we present the results.

### 5.1 Experimental Setup

**Datasets.** We have used three real-world datasets of spatio-textual web objects for our experiments. The *Flickr* dataset is derived from the Flickr Creative Commons dataset provided by Yahoo [35]. The whole dataset contains about 99.3 million images, about 49 million of which are geotagged. For our experiments, we concentrate on objects from the geographical boundaries of London, UK and we filtered out images that do not contain coordinates or tags as well as those that are created by stationary users. The resulting dataset contains 11,306 users and 1,116,348 objects. The *GeoText* dataset [18] is a geotagged microblog corpus available online.<sup>1</sup> It comprises 377,616 posts by 9,475 different users within the US. Finally, the *Twitter* dataset is a collection of geotagged tweets from the geographical area of London, UK, that we have collected and is part of the dataset used in [17]. It contains 9,724,579 tweets generated by 40,000 different users in 2014.

The NLTK toolkit<sup>2</sup> was employed to identify named entities from the text associated with the objects. The extracted named entities were used in combination with other related information, such as tokens, hashtags and mentions, as keywords associated with the respective objects. The characteristics of the three datasets are summarized in Table 1.

**Evaluation measures and parameters** The purpose of the experimental evaluation is to compare the performance of the proposed algorithms in terms of the execution time in different settings. For the case of the STPSJoin query, we investigate the effect of the following parameters: (a) the dataset size  $N$  in terms of number of users, (b) the query thresholds for spatial distance ( $\epsilon_{loc}$ ), textual similarity ( $\epsilon_{doc}$ ) and user similarity ( $\epsilon_u$ ) and (c) the *fanout* parameter of the R-tree structure. The effect of these parameters on the results of the STPSJoin query in the experimental datasets are described in Table 2. The largest deviation is observed on the Flickr dataset. This is consistent with the nature of this dataset, since popular POIs are often described using similar textual descriptions as well as photographs depicting these POIs are usually captured in nearby locations. On the contrary, the other datasets contain tweets, which are significantly more diverse both with respect to spatial locations and textual descriptions. For the top- $k$  STPSJoin query, we investigate the effect of the parameter  $k$  on execution time.

All algorithms were implemented in Java, and the experiments were executed on a machine with an Intel Core i5 2400 CPU and 16GB RAM, running on Ubuntu Linux. During the experiments, 15GB of memory were allocated to the JVM. All plots report running time in a logarithmic scale.

<sup>1</sup><http://www.ark.cs.cmu.edu/GeoText/>

<sup>2</sup><http://www.nltk.org/>

Dataset	Objects	Users	Tokens per Object	Objects per Token	Objects per User
Twitter	9,724,579	40,000	2.08 (1.43)	6.25 (141.80)	243.11 (344.86)
Flickr	1,116,348	11,306	8.04 (8.15)	26.41 (1,191.09)	98.73 (419.92)
GeoText	165,733	9,461	1.64 (1.01)	3.53 (39.36)	17.52 (12.99)

Table 1: Experimentation datasets, number of objects and users, and mean (standard deviation) for descriptive metrics.

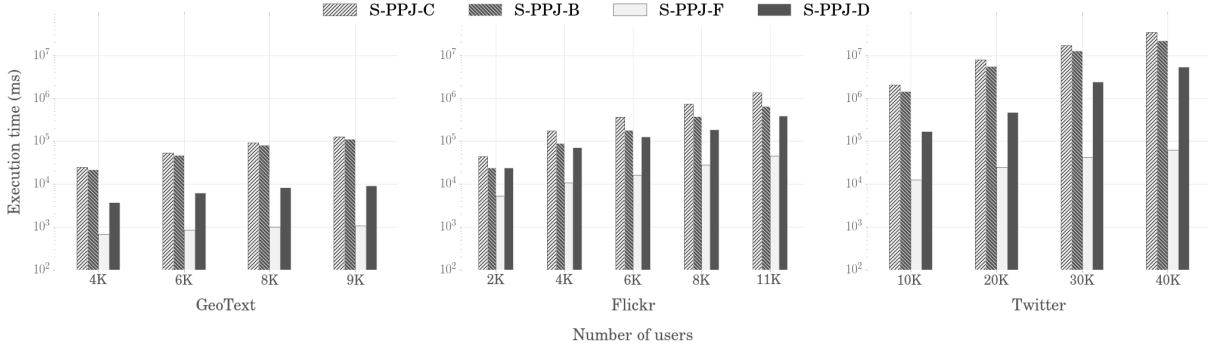


Figure 4: Scalability results for the GeoText, Flickr and Twitter datasets (parameter defaults: GeoText:  $\epsilon_{loc} = 0.001$ ,  $\epsilon_{doc} = 0.3$ ,  $\epsilon_u = 0.3$ ; Flickr  $\epsilon_{loc} = 0.001$ ,  $\epsilon_{doc} = 0.6$ ,  $\epsilon_u = 0.6$ ; Twitter:  $\epsilon_{loc} = 0.001$ ,  $\epsilon_{doc} = 0.4$ ,  $\epsilon_u = 0.4$ ).

	GeoText	Flickr	Twitter
<b>Scalability</b>	27.00 (8.51)	54.20 (46.22)	13.50 (6.54)
<b>Tuning</b>	18.00 (36.90)	326.00 (633.89)	14.14 (9.98)

Table 2: Mean (std-dev) of result-set sizes

## 5.2 Scalability

The scalability experiments evaluate the performance of our methods in datasets of different sizes. We divided the Twitter, Flickr and GeoText datasets for variable numbers of users. The resulting datasets range from 4,000 users with 72,094 objects to 40,000 users with 9,724,579 objects. Different parameter values are used for different datasets, in order to account for different sizes and token selectivity across datasets. Lower thresholds are set for the GeoText dataset in order to avoid empty result sets, whereas higher thresholds are set for the Flickr dataset to account for the higher similarity between user objects. This is due to the fact that a large amount of Flickr photos represent popular POIs that are described by similar textual content and are geo-located close to the location of the corresponding POIs.

Figure 4 shows the scalability evaluation results. The results clearly show that S-PPJ-F outperforms the other methods by several orders of magnitude. This is consistent for all datasets, irrespective of size. The efficiency of S-PPJ-F compared to the other approaches is attributed to the effect of the filter and refinement scheme, in combination with the suitability of the dynamic grid partitioning over the objects. The grid partitioning is tailor made to the spatial threshold parameter  $\epsilon_{loc}$ , which allows the search for matching objects to be limited exclusively in adjacent cells. Additionally, the inverted lists maintained within each cell of the grid, allow the effective filtering of candidate user pairs associated with spatially similar, but textually diverse, objects.

The performance of S-PPJ-B does not compare favourably against S-PPJ-F. This result is expected since S-PPJ-F builds on S-PPJ-B by leveraging the filter and refinement scheme.

Nevertheless, the comparison between S-PPJ-B and S-PPJ-C allows the evaluation of the early termination strategy, as well as the traversal mechanism, differentiating S-PPJ-B from S-PPJ-C. The results indicate that S-PPJ-B offers a consistent improvement in execution time compared to S-PPJ-C, confirming that the proposed techniques manage to prune the search space for similarity search among two point sets.

Finally, the results show that S-PPJ-D outperforms the baseline methods, but it is not comparable to the grid-based S-PPJ-F, which follows the same principles. The discrepancy in execution time can be attributed to the use of different spatial indexes. The data driven-partitioning imposed by the R-tree is independent of the spatial threshold given as a parameter to the STPSJoin query. As a result, the imposed partitioning leads to an ineffective division of the database. Inspection of the performance of S-PPJ-D shows that both partition size and overlap may lead to subpar performance, since objects within large partitions tend to be spatially irrelevant, and overlaps require the evaluation of multiple join operations. We revisit this issue in Section 5.4.

## 5.3 Effect of similarity thresholds

In the following experiments, we vary the parameters and evaluate the proposed algorithms for different combinations of textual, spatial and user similarity thresholds. Similar to the scalability experiments, different ranges in threshold values are used across datasets.

Figure 5 presents the results. We observe that the dominant parameter is the spatial threshold  $\epsilon_{loc}$ . High values on  $\epsilon_{loc}$  result in significantly higher execution times. This is particularly obvious for the Flickr and Twitter datasets, which contain significantly larger amounts of objects. When the spatial distance threshold reaches metropolitan level distances, the majority of the objects fall into adjacent partitions. As a result, the filtering step of S-PPJ-F and S-PPJ-D returns a high number of candidates. In these cases, the overhead imposed by the additional indexing maintained by S-PPJ-F and S-PPJ-D is apparent. We observe a peak in the case of S-PPJ-D, especially with respect to the Flickr

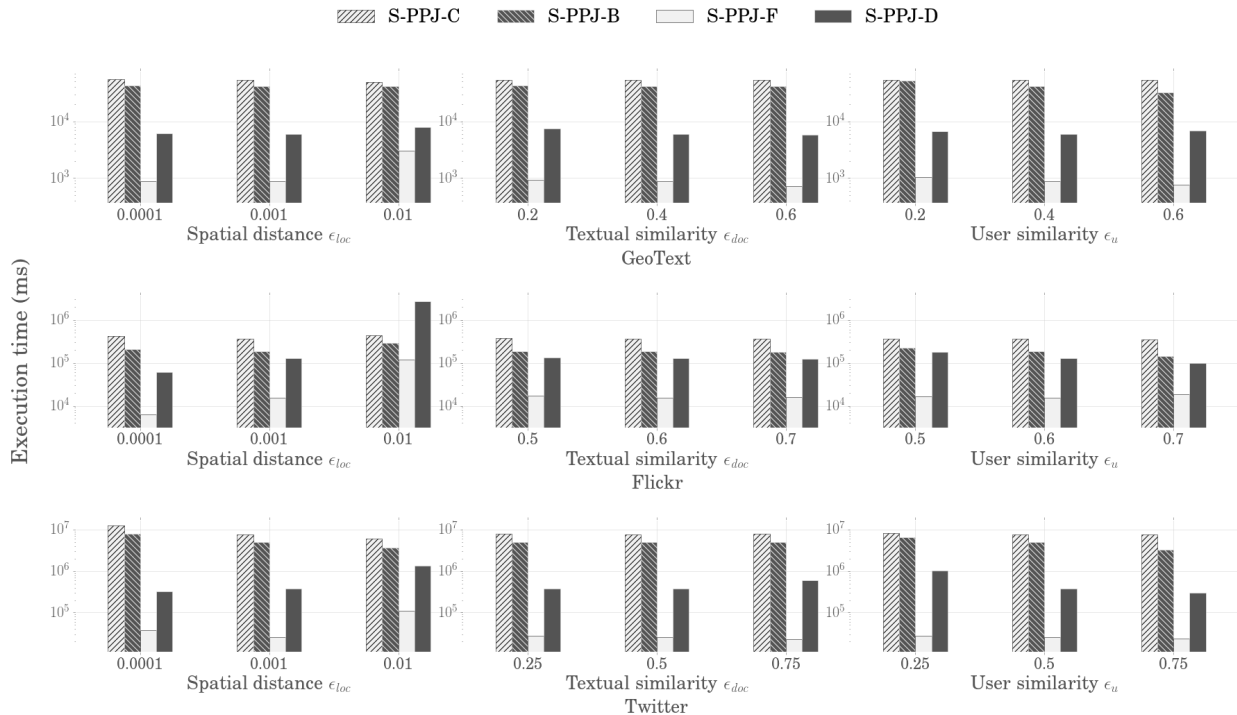


Figure 5: Results for varying similarity thresholds (GeoText: 6,000 users, 107,941 objects; Flickr: 6,000 users, 597,008 objects; Twitter: 20,000 users, 4,988,090 objects).

dataset. In this case, inspection shows that the R-tree partitioning does not manage to result in an efficient partition of the object database.

This does not apply for GeoText, mainly due to the fact that the objects in GeoText are scattered in the significantly larger area of the whole of USA. The results show that the proposed pruning strategies are highly functional in combination with a grid-based partitioning scheme. S-PPJ-F outperforms the other methods in every scenario, and apart from the case of the Flickr dataset with  $\epsilon_{loc} = 0.01$ , its performance is independent of the parameter values.

#### 5.4 Effect of Fanout on S-PPJ-D

An important parameter for data partitioning schemes based on R-trees is the *fanout* parameter. This parameter is associated with the number of objects that reside in a node of the R-tree. The effect of the fanout parameter on the performance of S-PPJ-D is twofold. First of all, S-PPJ-D executes a spatial distance join in order to identify spatial relations among the leaf nodes of the tree, which are treated by the algorithm as spatial data partitions. Since the fanout parameter affects both the depth and the breadth of the R-tree, it also affects the performance of the spatial join. Second, S-PPJ-D is built on top of the partitioning imposed by the leaf nodes. Therefore, the fanout affects both the number and the size of leaf nodes, which are relevant to S-PPJ-D.

In order to experimentally evaluate the effects of the fanout parameter, experiments with values ranging from 50 to 250 were conducted. The results are shown in Figure 6. The results verify that S-PPJ-D is sensitive to the fanout value. Even though no single fanout value achieves the best results in all datasets, we observe that an appropriate fanout value for STPSJoin queries falls within the range of 100 to 200.

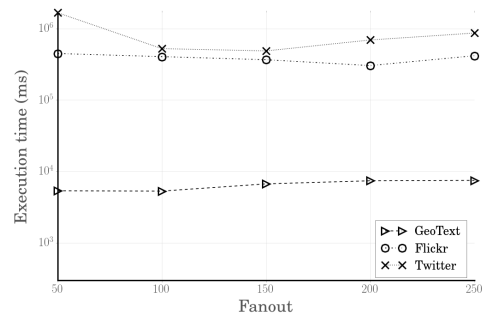


Figure 6: Tuning the R-Tree fanout parameter.

#### 5.5 Evaluation of top-k STPSJoin

In the following, we evaluate the proposed algorithms for the top- $k$  STPSJoin query. We vary the result size  $k$  in order to study the behaviour of the algorithms. Figure 7 shows the results of the experiments. The baseline TOPK-S-PPJ-F is competitive and is the better performing algorithm in the Flickr dataset. The poor performance of TOPK-S-PPJ-S shows that the simple ordering of the users based exclusively on the size of their object-sets is more efficient than the statistical approach that it follows, compared to the overhead that the additional computation imposes. TOPK-S-PPJ-P exploits an additional pruning step and offers a better performance in the cases of GeoText and Twitter. In the case of Flickr, while it is outperformed by TOPK-S-PPJ-F, it remains competitive. This is due to the fact that the Flickr dataset contains objects with very high similarity, mainly because of the nature of the Flickr service (ie. people describe popular places with nearly the same keywords). The

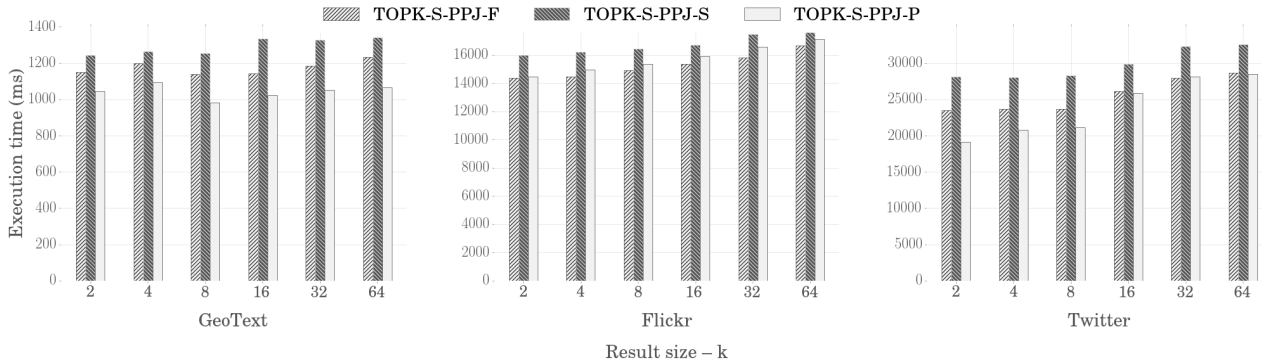


Figure 7: Results for the top- $k$  STPSJoin algorithms on GeoText, Flickr and Twitter datasets with varying  $k$  (GeoText:  $\epsilon_{loc} = 0.001$ ,  $\epsilon_{doc} = 0.3$ ,  $\epsilon_u = 0.3$ ; Flickr  $\epsilon_{loc} = 0.001$ ,  $\epsilon_{doc} = 0.6$ ,  $\epsilon_u = 0.6$ ; Twitter:  $\epsilon_{loc} = 0.001$ ,  $\epsilon_{doc} = 0.4$ ,  $\epsilon_u = 0.4$ ).

very high textual similarity between objects leads to high user similarities. Therefore, the additional filtering step of TOPK-S-PPJ-P cannot disqualify large numbers of user pairs.

## 5.6 Parameter Tuning

The STPSJoin query requires  $\epsilon_{loc}$ ,  $\epsilon_{doc}$ , and  $\epsilon_u$  to be provided as input. The values of these thresholds define what is “near” in terms of spatial, textual and user similarity, and are determined by the nature of the data and task in hand. In order to tackle situations in which there is no prior knowledge that can be used to determine the values of these thresholds, we present an automated process in order to discover sensible thresholds. In this case, the necessary input is an acceptable result set size.

The tuning algorithm is initialised with relaxed initial thresholds. Our experiments show that these can be predefined values regardless of the dataset. The only requirement is that they are relaxed enough to guarantee a result-set larger than the input value. Threshold steps are calculated as fractions of the initial values.

The algorithm follows a greedy strategy. Initially, it executes S-PPJ-F using the starting thresholds and populates a result-set. Then, the process traverses the parameter combination space in a depth-first manner. At any given step the algorithm selects probabilistically which parameter to tighten (an alternative strategy is to modify the least modified threshold). Tightening the parameters monotonically decreases the results-set that was the outcome of the previous step. As a result, S-PPJ-F is not executed again for the different parameters. Instead, PPJ-C is used to identify which pairs from the previous step adhere to the new thresholds. If the result set size reaches the desired value, the algorithm stops and the current threshold values are returned. If a step brings about thresholds that yield no results, the process backtracks to the previous step, and an alternative threshold is tightened.

Result size	S-PPJ-F		Tuning	
	5	25	50	
GeoText	2,229	145 (8)	124 (4)	110 (3)
Flickr	24,363	738 (23)	693 (17)	1,066 (10)
Twitter	82,412	1,278 (10)	3,085 (7)	1,439 (2)

Table 3: Parameter tuning including S-PPJ-F time and tuning time in ms (number of iterations) for varying result-sets.

Table 3 shows the time required for parameter tuning after the initial execution of S-PPJ-F. Initial thresholds were the minimum thresholds used in Section 5.3, and the datasets were those with the minimum number of users used in Section 5.2. It is worth noting that the initial running time of S-PPJ-F consumes a significant amount of the overall time.

## 5.7 Summary

Our experimentation verifies the superiority of the proposed algorithms for the treatment of the STPSJoin query, in terms of execution time for all datasets used. The pruning strategy employed by S-PPJ-F manages to significantly boost the algorithm’s performance. Furthermore, the S-PPJ-D algorithm which induces data partitioning is efficient enough to be considered as a viable choice in cases when the data are already partitioned with an R-tree (or any other data partitioning method). Our experimentation shows that S-PPJ-F can be directly modified to efficiently handle the top- $k$  STPSJoin query variant. Nevertheless, we propose an additional pruning strategy in top- $k$  STPSJoin that performs even better with datasets of lower degrees of similarity. Even in the case of the Flickr dataset, which does not fall into this category, TOPK-S-PPJ-P achieves competitive results. The experimental analysis is conducted on three real datasets of varied size (the two of them are publicly available) and different parameter settings have been examined in order to reach to the optimum configurations. The results show that the algorithms scale well in very large databases and can therefore be used effectively in real-world scenarios.

## 6. CONCLUSIONS

This paper studies the problem of similarity search on spatio-textual point sets. We formally define this problem as STPSJoin and present its top- $k$  variant. STPSJoin queries identify pairs of similar users, with respect to web documents such as tweets and photographs associated with these users.

In order to efficiently process (top- $k$ ) STPSJoin queries, we propose algorithms that leverage different spatio-textual indexes, and integrate early termination pruning mechanisms with filter and refinement approaches. We conducted large-scale experiments on real-world datasets for multiple values on the problem parameters. The better performing algorithm S-PPJ-F is orders of magnitude more efficient in terms of execution time than the baseline methods. Finally, S-PPJ-D shows improvement over the baseline methods for

the case of data-driven partitioned databases, even though it is significantly outperformed by S-PPJ-F. For the case of the top- $k$  STPSJoin query, the TOPK-S-PPJ-P algorithm offers the best results in the majority of the datasets, but also remains competitive in the case of the Flickr dataset, which contains significantly larger amounts of similar spatio-textual objects.

In the future, we plan to focus on distributed architectures in order to further enhance the efficiency of our methods. Furthermore, we intend to integrate additional characteristics in STPSJoin queries, which are often associated with web objects, such as temporal information.

## Acknowledgements

This work was partially supported by the EU Project City.Risks (H2020-FCT-2014-653747).

## 7 REFERENCES

- [1] M. D. Adelfio, S. Nutanong, and H. Samet. Searching web documents as location sets. In *ACM SIGSPATIAL*, 2011.
- [2] M. D. Adelfio, S. Nutanong, and H. Samet. Similarity search on a large collection of point sets. In *ACM SIGSPATIAL*, 2011.
- [3] J. Ballesteros, A. Cary, and N. Rishe. Spsjoin: Parallel spatial similarity joins. In *ACM SIGSPATIAL*, 2011.
- [4] J. Bao, Y. Zheng, D. Wilkie, and M. F. Mokbel. Recommendations in location-based social networks: a survey. *GeoInformatica*, 19(3):525–565, 2015.
- [5] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.
- [6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R\*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.
- [7] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *PVLDB*, pages 1–12, 2012.
- [8] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD*, 1993.
- [9] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD*, 2011.
- [10] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.
- [11] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.
- [12] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, 2006.
- [13] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top- $k$  most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.
- [14] J. Cranshaw, E. Toch, J. I. Hong, A. Kittur, and N. M. Sadeh. Bridging the gap between physical location and online social networks. In *UbiComp*, 2010.
- [15] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, 2008.
- [16] P. DeScioli, R. Kurzban, E. N. Koch, and D. Liben-Nowell. Best friends alliances, friend ranking, and the myspace social network. *Perspectives on Psychological Science*, 6(1):6–8, 2011.
- [17] H. Efstathiades, D. Antoniadis, G. Pallis, and M. D. Dikaiakos. Identification of key locations based on online social network activity. In *ASONAM*, 2015.
- [18] J. Eisenstein, B. O’Connor, N. A. Smith, and E. P. Xing. A latent variable model for geographic lexical variation. In *EMNLP*, 2010.
- [19] T. Eiter and H. Mannila. Distance measures for point sets and their computation. *Acta Informatica*, 34(2):109–133, 1997.
- [20] J. Fan, G. Li, L. Zhou, S. Chen, and J. Hu. Seal: Spatio-textual similarity search. *PVLDB*, 5(9):824–835, May 2012.
- [21] A. Goel, A. Sharma, D. Wang, and Z. Yin. Discovering similar users on twitter. In *MLG*, 2013.
- [22] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.
- [23] E. H. Jacox and H. Samet. Spatial join techniques. *TODS*, 32(1):7, 2007.
- [24] Y. Jiang, G. Li, J. Feng, and W.-S. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.
- [25] Q. Li, Y. Zheng, X. Xie, Y. Chen, W. Liu, and W. Ma. Mining user similarity based on location history. In *ACM SIGSPATIAL*, 2008.
- [26] Z. Li, K. C. K. Lee, B. Zheng, W.-C. Lee, D. Lee, and X. Wang. IR-tree: An efficient index for geographic document search. *TKDE*, 23(4):585–599, 2011.
- [27] S. Liu, G. Li, and J. Feng. Star-join: Spatio-textual similarity join. In *CIKM*, 2012.
- [28] S. Liu, G. Li, and J. Feng. A prefix-filter based method for spatio-textual similarity join. *TKDE*, 26(10):2354–2367, 2014.
- [29] C. Long, R. C.-W. Wong, K. Wang, and A. W.-C. Fu. Collective spatial keyword queries: a distance owner-driven approach. In *SIGMOD*, 2013.
- [30] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *SSTD*. Springer-Verlag, 2001.
- [31] J. Ramon and M. Bruynooghe. A polynomial time computable metric between point sets. *Acta Informatica*, 37(10):765–780, 2001.
- [32] J. Rao, J. Lin, and H. Samet. Partitioning strategies for spatio-textual similarity join. In *BigSpatial*, 2014.
- [33] J. a. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvgå. Efficient processing of top- $k$  spatial keyword queries. In *SSTD*, 2011.
- [34] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, 2004.
- [35] B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L.-J. Li. The new data and new challenges in multimedia research. *arXiv preprint arXiv:1503.01817*, 2015.
- [36] S. Vaid, C. B. Jones, H. Joho, and M. S. Spatio-textual indexing for geographical search on the web. In *SSTD*, 2005.
- [37] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*, pages 85–96, 2012.
- [38] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *TODS*, 36(3):15:1–15:41, Aug. 2011.
- [39] X. Xiao, Y. Zheng, Q. Luo, and X. Xie. Finding similar users using category-based location history. In *ACM SIGSPATIAL*, 2010.
- [40] D. Zhang, Y. M. Chee, A. Mondal, A. K. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, 2009.
- [41] D. Zhang, B. C. Ooi, and A. K. Tung. Locating mapped resources in Web 2.0. In *ICDE*, 2010.
- [42] Y. Zheng, L. Zhang, Z. Ma, X. Xie, and W. Ma. Recommending friends and locations based on individual location history. *TWEB*, 5(1):5, 2011.
- [43] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, 2005.



# Nearest Window Cluster Queries

Chen-Che Huang, Jiun-Long Huang, Tsung-Ching Liang, Jun-Zhe Wang,

Wen-Yuah Shih and Wang-Chien Lee<sup>†</sup>

Department of Computer Science

National Chiao Tung University, Hsinchu, Taiwan, ROC

<sup>†</sup>Department of Computer Science and Engineering

The Pennsylvania State University, University Park, PA 16802, USA

E-mail: cchuang.cs95g@nctu.edu.tw, jlhuang@cs.nctu.edu.tw, dy93\_@hotmail.com, jzwang@cs.nctu.edu.tw, wen21318@hotmail.com, wlee@cse.psu.edu

## ABSTRACT

In this paper, we study a novel type of spatial queries, namely Nearest Window Cluster (NWC) queries. For a given query location  $q$ , NWC  $(q, l, w, n)$  retrieves  $n$  objects within a window of length  $l$  and width  $w$ , where the distance between the query location  $q$  to these  $n$  objects is the shortest. To facilitate efficient NWC query processing, we identify several properties and accordingly develop an NWC algorithm. Moreover, we propose several optimization techniques to further reduce the search cost. To validate our ideas, we conduct a comprehensive performance evaluation using both real and synthetic datasets. Experimental results show that the proposed NWC algorithm, along with the optimization techniques, is very efficient under various datasets and parameter settings.

**Keywords:** Nearest window cluster query, spatial query processing, location-based service, spatial database.

## 1. INTRODUCTION

Spatial queries have received tremendous attention from the research community in past decades. In the past several years, owing to the emerging location-based services, a number of new spatial queries have been proposed to meet various application needs [16] [11][7][13][22][9]. However, many interesting/important applications still are not well supported by existing spatial queries. The following is an example.

- Suppose Bob is attending a business meeting in a foreign city. He wishes to buy some souvenirs for his family. With only a rough idea of what to buy (e.g., some local-brand clothes), he would like to search for some, say  $n$ , nearby clothes shops which are close to each other in a small area so he can walk around these clothes shops to find the souvenirs.

In this example, Bob aims to find the nearest area with sufficient choices (i.e.,  $n$  clothes shops) clustered in the area so he can go around to compare products and prices and even have fun doing some bargaining. Figure 1 illustrates the example above where each

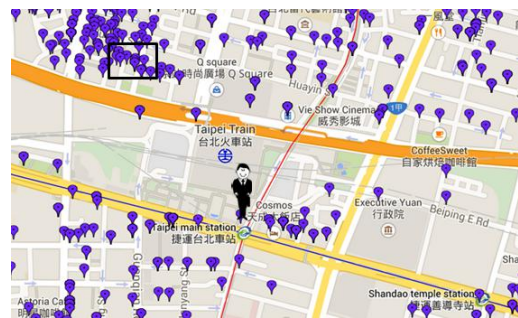


Figure 1: Example of a nearest window cluster query.

bubble indicates a clothes shop. Ideally, a location-based service would be able to suggest the clothes shops within the window back to Bob. Unfortunately, existing spatial queries such as kNN, trip-planning queries [15] and collective spatial keyword queries [2] do not meet Bob's need effectively and efficiently. To the best of our knowledge, no previous work on spatial queries address the query problem arising in the example scenario.

In this paper, we propose a novel type of spatial queries, namely *Nearest Window Cluster (NWC)* queries that finds a clustered of objects located in a spatial window nearest to a query point, e.g., the query issuer's location. Given a query point  $q$ , window length  $l$  and window width  $w$ , and the number of objects to find  $n$ , NWC  $(q, l, w, n)$  returns  $n$  objects located within a window of length  $l$  and width  $w$ , where the *distance* from these  $n$  objects to  $q$  is the shortest.<sup>1</sup>

Two main challenges arise in processing the NWC queries. First, while the locations of data objects are given, the locations of *qualified* windows are unknown in advance.<sup>2</sup> Second, the number of qualified windows may be huge. To address these challenges, we identify several properties that allow us to find qualified windows quickly. Accordingly, we develop an NWC algorithm that iteratively finds the nearest qualified window to return the objects within the qualified window. To facilitate efficient visits of data objects, we adopt R-tree to index the data objects.

Observing that the bottleneck of the NWC algorithm lies in finding the nearest qualified window, we propose four optimization techniques, namely *search region reduction (SRR)*, *distance-based pruning (DIP)*, *density-based pruning (DEP)* and *incremental window query processing (IWP)* to accelerate searching the nearest

©2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

<sup>1</sup>We will discuss distance measures in Section 2.

<sup>2</sup>A window is considered as qualified if it contains  $n$  objects.

qualified window, thereby improving the efficiency of the NWC algorithm. The SRR technique takes advantage of the distance of the best objects found so far to shrink the search regions of qualified windows and prune some objects from further processing. The DIP technique uses the distance of the best objects found so far to safely prune the index nodes distant from the query location  $q$ , thereby reducing the I/O cost. Inspired by the *clustering effect* of spatial objects [1] (e.g., certain objects such as clothes shops are usually clustered in some hot areas), the DEP technique maintains a density grid of the whole object space that records the number of objects in each grid cell. With the density grid, the DEP technique is able to prune the index nodes with insufficient objects and to avoid redundant window queries incurred during query processing. Finally, to alleviate the cost of window queries generated by the NWC algorithm, the IWP technique inserts *backward* and *overlapping pointers* into the leaf nodes and some intermediate nodes of the R-tree. With these pointers, fewer index nodes are involved for window queries, thereby achieving better performance.

The rest of this paper is organized as follows. In Section 2, we review related work and present the problem definition. In Section 3, we elaborate the properties of NWC queries and develop the NWC algorithm based on these properties. In addition, four optimization techniques are proposed to accelerate NWC query processing. The performance of the NWC algorithm is analyzed in Section 4, while the experimental results of the NWC algorithm are reported in Section 5. Finally, we conclude this paper in Section 6.

## 2. PRELIMINARIES

### 2.1 Problem Formulation and Transformation

Based on the application scenario discussed earlier, we consider a set of static data objects, denoted as  $P$ , in two-dimensional Euclidean space.<sup>3</sup> The nearest window cluster query is formally defined as below.

**Definition 1 (Nearest Window Cluster (NWC) Query)** Given a query point  $q$ , a spatial window area specified by length  $l$  and width  $w$ , and the number of data objects  $n$ , a nearest window cluster query  $NWC(q, l, w, n)$  aims to retrieve  $n$  objects satisfying the following criteria:

1. these  $n$  objects are clustered within a spatial window of length  $l$  and width  $w$ , and
2. the *distance* from the window with these  $n$  objects reside to  $q$  is the shortest among all other windows with  $n$  objects satisfying (1).

To facilitate our discussion, we define the notion of qualified window with respect to an NWC query as follows.

**Definition 2 (Qualified Window)** Given an NWC query  $(q, l, w, n)$ , a *qualified window*, denoted as  $qwin$ , is a window of length  $l$  and width  $w$  that contains data objects  $S_{qwin} = \{p_1, p_2, \dots, p_{|S_{qwin}|}\} \subseteq P$ , where  $|S_{qwin}| \geq n$ .

Let  $MINDIST(q, qwin)$  be the distance from  $q$  to the closest point in the qualified window  $qwin$  covering  $\{p_1, p_2, \dots, p_n\}$ . As the distance measure mentioned above aims to measure the distance between  $q$  and these  $n$  objects, denoted as  $\{p_1, p_2, \dots, p_n\}$ , we consider the following in our algorithm.

<sup>3</sup>We focus on 2D data objects in accordance with real-world applications. The proposed algorithms could be easily adjusted to three dimensional space.

- Minimum distance:

$$dist_{min}(q, \{p_1, p_2, \dots, p_n\}) = \min_{i=1,2,\dots,n} dist(q, p_i) \quad (1)$$

- Maximum distance:

$$dist_{max}(q, \{p_1, p_2, \dots, p_n\}) = \max_{i=1,2,\dots,n} dist(q, p_i) \quad (2)$$

- Average distance:

$$dist_{avg}(q, \{p_1, p_2, \dots, p_n\}) = \frac{1}{n} \sum_{i=1}^n dist(q, p_i) \quad (3)$$

- Nearest window distance:

$$dist_{nearest}(q, \{p_1, p_2, \dots, p_n\}) = \min_{\forall qwin \in qwins} (MINDIST(q, qwin)), \quad (4)$$

where  $qwins$  is a set of all qualified windows containing  $\{p_1, p_2, \dots, p_n\}$ .

With the above definitions, we can transform the problem of NWC query processing as below.

**Problem Transformation.** When  $MINDIST(q, qwin)$  is always smaller than or equal to  $dist(q, \{p_1, p_2, \dots, p_n\})$ , the processing of an NWC query can be performed by incrementally finding the next nearest qualified window to  $q$  and using the distance of the best objects found so far, denoted as  $dist_{best}$ , to prune the search space until no better qualified window is found.

Specifically, we can solve the NWC query by the following steps.

Step 1: Set  $dist_{best}$  to  $\infty$  and set  $objs$  to  $\emptyset$ .

Step 2: Find the nearest qualified window  $qwin$ .

Step 3: If  $qwin$  is found and  $MINDIST(q, qwin) < dist_{best}$ , perform Steps 4-6. Otherwise, go to Step 7.

Step 4: Let  $\{p_1, p_2, \dots, p_n\}$  be the  $n$  objects in  $qwin$  of the shortest distance to  $q$ .

Step 5: If  $dist(q, \{p_1, p_2, \dots, p_n\}) < dist_{best}$ , set  $objs$  to  $\{p_1, p_2, \dots, p_n\}$  and  $dist_{best}$  to  $dist(q, \{p_1, p_2, \dots, p_n\})$ .

Step 6: Find the next nearest qualified window  $qwin$  and go to Step 3.

Step 7: Return  $objs$ .

Clearly,  $MINDIST(q, qwin)$  is always smaller than or equal to minimum, maximum, average and nearest window distances between  $q$  and  $\{p_1, p_2, \dots, p_n\}$  (see Equations (1), (2), (3) and (4), respectively). As the bottleneck of the above procedure is in finding the nearest qualified window, we will focus on nearest qualified window search for the rest of this paper. The advantages of using nearest qualified window search to process an NWC query are twofold. First, the above procedure can be used for other distance measures as long as  $MINDIST(q, qwin)$  can be used as the lower bound of the employed distance measures. Second, the properties of nearest qualified window search can be used to efficiently process NWC queries.

**Table 1: List of used symbols**

Symbol	Description
$P$	Data object set
$T_p$	R-tree on $P$
$q$	query location
$n$	the desired number of data objects
$SR_p$	search region of object $p$
$qwin_p$	best qualified window of $p$
$S_{qwin_p}$	the set of the data objects inside $qwin_p$
$ S_{qwin_p} $	the cardinality of $S_{qwin_p}$
$dist(q, \{p_1, p_2, \dots, p_n\})$	the distance between $q$ and $\{p_1, p_2, \dots, p_n\}$
$objs$	the best objects found so far
$dist_{best}$	the distance of the best objects found so far

## 2.2 Related Work

In the past two decades, a large number of spatial queries have been proposed and studied by researchers in the database community. Here we focus on the variants of NN queries due to their relevance to our work. Constrained NN [8] queries find the nearest neighbor(s) constrained to a specific region instead of the entire data space. Nearest surround (NS) queries [13] consider the object orientation and retrieve the nearest neighbors at different angles with respect to the query location  $q$ . A reverse NN (RNN) query [12][21] finds all the data objects with  $q$  as their nearest neighbor. RkNN queries [19][3] search for all the data objects that have  $q$  as one of their  $k$  nearest neighbors. The ranked RNN (RRNN) query [14] allows to identify and rank the  $t$  data objects most influenced by  $q$ . Different from typical RNN queries, RFN [22] queries find the objects that have  $q$  as their *furthest* neighbor. With RFN queries, a plant producing hazardous gas could be constructed at a point with fewest residents being affected.

Group NN (GNN) queries [16] (also known as aggregate NN [17] queries) retrieve the data object(s) with the smallest sum of distance to  $Q$  where  $Q$  is the set of query points. GNN queries are useful when a group of friends intend to find a meeting restaurant with the minimum distances to them. Group nearest group (GNS) queries [6], a generic version of GNN queries, return more objects for gathering to reduce the traveling costs of users. With data object set  $P$  and target object set  $Q$ , optimal-location-selection (OLS) queries [9] find  $q \in Q$  outside a specific region  $R$  with maximal optimality where the optimality metric is determined by to the number of data objects in  $R$  and the accumulated distances to  $q$ . OLS query is useful for applications like optimal lifeguard station selection. Range NN (RangeNN) queries [11][5] search for the NNs for every point in a rectangle. It could be used to offer location privacy and computation saving. Given a specified window size, the maximizing range sum (MaxRS) problem [4] is to find the window  $win$  with the largest sum of weights of all objects within  $win$  among all candidate windows of the specified window size. Although bearing similarity to the proposed NWC queries, the MaxRS problem does not consider any query location and thus is naturally different from the proposed NWC query.

## 3. PROCESSING NEAREST WINDOW CLUSTER QUERIES

In this section, we elaborate the NWC query processing based on the procedure of nearest qualified window search mentioned in Section 2.1. First, we identify some unique properties of the nearest qualified windows in Section 3.1. Based on these properties, we develop an NWC algorithm to find the nearest qualified window of

the NWC query in Section 3.2. To improve the efficiency of NWC search, in Section 3.3, we further propose four optimization techniques, including (i) *search region reduction*, (ii) *distance-based pruning*, (iii) *density-based pruning* and (iv) *incremental window query processing* to reduce the search cost. To facilitate better readability, the symbols used throughout this paper are listed in Table 1.

### 3.1 Properties of the Nearest Qualified Windows

In this section, we introduce the following properties regarding the nearest qualified window to facilitate efficient NWC search.

**Lemma 1** The nearest qualified window of an NWC query, or one of its equivalent qualified windows, has at least one object on one vertical edge and at least one object on one horizontal edge.

PROOF. The proof is omitted for the interest of space.  $\square$

A qualified window  $qwin$  is said to be *generated* by a data object  $p$  when  $p$  is on at least one edge of  $qwin$ . Therefore, we use data objects as the basis to *generate* qualified windows and consider only those qualified windows generated by some data objects for NWC query processing. In other words, we consider only those qualified windows generated by data objects on one vertical or horizontal edge based on the relative position of  $q$  and object  $p$ . Furthermore, based on Lemma 1, we can utilize the lying quadrant of  $p$  (with  $q$  as the origin) to determine that we merely need to evaluate the qualified windows with  $p$  on the right or left edge (top or bottom edge) by the following two observations.

1. Consider a qualified window, say  $qwin$ , generated by data object  $p$  on one of the vertical edges (denoted by  $e_R$  or  $e_L$ ). When  $p$  is in the first or fourth quadrant with respect to the origin  $q$ ,  $p$  must be on the right edge  $e_R$  of  $qwin$ ; when  $p$  is in the second or third quadrant,  $p$  must be on the left edge  $e_L$  of  $qwin$ .
2. Consider a qualified window, say  $qwin$ , generated by data object  $p$  on one of the horizontal edges (denoted by  $e_T$  or  $e_B$ ). When  $p$  is in the first or second quadrant with respect to the origin  $q$ ,  $p$  must be on the top edge  $e_T$  of  $qwin$ ; when  $p$  is in the third or fourth quadrant,  $p$  must be on the bottom edge  $e_B$  of  $qwin$ .

### 3.2 NWC Algorithm

With the above properties and the steps discussed in Section 2.1, we present the NWC algorithm which incrementally finds the next qualified window to  $q$  and uses the distance of the best object found so far ( $dist_{best}$ ) to prune the search space until no better qualified window is found. Since the bottleneck of the NWC algorithm lies in finding the nearest qualified window, we focus on nearest qualified window search. The idea of the NWC algorithm is as follows. The NWC algorithm visits all data objects based on their distance to the query location  $q$  in ascending order. To facilitate efficient visits of data objects, we adopt R-tree to index the data objects. When visiting an object  $p$ , the NWC algorithm creates the *search region* for object  $p$  (denoted as  $SR_p$ ) to cover all qualified windows generated by  $p$ , and then find all qualified windows generated by  $p$  (i.e., qualified windows within  $SR_p$ ). When a qualified window, say  $qwin_p$ , is discovered and  $MINDIST(q, qwin_p) < dist_{best}$ , the NWC algorithm retrieves the  $n$  objects, say  $\{p_1, p_2, \dots, p_n\}$ , in  $qwin_p$  of the shortest distance to  $q$  and checks whether  $dist(q, \{p_1, p_2, \dots, p_n\}) < dist_{best}$ . If so, the NWC algorithm sets  $objs$  and  $dist_{best}$  to  $\{p_1, p_2, \dots, p_n\}$  and  $dist(q, \{p_1, p_2, \dots, p_n\})$ , respectively. The

NWC algorithm repeats the above steps until no better qualified window is found.

We now discuss how to build  $SR_p$  covering all qualified windows generated by  $p$ . According to the observations in Section 3.1,  $p$  has to be on the vertical and horizontal edges in order to guarantee all potential qualified windows generated by  $p$  are contained in  $SR_p$ . In fact, we consider  $p$  only on either the vertical or the horizontal edge for building  $SR_p$  because the other case is handled naturally while processing other objects. Specifically, if  $SR_p$  is built on the condition that  $p$  is on the vertical edge, the other potential qualified windows on the condition that  $p$  is on the horizontal edge will be contained within the search region of another object  $p'$  on the vertical edge (i.e.,  $SR_{p'}$ ). To avoid redundant computation, when visiting data object  $p$ , we consider  $p$  only on the vertical edge for  $SR_p$  construction. Due to space limitation, we mainly explain the case where  $p$  is in the first quadrant with respect to the origin  $q$  (i.e.,  $p$  is on the right edge) since the other cases are able to be addressed similarly.

With  $p$  on the right edge, the four vertexes  $v_1, v_2, v_3$  and  $v_4$  of  $SR_p$  are defined below, where  $x_p$  and  $y_p$  are the x and y coordinates of  $p$ , respectively.

$$\begin{aligned} x_{v_1} &= x_p - l & y_{v_1} &= y_p - w & x_{v_2} &= x_p & y_{v_2} &= y_p - w \\ x_{v_3} &= x_p & y_{v_3} &= y_p + w & x_{v_4} &= x_p - l & y_{v_4} &= y_p + w \end{aligned}$$

It is obvious that all windows generated by  $p$  are inside  $SR_p$ . Therefore, the NWC algorithm is able to evaluate only data objects inside  $SR_p$  for identifying the qualified windows generated by  $p$ . To do so, the NWC algorithm retrieves all the objects within  $SR_p$  by issuing a window query with  $SR_p$  as the query window. To efficiently obtain the qualified windows generated by  $p$ , the NWC algorithm first reorders the objects in  $S_{SR_p}$  based on their y coordinates in ascending order and then skips the objects with y coordinates lower than  $p$ . The data objects below  $p$  are skipped because their associated qualified windows would not contain  $p$ . Finally, the NWC algorithm sequentially visits each object remaining in  $S_{SR_p}$ , say  $p'$ , to consider the window, say  $qwin_{p'}$ , with  $p$  on the right edge and  $p'$  on the top edge for each  $p'$ .

When a window  $qwin_p$  is considered, the NWC algorithm evaluates whether  $qwin_p$  is qualified. If the number of objects within  $qwin_p$  is smaller than  $n$  (i.e.,  $|S_{qwin_p}| < n$ ),  $qwin_p$  is not qualified and the NWC algorithm skips  $qwin_p$ . When  $qwin_p$  is qualified, (i.e.,  $|S_{qwin_p}| \geq n$ ), the NWC algorithm retrieves the  $n$  objects, say  $\{p_1, p_2, \dots, p_n\}$ , in  $qwin_p$  of the shortest distance to  $q$ . If  $dist(q, \{p_1, p_2, \dots, p_n\}) < dist_{best}$ , the NWC algorithm sets  $dist_{best}$  and  $objs$  to  $dist(q, \{p_1, p_2, \dots, p_n\})$  and  $\{p_1, p_2, \dots, p_n\}$ , respectively. Otherwise, the NWC algorithm skips  $qwin_p$  and considers another window generated by  $p$ .

We use the example in Figure 2 to illustrate the process of identifying qualified windows in  $SR_p$ . Let's consider an intermediate step, where  $p_5$  is the candidate data object under examination. To find  $qwin_{p_5}$ , the NWC algorithm first builds  $SR_{p_5}$  based on the residing quadrant of  $p_5$  to  $q$  and retrieves all data objects within  $SR_{p_5}$ . Since  $p_5$  is in the first quadrant, the NWC algorithm sorts all data objects within  $SR_{p_5}$  based on their y coordinates in ascending order. As shown, the y coordinate of  $p_4$  is smaller than that of  $p_5$ , so the NWC algorithm skips  $p_4$  and sequentially considers  $p_5, p_6$ , and  $p_7$  on the top edge. Suppose that the desired number of objects in a qualified window is three (i.e.,  $n = 3$ ). When  $p_5$  is considered, the window with  $p_5$  on the right and top edges is not qualified since this window contains only two objects. When  $p_6$  is evaluated, the window with  $p_5$  on the right edge and  $p_6$  on

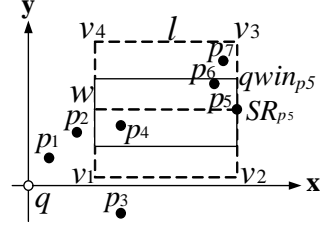


Figure 2: Discover  $qwin_p$  from  $SR_p$ .

the top edge is set to  $qwin_{p_5}$  since this window contains three objects. The NWC algorithm retrieves the three objects of the shortest distance to  $q$  from  $qwin_{p_5}$  (i.e.,  $\{p_4, p_5, p_6\}$ ) and checks whether  $dist(q, \{p_4, p_5, p_6\}) < dist_{best}$ . If so,  $dist_{best}$  and  $objs$  are set to  $dist(q, \{p_4, p_5, p_6\})$  and  $\{p_4, p_5, p_6\}$ , respectively. Then the NWC algorithm continues to find the next window in  $SR_p$  until all windows in  $SR_p$  have been evaluated.

### 3.3 Optimization Techniques

While being able to answer NWC queries, the NWC algorithm suffers from costly and redundant evaluations of objects and index nodes. In light of this, we design the following optimization techniques to mitigate the cost of NWC search.

- *Search region reduction (SRR)*: The search region reduction technique exploits the distance between  $q$  and the best objects found so far (i.e.,  $dist_{best}$ ) to reduce the search regions of the remaining data objects, thereby saving the cost of qualified window discovery. Besides, with  $dist_{best}$ , SRR tries to exclude those objects that are unlikely to create closer qualified windows to  $q$ , eliminating the redundant evaluation of those objects.
- *Distance-based pruning (DIP)*: The distance-based pruning technique takes advantage of  $dist_{best}$  to save the access to the index nodes that are too distant to create closer qualified windows, thereby achieving reductions in I/O cost.
- *Density-based pruning (DEP)*: We propose to build a density grid that maintains the number of objects residing in each grid cell. With the density grid, we propose a density-based pruning technique to prune the index nodes with insufficient objects (compared with the numbers of objects requested by NWC queries) to eliminate unnecessary I/O cost. In addition, DEP is able to prune some redundant window queries which do not produce any qualified window.
- *Incremental window query processing (IWP)*: To reduce the I/O cost to process the window queries generated by the NWC algorithm, we propose to enhance the R-tree by inserting *backward pointers* and *overlapping points* into the leaf nodes and some intermediate nodes, respectively. We design the incremental window query processing technique to use these backward pointers and overlapping pointers to efficiently process these window queries with less I/O costs.

The details of these optimization techniques are described in the following subsections.

#### 3.3.1 Search Region Reduction

To identify each qualified window  $qwin_p$  generated by object  $p$ , the NWC algorithm issues a window query with the search region

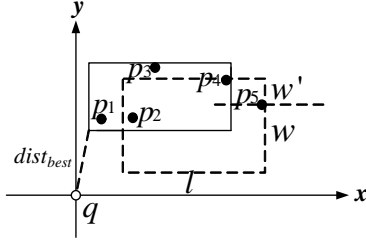


Figure 3: Reduced  $SR_p$ s with  $dist_{best}$ .

$SR_p$  of  $p$  as the query window. Obviously, the smaller  $SR_p$  is, the lower the I/O cost is. Thus, we propose the search region reduction technique (abbreviated as SRR) to 1) avoid evaluations of unnecessary search regions or 2) reduce the sizes of the search regions by exploiting the following two observations.

- Object  $p$  may be so distant to  $q$  that all qualified windows generated by  $p$  having distances greater than  $dist_{best}$ . For such object  $p$ , building  $SR_p$  is unnecessary and thus no window query is issued.
- The qualified windows in the specific portions of  $SR_p$  may never have a distance smaller than  $dist_{best}$ . Thus,  $SR_p$  can be shrunk, leading to smaller query windows.

With  $dist_{best}$ , when processing object  $p$ , SRR first checks whether the creation of  $SR_p$  is necessary based on  $x_p$  or  $y_p$ . Specifically, let the coordinates of the bottom-left vertex of  $SR_p$ , say  $v_1$ , be  $(x_{v_1}, y_{v_1})$ . When the distance from  $q$  to  $v_1$  is greater than  $dist_{best}$ , there is no need to build  $SR_p$  since no qualified window generated by  $p$  will be closer to  $q$  than  $dist_{best}$ . Thus, the reduced search region of  $p$ , denoted as  $SR'_p$ , is set to be empty. Otherwise, the building of  $SR_p$  is necessary. Then, SRR tries to utilize  $dist_{best}$  to shrink  $SR_p$  into a smaller search region  $SR'_p$  so that the distance of each qualified window in  $SR'_p$  is shorter than  $dist_{best}$ . The coordinates of the four vertices of  $SR'_p$  are calculated below.

$$\begin{aligned} x'_{v_1} &= x_p - l & y'_{v_1} &= y_p - w & x'_{v_2} &= x_p & y'_{v_2} &= y_p - w \\ x'_{v_3} &= x_p & y'_{v_3} &= y_p + w' & x'_{v_4} &= x_p - l & y'_{v_4} &= y_p + w' \end{aligned}$$

It is clear that  $w'$  is the maximal value making the following two equations satisfied.

$$0 \leq w' \leq w \quad (5)$$

$$(x_p - l - x_q)^2 + (y_p + w' - w - y_q)^2 \leq dist(q, qwin_{best})^2 \quad (6)$$

Equation (5) indicates that  $p$  should be within  $SR'_p$ , while Equation (6) indicates that the minimum distance from  $q$  to each window of length  $l$  and width  $w$  in  $SR'_p$  should be shorter than  $dist_{best}$ . According to Equations (5) and (6), the value of  $w'$  is

$$w' = \min \left( w, \sqrt{dist_{best}^2 - (x_p - l - x_q)^2} - (y_p - w - y_q) \right).$$

In Figure 3, SSR helps to reduce  $SR_{p_5}$  with  $w'$  being only

$$\sqrt{dist_{best}^2 - (x_{p_5} - l - x_q)^2} - (y_{p_5} - w - y_q).$$

### 3.3.2 Distance-Based Pruning

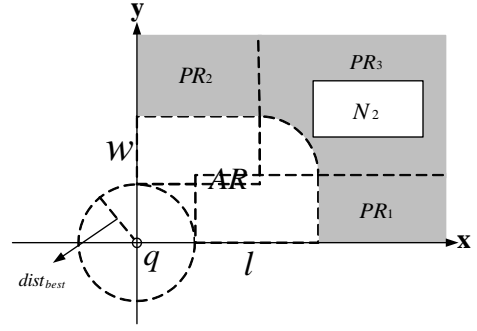


Figure 4: Node  $N_2$  can be safely pruned based on  $dist_{best}$ .

In addition to reducing search regions of objects,  $dist_{best}$  is able to be used to safely prune some index nodes as long as all qualified windows created by any object inside these pruned index nodes are guaranteed to be of distances to  $q$  greater than  $dist_{best}$ . For example, index node  $N_2$  in Figure 4 can be safely pruned because any qualified window created by any object inside  $N_2$  is unlikely to have the distance to  $q$  smaller than  $dist_{best}$ . Based on this observation, we propose the distance-based pruning technique (abbreviated as DIP) to facilitate the pruning of unvisited index nodes. DIP defines a *pruning region* (denoted as  $PR$ ) based on  $dist_{best}$ ,  $l$ , and  $w$ . If an unvisited index node  $N$  is completely inside  $PR$ ,  $N$  is able to be safely pruned without being visited. With  $dist_{best}$ ,  $PR$  is defined as below.

$$\begin{aligned} PR_1 &= \{(x, y) | x \geq x_q + dist_{best} + l, y_q \leq y \leq y_q + w\} \\ PR_2 &= \{(x, y) | x_q \leq x \leq x_q + l, y \geq y_q + dist_{best} + w\} \\ PR_3 &= \{(x, y) | x \geq x_q + l, y \geq y_q + w \\ &\quad \wedge (x - (x_q + l))^2 + (y - (y_q + w))^2 \leq dist_{best}^2\} \\ PR &= PR_1 \cup PR_2 \cup PR_3 \end{aligned} \quad (7)$$

As defined in Equation (7),  $PR$  is composed of three subregions:  $PR_1$ ,  $PR_2$ , and  $PR_3$ .  $PR_1$  guarantees that each qualified window generated by each object in  $PR_1$  has the vertical distance to  $q$  greater than  $dist_{best}$ , while  $PR_2$  ensures that the horizontal distance between  $q$  and each qualified window generated by each object in  $PR_2$  is greater than or equal to  $dist_{best}$ .  $PR_3$  excludes the region where an object can be used to generate a qualified window with the distance to  $q$  smaller than  $dist_{best}$ . Hence, if index node  $N$  is totally contained within  $PR$ , no closer qualified window can be generated by each object in  $N$  and thus  $N$  can be safely pruned to reduce I/O cost.

### 3.3.3 Density-Based Pruning

The spatial clustering effect in practice leads objects like clothes shops to be clustered in certain areas. Thus, an index node may be so sparse that all windows generated by each object in the index node are not qualified. With this observation, we devise the density-based pruning technique (abbreviated as DEP) to save I/O cost by avoiding visits to sparse index nodes. To facilitate DEP, the whole object space is divided into a  $g_d \times g_d$  density grid and each grid cell is associated with the number of objects within the cell. When processing an index node, DEP first extends the MBR of the index node and checks whether the summation of the objects in the grid

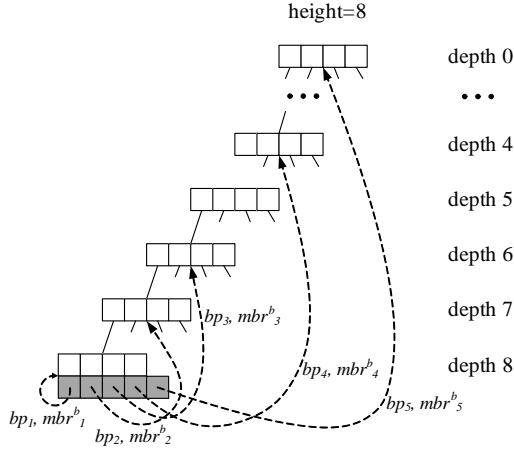


Figure 5: An illustrative example of backward pointers

cells intersecting the extended MBR is smaller than  $n$ . If so, this index node will be pruned. We now describe the method to extend an MBR in the first quadrant with respect to the origin  $q$ , and the other cases can be handled in a similar manner. Suppose that the counterclockwise order of the four vertices of the MBR is  $v_1, v_2, v_3$  and  $v_4$ , and  $v_1$  is the bottom-left vertex of the MBR. The MBR can be *extended* as follows to ensure that all windows generated by each object within the MBR must be within the extended MBR.

$$\begin{aligned} x'_{v_1} &= x_{v_1} - l & y'_{v_1} &= y_{v_1} - w & x'_{v_2} &= x_{v_2} & y'_{v_2} &= y_{v_2} - w \\ x'_{v_3} &= x_{v_3} & y'_{v_3} &= y_{v_3} + w' & x'_{v_4} &= x_{v_4} - l & y'_{v_4} &= y_{v_4} + w' \end{aligned}$$

Besides, it is possible that the search region of an object does not contain enough objects to generate any qualified window. DEP also utilizes this observation to eliminate the window queries of such objects with the aid of the density grid. Specifically, before issuing a window query for the currently processing object  $p$ , DEP checks whether the summation of the objects in the grid cells intersecting the search region  $SR_p$  is smaller than  $n$ . If so, DEP cancels the window query since  $SR_p$  never contains any qualified window.

### 3.3.4 Incremental Window Query Processing

As mentioned in Section 3.2, the NWC algorithm issues a window query with query window  $SR_p$  ( $SR'_p$  when SRR is used) to identify the qualified windows generated by object  $p$ . With traditional window query processing, solving a window query requires to access the R-tree from root node to some leaf nodes. However, we observe that some index nodes in the R-tree, especially the index nodes close to the root node, are usually unnecessary to visit when processing the window queries issued by the NWC algorithm. In view of this, we propose to add some *backward pointers* into each leaf node of the R-tree and some *overlapping pointers* into the nodes pointed by backward pointers. Based on backward and overlapping pointers, we design an incremental window query processing technique (abbreviated as IWP) to allow window queries to be processed from intermediate nodes instead of root node, thereby reducing the I/O cost.

Consider an R-tree with height  $h$ . Each leaf node is with depth  $h$ . Suppose that there are  $r$  backward pointers (denoted as  $(bp_1, mbr_1^b), (bp_2, mbr_2^b), \dots, (bp_r, mbr_r^b)$ ) for each leaf node. It is obvious that  $SR'_p$  is very likely to be totally covered by the intermediate nodes

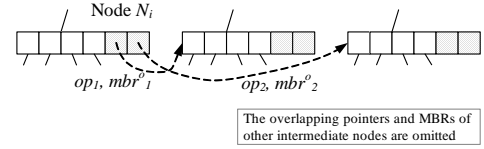


Figure 6: An illustrative example of overlapping pointers

close to the leaf node containing  $p$ . Thus, inspired by Exponential Index [20], for a leaf node  $s$ , the backward pointers are set by the following rules.

1. The first backward pointer  $bp_1$  points to  $s$ .
2.  $bp_i$ , where  $1 < i < r$ , points to the ancestor of  $s$  with depth  $h - 2^{i-2}$ .
3. The last backward pointer  $bp_r$  points to the root node.
4.  $mbr_i^b$ , where  $1 \leq i \leq r$ , is the MBR of the node pointed by  $bp_i$ .

According to the third rule,  $r$  is the smallest integer making the following equation true.

$$h - 2^{r-2} \leq 0$$

Thus, we can obtain that  $r = \lceil \log_2 h + 2 \rceil$ .

Figure 5 shows an illustrative example of backward pointers. Only part of an R-tree with height eight is shown for better readability. Since  $h = 8$ , each leaf node is of  $r = \lceil \log_2 8 + 2 \rceil = 5$  backward pointers (i.e., gray squares in Figure 5).  $bp_1$  points to the leaf node, while  $bp_2, bp_3, bp_4$  and  $bp_5$  point to the ancestors of the leaf node with depth 7, 6, 4 and 0, respectively.

Since most variants of R-tree do not guarantee the MBRs of intermediate nodes in the same depth level to be non-overlapped, using only backward pointers to incrementally process window queries may lead to wrong query results. Therefore, some overlapping pointers are needed for the nodes pointed by backward pointers (except the root node) to facilitate correct incremental window query processing. We use the example in Figure 6 to illustrate the overlapping pointers. Since overlapping with two other nodes with the same depth, the node  $N_i$  is of two overlapping pointers ( $(op_1, mbr_1^o), (op_2, mbr_2^o)$ ) where  $op_j$  is the pointer pointing to the  $j$ -th intermediate node with the same depth as  $N_i$  and overlapping with  $N_i$ , and  $mbr_j^o$  is the MBR of the node pointed by  $op_j$ .

With backward pointers and overlapping pointers, a window query can be incrementally processed as follows. When an object  $p$  is inserted into the priority queue  $PQ$ , the backward pointers of the leaf node where  $p$  is stored are also inserted into  $PQ$  along with  $p$ . When processing the window query with  $SR'_p$  as the query window, instead of searching from the root node of the R-tree, IWP retrieves the corresponding backward pointers, finds the smallest value of  $i$  so that  $SR'_p$  is totally covered by  $mbr_i^b$ , and searches from the node pointed by  $bp_i$ . In addition, for each overlapping pointer  $op_j$  of the node pointed by  $bp_i$ , IWP also executes the window query from the node pointed by  $op_j$  when  $mbr_j^o$  overlaps with the query window.

Finally, the NWC algorithm enhanced with optimizations as well as some companion functions are given in Algorithms 1, 2 and 3.

## 3.4 $k$ Nearest Window Cluster Query Processing

An NWC query is to provide the user with an area with  $n$  objects (choices). In practice, it is possible that the user even would like

---

**Algorithm 1:** NWC algorithm enhanced with optimizations

---

**Data:** NWC query  $(q, l, w, n)$ , priority queue  $PQ$   
**Result:** a set of  $n$  objects

```
1  $dist_{best} \leftarrow \infty, objs \leftarrow \emptyset, PR \leftarrow \emptyset;$ 
2  $PQ.enqueue(Root\ of\ T_p);$ 
3 while  $|PQ| > 0$  do
4    $p \leftarrow PQ.dequeue();$ 
5   if  $p$  is an index node then
6      $MBR_p \leftarrow$  the MBR of  $p$ ;
7     Extend  $MBR_p$  as  $MBR'_p$  based on DEP;
8     if  $MBR_p - PR \neq \emptyset$  and  $isPrunedByDEP(MBR'_p)$  is FALSE then
9       for each child  $c$  of  $p$  do
10         $PQ.enqueue(c);$ 
11   else
12     //  $p$  is an object
13     determine the lying quadrant of  $p$  with respect to  $q$ ;
14     determine whether  $p$  is on the left or right edge;
15     build  $SR_p$  and reduce  $SR_p$  as  $SR'_p$  by SRR;
16     if  $SR'_p \neq \emptyset$  and  $isPrunedByDEP(SR'_p)$  is FALSE then
17        $S_{SR'_p} \leftarrow IWP(SR'_p);$ 
18       sort  $S_{SR'_p}$  in ascending/descending order of  $y$  coordinates;
19       remove  $p_i$  from  $S_{SR'_p}$  when the  $y$  coordinate of  $p_i$  is
20         smaller/larger than the  $y$  coordinate of  $p$ ;
21       for  $i = 1$  to  $|S_{SR'_p}|$  do
22         build  $qwin_p$  by setting  $p$  on the right/left edge and  $p_i$  on the
23         top/bottom edge;
24         if  $|S_{qwin_p}| \geq n$  and  $MINDIST(q, qwin_p) < dist_{best}$  then
25           let  $\{p_1, p_2, \dots, p_n\}$  be the  $n$  objects in  $qwin_p$  of the
26           shortest distance to  $q$ ;
27           if  $dist(q, \{p_1, p_2, \dots, p_n\}) < dist_{best}$  then
28              $dist_{best} \leftarrow dist(q, \{p_1, p_2, \dots, p_n\});$ 
29              $objs \leftarrow \{p_1, p_2, \dots, p_n\};$ 
30             update  $PR$  accordingly;
31 return  $objs;$ 
```

---

---

**Algorithm 2:** Function  $isPrunedByDEP(rect, n)$ 

---

**Data:** a rectangle  $rect$   
**Result:** whether  $rect$  is pruned by DEP

```
1  $ub \leftarrow 0;$ 
2 for each cell  $cell$  in the density grid do
3   if  $cell$  intersects  $rect$  then
4      $ub \leftarrow ub +$  the number of objects in  $cell$ ;
5 if  $ub < n$  then
6   return TRUE;
7 else
8   return FALSE;
```

---

---

**Algorithm 3:** Function  $IWP(rect)$ 

---

**Data:** a rectangle  $rect$   
**Result:** the set of objects within  $rect$

```
1  $result \leftarrow \emptyset, nodes \leftarrow \emptyset;$ 
2 fetch the backward pointers associated with  $p$ ;
3 for  $i=1$  to  $r$  do
4   if  $rect \subseteq mbr_i^p$  then
5      $N_i \leftarrow$  the index node pointed by  $bp_i$ ;
6     insert  $N_i$  into  $nodes$ ;
7     break;
8 for each overlapping pointer  $op_j$  of  $N_i$  do
9   if  $mbr_j^p \cap rect \neq \emptyset$  then
10    insert the index node pointed by  $op_j$  into  $nodes$ ;
11 for each node  $N$  in  $nodes$  do
12   perform traditional window query processing with query window  $rect$ 
13   starting from  $N$ ;
14   insert the resultant objects into  $result$ ;
15 return  $result;$ 
```

---

to retrieve multiple areas so that the user is able to pick a proper area from them. To satisfy such needs, we extend NWC queries to  $k$ NWC queries that enable users to retrieve  $k$  object groups where each group consists of  $n$  objects located within a window of length  $l$  and width  $w$ . It is obvious that the user is not willing to get  $k$  object groups and each pair of groups consist of almost the same objects. Thus, we introduce a new parameter  $m$  which allows a user to specify the maximal number of identical objects allowed in each pair of groups. The formal definition of a  $k$ NWC query is given below.

**Definition 3 ( $k$ NWC Query)** Given a query location  $q$ , a spatial window area specified by length  $l$  and width  $w$ , the number of data objects  $n$ , and the maximal number of identical objects, say  $m$ , in any two object groups, a  $k$  nearest window cluster ( $k$ NWC) query  $(k, q, l, w, n, m)$  retrieves  $k$  object groups,  $objs_1, objs_2, \dots, objs_k$ , satisfying the following criteria.

- Each object group consists of  $n$  objects within a window of length  $l$  and width  $w$ .
- For each pair of object groups,  $objs_1$  and  $objs_2$ , there are at most  $m$  objects in both  $objs_1$  and  $objs_2$  (i.e.,  $|objs_1 \cap objs_2| \leq m$ ).
- The  $k$  object groups are ordered by their distances to  $q$  in ascending order. That is,  $dist(q, objs_i) \leq dist(q, objs_j) \forall i, j$  where  $i < j$ .
- For each group of  $n$  objects, say  $obj^j$ , within a window of length  $l$  and width  $w$  and  $obj^j \neq obj_i$  where  $i = 1, 2, \dots, k$ , at least one of the following conditions should be satisfied.
  1.  $dist(q, objs_k) \leq dist(q, obj^j)$ .
  2. There exists an integer  $i$  ( $1 \leq i \leq k$ ) so that  $dist(q, obj_i) \leq dist(q, obj^j)$  and  $|obj_i \cap obj^j| > m$ .

It is obvious that Lemma 1 also holds for  $k$ NWC queries. To answer  $k$ NWC queries, similar to NWC queries, we are allowed to consider only those qualified windows with objects on their vertical and horizontal edges. Based on Lemma 1, we now design the  $k$ NWC algorithm, an extension of the NWC algorithm, to support  $k$ NWC queries as below. The  $k$ NWC algorithm maintains the  $k$  object groups  $\{objs_1, objs_2, \dots, objs_k\}$  found so far in  $groups$  and sorts the  $k$  object groups by their distances to query location  $q$  in ascending order. The optimization techniques proposed in Section 3.3 can also be used to mitigate the I/O cost of identifying the nearest qualified windows of a given  $k$ NWC query. Specifically, when  $k$  object groups are obtained, the distance between  $q$  and the  $k$ -th object group (i.e.,  $dist(q, objs_k)$ ) is employed in SRR to reduce the search regions of remaining objects and in DIP to prune remaining index nodes. When finding a qualified window  $qwin_p$ , the  $k$ NWC algorithm performs the following steps to handle  $qwin_p$ .

- Step 1: Let  $objs_p = \{p_1, p_2, \dots, p_n\}$  be the  $n$  objects in  $qwin_p$  of the shortest distance to  $q$ .
- Step 2: Scan  $groups$  in reverse order to find the first object group, say  $objs_i$ , which is of distance shorter than  $objs_p$ . In case that no such  $i$  exists, set  $i$  to 0 and go to Step 4. If  $i = k$ , drop  $objs_p$  and stop this procedure.
- Step 3: Check whether  $|objs_p \cap objs_j| \leq m$  for each  $objs_j$ ,  $j = 1, 2, \dots, i$ . If not, drop  $objs_p$  and stop this procedure.
- Step 4: Remove  $objs_k$  from  $groups$  and insert  $objs_p$  into  $groups$  at position  $i + 1$  (i.e., as  $objs_{i+1} = objs_p$ ).

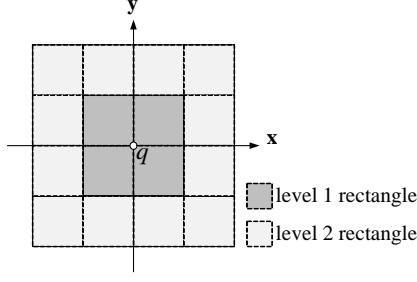


Figure 7: Illustration of analysis

- Step 5: Check whether  $|objs_p \cap objs_j| \leq m$  for each  $objs_j$ ,  $j = i+2, i+3, \dots, k-1$ . Remove  $objs_j$  from *groups* when  $|objs_p \cap objs_j| > m$ .

## 4. THEORETICAL ANALYSIS

### 4.1 Time Complexity Analysis of NWC Algorithm

In this section, we develop a cost model to analyze the I/O cost of the NWC algorithm. To facilitate the following discussion, as shown in Figure 7, the space is divided into multiple disjoint rectangles of height  $l$  and width  $w$ . Since the NWC algorithm visits the objects according to their distances to  $q$  in ascending order [10], it is very likely that the NWC algorithm visits the objects in all level- $i$  rectangles and then visits the objects in all level- $i+1$  rectangles until the best objects are found.

We assume that the objects in an area are Poisson distributed with mean  $\lambda$ . For simplicity, we also assume that the probability that a window is not qualified is independent of the probability of any other window. Thus, the average number of objects in a window of length  $l$  and width  $w$  is  $\lambda \times l \times w$  and the probability that a window is not qualified is

$$\mathbf{P} = P\{X \leq n-1\} = e^{-\lambda \times l \times w} \sum_{i=0}^{n-1} \frac{(\lambda \times l \times w)^i}{i!}. \quad (8)$$

An object in a level- $i$  rectangle is called a level- $i$  object, while a qualified window generated by a level- $i$  object is called a level- $i$  qualified window. Due to the effect of DIP, we also assume that the objects within a level- $i$  qualified window can be verified as the best objects only when 1) there is no level- $j$  qualified window, where  $j = 1, 2, \dots, i-1$ , and 2) all level- $i$  qualified windows have been checked.

Consider an object  $p$ . The NWC algorithm issues a window query to retrieve all objects within the search region of  $p$  (i.e.,  $SR_p$ ) and the average number of objects in the upper-half of  $SR_p$  is  $\lambda \times l \times w$ . For each object  $p'$  on the upper-half of  $SR_p$ , the NWC algorithm then generates one window with  $p$  on the right edge and  $p'$  on the top edge and checks whether the window is qualified or not. Thus, the average number of windows generated by an object is  $\lambda \times l \times w$ , and the probability that an object cannot generate a qualified window is  $\mathbf{P}^{\lambda \times l \times w}$ .

Let  $\mathbf{N}(i)$  be the number of level- $i$  rectangles and it is clear that

$$\mathbf{N}(i) = (2i)^2 - (2(i-1))^2 = 8i - 4. \quad (9)$$

We can obtain that the average number of level- $i$  objects is  $\mathbf{N}(i) \times \lambda \times l \times w$ . Denote the probability that there is no level- $i$  qualified window (that is, all windows generated by all level- $i$  objects are not qualified) to be  $\mathbf{Q}(i)$ . Since there is no level-0 rectangle,  $\mathbf{Q}(0) = 1$ .

For each positive integer  $i$ , we can derive that

$$\mathbf{Q}(i) = \prod_{j=1}^i \mathbf{P}^{\lambda \times l \times w} = \mathbf{P}^{\mathbf{N}(i) \times (\lambda \times l \times w)^2}.$$

The probability that the qualified window consisting of the best objects is a level- $i$  qualified window is  $(1 - \mathbf{Q}(i)) \times \prod_{j=0}^{i-1} \mathbf{Q}(j)$ .

Consider the case that the qualified window consisting of the best objects is a level- $i$  qualified window. All level- $j$  objects, where  $j = 1, 2, \dots, i$ , should be retrieved. Let the average number of objects to be retrieved in this case be  $\mathbf{O}(i)$ . We can obtain

$$\mathbf{O}(i) = \sum_{j=1}^i \mathbf{N}(j) \times \lambda \times l \times w = 2 \times i^2 \times \lambda \times l \times w. \quad (10)$$

Since these objects are retrieved in accordance with their distances to  $q$ , the average I/O cost to retrieve them is close to the average I/O cost of using  $K$  nearest neighbor query to retrieve  $\mathbf{O}(i)$  objects. For each retrieved object  $p$ , the NWC algorithm issues a window query to get the objects within  $SR_p$ , and thus, the average number of issued window queries is  $\mathbf{O}(i)$ . Let the average I/O cost to use  $K$  nearest neighbor query to retrieve  $K$  objects be  $\mathbf{KNN}(K)$ , and let the average I/O cost of a window query of length  $l$  and width  $w$  be  $\mathbf{WIN}(l, w)$ . The average I/O cost when the qualified window consisting of the best objects is a level- $i$  qualified window is  $\mathbf{O}(i) \times \mathbf{WIN}(l, w) + \mathbf{KNN}(\mathbf{O}(i))$ .

Suppose that the whole space contains at most level- $MaxLV$  rectangles. The average I/O cost of the NWC algorithm is

$$\sum_{i=1}^{MaxLV} \left\{ \left[ (1 - \mathbf{Q}(i)) \times \prod_{j=0}^{i-1} \mathbf{Q}(j) \right] \times \left[ \mathbf{O}(i) \times \mathbf{WIN}(l, w) + \mathbf{KNN}(\mathbf{O}(i)) \right] \right\},$$

where  $\mathbf{WIN}(l, w)$  can be obtained from [18] and  $\mathbf{KNN}(K)$  can be obtained from [10].

### 4.2 Time Complexity Analysis of $k$ NWC Algorithm

Let the probability that a window is not qualified be  $\mathbf{P}$  where  $\mathbf{P}$  can be obtained by Equation (8). Let the probability that a qualified window consists of at most  $m$  identical objects with each object group in *groups* be  $Pr(m, k)$ . Thus, the probability that the objects within a window cannot be inserted into *groups* be  $\mathbf{P}' = 1 - [(1 - \mathbf{P}) \times Pr(m, k)]$ . Suppose that the object group within a level- $i$  qualified window will not be removed from *groups* due to the insertion of any object group within a level- $j$  qualified window where  $j > i$ . Therefore, when the object group within a level- $i$  qualified window becomes the  $k$ -th nearest object group, the  $k$ NWC algorithm will terminate when all object groups within level- $i$  qualified windows have been checked.

We now derive the probability that the  $k$ -th nearest object group is within a level- $i$  qualified window. Due to the effect of SRR and DIP, we assume that the  $k$ -th nearest object group is within a level- $i$  qualified window only when 1) the number of object groups within level- $j$ , where  $j = 1, 2, \dots, i-1$ , qualified windows inserted into  $qwins_{best}$  is smaller than  $k$  and 2) the number of object groups within level- $j$ , where  $j = 1, 2, \dots, i$ , qualified windows inserted into  $qwins_{best}$  is larger than or equal to  $k$ .

As shown in Equation (10), the average number of all level- $j$  objects, where  $j = 1, 2, \dots, i$ , is  $\mathbf{O}(i)$ . Since the average number of windows generated by an object is  $\lambda \times l \times w$ , the probability that there are  $a$  object group within level- $j$ , where  $j = 1, 2, \dots, i$ , qualified windows inserted into *groups* is

$$\mathbf{R}(i, a) = C_a^{\mathbf{O}(i) \times \lambda \times l \times w} \times (1 - \mathbf{P}')^a \times \mathbf{P}'^{\mathbf{O}(i) \times \lambda \times l \times w - a}.$$



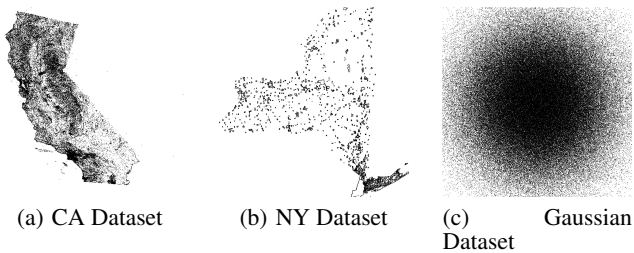


Figure 8: Distributions of the used datasets

Table 2: Description of datasets

Dataset	Cardinality	Description
CA	62,556	Real places in California
NY	255,259	Real places in New York
Gaussian	250,000	Generated by Gaussian distribution

According to Equation (9), there are  $N(i)$  level- $i$  rectangles and the average number of level- $i$  objects is  $N(i) \times \lambda \times l \times w$ . Therefore, the probability that there are at least  $b$  object groups within level- $i$  qualified windows inserted into *groups* is

$$S(i, b) = 1 - \sum_{d=1}^{b-1} \left[ C_d^{N(i) \times (\lambda \times l \times w)^2} \times (1 - \mathbf{P}')^d \times \mathbf{P}'^{N(i) \times (\lambda \times l \times w)^2 - d} \right].$$

Therefore, the probability that the  $k$ -th nearest object group within a level- $i$  qualified window is

$$\sum_{j=0}^{k-1} \left[ \mathbf{R}(i-1, j) \times S(i, k-j) \right].$$

When the  $k$ -th nearest object group is within a level- $i$  qualified window, the  $k$ NWC algorithm visits all objects in all level- $j$  rectangles, where  $j = 1, 2, \dots, i$ . Similar to the derivations in the previous subsection, the average I/O cost when the  $k$ -th nearest object group is a level- $i$  qualified window is  $\mathbf{O}(i) \times \mathbf{WIN}(l, w) + \mathbf{KNN}(\mathbf{O}(i))$ . Suppose that space contains at most level- $MaxLV$  rectangles. The average I/O cost of the  $k$ NWC algorithm is

$$\sum_{i=1}^{MaxLV} \left\{ \left[ \sum_{j=0}^{k-1} \left[ \mathbf{R}(i-1, j) \times S(i, k-j) \right] \right] \times \left[ \mathbf{O}(i) \times \mathbf{WIN}(l, w) + \mathbf{KNN}(\mathbf{O}(i)) \right] \right\}.$$

## 5. PERFORMANCE EVALUATION

In this section, we conduct experiments to evaluate the performance of the proposed NWC algorithm and the proposed optimization techniques. All algorithms are implemented in Java. Three datasets, two real and one synthetic, are used in the evaluation. The CA dataset contains 62,556 places in California<sup>4</sup>, while the NY<sup>5</sup> dataset contains 255,259 places in New York. The data space for these two real datasets are normalized to a square of width 10,000. The synthetic dataset is created based on Gaussian distribution with mean 5000 and standard deviation 2000. The cardinality of the Gaussian dataset is default at 250,000. These datasets are summarized in Table 2, while Figure 8 depicts the object distributions of

<sup>4</sup><http://www.chorochronos.org/>

<sup>5</sup><http://www.census.gov/geo/www/tiger>

Table 3: Description of schemes

Scheme	Optimization Technique(s) Used			
	SRR	DIP	DEP	IWP
NWC	-	-	-	-
SRR	✓	-	-	-
DIP	-	✓	-	-
DEP	-	-	✓	-
IWP	-	-	-	✓
NWC+	✓	✓	-	-
NWC*	✓	✓	✓	✓

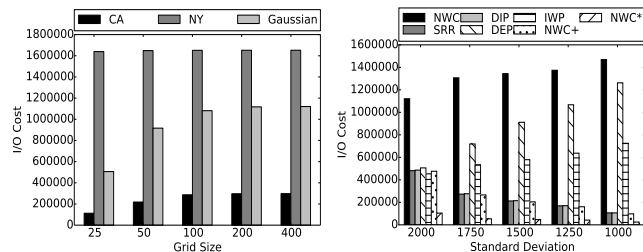


Figure 9: Effect of grid size

Figure 10: Effect of object distribution

these datasets.

All datasets are indexed by R\*-trees with the page size set to 4096 bytes. The maximum number of entries in a node is 50. The default value of  $n$  is 8 and the window length and width are both 8. The grid cell size is set to 25. Similar to [18][10], we consider I/O cost as the performance metric, which is the number of R\*-tree nodes visited, since I/O cost dominates the total execution time of the NWC algorithm. We run 25 queries for each experiment and report the average as the experimental result. To measure the benefit of each optimization technique, we separately run the experiments of the NWC algorithm augmented with each optimization technique (labelled as SRR, DIP, DEP and IWP, respectively). In addition, we devise a scheme NWC+ by enabling only SRR and DIP (which do not incur extra storage overhead). Finally, we devise a scheme NWC\* which enables all optimization techniques proposed in this work. The schemes compared in the experiments are summarized in Table 3. In the following, we show our experimental results by varying various parameter settings, including grid size, object distribution, number of objects and window size.

### 5.1 Effect of Grid Size

In this experiment, we investigate the effect of the grid size by varying the grid size from 25 to 400. Since only scheme DEP uses the density grid, we present only the experimental results of scheme DEP here. Figure 9 shows that for the CA and Gaussian datasets, the I/O cost increases along with the grid size. With the smaller grid size, the granularity of the density grid gets finer. Thus scheme DEP is able to obtain tighter upper bounds during query processing, thus achieving better pruning effect. For the NY dataset, it is interesting to see that the I/O cost of scheme DEP stays nearly constant regardless of the grid size, as depicted in Figure 9. The reason is that the objects in the NY dataset are highly clustered in certain areas, resulting in less effective pruning even when the grid size is small. This result indicates that scheme DEP could not benefit from the density grid for extremely clustered data distributions.

### 5.2 Effect of Object Distribution

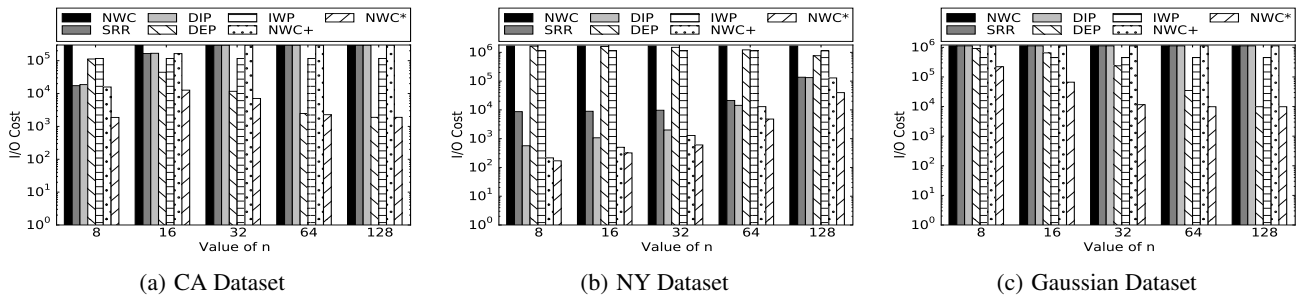


Figure 11: Effect of the number of search objects

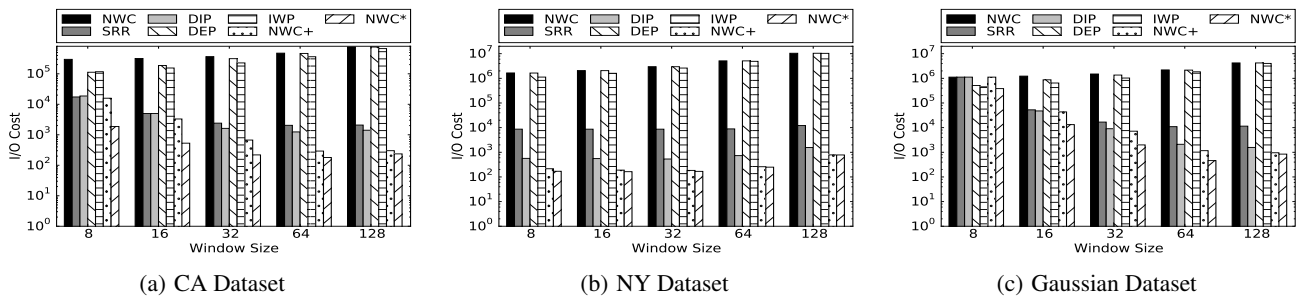


Figure 12: Effect of window size

We study the effect of the object distribution on the performance of all the schemes in this experiment by generating five Gaussian datasets. We fix the same mean 5,000 but varying standard deviations from 2,000 to 1,000. As shown in Figure 10, the I/O cost of scheme NWC increases as the standard deviation decreases. When the standard deviation gets smaller, the data objects are more clustered and more objects are within search regions. Thus, scheme NWC has to access more nodes to process these window queries issued for the search regions. On the contrary, for schemes SRR, DIP, and NWC+, their I/O costs decrease as the standard deviation gets smaller. In our experiment, the I/O cost reduction rates of schemes SRR, DIP and NWC+ over scheme NWC increase from 57% to 93% as the standard deviation decreases from 2000 to 1000. This is because the more clustered object distribution leads these schemes to be able to find *locally best qualified windows* (a qualified window  $qwin$  is called locally best if  $MINDIST(q, qwin) < dist_{best}$  when  $qwin$  is discovered) more easily, achieving better pruning effect. It is not surprising that scheme NWC+ outperforms schemes SRR and DIP by using SRR and DIP together.

On the other hand, the I/O costs of schemes DEP and IWP increase as the standard deviation decreases. As discussed above, scheme DEP performs well in nearly uniformly distributed datasets, but achieves relatively poor performance when the object distribution is highly clustered. In our experiment, the I/O cost reduction rate of scheme DEP over scheme NWC decreases from 54.8% to 14.1% along with the decrease of the standard deviation. Regarding scheme IWP, the performance downgrades owing to that the more highly clustered data objects cause more index nodes to overlap together and thus increases the number of overlapping pointers. The more overlapping pointers are, the more index nodes scheme IWP accesses. In our experiment, the I/O cost reduction rate of scheme IWP over scheme NWC decreases from 59.5% to 55.6% as the standard deviation decreases from 2000 to 1000. From the

experimental result, we can observe that the proposed optimization techniques are complementary with each other. SRR and DIP perform well on highly clustered datasets (i.e., with small standard deviations) while DEP and IWP outperform SRR and DIP on nearly uniformly distributed datasets (i.e., with large standard deviations). By combining the advantages of all the optimization techniques, scheme NWC\* performs the best in terms of I/O cost and significantly reduces 98.3% I/O cost compared with scheme NWC. Moreover, the I/O cost reduction of scheme NWC\* over scheme NWC+ is ranging from 73.8% to 79.7%, showing that the small storage overhead produced by DEP and IWP really pays off especially on the cases not suitable for SRR and DIP.

We now evaluate the storage overheads of scheme DEP and scheme IWP. When the grid size is set to 25, the density grid is of 160000 grids. As we use short integer to store the number of object in each cell, the storage overhead of DEP (the size of the density grid) is about 312KB. On the other hand, it is obvious that the numbers of backward and overlapping pointers are proportional of object numbers. The numbers of backward and overlapping pointers for the CA, NY and Gaussian datasets are 26473, 6236 and 29037, respectively. Suppose that the size of one pointer is 4 bytes. The storage overheads of IWP (the size of these pointers) in the CA, NY and Gaussian datasets are about 103KB, 24KB and 113KB, respectively. These storage overheads are acceptable.

### 5.3 Effect of the Number of Searched Objects

This experiment evaluates the effect of the number of searched objects  $n$ . In the experiment, we vary the value of  $n$  from 8 to 128. Note that we set the y axis in logarithmic scale in this and the following experiments due to the varied scale of I/O cost. Figure 11 shows that the I/O cost of scheme NWC almost stays constant because scheme NWC accesses all the objects in R\*-tree regardless of the value of  $n$ . The other schemes suffer from higher I/O costs with the value of  $n$  increasing, because a larger value of  $n$  causes more

index nodes to be accessed in order to find the locally best qualified windows. When the value of  $n$  is too large to find any qualified window, schemes SRR, DIP and NWC+ would degenerate to scheme NWC since no pruning is achieved. As shown in Figure 11a, the I/O costs of scheme SRR, DIP, and NWC+ are equal to the I/O cost of scheme NWC when the value of  $n$  is larger than or equal to 32. Similarly, Figure 11c shows that schemes SRR, DIR, and NWC+ incur the same I/O cost as scheme NWC when the value of  $n$  is 8 or larger. These three schemes degenerate faster in the Gaussian dataset because the data objects are nearly uniformly distributed in the Gaussian dataset. Different from the CA and Gaussian datasets, schemes SRR, DIP, and NWC+ still outperform scheme NWC in the NY dataset even when the value of  $n$  is 128. The reason is that the highly clustered objects in the NY dataset allow these schemes to find locally best qualified windows quickly even for large values of  $n$ .

On the other hand, schemes DEP and IWP are more resilient to the increase of the value of  $n$ . For scheme DEP, it prunes more index nodes and search regions when the value of  $n$  gets larger. Thus, scheme DEP remains to perform well in the Gaussian dataset as long as the value of  $n$  is large to a certain extent. As shown in Figure 11c, when  $n$  increases from 8 to 128, the I/O cost reduction rate of scheme DEP over scheme NWC in the Gaussian dataset increases from 18% to 99.1%, showing the benefit of scheme DEP. For scheme IWP, a larger value of  $n$  only affects the performance slightly because IWP mainly focuses on saving the I/O cost of window query processing. As such, scheme IWP is able to reduce the I/O cost in the case that schemes SRR, DIP and DEP performs poorly (e.g.,  $n = 8$  in the Gaussian dataset). We can observe from Figure 11c that the I/O cost reduction rate of scheme IWP over scheme NWC keeps in 59.6% when  $n$  increases from 8 to 128. Due to the complementary advantages of the optimization techniques, scheme NWC\* performs the best in all the cases, and compared with scheme NWC, is able to reduce 95.7%~99.4%, 97.6%~99.9% and 88.2%~99.1% I/O costs in the CA, NY and Gaussian datasets, respectively.

## 5.4 Effect of the Window Size

In this experiment, we explore the effect of the window size on I/O costs for all schemes by increasing the window length and width from 8 to 128. We can see in Figure 12 that the I/O cost of scheme NWC gets larger as the window size increases. This is because the larger window sizes result in larger search regions and more data objects involved in the discovery of qualified windows. On the contrary, as the window size increases, it is easier to find locally best qualified windows. With more locally best qualified windows, schemes SRR and DIP achieve better performance. We can see in Figure 12c that schemes SRR and DIP degenerate to scheme NWC in the Gaussian dataset when window size is set to 8. Setting window size to 8 is too small to find any qualified window in the Gaussian dataset since distribution of the objects in the Gaussian dataset is close to uniform distribution. Except for such an extreme case, the I/O cost reduction rates of schemes SRR and DIP over scheme NWC increase from 93.7% to 99.8% and from 95.5% to 99.9% in the CA and Gaussian datasets, respectively, as the window size gets from 16 to 128. As depicted in Figure 12b, the I/O cost reduction rates of schemes SRR and DIP over scheme NWC in the NY dataset keep in 99.5%~99.9%. The reason is that in the NY dataset, there are a large number of data objects and these data objects are highly clustered. Schemes SRR and DIP are still able to get enough locally best qualified windows even window size is set to 8. Thus, increasing the window size does not significantly increase I/O costs of scheme SRR and DIP. Similar to previous ex-

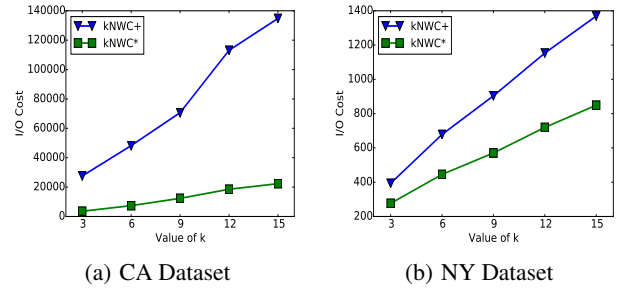


Figure 13: Effect of  $k$

periments, scheme NWC+ outperforms schemes SRR and DIP in all cases.

On the contrary, schemes DEP and IWP do not benefit from larger window sizes. When the window size gets larger, the number of qualified windows increases, thereby reducing the pruning effect of DEP. As the window size is large to a certain extent, scheme DEP could not achieve any pruning and thus will degenerate to scheme NWC, referring to Figure 12a and Figure 12b. For scheme IWP, the large window size results in more overlapping index nodes when processing window queries, reducing the benefit of scheme IWP. As such, scheme IWP is less effective for large window sizes. Fortunately, DEP and IWP are still able to reduce some I/O costs when SRR and DIP are used. Therefore, scheme NWC\* achieves the best performance and the I/O cost reduction rates of scheme NWC\* over scheme NWC+ are from 88.1% to 21.4%, 21% to 0.6% and 65.8% to 11.9% in the CA, NY and Gaussian datasets, respectively, when window size gets from 8 to 128.

## 5.5 Effect of $k$

We now evaluate the performance of these schemes for  $k$ NWC queries. We can observe from the above experiments that scheme NWC+ is of the best performance when extra storage except R-tree is not available. On the other hand, scheme NWC\* performs the best when extra storage is available. Thus, we only compare the performance of scheme  $k$ NWC+ and scheme  $k$ NWC\* (extensions of scheme NWC+ and scheme NWC\*, respectively, for  $k$ NWC queries) in this and the following experiments.

Figure 13 shows the effect of  $k$  on the CA and NY datasets. It is obvious that when  $k$  gets larger, both schemes need to spend more time in exploring more qualified windows. As shown in Figure 13, the I/O costs for both schemes almost linearly increase. As mentioned in Section 5.4, since the NY dataset is of many highly clustered objects, the I/O costs of both scheme in the CA dataset are higher than the I/O costs in the NY dataset. In addition, due to the effect of DEP and IWP, scheme  $k$ NWC\* outperforms scheme  $k$ NWC+ in both datasets. Since both schemes perform well in the NY dataset, the I/O cost reduction rate of scheme  $k$ NWC\* over scheme  $k$ NWC+ in the CA dataset is higher than that in the NY dataset. In our experiment, the average I/O cost reduction rates of scheme  $k$ NWC\* over scheme  $k$ NWC+ in the CA and NY datasets are around 84.3% and 35.3%, respectively.

## 5.6 Effect of $m$

Figure 14 shows the effect of  $m$  on the CA and NY datasets. Setting  $m$  to a larger value means that users accept more identical objects in the nearest qualified windows. When a nearest qualified window is found, the qualified windows nearby are of high likelihood to be the qualified windows. Thus, it is easier

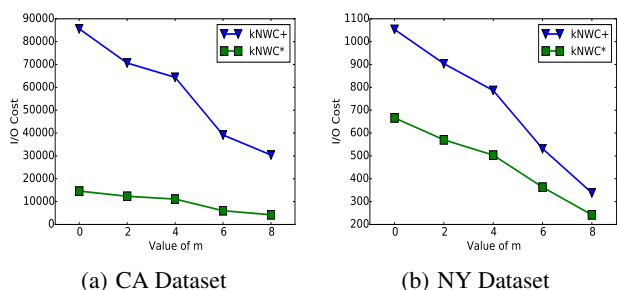


Figure 14: Effect of  $m$

for both schemes to find  $k$  nearest qualified windows when  $m$  gets larger. Similarly, both schemes are of higher I/O costs in the CA dataset than in the NY dataset. Fortunately, with the aid of DEP and IWP, scheme  $kNWC^*$  outperforms scheme  $kNWC^+$  in both datasets. In our experiment, the average I/O cost reduction rates of scheme  $kNWC^*$  over scheme  $kNWC^+$  in the CA and NY datasets are around 83.8% and 33.9%, respectively.

## 6. CONCLUSIONS

In this paper, we propose a novel type of spatial queries, namely nearest window cluster (NWC) queries. To process NWC queries, we identify several properties to find qualified windows, leading to the development of the NWC algorithm. To further accelerate NWC search, we present four optimization techniques to reduce I/O cost. We conduct several experiments to evaluate the performance of the NWC algorithm and the proposed optimization techniques. Experimental results show that these optimization techniques are complementary with each other, and the NWC algorithm with these optimization techniques performs the best in terms of I/O cost.

## Acknowledgement

This work was supported in part by Ministry of Science and Technology, Taiwan, under contracts 102-2221-E-009-147 and 103-2221-E-009-126-MY2.

## 7. REFERENCES

- [1] X. Cao, G. Cong, and C. S. Jensen. Retrieving top- $k$  prestige-based relevant spatial web objects. *Proc. of the VLDB Endowment (PVLDB)*, 3(1), 2010.
- [2] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *Proc. of the ACM International Conference on Management of Data (SIGMOD)*, June 2011.
- [3] M. A. Cheema, X. Lin, W. Zhang, and Y. Zhang. Influence zone: Efficiently processing reverse  $k$  nearest neighbors queries. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, pages 577–588, 2011.
- [4] D.-W. Choi, C.-W. Chung, and Y. Tao. A scalable algorithm for maximizing range sum in spatial databases. *Proc. of the VLDB Endowment (PVLDB)*, 5(11), 2012.
- [5] C.-Y. Chow, M. F. Mokbel, J. Naps, and S. Nath. Approximate evaluation of range nearest neighbor queries with quality guarantee. In *Proc. of the International Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 283–301, 2009.
- [6] K. Deng, S. Sadiq, X. Zhou, H. Xu, G. P. C. Fung, and Y. Lu. On group nearest group query processing. *IEEE Transactions on Knowledge and Data Engineering*, 24(2):295–308, February 2012.
- [7] Y. Du, D. Zhang, and T. Xia. The optimal-location query. In *Proc. of the International Conference on Advances in Spatial and Temporal Databases (SSTD)*, pages 163–180, 2005.
- [8] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A. E. Abbadi. Constrained nearest neighbor queries. In *Proc. of the International Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 257–278, 2001.
- [9] Y. Gao, B. Zheng, G. Chen, and Q. Li. Optimal-location-selection query processing in spatial databases. *IEEE Transactions on Knowledge and Data Engineering*, 21(8):1162–1177, Aug. 2009.
- [10] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999.
- [11] H. Hu and D. L. Lee. Range nearest-neighbor query. *IEEE Transactions on Knowledge and Data Engineering*, 18(1):78–91, January 2006.
- [12] F. Korn and S. M. Krishnan. Influence sets based on reverse nearest neighbor queries. In *Proc. of the ACM International Conference on Management of Data (SIGMOD)*, pages 201–212, 2000.
- [13] K. C. Lee, W.-C. Lee, and H. V. Leong. Nearest surrounder queries. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, page 85, 2006.
- [14] K. C. K. Lee, B. Zheng, and W.-C. Lee. Ranked reverse nearest neighbor search. *IEEE Transactions on Knowledge and Data Engineering*, 20(8):894–910, July 2008.
- [15] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. In *Proc. of the International Conference on Advances in Spatial and Temporal Databases*, August 2005.
- [16] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, pages 301–312, 2004.
- [17] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Transactions on Database Systems*, 30(2):529–576, June 2005.
- [18] G. Proietti and C. Faloutsos. I/O complexity for range queries on region data stored using an R-tree. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, March 1999.
- [19] Y. Tao, D. Papadias, X. Lian, and X. Xiao. Multidimensional reverse knn search. *The VLDB Journal*, 16(3):293–316, Jul. 2007.
- [20] J. Xu, W.-C. Lee, and X. Tang. Exponential index: A parameterized distributed indexing scheme for data on air. In *Proc. of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2004.
- [21] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, pages 485–492, 2001.
- [22] B. Yao, F. Li, and P. Kumar. Reverse furthest neighbors in spatial databases. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, pages 664–675, 2009.

# Adaptive query parallelization in multi-core column stores

Mrunal Gawade  
CWI, Amsterdam  
mrunal.gawade@cwi.nl

Martin Kersten  
CWI, Amsterdam  
martin.kersten@cwi.nl

## ABSTRACT

With the rise of multi-core CPU platforms, their optimal utilization for in-memory OLAP workloads using column store databases has become one of the biggest challenges. Some of the inherent limitations in the achievable query parallelism are due to the degree of parallelism dependency on the data skew, the overheads incurred by thread coordination, and the hardware resource limits. Finding the right balance between the degree of parallelism and the multi-core utilization is even more trickier. It makes parallel plan generation using traditional query optimizers a complex task.

In this paper we introduce *adaptive parallelization*, which exploits execution feedback to gradually increase the level of parallelism until we reach a sweet-spot. After each query has been executed, we replace an expensive operator (or a sequence) by a faster parallel version, i.e. the query plan is morphed into a faster one. A convergence algorithm is designed to reach the optimum as quick as possible.

The approach is evaluated against a full-fledged column-store using micro-benchmarks and a subset of the TPC-H and TPC-DS queries. It confirms the feasibility of the design and proofs to be competitive against a statically optimized heuristic plan generator. Adaptively parallelized plans show optimal multi-core utilization and up to five times improvement compared to heuristically parallelized plans on the workload under evaluation.

## 1. INTRODUCTION

Column store databases are designed with a focus on analytical workloads. Almost all database vendors these days have a column store implementation. A recent study by Microsoft showed that a majority of real world analytic jobs process less than 100 GB of input [3]. This can be accommodated by an in-memory solution on a single high-end server. They come with an abundance of CPU power using tens of cores [27, 23]. Query parallelization is one of the ways to utilize multi-cores. This calls for a renewed look at the traditional query parallelization techniques, such as the *exchange operator* based parallelization [15], since the state of the art column store systems such as IBM BLU accelerator [24], HyPer [30] use work stealing based approach for multi-core scalability.

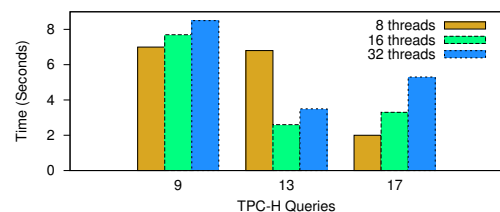


Figure 1: Response time variations due to varying degree of parallelism under concurrent workload (32 hyper-threaded cores).

An important issue is the degree of parallelism (DOP) of a plan which reflects the maximum number of parallel operator executions. With tens of cores on CPUs, finding the optimal degree of parallelism of a query plan using heuristic and cost model based exchange operator approach is difficult [2]. Some of the prominent problems are a huge multi-core aware plan search space, parallelism aware accurate cost model estimations, and the optimal placement of exchange operators in the plan. The degree of parallelism problem becomes even more difficult under a concurrent workload due to competition for shared resources, such as CPU cores, memory, and memory controllers. This forces many systems to take a conservative approach towards plan parallelization decisions, as a sub-optimal parallel plan could often degrade performance. Often a serial plan is preferred as long as it ensures a robust performance [1].

For example, consider Figure 1, which shows execution of three TPC-H heuristically parallelized queries for different DOP under a heavy concurrent CPU bound workload, which ensures 0% CPU core idleness (Scale factor 10 on 256 GB RAM with 32 hyper-threaded cores). The queries show varying performance under different DOP. The traditional plan generation approaches based on heuristic and cost model [10] fall short, as the plans do not reflect runtime resource variations, making them suboptimal under a concurrent workload.

We introduce *adaptive parallelization*, a new mechanism to generate *range partitioned* parallel plans using query execution feedback, while taking into account the run-time resource contention. Adaptive parallelization generates a better plan (P1) from an old plan (P0) in a greedy manner, by parallelizing the most expensive operator from P0, under repeated query invocations. The inspiration is derived from the observation that in real world systems the same query templates get reused multiple times only changing some parameters. Starting with a serial plan, each successive query invocation results in a new parallel plan, until a near minimal execution time parallel plan is detected, which ensures a near optimal DOP. Adaptive parallelization under concurrent workload reflects resource contention, making adaptive parallelized plans resource

contention aware [11]. The success of adaptive parallelization depends on its ability to converge quickly, while ensuring a near minimal execution parallel plan.

Adaptive parallelization also allows to analyze the relation between DOP and multi-core utilization. Multi-core utilization represents the fraction of actual CPU cores used versus the available cores during query processing. Maximum multi-core utilization however need not improve performance, as it might lead to memory bandwidth pressure due to parallel operator executions [22]. Hence, finding the right balance between the DOP and multi-core utilization is important. Since adaptive parallelization generates new parallel plans incrementally, it enables us to analyze the relation between DOP and multi-core utilization. Adaptive parallelized plans have minimal multi-core utilization and a near optimal degree of parallelism, which helps in achieving better response time during concurrent workloads.

We summarize our main contributions as follows.

- We introduce *adaptive parallelization*, a new execution feedback based parallel plan generation technique, that ensures near optimal degree of parallelism.
- We introduce an adaptive parallelization convergence algorithm for different scenarios.
- We analyze the parameters affecting the speedup of the core relational algebra operators.
- A near optimal DOP allows adaptive parallelized plans to show up to five times response time improvement compared to heuristically parallelized plans.

**Paper outline:** The paper is structured as follows. In Section 2 we describe the architecture of adaptive parallelization. We also provide parallelization heuristics for operators and illustrate the dynamic partitioning scheme and discuss related problems. Section 3 describes the convergence algorithm to find the near minimal execution parallel plan along with various convergence scenarios. In Section 4 we provide a detailed experimental evaluation. Related work is described in Section 5. We conclude citing the major lessons learned in Section 6.

## 2. ARCHITECTURE

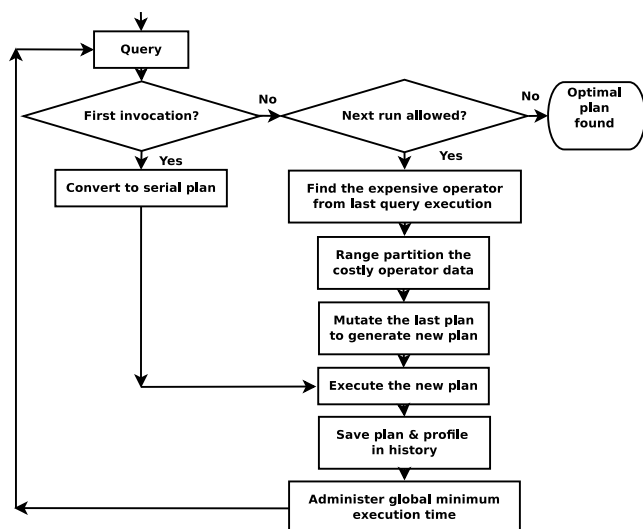


Figure 2: Adaptive parallelization workflow.

Adaptive parallelization could be used by any columnar database system as long as its plan representation allows identification of individual expensive operators.

**Run-time environment:** It consists of a scheduler, an interpreter, and a profiler. The scheduler uses a data-flow graph based scheduling policy, where an operator is scheduled for execution once all its input sources are available. While an interpreter per CPU core executes the scheduled operators, the profiler gathers performance data on an executed operator basis. The profiling overhead is minimal due to vectorized nature of execution. The profiled data consists of operator’s execution time, memory claims, and thread affiliation id. Cost model based plan generation approaches often suffer from incorrect cardinality estimates. We use a heuristic plan generation approach where parallelization decisions are based on execution time feedback, without a need for operator’s cardinality statistics.

The execution time is a good metric for parallelization decisions as it reflects the system state such as the memory bandwidth pressure and the processor usage. Though the presence of system noise might affect execution time, such disturbances level out during adaptation.

**Infrastructure components:** The Adaptive Parallelization (AP) infrastructure is implemented using the following components a) operator stubs to morph a plan based on past behavior, b) the plan administration policies to choose a suitable plan from the plan history, and c) the AP convergence algorithm, which we describe in Section 3.

**Workflow:** The adaptive parallelization work-flow is summarized in Figure 2. The first phase is similar to most systems [16] where an optimal serial plan (Figure 3 Plan 1) is generated. Our approach differs in the second phase where the the query is cached, plan is fed to the framework, executed, and the profiling information such as the query execution time, the operator execution time, the number of invocations, etc. are stored. On the next query invocation a new parallel plan (Figure 3 Plan 2) is derived from the immediate old plan (Plan 1) by parallelizing the most expensive operator (Select on input A). The AP process iterates by invoking the same query again and generating parallel plans in an incremental manner by parallelizing the most expensive operators in successive steps. The number of iterations to find the minimal execution time parallel plan is controlled by a convergence algorithm described in Section 3. As the book keeping and compilation time is minimal we only report the execution time.

**Why feedback based approach?** Like parallel databases, multi-core CPUs make the parallel plan search difficult [19]. The main problem is finding the *optimal number of partitions* per operator for an *optimal input serial plan*. Finding an optimal input serial plan is out of the scope of this paper. During parallelization when an operator’s data is partitioned, there are combinatorial possible choices for the partition size. For example, in the worst case, each operator’s data could be partitioned in a single tuple, such that the total number of operators equal the number of tuples. In the best case a single operator could work on the entire non-partitioned input. The possible partition size choices for different operators represent multiple parallel plans with different execution times, making this a combinatorial search problem. The plan search space exploration is usually done using a combination of both the heuristic and the cost model based approach. It allows to prune the search space for an efficient search. Overall, finding an optimal multi-core aware parallel plan using traditional approaches is difficult. In comparison the feedback based approach we propose is relatively easy, as the assumption is the input serial plan we start with is an *optimal plan*. Since the approach explores the search space in a guided way

by parallelizing only the most expensive operator, we avoid a large space of uninterested plans.

## 2.1 Plan mutation

We refer to the process described in the workflow as *plan mutation*. Plan mutation could be guided by different policies. In this paper we use parallelization of the *expensive operator* in a plan as the guiding principle. An operator is considered *expensive* if its execution time is the highest amongst all operators. Based on the complexity, we categorize mutation in three types as Basic, Medium, and Advanced.

**Basic:** Basic mutation involves parallelization of an expensive operator by introducing two new operators of the same type, called expensive operator's **cloned operators**. The cloned operators work on the expensive operator's partitioned data. Partitioning is cheap when it involves no data copy, but introducing range partitioned sliced view of the columnar data. (Value / hash based partitioning needs the presence of a partition operator, about which we discuss in Section 5.) An *exchange union* operator (either a newly introduced or an existing one) combines the result of the cloned operators. In Figure 3 we see one such example for select operator parallelization.

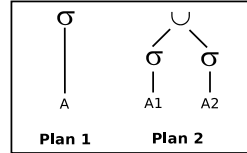


Figure 3: **Basic mutation.**

The two most popular algorithms for the join operator are the hash join and the sort merge join. We analyze the hash join implementation as it suits most workloads due to the omnipresence of non-sorted data. We consider adaptive

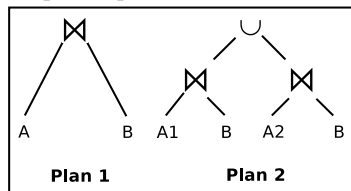


Figure 4: **Join parallelization.**

parallelization of the join operator plan (Figure 4 Plan 1) when only the larger (outer) input is split into equi-range partitions on consecutive runs. Figure 4 Plan 2 shows the parallelized plan with the two new join cloned operators. An exchange union operator combines the output of the cloned operators.

**Medium:** Medium mutation handles plan parallelization when the *exchange union* operator (U) itself turns out to be expensive, as a result of intermediate data copying due to low selectivity input. This mutation stage arises, when the exchange union operator is introduced as a result of the basic mutation.

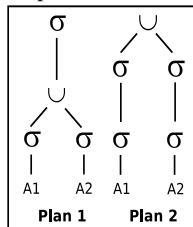


Figure 5: **Medium mutation.**

Figure 5 shows one such example where Plan 1 with an expensive exchange union operator is mutated into Plan 2. The mutation process involves propagating the inputs to the exchange union operator, to its data flow dependent operators. The data flow dependent operators are cloned to match the exchange union operator's input. Finally a newly introduced exchange union operator combines the result of the cloned operator's output.

**Advanced:** Advanced mutation involves parallelization of operators such as group-by and sort, that do not exhibit the filtering property (selectivity = 0).

Figure 6 shows parallelization of a group-by operator with the advanced mutation. The expensive operator (group-by) is parallelized by introduc-

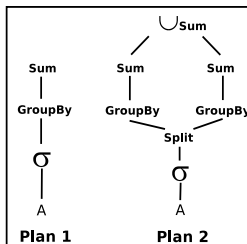


Figure 6: **Advanced mutation.**

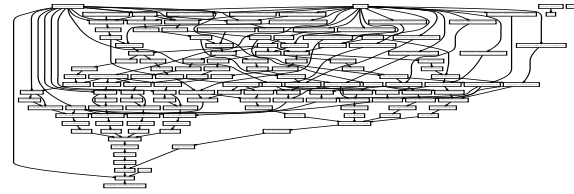


Figure 7: **Complex operator dependencies in TPC-H Q14 parallel plan.** Rectangles represent operators, and edges between them represent the dependencies. The graph is only meant to give a high level perspective of the plan's complexity, abstracting individual operator details. [12] shows graphs where operators are visible.

ing two cloned operators on its equi-range partitioned data. Next the aggregation operators such as sum and average are parallelized by introducing two aggregation cloned operators. The cloned operators (group-by) result is propagated to the aggregation cloned operators (sum). Finally, an exchange union operator combines the parallelized aggregation operators result. Since the aggregation cloned operators always show very high filtering property, the exchange union operator combining their result is cheap.

**Summary:** A relational operator gets parallelized in two cases. In the first case, the operator itself might be expensive and gets parallelized using either the basic or the advanced mutation. In the second case, operator parallelization occurs as a result of using the medium mutation, where the operator is in the data flow dependent path of the expensive exchange union operator. In both cases identifying and resolving the parallelizable operator's output propagation dependency across the entire plan is an essential step.

The three mutation schemes we described cover all possible mutations as an operator could either get parallelized due to its own expensiveness or as a result of its presence in the data flow path of another parallelizable operator.

## 2.2 Making plans simpler to mutate

Most columnar systems [1, 4, 7, 18] use a simple representation of plan with operators represented using physical algebra. The operators use standardized interfaces for individual columnar data and related argument passing. Column store specific functionality such as operations on multiple columns and tuple reconstruction are mostly hidden away as the internal logic in the execution engine framework. Some column stores like the open-source system MonetDB, however use an abstract language to represent plans [6], where column store specific functionality such as the tuple reconstruction and other columnar operations is exposed in the plan representation itself. Use of operators with different semantics and specialized operators such as the tuple reconstruction operators is common. Figure 7 shows one such plan with complex data flow dependencies.

Plan mutations using either the medium or advanced mutation involves resolving parallelized operator's propagation dependencies. Hence, care has to be taken to resolve parallelized operator's propagation dependencies. To make plan mutations simpler, modification of some of the operator's semantic representation is needed. We describe the related aspects in the rest of the section.

**Adaptive parallelization and operator semantics:** Operators can have different semantics depending on primitives being used. Adaptive parallelization could further add more information such as the partition under use, total number of partitions, etc. Plan mutations thus generate combination of different operator semantics.

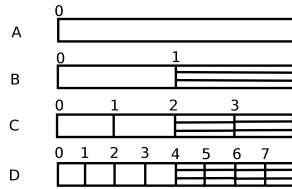


Figure 8: **Column A dynamically partitioned with iterations.**

For example, consider the case of the filter operator which can have two representations. One which accepts normal columnar input and the other which accepts column and also a bit vector from another selection operator's output. Hence, the filter operator could be represented using two primitives depending on the number and the type of inputs. Depending on the data flow dependency, a suitable filter operator gets parallelized during plan mutations.

Adaptive parallelization uses different parallelization rules catered to different operator semantics. Since any operator could be expensive, resulting in its parallelization, the challenge for different mutation schemes lies in how well they are able to resolve the data flow dependencies across different operator semantics.

**Plan rewriting:** One of the techniques to ease the mutation process is to modify the original input serial plan from the SQL compiler using a query rewriter. The rewriter substitutes original operators (for example, aggregation operators and tuple reconstruction operators) with new adaptively parallelization aware operators. These new operators use modified implementation of operators such as group-by, aggregation operators (sum, avg), and sort, by keeping their original semantics, but with changed arguments ordering, to resolve possible operator propagation dependencies. For systems with simple plan representations the operator propagation dependencies due to multiple columns could be handled in the execution engine framework logic.

### 2.3 Adaptive parallelization aware partitioning

In a column store the operators operate on an array or vector representation of the data. For readability, we consider the array representation with range partitioning. It involves creating read only slices on the base or the intermediate column. Creating slices involves marking the boundary ranges for the base or intermediate columnar data and is cheap, as there is no data copying involved. This technique could be also used during vectorized execution where the vectors are derived from the partitioned range of the base and intermediate input. We briefly describe a value based partitioning approach use case in Section 5.

**Dynamic partitioning:** Adaptive parallelization generates dynamically sized partitions on the base or intermediate column, as any operator could be parallelized during successive iterations. In contrast a heuristically parallelized plan often uses a fixed number of partitions based on the available CPU cores. To explain dynamic partitioning of a column using a select operator, we use Figure 8.

When the select operator on the column in 8A turns expensive, the column is sliced in two partitions represented by 8B. When the select operator on partition 1 in 8B turns expensive, two new partitions are introduced, represented by 2nd and 3rd in 8C. Now there are three select operators, one on 0th partition of 8B and two on 2nd and 3rd partition of 8C. When the select operator on 2nd partition in 8C becomes expensive, it is divided further and two new partitions 4th and 5th in 8D are introduced. So now there are total 4 select operators working on 0th partition of 8B, 3rd partition of 8C and 4th,5th partition of 8D. Please note that the partitions are of different sizes and their boundaries are aligned on the base column in 8A. Maintaining the alignment during dynamic partitioning is important, as misalignment could lead to problems such as a) repetition of data b) omission of the data across different operator

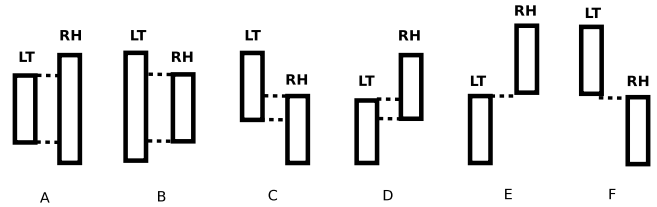


Figure 9: **Different alignment scenarios between columns during tuple reconstruction due to dynamic partitioning.**

partitions. Thus, dynamic partitioning allows the operators to work on different sized partitions of the same column in parallel.

**Dynamic partitioning and tuple reconstruction:** Tuple reconstruction is a well known problem in column stores [17] and is implemented as join lookup. Column stores use either early or late materialization strategies for column projection using tuple reconstruction, which involves using row-ids to fetch values from the column that needs projection. For example, consider the columnar representations in Figure 10, which shows head (H) and tail (T) columns grouped as Left (L) and Right (R). The head column (LH / RH) contains row-ids, whereas the tail column contains either row-ids (LT) or actual values (RT). The @ indicates a row-id. When the head column (LH / RH) contains consecutive row-ids, it is not materialized and used as a virtual column. During tuple reconstruction, the row-ids in the left tail (LT) are used as an index in (RH) to fetch the corresponding values from the right tail (RT). For example, row-ids 2, 4, 5, 7 from LT are probed in the RH, whose corresponding values in RT are 12, 11, 20, and 13.

One important aspect is the effect dynamic partitioning has on the tuple reconstruction due to possible misalignment between LT and RH. When row-ids from LT are used as an index to fetch values from RT, the row-ids in LT should be a subset of row-ids in RH. If not, then a lookup using row-id in LT, for the row-id index in RH does not exist, resulting in an invalid access.

LH LT		RH RT	
1@	2@	1@	15
2@	4@	2@	12
3@	5@	3@	44
4@	7@	4@	11
5@	8@	5@	20
		6@	16
		7@	13

Figure 10: **Tuple reconstruction between two columns.**

Since adaptive parallelization generates variable sized partitions, it gives rise to different alignment scenarios as shown in Figure 9 (B,C,...F). Consider the mis-alignment example in Figure 10. Here LT start row-id=2, which is greater than RH start row-id=1, and LT end row-id=8, which is greater than RH end row-id=7. Hence, LT's upper boundary starts after RH's upper boundary, whereas LT's lower boundary extends beyond RH's lower boundary as represented in Figure 9D. In Figure 9 the lengths of columns provide just a logical representation of over and undershooting of boundaries, and do not represent the actual content. To maintain the alignment the lower boundary of LT is adjusted by removing row-id=8, to match the lower boundary of RH. The correct boundary alignment is represented by dashed lines in Figure 9D. Adaptive parallelization depending on the operator semantics uses one of the alignment scenarios, to make sure that the partitions align correctly. Fixed size partitions always lead to correct alignment (See Figure 9A), resulting in a valid access.

Another important aspect arises when the output of operators working on the dynamically partitioned data is packed together. Here the exchange union operator must maintain the correct ordering to avoid the incorrect results. The correct ordering is maintained, as the operators whose results are packed follow the mutation sequence order, hence the results being packed together fol-



low the same order. Adaptive parallelized plans could become very large due to successive partitioning and operator propagation, which could make partition misalignment related problems, if any, hard to identify and resolve.

**Plan explosion:** As adaptive parallelization involves propagating the parallelized operator’s output on its dependent operators, the plans could quickly grow large. Plan explosion results as a side effect of the exchange union operator removal during the *medium* mutation. For example, when a descendant of the same type of operator stays expensive during successive invocations, it gets parallelized and a single exchange union operator combines the output of all such parallelized operators. As a result the number of input parameters to the exchange union operator could become very large. Eventually if the exchange union operator itself turns expensive, it is removed using the *medium* mutation. This leads to a plan explosion, as the medium mutation propagates inputs of the exchange union operator on its data flow dependent operators. For each operator in the data flow path, new instructions (operators) which equals the number of the exchange union operator inputs are added in the plan. Hence, if the number of input parameters to the exchange union operator is large, the plans could grow very large.

The growth of large plans is suppressed by not removing the exchange union operator if its input parameters cross a certain threshold. The threshold in the current implementation is 15 parameters, chosen on the basis of empirical observations from different parallelization cases. Suppressing the exchange union operator removal however stops further plan parallelization, as the exchange union operator stays the most expensive operator in all further query invocations.

We have described adaptive parallelization aware infrastructure changes so far. Obtaining a minimal execution parallel plan however depends on how fast the adaptive parallelization process converges. In the next section we describe a new algorithm that ensures convergence in different scenarios.

### 3. ALGORITHM

In this section we introduce the heuristics for the global minimum execution identification from the set of available plans and the corresponding convergence algorithm. The algorithm is loosely inspired by the hill climbing approach [26]. Figure 11 shows different cases of the presence of minima, plateaus, and up-hills in the execution times, during adaptive parallelized runs of a join operator plan. We refer to the minimal execution time amongst them as the *global minimum execution (GME)*. Like most systems that generate parallel plans, our base assumption is an optimal input serial plan. Hence, the focus of parallelization is to identify the optimal number of partitions for operator’s data in the input plan. Problems such as sub-optimal parallel plans due to poor ordering which might require backtracking are not considered, as the input is an optimal serial plan.

The convergence algorithm should be able to find the GME in all cases of minima, plateaus, and up-hills in the execution time, and converge in minimal number of runs. Next we formally define the GME first, the convergence algorithm next, and then illustrate different convergence scenarios.

#### 3.1 Global minimum execution (GME)

As the runs progress, the GME is the minimal execution time amongst so far observed runs, and keeps on changing, during an active adaptive parallelization instance.

We denote the current run’s execution time as *CurExec*. The *execution time improvement (CurExecImprv)* at the current run is calculated with respect to 0th run’s (*SerialExec*) execution time.

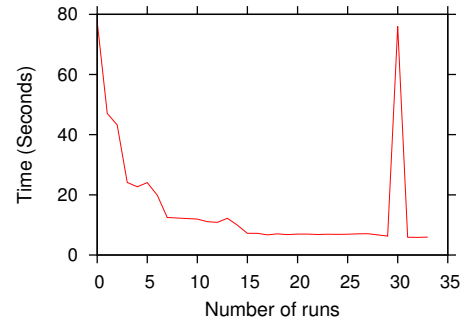


Figure 11: Adaptive parallelization convergence algorithm scenarios for join operator parallelization.

$$CurExecImprv = |(SerialExec - CurExec)| / SerialExec.$$

To calculate the first GME improvement, we initialize GME to the first run’s execution time after the serial execution (0th run).

$$GMEimprv = |(SerialExec - GME)| / SerialExec.$$

As the runs progress, new GME needs to be identified. A new run’s execution time becomes the new GME, if the run’s execution time improvement is better than the current GME’s execution time improvement by a certain threshold.

$$GME = CurExec \quad \{if(CurExecImprv - GMEimprv) > threshold\}.$$

As the runs progress, the new execution times could be slightly lower than the existing GME, hence, selecting the correct threshold is important in discarding such new execution times, which otherwise could become the new GME. For example, consider a hypothetical adaptive parallelization instance. Let *CurExecImprv* at the 8th run be 96%, *GMEimprv* at the 3rd run = 90%, and *threshold* = 5%, then  $(CurExecImprv - GMEimprv) > 5$ . Hence, the Current run Execution time at the 8th run is considered the new Global Minimum Execution (GME). Correct tuning of the *threshold* parameter is thus crucial as it helps to discard multiple possible GMEs and to choose the optimal execution time amongst them, as the new GME.

Finding GME could be also difficult due to the presence of many local minima, about which we illustrate next.

**The global minimum detection problem:** The problem could be formally stated as finding the global minimum execution from many local minima that occur as the runs progress. When the execution time of a run is more than its previous run, a local minimum results at the previous run. For example, a local minimum occurs at the 4th run in Figure 11. The convergence algorithm has to overcome many such local minima during its exploration of the global minimum. We use the rate of improvement in the execution time of the runs as a heuristic, to avoid the local minimas.

The execution time of consecutive runs could improve or worsen depending on the run-time conditions (execution skew, operating system interference), giving rise to positive or negative *rate of execution time improvement (ROI)*. The ROI of a run is defined with respect to its previous run’s execution time (*PrevExec*). We define ROI as follows.

$$ROI = (PrevExec - CurExec) / MAX(CurExec, PrevExec).$$

In Section 3.2 while describing the core convergence algorithm, we illustrate how to use ROI to avoid local minimas. Finding GME is difficult, however, another equally difficult task is to find it in the minimal convergence runs, about which we illustrate next.

**The minimal run convergence problem:** The problem could be formally stated as finding GME in a minimal number of runs, dur-

ing consecutive query invocations. Too few runs have a risk of non-occurrence of the global minimum and the algorithm converging on a local minimum. Too many runs might ensure a global minimum at the cost of a slow convergence. Hence, finding the right balance between the minimum convergence runs and the GME is of prime importance for the convergence algorithm.

## 3.2 Convergence algorithm

We describe the convergence algorithm using the context described so far in Section 3. The aim is to find the GME in minimal number of convergence runs. We model the number of convergence runs (*Convergence\_Runs*) using the parameters *credit* and *debit*. A credit reflects the number of runs accumulated at each run due to a positive ROI. A debit reflects the number of runs accumulated at each run due to a negative ROI.

$$\begin{aligned} \text{Credit} &= \text{Credit} + (\text{ROI} * \text{Number\_Of\_Cores}). \\ \text{Debit} &= \text{Debit} + (|\text{ROI}| * \text{Number\_Of\_Cores}). \end{aligned}$$

The value of (*credit - debit*) at each run reflects the balance (*Convergence\_Runs*) available for the system to converge. Hence, the next run is allowed only if the balance is positive i.e. (*credit - debit*) > 0).

$$\text{Convergence\_Runs} = \text{Credit} - \text{Debit} = f(\text{ROI}).$$

The algorithm starts with the value of credit = 1 and debit = 0. When parallelism reduces the execution time, the ROI of the first run is positive and very high (Figure 11 - The algorithm starts with the 0th run). With an increase in runs, the ROI decreases. During the initial few runs the algorithm should ensure availability of sufficient runs as a balance, to avoid premature convergence. During the later runs, as the ROI slows down, the algorithm should ensure as few balance runs as possible, to ensure fast convergence. From the formula above, as both credit and debit are dependent on ROI, they are a function of ROI, which makes the *Convergence\_Runs* also a function of ROI. The algorithm convergence is hence guaranteed, since the heuristic *Credit - Debit* > 0, which decides the available *Convergence\_Runs* becomes invalid eventually. Next we describe various convergence scenarios and how the heuristic *Credit - Debit* > 0 becomes invalid, which guarantees the algorithm's convergence.

## 3.3 Convergence scenarios

We identify three scenarios during which the algorithm should ensure the convergence, 1. No premature convergence in a local minimum before identifying a global minimum. 2. No extended convergence, and 3. The convergence in a noisy environment. We expect these scenarios to cover the entire spectrum as the aim is to find the global minimum, and the possible problems for the convergence algorithms could be its early termination, late termination, and termination during noisy environment. We describe these scenarios next.

### 3.3.1 No premature convergence

When parallelism improves the execution time, the first run always has a very high ROI (Figure 11 - The algorithm starts with the 0th run). Hence, the credit accumulated after the first run is very high with an upper limit of (*Number\_Of\_Cores + 1*). This ensures that there are sufficient runs available as a balance in the system during the initial stages to overcome plateaus and up-hills. Each run after the first run contributes more credit, ensuring more runs. This is also analogous to the concept of accumulation of the *potential energy* by a body when it falls from great heights. The greater the height, the higher the potential energy. The energy allows the body to keep moving in plateaus and climb high hills, as long as

there is a balance energy.

### 3.3.2 No extended convergence

Accumulation of high credit in the few initial runs on a stable system could result in a state where the algorithm *never* converges. In a stable system the execution time variations are minimal, leading to fewer *debts* being made. In such a system, the proportion of accumulated *credit* will always be much higher than the accumulated *debit* after a few initial runs. For example, consider Figure 11. After 15 runs the ROI is minimal, ensuring that no new significant credit or debit is introduced. However, the accumulated credit till 7 runs is very high, as the ROI till 7 runs is very high. This situation leads to non-convergence as there are always balance runs available i.e. (*credit - debit* <= 0) is never true.

**Leaking debit:** To ensure the algorithm converges in a finite number of runs we introduce the concept of *leaking debit*. In this scheme after a *threshold* on the number of runs is crossed, a constant debit gets deducted from the available credit at each run. *It ensures the available credit is drained to 0, so that the algorithm converges in a finite number of runs.* Hence, *leaking debit* is a function of the available *credit* at the threshold run. The threshold run value is calibrated to be the *Number\_Of\_Cores* on the CPU. It ensures at least those many runs are used to find the optimal execution time. The *Leaking\_Debit* is calculated by dividing the available credit at the threshold run amongst the possible remaining number of runs during the global minimum search.

$$\begin{aligned} \text{Remaining\_Runs} &= \text{Extra\_Runs} * \text{Number\_Of\_Cores}. \\ \text{Leaking\_Debit} &= \text{Credit} / \text{Remaining\_Runs}. \end{aligned}$$

Based on plan complexity, some queries converge early, while some take longer after crossing the *threshold run* reference. To avoid premature convergence, the system specific tunable parameter *Extra\_Runs* is used, which ensures that the *remaining number of runs* to search the global minimum are sufficient. Note that *Remaining\_Runs* is just an approximate bound. Plan representations vary considerably across systems. Hence, based on empirical observations from different parallelization cases, and multiple experimental runs (five), for the current platform, *Extra\_Runs*=eight is considered a safe boundary value to avoid the premature convergence. Higher values result in an extended convergence.

### 3.3.3 Convergence in a noisy environment

Depending on the stability of the run-time environment (operating system process interference, memory flushes, etc.) the execution time of an individual run could vary considerably. The execution time of some of the runs in a noisy environment is often greater than the serial plan execution time. One such peak is visible in Figure 11 at the 30th run. Most peak executions are followed and preceded by a normal execution. If care is not taken such peaks will make the algorithm halt *immediately* as the debit due to peak ascent will be higher than the accumulated credit. Hence, the algorithm should converge gracefully in such a noisy environment.

Our solution is to mark all such unique peaks as outliers, and ignore their presence. The algorithm incorporates this by allowing the immediate next run to execute. This ensures the balance runs stay unaffected, as the *debit* made during the peak *ascent* is compensated by an equivalent *credit* during the peak *descent*, during the next run. Concurrent workload could also affect the convergence, however, tuning the *Extra\_Runs* parameter to find the leaking debit should take care of it.

### 3.3.4 Global minimum plan identification proof

The convergence algorithm should ensure a global minimum plan while converging in a reasonable number of runs. The lower bound

Table 1: System configuration

CPU	Sockets	Threads	L1 cache	L2 cache	Shared L3 cache	Memory	OS
Intel Xeon E5-2650@ 2.00GHz	2	32	32KB	256 KB	20MB	256GB	Fedora 20
Intel Xeon E5-4657Lv2@ 2.40GHz	4	96	32KB	256KB	30MB	1TB	Fedora 20

on the convergence runs is  $Number\_Of\_Cores + 1$ , while the upper bound approximates between  $(Number\_Of\_Cores + 1 + Remaining\_Runs)$  and extra runs added to the previous upper bound, if any, due to a large credit accumulation. The convergence runs are directly influenced by the *Leaking Debit*, and credit / debit accumulation.

The global minimum plan’s existence beyond the upper bound of the convergence runs is not possible. We provide a proof by contradiction. If such a plan exists then its execution should be significantly better. In that case the corresponding expensive operator should have been identified much earlier, even before the first upper bound on the convergence runs is reached. If multiple such plans exists, then that indicates improved execution with each run. Such improvement should then add extra runs (more credit) to the first upper bound on the convergence runs, which would prolong the global minimum search further, to find a more optimal plan. Hence, no matter the situation, a near global minimum plan is identified in the available convergence runs. In all the convergence scenarios when the heuristic  $Credit - Debit > 0$ , that decides available *Convergence\_Runs*, turns invalid, the algorithm converges.

## 4. EXPERIMENTS

Adaptive parallelization is implemented in MonetDB, being the only full fledged open-source columnar system, with memory mapped columnar representation for the base and the intermediate data. The operators are represented in an intermediate language called MonetDB Assembly Language (MAL) [6], with their implementation in C. The operators have variable number of arguments depending on their semantics, and form complex data flow patterns in MAL plans, as shown in Figure 7.

Table 1 summarizes our experimental hardware platform, which consists of two types of machines, with two and four socket CPUs each. All experiments, unless mentioned, use in-memory data (without disk IO) on the two socket machine. Heuristic parallelization unless mentioned uses 32 threads. Each graph plots an average of four runs of the same experiment. We use the four socket machine to test one of the workload’s scalability from NUMA perspective.

The experimental section is divided into two broad categories. In the first we analyze how parallelization gets affected by various operator level parameters. In the second we analyze it at the SQL query level. We use a mix of micro-benchmarks, simple, and complex SQL queries to gain parallelization behavior insights.

We use TPC-H and TPC-DS workload for SQL query level performance comparison. We observe the TPC-H isolated execution of both the adaptive and the heuristic parallelization shows similar performance. However, adaptive plans are better as they use fewer number of cores, which helps during concurrent workload. Adaptive plans show better performance than the heuristic plans for the TPC-DS workload isolated execution, due to optimal number of partitions, and the presence of the skewed data. In the rest of the section we describe the experimental details.

### 4.1 Operator level analysis

Adaptive parallelization helps to analyze the role of individual operators in influencing parallelized execution, as it uses expensive operator parallelization as a heuristic. Getting insights into the issues such as the execution skew becomes easier. An operator’s execution time varies on the basis of type of computation, data distribution, amount of data being read / written, type of data access (serial / random), and memory hierarchy of the access (cache / main

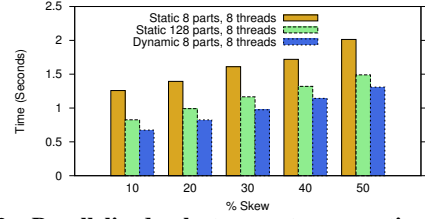


Figure 12: **Parallelized select operator execution on skewed data using static and dynamic (adaptive) sized partitioning. The second bar indicates a work stealing based approach.**

memory / disk IO). We analyze some of these factors next.

#### 4.1.1 Data skew

This experiment highlights the role of *dynamic* sized partitions to avoid execution skew during parallelized execution, when the data distribution is *non-uniform* (skewed). The execution skew occurs when at least one of the parallelized operators takes longer to execute than the rest.

Static partitioning (equi-range partitioning) of skewed data leads to execution skew as some partitions have more matches than the rest. Adaptive parallelization performs well in skewed data scenario as the operator with the skewed partition turns expensive, and gets parallelized until expensiveness balances out.

Figure 12 shows the execution time when parallel select operators work on statically or dynamically (adaptively) partitioned skewed column of type *long* (8 bytes). The number of tuples in the input column are 1000 million (M) (size = 8GB). Figure 13 shows the column’s data distribution with 500 million random tuples in the first half. The second half contains skewed data with 5 sequential clusters of 100 million identical tuples. We vary the select operator’s condition to generate the execution skew.

500M Random Numbers	100M Same	100M Same	100M Same	100M Same	100M Same
---------------------	-----------	-----------	-----------	-----------	-----------

Figure 13: **Data distribution for a skewed column.**

Figure 12 shows execution with 8 threads on 8 dynamically sized partitions (blue) is up to 60% better than the execution with 8 threads on 8 static partitions (khaki). One may argue that the work stealing approach [5] could solve the problem of execution skew due to the static partitions. We analyze it by creating a large number of smaller partitions (128) operated upon by 8 threads. Large number of smaller partitions allows those threads that finish work early to operate on remaining partitions, while threads on skewed partitions stay busy. Identifying the optimal combination of static partitions and threads is however non trivial, as in some cases more partitions might lead to plan blow-up resulting in scheduling overheads. In contrast we observe that the dynamic sized (adaptive) partitioning approach with 8 threads and just 8 partitions fares competitively with static 8 threads, 128 partitioned approach.

**Summary:** Skew handling is a natural property of adaptive parallelization. It is a result of dynamically sized range partition creation, and a side effect of the expensive operator parallelization heuristic.

#### 4.1.2 Selectivity, Input size and Exchange union operation

This experiment analyzes the effect of selectivity, input size, and the exchange union operation on the parallelized execution of select and join operators, in terms of their speed-up. *Speedup* is defined as the ratio of serial to parallel plan execution time. The experiment also allows to analyze the speed-up effect when the number of threads varies from 1 to 32. This is possible since with each it-

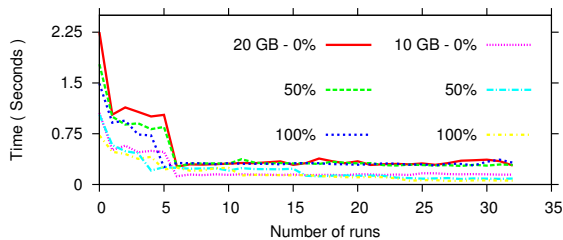


Figure 14: Effect of variations of data size and selectivity on the speedup of the adaptively parallelized *Select* operator plan.

Table 2: *Select* operator plan speedup (compared to serial execution) using adaptive and heuristic parallelization.

Size (GB)	AP = Adaptive, HP = Heuristic parallelization					
	Selectivity					
	0%		50%		100%	
	AP	HP	AP	HP	AP	HP
100	10	10	8.5	10	7	9
20	10.5	12	8.5	12	8	12
10	16	11	14.5	11	12	9.5

eration one more partition gets added and is available for one more execution thread (from a pool of 32 threads) to operate in parallel.

**Exchange union operation:** Many systems use the *exchange operator* based parallelism [15], where one of the concerns is to identify the correct placement of the exchange operator in a plan to minimize its overhead [2]. Most systems use a cost model based approach for this decision. A good example is [30], where Vectorwise is shown to have a limited speed-up due to the exchange operator overheads.

As the exchange union operator combines parallelized operator’s result, its expensiveness varies depending on the size of the data being packed. Low selectivity reflects more matching data, hence more data to be packed. The packing overhead is minimized by pushing the exchange union operator as high as possible. It ensures the final data to be packed is relatively small, as it gets filtered by the intermediate operators.

Adaptive parallelization (AP) enables to analyze the exchange union operator’s placement with successive iterations of parallelized plans, as it directly affects the speed-up. In the next two experiments we observe that the AP plan’s speed-up is comparable to the speed-up of the heuristically parallelized (HP) plans. The speed-up gets hindered due to operator dependencies that form critical paths, which can not be parallelized. However, the AP plans benefit by their optimal multi-core utilization due to less partitions, which ensures improved *concurrent* execution performance, as described in Section 4.2.5.

**Select operator adaptive parallelization:** We use query 6 from the TPC-H benchmark to analyze the speedup of the select operator (See Table 2). Query 6 is a simple query with only selection predicates on the *Lineitem* table. We vary the selectivity by varying the parameter *l\_quantity* from the selection predicate. Figure 14 plots the execution time of AP plans on the Y axis with respect to iterations (X axis), when selectivity is varied from 0% (all output) to 100% (no output), and scale factor is varied from 10 GB to 20 GB. We do not plot the graph for 100GB for readability purpose, and only list its speedup in Table 2.

From Table 2 as the selectivity increases the speedup decreases. During low selectivity a single select operator in a serial plan writes a large number of output tuples, as compared to its parallel plan counterparts. This results in the large speedup as serial execution time is much higher, whereas parallel execution time is much lower. During highest selectivity (100%) since there is no output the serial execution is less expensive as compared to 0% selectivity serial execution. This results in lower speedup. The speedup increases with a decrease in the input size. This is a result of lower minimum time

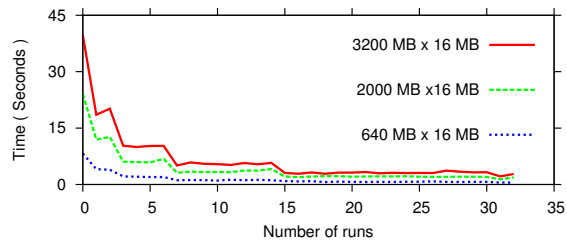


Figure 15: Effect of variations of data size on the speedup of the adaptively parallelized *Join* operator plan. Outer input partitioned and inner input used to build hash table.

during parallel execution, due to less input data. With increased selectivity the speedup for AP is less compared to HP. This is due to the presence of less expensive exchange union operators, which do not get pushed higher in the plan.

Table 3: *Join* operator plan speedup (compared to serial execution) using adaptive and heuristic parallelization.

Size (MB)	AP = Adaptive, HP = Heuristic parallelization			
	64		16 (Smaller Input)	
	(Larger Input)	AP	HP	AP
3200	15.75	14	18.5	18
2000	15	13.5	17.75	17.75
640	13.75	13	17	15

**Join operator adaptive parallelization:** For the join operator (hash) plan parallelization analysis, we partition the outer input and build up the hash table on the inner input. We use a micro-benchmark for a fine grained control, where the outer input has 400 M, 250 M, and 80 M (M = Millions) random tuples of type *long* (8 bytes), and the inner input has 8 M and 2 M tuples. The outer inputs stay larger than the inner input of size 16 MB (2 M tuples) even after 32 partitions (threads on CPU). The 16 MB input fits in the shared L3 cache (20 MB).

Figure 15 shows the join operator plan speedup and Table 3 quantifies it. The speedup of 16 MB input join is more than the 64 MB input join, as the 16 MB input join’s hash table fits partially in the L3 cache (20 MB), which improves the probe phase, due to reduced cache thrashing. Speedup also decreases as the outer table size decreases, as the serial execution time is directly proportional to the outer table size. For all sizes the best speedup is obtained when the number of partitions are 32, with 32 threads (hyper-threading enabled). Maximum speed-up observed is around the number of physical cores (16). Both AP and HP show a similar performance unlike the previous select operator plan analysis case, as the join plan contains only join and union operators.

**Summary:** Adaptive parallelization works for both the select and the join operator and these operators scale linearly with the number of physical cores. Input size, selectivity, and properties such as cache consciousness affects the speedup.

Having analyzed how individual operators affect parallel execution, in the next section we focus on holistic SQL query level analysis, from execution performance and convergence perspective.

## 4.2 SQL query level analysis

### 4.2.1 TPC-H queries

Since the TPC-H benchmark is considered the de-facto workload for performance comparison, in this section we use a subset of queries (see Table 4) from TPC-H (scale factor 10). TPC-H has uniformly distributed data. The adaptively parallelized group-by operator implementation at present supports single attribute group-by queries. Hence, we modify some queries so that they have a single attribute group-by representation. Since we use the same set of queries to evaluate multiple parallelization approaches, the

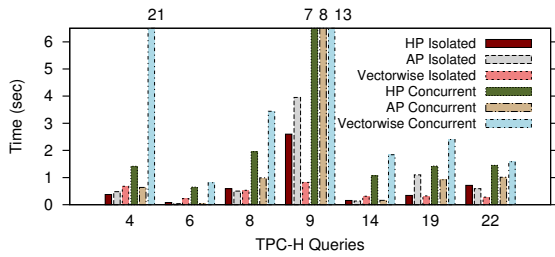


Figure 16: **Heuristic vs adaptive parallelization performance in isolated and concurrent environment for MonetDB and Vectorwise.**

comparison is fair. We plot an average of four executions using the experimental set-up described towards beginning of Section 4.

Table 4: TPC-H queries.

Simple	Q6	Q14			
Complex	Q4	Q8	Q9	Q19	Q22

We compare adaptive parallelization (AP) with heuristic parallelization (HP), the default parallelization technique in MonetDB, under isolated execution setting. HP uses parameters such as the number of threads, physical memory size, and the largest table size to identify the number of partitions for the largest table in the serial plan. A plan re-writer generates a parallel plan from a serial plan by propagating the partitions to data flow dependent operators. Though both HP and AP start with the same serial plan, the final parallel plan is different for both techniques as in AP only the most expensive operator gets parallelized unlike in HP, where all possible parallelizable operators are parallelized. In Figure 16 the first two bars show HP vs AP performance when queries execute in isolation. All AP queries except Q9 and Q19 show similar performance as HP. Q9 and Q19 show a degraded performance due to the presence of some non-parallelizable operators, which prolong the query execution. The robustness of individual query execution could be observed in Figure 18C, where queries show minimal execution time variations. Though the execution performance of adaptive plans is similar to the heuristic plans, the adaptive plans are better as they use much fewer number of partitions (See Table 5). It helps during concurrent workload execution, where adaptive plans exhibit better execution performance due to better resource utilization. Table 5: AP and HP Q14 plan statistics.

	AP	HP
# Select operators	10	65
# Join operators	16	32
% Multi-core Utilization	35	75

#### 4.2.2 TPC-DS queries

TPC-DS benchmark has 25 tables, out of which 6 tables are relatively large (above 1GB in size), in a scale factor 100 dataset. The benchmark supports 99 query templates. We use a few modified queries. These queries are a subset of the original TPC-DS queries and are chosen such that they contain the large tables and a few smaller dimension tables. Since we compare both the adaptive and the heuristic parallelization technique with the same queries, the comparison gives a perspective of their respective performance.

We experiment on both the two socket and the four socket machine (See Table 1 for configuration) with 100GB dataset, to get a perspective of the NUMA effects. Graphs in Figures 17a and 17b show the comparison. Adaptive plans exhibit a maximum of 5 times better performance compared to heuristic plans, which could be attributed to *correct partitioning by adaptive parallelization compared to heuristic parallelization and the skewed data distribution*. The execution time for both two and four socket machine shows similar time, which indicates minimal NUMA effects. As authors in [14] observe, since MonetDB uses a memory mapped representation for the buffer data, as the number of partitions increase, we expect them to get assigned to the memory modules of

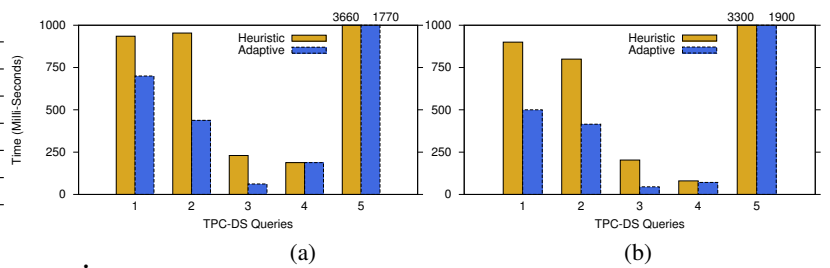


Figure 17: **Isolated execution performance of TPS-DS queries on a) 2 socket machine with 2.00 GHz CPU b) 4 socket machine with 2.40 GHz CPU, on 100GB data.**

the sockets on which operator execution gets scheduled. We also observe a limit on the execution improvement, even though a higher number of cores are used, which indicates increased parallelism need not improve performance beyond a threshold.

#### 4.2.3 Concurrent workload execution

This experiment highlights the effectiveness of adaptively parallelized plans compared to heuristically parallelized plans in a concurrent workload setting. Concurrent query executions in batch workload leads to resource contention, which in turn affects the degree of parallelism of individual queries under execution. Resource contention varies with random workload, however for the current set-up we consider a homogeneous concurrent workload. In Figure 16 the 4th and 5th bar shows HP vs AP execution under a concurrent workload. The workload consists of random simple and complex queries from the TPC-H benchmark (10GB), where 32 clients invoke queries repeatedly. AP Q8 shows 50% improved execution compared to HP Q8. Simple queries such as Q6 and Q14 show around 90% execution improvement in AP. HP plans have too many partitions compared to the AP plans as shown in Table 5. AP plans also reflect the resource contention through execution feedback. Hence, AP plans are more robust and better performing under a concurrent workload, compared to statically generated HP plans. In [11] authors discuss HP vs AP plans comparison under different concurrent workload resource contention scenarios in a detailed manner.

#### 4.2.4 Comparison with Vectorwise

We compare the concurrent workload performance of Vectorwise (version 3.5.1 with histogram build feature enabled to generate optimized plans), a leading analytical columnar database using pipelined vectorized execution [7], with adaptive parallelization in MonetDB. Vectorwise uses cost model based exchange operator dependent parallel plans. The resources are allocated based on the number of connected clients and the system load. During a heavy concurrent workload (32 clients invoking random TPC-H queries repeatedly on 10GB data), the first client's query gets all the resources, while the queries from the remaining clients get less resources based on an admission control scheme. Figure 16 shows MonetDB adaptive parallelized query execution performance is better than the Vectorwise execution performance, during the concurrent workload. MonetDB does not have explicit resource control based plan generation scheme, which helps in the current case. We hypothesize that as workload queries are invoked repeatedly, Vectorwise queries under analysis execute serially due to lack of resources.

#### 4.2.5 Multi-core utilization

This experiment highlights that an AP plan is better than a HP plan from the multi-core utilization perspective. Multi-core utilization represents the fraction of actual CPU cores used versus the available cores during query processing. AP ensures minimal multi-core utilization as each operator is parallelized with a different degree of parallelism unlike HP. Figures 19 and 20 visualize AP vs HP plan execution of TPC-H Q14, in an isolated execution

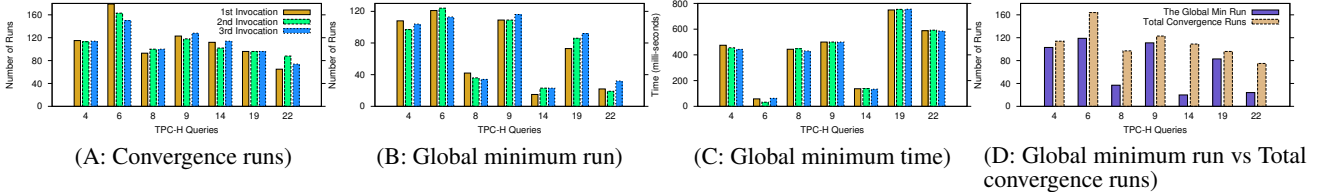


Figure 18: Convergence algorithm shows minimal variations across multiple invocations in the graphs A, B, and C for adaptively parallelized query execution. Graph D shows most queries converge quickly after detection of the global minimum.

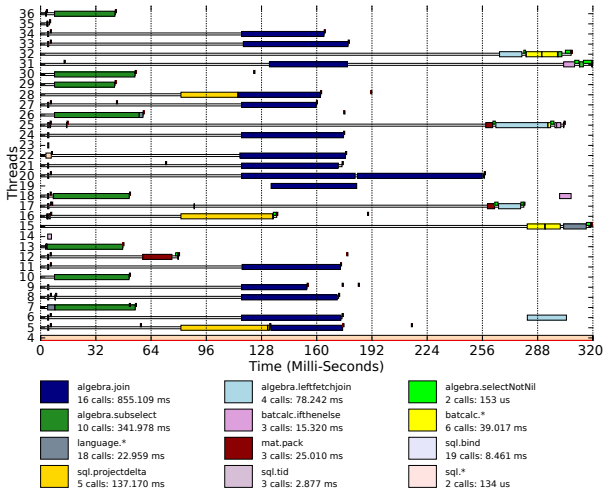


Figure 19: Adaptive parallelization multi-core utilization (35%) during isolated execution of TPC-H Q14. Green- Select, Blue-Join, Brown- Exchange union operator.

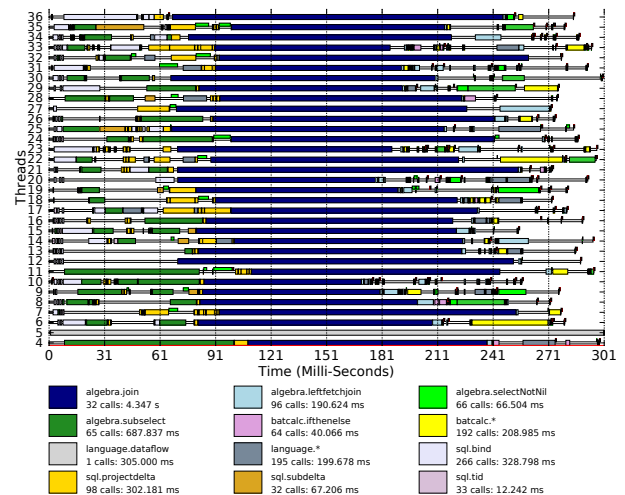


Figure 20: Heuristic parallelization multi-core utilization (75%) during isolated execution of TPC-H Q14. Green- Select, Blue-Join, Brown- Exchange union operator.

setting. The length of a colored box represents an operator’s execution interval (In an operator-at-a-time execution model an operator executes completely. Blue- join, Green-select, Brown- exchange union operator.) A whitespace indicates no execution. The amount of whitespace in Figure 19 is much more than in Figure 20, indicating lower multi-core utilization for AP. In the HP execution (Figure 20) the length of the join operators is much longer than the corresponding operators in the AP execution (Figure 19), which hints at the memory bandwidth pressure. In [13] authors analyze parallel plans in detail using this visual scheme.

The degree of parallelism per operator thus influences the overall multi-core utilization. For example, while only ten select operators execute in AP, many more execute in HP (See Table 5). Since AP shows lower multi-core utilization (35%) during isolated execution, the spare resources ensure better response time during concurrent workload, as further elaborated in [11].

### 4.3 Convergence algorithm robustness

Adaptive parallelization not only should converge in minimal number of runs, but also should exhibit robustness. The robustness implies during *multiple* adaptive parallelization invocations of a query a) the total number of convergence runs b) the run at which the global minimum occurs, and c) the global minimum execution time should not show much variations. In this experiment we test the robustness of the convergence algorithm in an isolated execution setting. Graphs in Figure 18 (A,B,C) show these three cases.

Graph 18(A) shows the number of convergence runs to find the optimal execution time for three invocations. Except for Q6 and Q22 all other queries show minimal variations for convergence runs. Q6 is the most simple query in the given set of queries. It shows the most speed-up amongst all queries, but that also makes it vulnerable to external factors such as operating system noise interference, etc. Since the global minimum time is very low, even small inter-

ference affects its performance. Q22 is a complex query where join operator is always the most expensive operator.

Graph 18(B) shows the run where the global minimum time occurs during three invocations of the query set under evaluation. Depending on the resource contention and the run-time interference from the operating system we get small variations across different runs for all queries. The highest difference is observed between the first and the third run for Q19. However, overall the number of runs do not show much deviations.

Graph 18(C) shows the global minimum time for adaptively parallelized queries for three invocations. The global minimum time for all queries is almost stable across multiple invocations. This indicates the robustness of the generated plans.

Graph 18(D) shows that the queries 4, 6, 9, 19 converge quickly after detection of the global minimum. Different types of queries show different convergence properties and the algorithm gets tuned to converge in the least possible number of runs. For example, Q8, Q14, and Q22 show the global minimum with fewer than 40 runs, while the total convergence runs are around 100. The slow convergence is a result of the *Leaking\_Debit* being too low, which leads to the *credit* getting drained slowly. The convergence runs are close to 60 for the same global minimum, when the *Leaking\_Debit* is high.

**How to lower number of convergence runs?** At present plan parallelization introduces only a single new operator per invocation, which results into a higher number of convergence runs, as the execution skew introduced by a single new operator needs to level out. The number of runs could be made much lower if more and even number of operators are introduced per invocation. We avoid it at present to analyze the parallel plan evolution with each new operator addition.

## 5. RELATED WORK AND APPLICABILITY TO OTHER SYSTEMS

The basic optimizer approach of “optimize once and execute many” as proposed by System R has reached its limits [28]. Hence, adaptive query processing techniques are being proposed to address query optimization problems due to unreliable cardinality estimates, data skew, parameterized query execution, changing workload, complex queries with many tables, etc. [9]. In this section we describe some state of the art adaptive techniques.

**Adaptive aggregation** is used by the authors in [8] to handle different group-by based parallelization cases. The operator performs a lightweight sampling of the input to choose the best aggregation strategy with high accuracy, at runtime. Algorithmic approaches are based on using independent and shared hash tables with locking and atomic primitives to minimize hash table access contention. Three cases are identified that affect performance based on the average run-length of identical group-by values, locality of references to the aggregation hash table, and frequency of repeated access to the same hash table location. This work targets adaptivity from a single operator’s perspective, whereas our work targets it at the plan level. The approach used here could be combined with our adaptive approach to improve per operator performance.

**Vectorwise** uses micro-adaptivity to improve query execution time by using run-time execution feedback [25]. Micro-adaptivity is defined as the ability to choose the most promising execution primitive at run-time, based on real time statistics. Most methods, like adaptive parallelization, use plan level modification, whereas micro-adaptivity uses the available execution primitives at run time. It chooses primitives based on the platform, instance, and call adaptivity using parameters such as compiler, branch prediction, selectivity, loop unrolling, etc.

**The Learning Optimizer (LEO) in DB2** uses query execution feedback for cardinality estimation corrections [29]. It uses learning and feedback based infrastructure to monitor query execution and generates feedback for correction in the cardinality estimation and related statistics. More learning helps in better cost model predictions. LEO has improved query execution performance by orders of magnitude. MonetDB does not use cardinality related statistics, however if used with the statistics correction methods, the selection of operator’s to parallelize could be improved further.

In [4] authors illustrate adaptive parallel execution in Oracle for big data analytics. In Oracle problems such as reliance on query optimizer estimates are handled by changing the data distribution decisions adaptively, during query execution.

Column store architectures differ in various aspects such as plan representation, partitioning strategy, scale out support, etc. Encompassing all the requirements in a single architecture is not possible due to their architectural confinements. While describing the related state of the art column stores, we also describe the possibility of adaptive parallelization’s application to them.

**Vertica** uses a value based partitioning approach [18]. It uses a Read Optimized Store (ROS), where the data is stored in multiple ROS containers on a standard file system. Two files per column within a ROS container are stored, one with the actual column data and the other with position index. This representation is very similar to the representation in Figure 10, where RH is the index, while RT is the actual value. Vertica also supports grouping multiple columns together in a file, however this hybrid row-column storage is rarely used in practice because of the performance and compression penalty it incurs.

Vertica execution engine uses a multi-threaded pipelined vector-

ized execution where the execution plan consists of standard relational algebra operators. Operators such as StorageUnion are used for partitioning data across operators. Hence, StorageUnion is equivalent to a partition operator. Operators such as ParallelUnion are used for directing execution to multiple threads and to combine the parallelized operator’s results. Hence, ParallelUnion is equivalent to an exchange union operator.

To understand the feasibility of applying adaptive parallelization in Vertica, let’s assume that the execution starts with a serial plan and incrementally introduces value based partitions to partition the expensive operator’s data. For example, when a select operator becomes expensive and needs to be parallelized, a partition operator is introduced which creates two value based partitions, which would be consumed by two new select operators. The two new select operator’s output is combined using an exchange union operator. This is similar to the basic mutation scheme with the addition of a partition operator that feeds two newly introduced select operators.

When one of the newly introduced select operators itself becomes expensive, we further partition that operator’s data into two more value based partitions, by introducing a new partition operator in the data flow path after the previous partition operator, and before the input of the expensive select operator. Thus in a hypothetical case when one of the select operators stays expensive during consecutive invocations, new partition operators would keep on getting added to the existing plan. We expect the cost of the partitioning operator to be small considering its presence in the existing Vertica execution plans. Quantifying the exact cost is difficult due to lack of sufficient references. Similar logic could be applied for other operator’s parallelization.

**Apollo** creates column store indexes in a traditional row store database like SQL server [20]. It is the first database which uses the existing row store to create new column store indexes. The method involves creation of batches of rows to create segments from which individual columns are stored in individual column representations. The column segments information is stored in the directory structure, with a catalog.

The columns are compressed and encoded using different types of encoding. New operators called batch operators are introduced which get called if there is bulk data to be processed. The valid rows to process are noted down in bitvector formats.

Apollo uses range partitioning of data. Since traditional SQL server uses cost model based exchange operator induced parallelization, Apollo leverages the existing SQL server parallelization technique using the exchange operator based parallelization.

To understand how adaptive parallelization might be applied in Apollo type of column store, we need to find similarities between adaptive parallelization and Apollo architecture. Both do range partitioning of data, hence the fundamental assumption of range partitioned access stays the same, and could change in the way individual operators are implemented. For example, the operations like the join operator consists of separate build and probe operators, where build uses a shared hash table, where all threads build a hash table, and then probe operator probes it in parallel. As Apollo extends the exchange operator based parallelization as used in SQL server, we expect adaptive parallelization to be useful, due to its dependence on the exchange operator based parallelization.

**Hyper** uses LLVM [21] generated Just In Time (JIT) compiled plans. The longest pipeline in a plan is identified, by looking for a pipeline breaker operator. The operators in the longest pipeline are fused using JIT compilation such that their highly efficient machine language code represents a single task. The fusing allows tuples to be kept in registers to process them without generating intermedi-

ate results. Hyper's morsel driven parallelism uses work stealing based approach to assign the fused pipeline tasks to a fixed number of pre-created threads. The task allocation based approach allows controlling the number of tasks executing in parallel dynamically, at run-time, and allows better control over resource allocation during concurrent execution of queries.

Adaptive parallelization technique is based on the fundamental assumption that an expensive operator is always identifiable in an execution plan. This is a basic requirement since plan parallelization is a result of incrementally parallelizing the expensive operator during successive query invocations, until a global minimum plan is identified.

Identification of a single expensive operator is not feasible in Hyper's execution plans due to the JIT compiled fused nature of operator's pipeline, which prevents a direct application of adaptive parallelization. However, in a broader sense if the entire task is considered to be expensive and treated as an expensive operator, application of the adaptive parallelization logic could be possible. Hence, the feasibility of adaptive parallelization technique in Hyper depends on how to categorize the expensiveness metric.

**DB2 BLU** accelerator [24] uses evaluator chains, which comprises DB2 BLU operators working on columnar data. The data is accessed in strides. It uses novel data structures that minimize latching allowing seamless scaling with multi-cores. Parallelism involves cloning of evaluator chains once per thread, where the number of threads is decided by cardinality estimates, system resources and system load. Each thread requests strides for its evaluator chains until no more strides are available. DB2 BLU also uses work stealing based approach where worker threads operate on tasks comprising of evaluator chain based work.

## 6. CONCLUSION

Adaptive parallelization uses query execution feedback to generate resource contention and skew aware range partitioned multi-core parallel plans. It helps in finding the right balance between the multi-core utilization and the degree of parallelism for the exchange operator based parallel plans. We observe a near linear speedup with the number of cores while analyzing the parallel plan evolution using parameters such as the input size and selectivity. During TPC-DS isolated workload execution, the adaptively parallelized plans show up to five times better performance compared to heuristically parallelized plans. During TPC-H concurrent workload, they show minimal multi-core utilization, allowing better resource utilization. They also fare competitively with work stealing based scheduling approach.

Using different convergence scenarios we show that the adaptive parallelization convergence algorithm behaves robustly, and converges in a reasonable number of runs.

## 7. ACKNOWLEDGEMENT

We thank the MonetDB team for their support. This project is funded through COMMIT grant WP-02.

## 8. REFERENCES

- [1] Sql server parallelization. "<http://www.simple-talk.com/sql/learn-sql-server/understanding-and-using-parallelism-in-sql-server/>".
- [2] K. Anikiej. Multi-core parallelization of vectorized query execution. <http://homepages.cwi.nl/~boncz/msc/2010-KamilAnikiej.pdf>.
- [3] R. Appuswamy, C. Gkantsidis, et al. Nobody ever got fired for buying a cluster. Technical report, Microsoft Technical Report MSR-TR-2013, 2013.
- [4] S. Bellamkonda et al. Adaptive and big data scale parallel execution in oracle. *Proc of VLDB*, 6(11):1102–1113, 2013.
- [5] R. D. Blumofe et al. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.
- [6] P. Boncz and M. Kersten. Mil primitives for querying a fragmented world. *The VLDB Journal*, 8(2):101–119, 1999.
- [7] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.
- [8] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *Proc of VLDB*, pages 339–350. VLDB Endowment, 2007.
- [9] A. Deshpande et al. Adaptive query processing: why, how, when, what next? In *Proc of VLDB*, pages 1426–1427, 2007.
- [10] S. Ganguly et al. Query optimization for parallel execution. In *ACM SIGMOD Record*, volume 21, pages 9–18, 1992.
- [11] M. Gawade and M. Kersten. Multi-core query parallelism under resource contention - under review vldb 2016. <https://sites.google.com/site/confproceed/AdParConcur.pdf>.
- [12] M. Gawade and M. Kersten. Stethoscope: a platform for interactive visual analysis of query execution plans. *Proc of VLDB*, 5:1926–1929, 2012.
- [13] M. Gawade and M. Kersten. Tomograph: Highlighting query parallelism in a multi-core system. In *Proc of DBTest*, page 3. ACM, 2013.
- [14] M. Gawade and M. Kersten. Numa obliviousness through memory mapping. In *Proc of DAMON*, page 7, 2015.
- [15] G. Graefe. *Encapsulation of parallelism in the Volcano query processing system*, volume 19. ACM, 1990.
- [16] W. Hong. *Parallel query processing using shared memory multiprocessors*. PhD thesis, UC, Berkeley, 1992.
- [17] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *Proc of SIGMOD*, pages 297–308. ACM, 2009.
- [18] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *Proc of VLDB*, 5(12):1790–1801, 2012.
- [19] R. S. Lancelotte et al. On the effectiveness of optimization search strategies for parallel execution spaces. In *VLDB*, volume 93, pages 493–504, 1993.
- [20] P.-Å. Larson et al. Sql server column store indexes. In *Proc of Sigmod*, pages 1177–1184, 2011.
- [21] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.
- [22] Z. Majo et al. Memory system performance in a numa multicore multiprocessor. In *Proc of ICSS*, page 12, 2011.
- [23] C. Mohan. Impact of recent hardware and software trends on high performance transaction processing and analytics. In *Proc of PEMCCS*, pages 85–92. Springer, 2011.
- [24] V. Raman et al. Db2 with blu acceleration: So much more than just a column store. *Proc of VLDB*, pages 1080–1091, 2013.
- [25] B. Răducanu, P. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In *Proc of SIGMOD*, pages 1231–1242, 2013.
- [26] R. A. Rutenbar. Simulated annealing algorithms: An overview. *Circuits and Devices, IEEE*, 5(1):19–26, 1989.
- [27] M. Saecker and V. Markl. Big data analytics on modern hardware architectures: A technology survey. In *Business Intelligence*, pages 125–149. Springer, 2013.
- [28] P. G. Selinger, M. M. Astrahan, Chamberlin, et al. Access path selection in a relational database management system. In *Proc of SIGMOD*, pages 23–34, 1979.
- [29] M. Stillger, G. M. Lohman, V. Markl, et al. Leo-db2's learning optimizer. In *Proc of VLDB*, pages 19–28, 2001.
- [30] L. Viktor et al. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proc of SIGMOD*, 2014.



# PARAGON: Parallel Architecture-Aware Graph Partition Refinement Algorithm

Angen Zheng  
University of Pittsburgh  
Pittsburgh, PA, USA  
anz28@cs.pitt.edu

Alexandros Labrinidis  
University of Pittsburgh  
Pittsburgh, PA, USA  
labrinid@cs.pitt.edu

Patrick Pisciuneri  
University of Pittsburgh  
Pittsburgh, PA, USA  
php8@pitt.edu

Panos K. Chrysanthis  
University of Pittsburgh  
Pittsburgh, PA, USA  
panos@cs.pitt.edu

Peyman Givi  
University of Pittsburgh  
Pittsburgh, PA, USA  
pgivi@pitt.edu

## ABSTRACT

With the explosion of large, dynamic graph datasets from various fields, graph partitioning and repartitioning are becoming more and more critical to the performance of many graph-based Big Data applications, such as social analysis, web search, and recommender systems. However, well-studied graph (re)partitioners usually assume a homogeneous and contention-free computing environment, which contradicts the increasing communication heterogeneity and shared resource contention in modern, multicore high performance computing clusters. To bridge this gap, we introduce PARAGON, a *parallel architecture-aware* graph partition refinement algorithm, which mitigates the mismatch by modifying a given decomposition according to the nonuniform network communication costs and the contentiousness of the underlying hardware topology. To further reduce the overhead of the refinement, we also make PARAGON itself architecture-aware.

Our experiments with a diverse collection of datasets showed that on average PARAGON improved the quality of graph decompositions computed by the de-facto standard (hashing partitioning) and two state-of-the-art streaming graph partitioning heuristics (deterministic greedy and linear deterministic greedy) by 43%, 17%, and 36%, respectively. Furthermore, our experiments with an MPI implementation of Breadth First Search and Single Source Shortest Path showed that, in comparison to the state-of-the-art streaming and multi-level graph (re)partitioners, PARAGON achieved up to 5.9x speedups. Finally, we demonstrated the scalability of PARAGON by scaling it up to a graph with 3.6 billion edges using only 3 machines (60 physical cores).

## 1. INTRODUCTION

It is well-known that graph (re)partitioning has been extensively studied in the area of scientific simulations [14, 34]. Yet, its importance is continuously increasing due to the explosion of large graph datasets from various fields, such as the World Wide Web, Pro-

tein Interaction Networks, Social Networks, Financial Networks, and Transportation Networks. This has led to the development of graph-specialized parallel computing frameworks, e.g., Pregel [21], GraphLab [19], and PowerGraph [13].

Pregel, as a representative of these computing frameworks, embraces a vertex-centric approach where the graph is partitioned across multiple servers for parallel computation. Computations are often divided into a sequence of *supersteps* separated by a global synchronization barrier. During each superstep, a user-defined function is computed against each vertex based on the messages it received from its neighbors in the previous step. The function can change the state and outgoing edges of the vertex, send messages to the neighbors of the vertex, or even add or remove vertices/edges to the graph.

**Traditional Graph Partitioners** Clearly, the distribution of the graph data across servers may impact the performance of target applications significantly. *Graph partitioning* has been studied for decades [14, 34], attempting to provide a good partitioning of the graph data, whereby both the skewness and the communication (edge-cut) among partitions are minimized as much as possible, in order to minimize the total response time for the entire computation. However, classic graph partitioners such as METIS [23] and Chaco [7] do not scale well with large graphs.

**Streaming Graph Partitioners** Streaming graph partitioners (e.g., DG/LDG [39], arXiv'13 [11], and Fennel [42]) have been proposed in order to overcome the scalability challenges of classic graph partitioners, by examining the graph incrementally. One of the main shortcomings of these approaches is that they also assume uniform network communication costs among partitions as classic graph partitioners do. That is, they all assume that the communication cost is proportional only to the amount of data communicated among partitions. *This assumption is no longer valid in modern parallel architectures due to the increasing communication heterogeneity* [47, 8]. For example, on a  $4 * 4 * 4$  3D-torus interconnect, the distance to different nodes starting from a single node varies from 0 to 6 hops.

**Architecture-Aware Graph Partitioners** Architecture-aware graph partitioners [24, 8, 46] have been proposed to improve the mapping of the application's communication patterns to the underlying hardware topology. Chen et al. [8] (SoCC'12) took architecture-awareness a step further, by making the partitioning algorithm itself partially aware of the communication heterogeneity. However, both [8] and [24] (ICA3PP'08) are built on top of existing heavyweight graph partitioners, namely, METIS [23] and PARMETIS [30], which

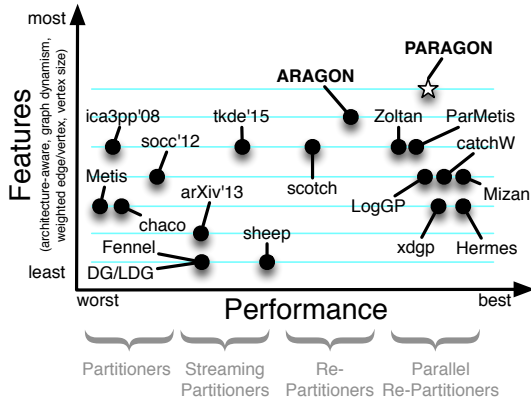


Figure 1: Classification of all graph partitioners/re-partitioners according to their features vs performance profile.

are known to be the best graph partitioners and repartitioners in terms of partitioning quality but have poor scalability. Finally, although Xu et al. [46] (TKDE'15) proposed a lightweight architecture-aware streaming graph partitioner, the partitioner may lead to sub-optimal performance for dynamic graphs [43].

**Traditional Graph Repartitioners** Most real-world graphs are often non-static, and continue to evolve over time. Because of this *graph dynamism*, both the quality of the initial partitioning and the mapping of the application communication pattern to the underlying hardware topology will continuously degrade over time, leading to (a potentially significant) load imbalance and additional communication overhead. Considering the sheer scale of real-world graphs, repartitioning the entire graph from scratch using [46, 8, 24], even in parallel, is often impractical, either because of the long partitioning time or the huge volume of data migration the repartitioning may introduce. To address this, several graph repartitioning algorithms have been proposed, such as Zoltan [1, 6] and PARMETIS [30, 33]. Although they are able to greatly reduce the data migration cost, they are all architecture-agnostic and do not scale well with massive graphs.

**Parallel/Lightweight Graph Repartitioners** Parallel lightweight graph repartitioners (e.g., CatchW [37], xdgp [43], Hermes [26], Mizan [17], arXiv'13 [11], and LogGP [45]) have been proposed to improve the performance and scalability of graph repartitioning. Instead of seeking an optimal partitioning at once, these algorithms adapt the graph decomposition to changes efficiently by incrementally migrating vertices from one partition to another based on some local heuristics. However, they are all oblivious of the nonuniform network communication costs among partitions.

**Limitations of the State-of-the-Art** Despite the plethora of graph partitioners and repartitioners (Figure 1), the current state-of-the-art is suffering from two main problems:

- Graph (re)partitioners either consider architecture-awareness (for CPU/network heterogeneity) or consider performance (i.e., parallel/lightweight implementation), but never both. This is illustrated in Figure 1, where the top-right corner is empty (except for PARAGON, which is presented in this paper).
- No existing graph (re)partitioner considers the issue of *shared resource contention* in modern multicore high performance computing (HPC) clusters. Shared resource contention is a well-known issue in multicore systems and has received a lot of attention in system-level research [15, 41].

**Our prior work** We have previously presented an architecture-aware graph repartitioner, ARAGONLB [48]. Although ARAGONLB considers the communication heterogeneity for target applications, it disregards the issue of shared resource contention, and the repartitioning itself is not architecture-aware. Moreover, the refinement algorithm that ARAGONLB uses to improve the mapping of the application communication pattern to the underlying hardware topology requires the entire graph to be stored in memory by a single server, which is infeasible for large graphs. Furthermore, the refinement algorithm is performed sequentially, which may become a performance bottleneck. Finally, ARAGONLB assumes that compute nodes used for parallel computation have the same number of cores and memory hierarchies, which may not always be true.

**Contributions** In this paper, we present PARAGON, which overcomes both limitations of the current state-of-the-art graph repartitioners by extending ARAGONLB in the following aspects.

1. We separate the refinement algorithm, ARAGON, from ARAGONLB as an independent component, and develop a parallelized version of ARAGON, PARAGON, for large graphs (Section 3, 4, 5). We further reduce the overhead of PARAGON by making it aware of the nonuniform network communication costs (explained in Section 2.1).
2. We identify and consider the issue of shared resource contention in modern HPC clusters for graph partitioning (Section 2.2 & 6).
3. We perform an extensive experimental study of PARAGON with a diverse set of 13 datasets and two real-world applications, demonstrating the effectiveness and scalability of PARAGON (Section 7).

## 2. MOTIVATION

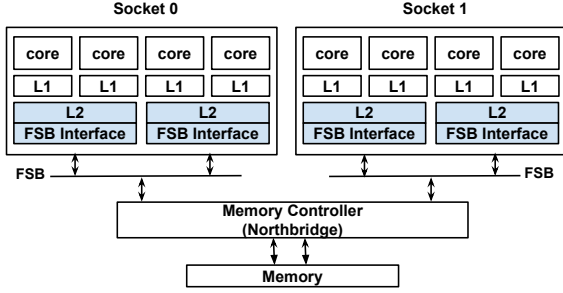
In this section we explain the importance of architecture-awareness (i.e., communication heterogeneity and shared resource contention) for efficient graph (re)partitioners.

### 2.1 Communication Heterogeneity

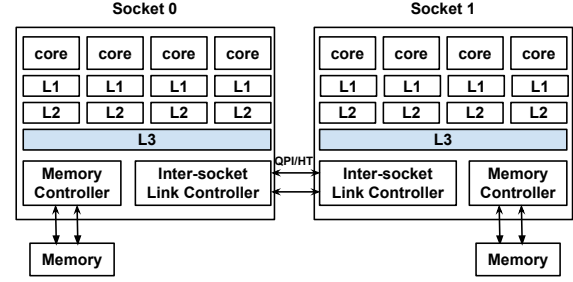
For distributed graph computations on multicore systems, communication can be either *inter-node* (i.e., among cores of different compute nodes) or *intra-node* (i.e., among cores of the same compute node). In general, intra-node communication is an order of magnitude faster than inter-node communication. This is because in many modern parallel programming models like MPI [27, 25], a predominant messaging standard for HPC applications, intra-node communication is implemented via shared memory/cache [16, 5], while inter-node communication needs to go through the network interface. Additionally, both inter-node and intra-node communication are themselves nonuniform.

**Nonuniform Inter-Node Network Communication** Modern parallel architectures, like supercomputers, usually consist of a large number of compute nodes linked via a network. Consequently, the communication costs among compute nodes vary a lot because of their varying locations. For example, in the Gordon supercomputer [28], the network topology is a 4x4x4 3D torus of switches with 16 compute nodes attached to each switch. As a result, the distance to different compute nodes starting from a single node varies from 0 to 6 hops. Also, supercomputers often allow multiple jobs to concurrently run on different compute nodes and contend for the shared network links, limiting the effective network bandwidth available for each job and thus amplifying the heterogeneity.

**Nonuniform Intra-Node Network Communication** Communication among cores of the same compute node is also nonuniform because of the complex memory hierarchy. Communication among



(a) Uniform Memory Access (UMA) Architecture



(b) Nonuniform Memory Access (NUMA) Architecture

Figure 2: Example Architectures of Modern Compute Nodes

cores sharing more cache-levels can achieve lower latency and higher effective bandwidth than cores sharing fewer cache-levels. For example, in the architecture described by Figure 2a, communication among cores sharing L2 caches (e.g., between the first and second core of Socket 0) offers the highest performance, while communication among cores of the same socket but not sharing any L2 cache (e.g., between the first and third core of Socket 0) delivers the next highest performance. Communication among cores of different sockets performs the worst. Similarly, in Figure 2b, cores of the same socket (intra-socket communication) usually communicate faster than cores residing on different sockets (inter-socket communication). This is because intra-socket communication can be achieved via the shared caches, while inter-socket communication has to go through the front-side bus and the off-chip memory controller (Figure 2a) or the inter-socket link controller (Figure 2b).

**Take-away** *To improve the performance of graph-based big-data applications, we should not only minimize the number of edges across different partitions (edge-cut), but also the number of edge-cuts among partitions having higher network communication costs (hop-cut).* This is the major difference between architecture-agnostic solutions (that only minimize edge-cut) and architecture-aware ones (that try to minimize both edge-cut and hop-cut).

## 2.2 Intra-Node Shared Resource Contention

As mentioned above, MPI intra-node communication is implemented via shared memory, which can either be user-space or kernel-based [16, 5]. Current MPI implementations often use the former for small messages and the latter for large messages. The user-space approach requires two memory copies. The sender first needs to load the send buffer into its cache and then write the data to the shared buffer (which may require loading the shared buffer block into the sender’s cache first). Then, the receiver reads the data from the shared memory (which may demand loading the shared memory block and receiving buffer into the receiver’s cache first). For kernel-based approaches, the receiver first loads the send buffer directly to its cache with the help of the OS kernel. Then, the receiver writes the data to the receiving buffer (which may require loading the receiving buffer into its cache first). Clearly, kernel-based approaches reduce the number of memory copies to be one, mitigating the traffic on the memory subsystem. However, it demands a trap to the kernel on both the sender and receiver, making it inefficient for small messages. As can be seen, intra-node communication generates lots of memory traffic and cache pollution, which may saturate the memory subsystem if we put too much communication within each compute node. This issue is further amplified by the increasing contentiousness of the shared resources in modern multicore systems. Table 1 summarizes the resources that different cores may have to compete for when they are communicating with each other for the architectures presented in Figures 2a and 2b. The

Table 1: Intra-Node Shared Resource Contention

Cores/Resources Core Groups		Sharing		Contention		
		Socket	LLC	LLC	FSB/QPI(HT)	Memory Controller
UMA Fig. 2a	G1	✓	✓	✓	✓	✓
	G2	✓			✓	✓
	G3					✓
NUMA Fig. 2b	G1	✓	✓	✓		✓
	G2				✓	

summary is based on whether the cores are on the same socket and whether they share the last level cache (LLC).

**Take-away** *Focusing solely on placing neighboring vertices as close as possible is not sufficient to achieve superior performance. In fact, putting too much communication within each compute node may even hurt the performance due to the traffic congestion on memory subsystems.* Counter-intuitively, offloading a certain amount of intra-node communication across compute nodes may sometimes achieve better performance. This is because inter-node communication is often implemented using Remote Direct Memory Access (RDMA) and rendezvous protocols [40], which allow a compute node to read data from the memory of another compute node without involving the processor, cache, or operating system of either node, thus alleviating the traffic on memory subsystems and cache pollution. Additionally, it is reported in [3] that modern RDMA-enabled networks can deliver comparable network bandwidth as that of memory channels. This requires us to examine the impact of multi-core architecture on graph partitionings more carefully, especially for small HPC clusters, since the network may no longer be the bottleneck.

## 3. PROBLEM STATEMENT

Let  $G = (V, E)$  be a graph, where  $V$  is the vertex set and  $E$  is the edges set, and  $P$  be a partitioning of  $G$  with  $n$  partitions, where

$$P = \{P_i : \cup_{i=1}^n P_i = V \text{ and } P_i \cap P_j = \phi \text{ for any } i \neq j\} \quad (1)$$

and let  $M$  be the current assignment of partitions to servers, where  $P_i$  is assigned to server  $M[i]$ . The server can be either a hardware thread, a core, a socket, or a machine.

**Architecture-aware graph partition refinement** aims to improve the mapping of the application communication pattern to the underlying hardware topology by modifying the current partitioning of the graph, such that the communication cost of the target application, given the specific hardware topology, is minimized. The modification usually involves migrating vertices from one partition to another partition. Hence, in addition to the communication cost, the refinement should also minimize the data migration cost among partitions. Also, to ensure balanced load distribution in terms of the computation requirement, the refinement should keep the skewness of the partitioning as small as possible.

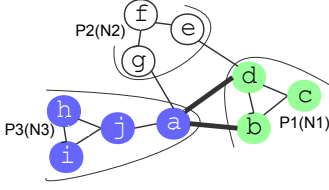


Figure 3: Old Decomposition

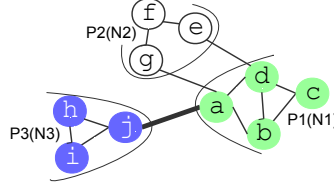


Figure 4: Better Decomposition

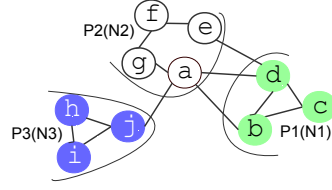


Figure 5: Best Decomposition

	$N_1$	$N_2$	$N_3$
$N_1$		1	6
$N_2$	1		1
$N_3$	6	1	

Figure 6: Relative Network Communication Costs

We define the *communication cost* of a partitioning  $P$  as:

$$comm(G, P) = \alpha * \sum_{\substack{e=(u,v) \in E \\ \text{and } u \in P_i \text{ and } v \in P_j \text{ and } i \neq j}} w(e) * c(P_i, P_j) \quad (2)$$

where  $\alpha$  specifies the relative importance between communication and migration cost, which is usually set to be the number of supersteps carried out between two consecutive refinement/repartitioning steps,  $w(e)$  is the edge weight, indicating the amount of data communicated along the edge per superstep, and  $c(P_i, P_j)$  can be either the relative network communication cost, the degree of shared resource contentiousness between  $P_i$  and  $P_j$  or a hybrid of both. Existing architecture-agnostic graph (re)partitioners usually assume  $c(P_i, P_j) = 1$ .

The *migration cost* of the refinement is defined as:

$$mig(G, P, P') = \sum_{\substack{v \in V \\ \text{and } v \in P_i \text{ and } v \in P'_j \text{ and } i \neq j}} vs(v) * c(P_i, P'_j) \quad (3)$$

where  $vs(v)$  is the vertex size, reflecting the amount of application data represented by  $v$ , and  $P'$  denotes the partitioning after being refined/repartitioned.

The *skewness* of a partitioning,  $P$ , is defined as:

$$skewness(G, P) = \frac{\max\{w(P_1), w(P_2), \dots, w(P_n)\}}{\frac{\sum_{i=1}^n w(P_i)}{n}} \quad (4)$$

where  $w(P_i) = \sum_{v \in P_i} w(v)$  with  $w(v)$  denoting the vertex weight (i.e., the computation requirement of the vertex).

**Self Architecture-Awareness** In fact, the refinement algorithm itself should be architecture-aware (during its execution), since the refinement may also result in a lot of communication.

## 4. OUR PRIOR WORK: ARAGON

ARAGON is a serial, architecture-aware graph partition refinement algorithm proposed by us in [48]. It is a variant of the Fiduccia-Mattheyses (FM) algorithm [12]. It tries to reduce the application communication cost by modifying the current decomposition according to the nonuniform network communication costs of the underlying hardware topology. Each time it takes as input two partitions of the  $n$ -way decomposition and the relative network communication costs among partitions. For each input partition pair, it attempts to improve the mapping of the application communication pattern to the underlying hardware topology by iteratively moving vertices between them. During each iteration, it tries to find a single vertex such that moving it from its current partition to the alternative partition would lead to a maximal gain, where the gain is defined as the reduction in the communication and migration cost. Upon each movement of a vertex,  $v$ , it also updates the gain of  $v$ 's neighbors of the partition pair. This process is repeated until all vertices are moved once or the decomposition cannot be further improved after a certain number of vertex movements. Since

ARAGON can only refine one partition pair at a time, it is repeatedly applied to all partition pairs sequentially.

The gain of moving vertex  $v$  from its current partition,  $P_i$ , to its refinement partner,  $P_j$ , is defined as:

$$g^{i,j}(v) = g_{std}^{i,j}(v) + g_{topo}^{i,j}(v) + g_{mig}^{i,j}(v) \quad (5)$$

Here,  $g_{std}^{i,j}(v)$  considers the impact of the movement on the communication between  $P_i$  and  $P_j$ , defined as:

$$g_{std}^{i,j}(v) = \alpha * (d_{ext}(v, P_j) - d_{ext}(v, P_i)) * c(P_i, P_j) \quad (6)$$

where  $d_{ext}(v, P_i)$  denotes the amount of data  $v$  communicates with vertices of partition  $P_i$ , formally defined as

$$d_{ext}(v, P_i) = \sum_{\substack{e=(v,u) \in E \\ \text{and } v \in P_i \text{ and } u \in P_j \text{ and } i \neq j}} w(e) \quad (7)$$

The second term of Equation 5,  $g_{topo}^{i,j}(v)$ , considers the impact of the movement on the communication between  $v$  and its neighbors in other partitions in addition to  $P_i$  and  $P_j$ . We define it as:

$$g_{topo}^{i,j}(v) = \alpha * \sum_{\substack{k=1 \\ \text{and } k \neq i \text{ and } k \neq j}}^n d_{ext}(v, P_k) * (c(P_i, P_k) - c(P_j, P_k)) \quad (8)$$

The third term of Equation 5,  $g_{mig}^{i,j}(v)$ , considers the impact of the movement on migration cost, which is defined as:

$$g_{mig}^{i,j}(v) = vs(v) * (c(P_i, P_k) - c(P_j, P_k)) \quad (9)$$

where  $P_k$  is the owner of  $v$  in the original decomposition. The current owner of  $v$ ,  $P_i$ , may be different from its original owner,  $P_k$ , due to the refinement.

**Example** In the decomposition shown in Figure 3, we have a graph with unit weights and sizes and is initially distributed across 3 machines:  $N_1$ ,  $N_2$ , and  $N_3$ . The relative network communication costs among partitions are shown in Figure 6. Clearly, the number of edges among partitions goes from 4 in Figure 3, to 3 in Figure 4. In fact, if we assume uniform network communication costs among partitions, Figure 4 would be the optimal decomposition of the graph. However, if we consider the case where all network costs are not equal (as in Figure 6), then the decomposition in Figure 4 can be further improved by moving vertex  $a$  to  $P_2$  (Figure 5). Even though moving vertex  $a$  from  $P_1$  to  $P_2$  increases the communication cost between  $P_1$  and  $P_2$  by 1, it actually reduces the communication cost between  $a$  and  $j$  by 5, since the relative network communication cost between  $P_1$  and  $P_3$  is 6, while that of  $P_2$  and  $P_3$  is 1. For the same reason, moving  $a$  to  $P_2$  also decreases the migration cost of  $a$  by 5, since vertex  $a$  was originally in  $P_3$ .

## 5. PARAGON

**Motivation** Clearly, one naive implementation of ARAGON could be as follows: server  $M[i]$  is responsible for the refinement of  $P_i$  with all its partners  $P_{i+1}, P_{i+2}, \dots, P_n$ , and server  $M[i+1]$  can

not start its refinement for  $P_{i+1}$  until server  $M[i]$  finishes its refinement. One major issue of this approach is that it requires the entire graph to be sent across network  $\frac{n-1}{2}$  times. An advantage of this approach is that each server only needs to hold two partitions in memory at a time (one for its local partition and the other one for the refinement partner). In our prior work [48], ARAGON goes for another extreme, where all servers send their local partitions to a single server that is responsible for the refinement of all partition pairs. By doing this, ARAGON only needs to send the entire graph over network once, significantly reducing the communication traffic. One drawback of this approach is that it requires the server to store the entire graph in memory. Another issue is that the server can easily become a performance and scalability bottleneck.

**Overview** Based on the observation above, PARAGON takes a middle point of the two extremes, where it allows multiple servers to do the refinement in parallel, each of which is responsible for the refinement of a group of partitions. In this way, we can enjoy the benefits of both extremes without worrying about their drawbacks. Algorithm 1 describes the main idea of PARAGON. During refinement, each server runs an instance of the algorithm with its local partition  $P_l$  and the relative network communication cost matrix  $c$  as its input. The algorithm first selects a server as master node (Line 1), and then computes everything needed by the master node to make the parallelization decision (Line 2). The master node decides how to split partitions into groups such that each group can be refined independently on different servers and the selection of group servers (Line 4–6). The group servers take responsibility of the refinement of each group. Once the decision has been made, each server will send their vertices to the corresponding group servers (Line 7). Upon receiving all the vertices from their group members, group servers will start to do the refinement of each group independently (Line 8–13). After finishing the refinement of its group, group servers will notify their group members about the new locations of their vertices (Line 15). Then, each server will physically migrate vertices to their new owners accordingly (Line 16).

**Partition Grouping** To assign a partition to a group, we consider three factors: (1) to minimize the refinement time, each group should have roughly equal number of partitions; (2) members of each group should be carefully selected, since the gain of refining each partition pair may vary a lot. Thus, to maximize the effectiveness of refinement, we should group together partitions leading to high refinement gain; and (3) we should minimize the cross-group refinement interference, because the gain of refining one partition pair heavily relies on the amount of data they communicate with other partitions. This is different from the standard FM algorithms, which solely compute the gain of migrating each vertex based on the data it communicates with vertices of the partition pair. For example, in the decomposition of Figure 4, the communication between vertex  $a$  and  $j$  contributes most to the gain of moving  $a$  from  $P_1$  to  $P_2$  for PARAGON. However, for standard FM algorithms, the gain of migrating  $a$  to  $P_2$  will be -1, since  $a$  has two neighbors in  $P_1$  and 1 in  $P_2$ . Unfortunately, there is no clear way to do the grouping, since we could not use the state-of-the-art graph partitioners (i.e., METIS) to compute a high-quality initial decomposition, due to their poor scalability. As a result, the input decomposition to PARAGON will probably have edge-cuts across all partitions. Fortunately, we find that random grouping along with the *shuffle refinement* (the remedy technique presented below) works quite well.

**Shuffle Refinement** To mitigate the impact of cross-group refinement interference and increase the gain of the refinement, we perform an additional round of refinement once all the group servers finish the refinement of their own groups. We call this *shuffle refine-*

---

**Algorithm 1: PARAGON**


---

**Data:**  $P_l, c$   
**Result:** new locations of vertices of  $P_l$

```

1  masterNodeSelection( $c$ )
2  partitionStat( $P_l, ps$ )
3  if server  $M[l]$  is master node then
4  |    $pg = \text{partitionGrouping}()$ 
5  |    $gs = \text{optGroupServerSelection}(pg, ps, c)$ 
6  |   partitionGroupServerBcast( $gs$ );
7  sendPartitionToGroupServers( $P_l, gs$ )
8  if server  $M[l]$  is a group server then
9  |    $pg = \text{recvPartitionsFromMyGroupMembers}(gs)$ 
10 |   foreach  $P_i \in pg$  do
11 |     |   foreach  $P_j \in pg$  do
12 |       |   if  $i \neq j$  then
13 |         |   AragonRefinement( $P_i, P_j, c$ )
14 |     shuffleRefinement( $pg$ )
15 |     vertexLocationUpdate( $pg$ )
16 physicalDataMigraton( $P_l$ )

```

---

*ment*. In this round, each group server first exchanges the changes it made to the decompositions such that each group server has the up-to-date load information of each partition and the up-to-date locations of the neighbors of each vertex. Then, each group server swaps some of its partitions randomly with other group servers. Subsequently, each group server starts another round of refinement with the new grouping.

The reason why shuffle refinement is a remedy to the above issue is because it increases the number of partition pairs refined by PARAGON and thus the solution space that PARAGON explores. For example, for a graph with 4 partitions and 2 groups, PARAGON originally only refines 2 out of the 6 partition pairs. However, if the group servers swap one of their partitions, PARAGON will refine 4 partition pairs instead of 2. In fact, we can repeat this shuffle refinement multiple times to further expand the solution space PARAGON explores, thus further alleviating the impact of cross-group refinement interference and increasing the gain we can obtain.

The idea of shuffle refinement is very straightforward, but it is not easy to efficiently implement, especially the propagation of the changes that each group server made. One easy way to achieve this is to use a distributed data directory, like the one provided by Zoltan [1]. In this scheme, each group server only needs to make an update to the data directory first, and then all the group servers can pull the up-to-date locations for the neighbors of their vertices. We found that this approach is very inefficient for really big graphs in terms of both memory footprint and execution time. It requires around  $O(|V|+|E|)$  data communication.

Another way to achieve this is to maintain an array at each group server, forming a mapping from vertex global identifiers<sup>1</sup> to their locations. In this way, the exchange can easily be achieved via a single (MPI) reduce operation, requiring only  $O(|V|)$  data communication. This approach is much more efficient than the distributed data directory approach in terms of execution time, but it is not memory scalable for large graphs.

In our implementation, we adopt a variant of the second approach. That is, we first chunk the entire global vertex identifier space into multiple smaller equal sized regions. Each region contains vertices within a contiguous range. By default, the region size

<sup>1</sup>In distributed graph computation, each vertex has a unique global identifier across all partitions and a unique local identifier within each partition.

equals  $k = \min\{2^{26}, |V|\}$ , where  $V$  is the vertex set of the entire graph. Correspondingly, the exchange is split into multiple rounds. Each round only exchanges the locations of vertices of one region. With this scheme, we only need to maintain a smaller array at each group sever and thus the amount of data communication remains unchanged. Although this scheme requires scanning the edge lists of each partition multiple times, it is much more efficient than the distributed data directory approach.

**Degree of Refinement Parallelism** Theoretically, the number of groups we can have can be any integer between 1 and  $\frac{n}{2}$ , where  $n$  is the number of partitions of the graph. Clearly, if the number of groups equals 1, PARAGON degrades to ARAGON, in which all servers will send their local partitions to a single group server for sequential refinement. The reason why there is an upper bound is because each group needs to have at least 2 partitions for the refinement to proceed. Typically, the higher the number is, the faster the refinement will finish. However, there is a tradeoff between the degree of parallelism and the quality of the resulting decomposition we can have. This is because the higher the number is, the fewer partitions each group will have and thus the fewer partition pairs will be refined. Given a graph with  $n$  partitions and  $m$  groups, PARAGON only refines  $\frac{n(n-m)}{2m}$  partition pairs, while ARAGON refines all  $\frac{n(n-1)}{2}$  partition pairs. In other words, ARAGON will eventually select an optimal migration destination among all partitions for each vertex, whereas PARAGON only considers a subset of the partitions for each vertex. This also explains the reason why the resulting decompositions computed by PARAGON are usually poorer than those of ARAGON. Fortunately, the shuffle refinement technique we proposed helps to address the issue.

**Group Server Selection** Once the master node finishes the grouping process, it will select an optimal server for each group, such that the cost of sending partitions of the group to the group server is minimized. For example, in case of Figure 4, where we assume that  $P_1, P_2$ , and  $P_3$  are of one group, we should select server  $M[2]$  as the group server intuitively since  $c(P_1, P_2) = c(P_2, P_3) = 1$  while  $c(P_1, P_3) = 6$ . To achieve this, we define the cost of selecting server  $M[s]$  as the group server for group  $g$  as:

$$\sum_{P_i \in g} ps[i] * c(P_i, P_s) * (1 + \frac{\sigma(s)}{drp}) \quad (10)$$

Here,  $ps[i]$  denotes the number of edges associated with vertices of  $P_i$ , which is a good approximation for the amount of data each server needs to send to their group servers.  $\sigma(s)$  is the number of group servers that have been designated on the compute node that server  $M[s]$  belongs to. It should be noticed that server  $M[s]$  can be a hardware thread, a core, a socket, or a machine.  $drp$  is the degree of refinement parallelism (number of group servers). The last term  $(1 + \frac{\sigma(s)}{drp})$  is the penalty that is added to avoid the concentration of multiple group servers into a single compute node, reducing the chance of memory exhaustion. Once all group servers are selected, the master node will broadcast the group servers of all groups to all slave nodes. Then, each server will send its vertices (as well as their edge lists) to their corresponding group servers, after which the group servers will start to refine partitions of their own groups independently.

**Reducing Communication Volume** Clearly, PARAGON with the shuffle refinement disabled requires the entire graph to be sent over the network once, and PARAGON with the shuffle refinement enabled demands more data communication. For really big graphs, the communication volume may get very high. Thus, we follow the same approach proposed in [35] to reduce the communication

volume. Specifically, instead of sending the entire partition to their group servers, each server only needs to send vertices that can be reached by a breadth-first search from boundary vertices of each partition within  $k$ -hop traversal. Boundary vertices are vertices that have neighbors in other partitions. The rationale behind this is that if a vertex is very far from the boundary vertex, the chance that it get moved by PARAGON to another partition to improve the decomposition is very small. Surprisingly, we find that PARAGON is not sensitive to  $k$  in terms of the partitioning quality, and that a larger  $k$  does not always lead to partitionings of higher quality. However, it may increase the refinement time greatly. Thus, in our implementation, we set  $k = 0$  by default. In other words, we only send boundary vertices of each partition to the group servers.

In fact, [35] has presented a solution to parallelize the standard FM algorithms [12]. However, it may require a graph with  $n$  partitions to be sent over the network  $n - 1$  times in case the initial decomposition has edge-cuts across all partition pairs. Furthermore, the presence of communication heterogeneity complicates things greatly. First, ARAGON has to be applied to all partition pairs, whereas standard FM algorithms, which assume uniform network communication costs, only need to refine partition pairs that have edge-cuts between them. Second, during each refinement iteration of a single partition pair, standard FM algorithms only need to consider migrating vertices of both partitions that have neighbors in the alternative partition. On the other hand, PARAGON has to consider migrating all boundary vertices.

**Master Node Selection** As presented so far, each server (slave node) needs to send some auxiliary data (i.e., the number of vertices/neighbors) of their local partitions to the master node for the parallelization decision, and the master needs to broadcast the decision it made to all slave nodes. To reduce the communication cost between the master node and the slave nodes, we also select the master node in an intelligent way using the following heuristic:

$$\min_{m \in [1, n]} \sum_{i=1 \text{ and } i \neq m}^n c(P_i, P_m) \quad (11)$$

The heuristic tries to find a server  $M[m]$  that will result in minimal network communication cost as the master node. For example, in case of Figure 4, we should select server  $M[2]$  as the master node. Clearly, the selection of master node can be made locally by each server without synchronizing with each other.

**Physical Data Migration** To support efficient distributed computation, we also provide a basic migration service for graph workloads. Considering that physical data migration is highly application-dependent, the migration service only takes responsibility for the redistribution of the graph data itself. It is the users who are responsible for the migration of any application data associated with each vertex. That is, the users should save the application context before using our migration service and restore the context afterwards. For example, in breadth first search, each vertex is usually associated with a value indicating its current distance to the source vertex. Users need to keep track of the distance value of each vertex while migrating. For complicated workloads, users can exploit the migration service provided by Zoltan [1] to simplify the migration.

## 6. CONTENTION AWARENESS

So far, we have presented how we parallelize ARAGON. In this section, we will cover how we make PARAGON aware of the issue of shared resource contention in multicore systems. We know that, guided by a given network communication cost matrix, PARAGON is able to gather neighboring vertices as close as possible, and that

the contention is caused by the fact that we put too much communication within the compute nodes. Hence, to avoid serious intra-node shared resource contention, we can simply penalize intra-node network communication costs by a score. The score is computed based on the degree of contentiousness between the communication peers. By doing this, the amount of intra-node communication will decrease accordingly. In our implementation, we refine the intra-node communication costs as follows:

$$c(P_i, P_j) = c(P_i, P_j) + \lambda * (s_1 + s_2) \quad (12)$$

where  $P_i$  and  $P_j$  are two partitions collocated in a single compute node;  $\lambda$  is a value between 0 and 1, denoting the degree of contention; and  $s_1$  denotes the maximal inter-node network communication cost, while  $s_2$  equals 0 if  $P_i$  and  $P_j$  reside on different sockets and equals the maximal inter-socket network communication cost otherwise. Clearly, if  $\lambda = 0$ , PARAGON will only consider the communication heterogeneity, and  $\lambda = 1$  means that intra-node shared resource contention is the biggest performance bottleneck, which should be prioritized over the communication heterogeneity. It should be noticed that PARAGON with any  $\lambda \in (0, 1]$  considers both the contention and the communication heterogeneity. Considering the impact of both resource contention and communication heterogeneity is highly application- and hardware-dependent, users will need to do simple profiling of the target applications on the actual computing environment to determine the ideal  $\lambda$  for them.

## 7. EVALUATION

In this section, we first evaluate the sensitivity of PARAGON to varying input decompositions computed by different initial partitioners and the impact of its two important parameters: the degree of parallelism and the number of shuffle refinement times (Section 7.1). We then validate the effectiveness of PARAGON using two real-world graph workloads: Breadth-First Search (BFS) [4] and Single-Source Shortest Path (SSSP) [20], which we implemented using MPI (Section 7.2). Finally, we demonstrate the scalability of PARAGON via a billion-edge graph (Section 7.3).

**Datasets** Table 2 describes the datasets used. By default, the graphs were (re)partitioned with vertex weights (i.e., computational requirement) set to be their vertex degree, with vertex sizes (i.e., amount of the data of the vertex) set to be their vertex degree, and with edge weights (i.e., amount of data communicated) set to 1. The degree of each vertex is often a good approximation of the computational requirement and the migration cost of each vertex, while a uniform edge weight of 1 is a close estimation of the communication pattern of many graph algorithms, like BFS and SSSP. Given the fact that communication cost is usually more important than migration cost, all the experiments were performed with  $\alpha = 10$  (Eq. 2). Unless explicitly specified, all the graphs were initially partitioned by DG (deterministic greedy heuristic), a state-of-the-art streaming graph partitioner [39], across cores of the compute node used (one partition per core). The partitionings were then improved by PARAGON. During the (re)partitioning, we allowed up to 2% load imbalance among partitions. For fairness, DG/LDG were extended to support vertex- and edge-weighted graphs.

**Platforms** We evaluated PARAGON on two clusters: PittMPIcluster [32] and Gordon supercomputer [28]. PittMPIcluster had a flat network topology, with all 32 compute nodes connected to a single switch via 56GB/s FDR Infiniband. On the other hand, the Gordon network topology was a 4x4x4 3D torus of switches connected via 8GB/s QDR Infiniband with 16 compute nodes attached to each switch. Table 3 depicts the compute node configuration of both clusters. The results presented were the means of 5 runs, except the

Table 2: Datasets used in our experiments

Dataset	$ V $	$ E $	Description
wave [38]	156,317	2,118,662	2D/3D FEM
auto [38]	448,695	6,629,222	3D FEM
333SP [10]	3,712,815	22,217,266	2D FE Triangular Meshes
CA-CondMat [2]	108,300	373,756	Collaboration Network
DBLP [18]	317,080	1,049,866	Collaboration Network
Email-Eron [2]	36,692	183,831	Communication Network
as-skitter [2]	1,696,415	22,190,596	Internet Topology
Amazon [2]	334,863	925,872	Product Network
USA-roadNet [9]	23,947,347	58,333,344	Road Network
PA-roadNet [2]	1,090,919	6,167,592	Road Network
YouTube [18]	3,223,589	24,447,548	Social Network
com-LiveJournal [2]	4,036,537	69,362,378	Social Network
Friendster [2]	124,836,180	3,612,134,270	Social Network

Table 3: Cluster Compute Node Configuration

Node Configuration	PittMPIcluster (Intel Haswell Processor)	Gordon (Intel Sandy Bridge Processor)
Sockets	2	2
Cores	20	16
Clock Speed	2.6 GHz	2.6 GHz
L3 Cache	25 MB	20 MB
Memory Capacity	128 GB	64 GB
Memory Bandwidth	65 GB/s	85 GB/s

execution of SSSP on Gordon (Section 7.2) and the scalability test (Section 7.3).

**Network Communication Cost Modelling** The relative network communication costs among partitions (cores) were approximated using a variant of the `osu_latency` benchmark [29]. To ensure the correctness of the cost matrix, each MPI rank (process) was bound to a core using the mechanism provided by `MVAPICH2 1.9` [25] on Gordon and `OpenMPI 1.8.6` [27] on PittMPIcluster. `MVAPICH2` and `OpenMPI` were two different MPI implementations available on the clusters.

## 7.1 MicroBenchmarks

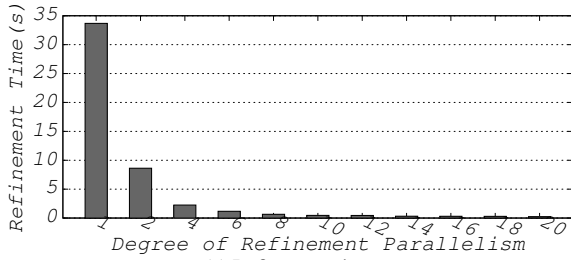
### 7.1.1 Varying Degree of Parallelism

**Configuration** In this experiment, we examined the impact of the degree of parallelism in terms of both the refinement time (i.e., the time that the refinement took) and the refinement quality (i.e., the communication cost of the resulting decomposition). Towards this, we first partitioned the `com-lj` dataset into 40 partitions using DG across 2 compute nodes of PittMPIcluster, and then applied PARAGON to the decompositions with varying degree of refinement parallelism but with shuffle refinement disabled.

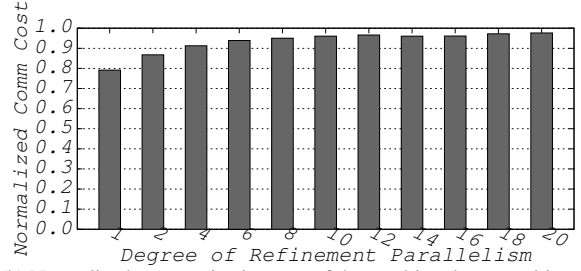
**Results (Figures 7a & 7b)** Figure 7a plots the runtime of PARAGON on the `com-lj` dataset for various degrees of parallelism. As expected, the higher the degree of parallelism, the faster the refinement would finish, and PARAGON significantly reduced the refinement time of ARAGON (PARAGON with degree of parallelism of 1). However, the speedup was achieved at the cost of higher communication cost of the resulting decompositions (Figure 7b). The communication costs presented were normalized to that of the initial decomposition computed by DG. However, in the end, PARAGON still resulted in lower communication cost in all cases when compared to the initial decompositions.

### 7.1.2 Impact of Shuffle Refinement

**Configuration** In our second experiment, we were interested to see whether the shuffle refinement technique could address the issue we identified in the previous experiment. Towards this, we repeated the same experiment but with a fixed degree of refinement parallelism (8) and varying number of shuffle refinement times (from 8 to 15).



(a) Refinement time



(b) Normalized communication cost of the resulting decompositions

Figure 7: Refinement time and communication costs of the com-lj decompositions after being refined with varying degree of refinement parallelism on two 20-core compute nodes. The communication costs presented were normalized to that of the initial decomposition.

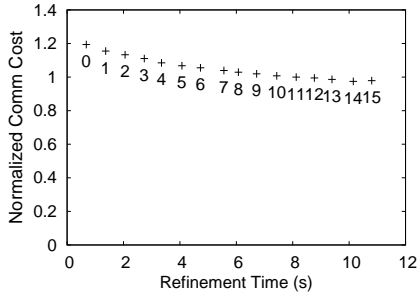


Figure 8: Y-axis corresponds to the communication costs of the com-lj decompositions after being refined with varying number of shuffle refinement times on two 20-core compute nodes when they were normalized to that of the decompositions refined by ARAGON; X-axis denotes the corresponding refinement time; the labels on each data point were the number of refinement times.

**Results (Figure 8)** Figure 8 shows the corresponding refinement time and the normalized communication costs of resulting decompositions with the decompositions computed by ARAGON as the baseline. As shown, PARAGON (with shuffle refinement enabled) not only produced decompositions of lower communication costs than ARAGON (when the number of shuffle refinement times was greater than 11), but also completed the refinement faster (ARAGON took around 33s to finish the refinement vs 8.12s by PARAGON with 11 shuffle refinement times).

### 7.1.3 Impact of Initial Partitioners

**Configuration** This experiment examined the refinement overhead and the quality of the resulting decompositions, when PARAGON was provided with decompositions computed by four different partitioners: (a) HP, the default graph partitioner of many parallel graph computing engines; (b) DG and LDG, two state-of-the-art streaming graph partitioning heuristics [39]; and (c) METIS, a state-of-the-art multi-level graph partitioner [23]. The graphs were initially partitioned across the same two machines used in our prior experiments but with both the degree of refinement parallelism and the number of shuffle refinement times set to 8.

**Quality of the Initial Decompositions (Figure 9)** Figure 9 denotes the communication cost of the initial decompositions computed by HP, DG, LDG, and METIS for a variety of graphs. As anticipated, METIS performed the best and HP the worst. However, METIS is a heavyweight serial graph partitioner, making it infeasible for large-scale distributed graph computation either as an initial partitioner or as an online repartitioner (repartitioning from

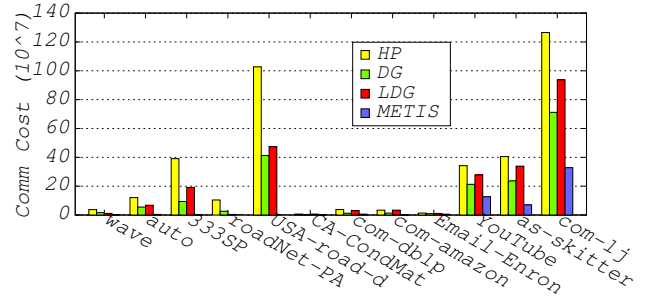


Figure 9: Communication cost of the initial decompositions computed by HP, DG, LDG, and METIS across cores of two 20-core compute nodes for a variety of graphs.

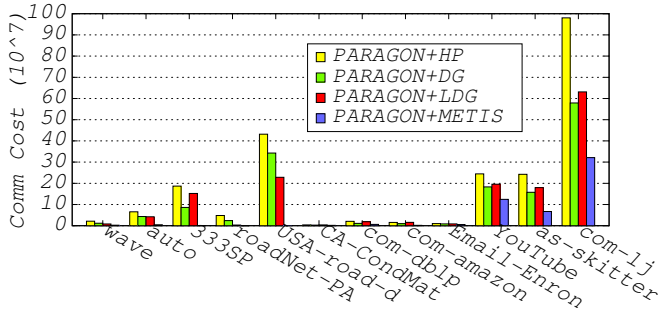
scratch). It was reported in prior work [42] that METIS took up to 8.5 hours to partition a graph with 1.46 billion edges. Unexpectedly, DG outperformed LDG, the best streaming partitioning heuristic among the ones presented in [39]. This was probably because the order in which the vertices were presented to the partitioner favored DG over LDG (the results of DG and LDG rely on the order in which vertices are presented). This was also the reason why we picked DG as the default initial partitioner for PARAGON.

### Quality of the Resulting Decompositions (Figures 10a & 10b)

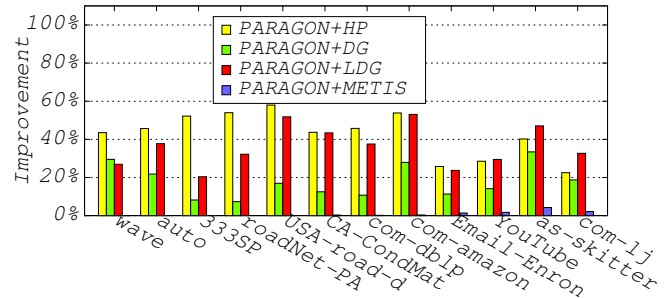
Figures 10a and 10b show the corresponding communication cost of the resulting decompositions and the improvement achieved by PARAGON in terms of the communication cost when compared to the initial decompositions. As shown, the better the initial decomposition was, the better the resulting decomposition would be. In comparison with the initial decompositions computed by HP, DG, and LDG, PARAGON reduced the communication cost of the decompositions by up to 58% (43% on average), 29% (17% on average), and 53% (36% on average), respectively. Although PARAGON did not improve significantly the decompositions computed by METIS for easily partitioned FEM and road networks (left 7 datasets), it achieved an improvement of up to 4.5% for complex networks (right 5 datasets). Given the size of the dataset, the improvement was still non-negligible. Fortunately, we found that PARAGON with DG as its initial partitioner can achieve even better performance than METIS on real-world workloads (Section 7.2).

**Refinement Overhead (Figures 11a & 11b)** We also noticed that the quality of the initial decomposition impacted the refinement overhead greatly. Figures 11a and 11b plot the migration cost (Eq. 3) and the refinement time. Clearly, the poorer the initial decomposition was, the higher the migration cost and the longer the refinement time would be. Finally, for decompositions, which PARAGON failed to make much improvement, PARAGON only led to a very small amount of overhead.



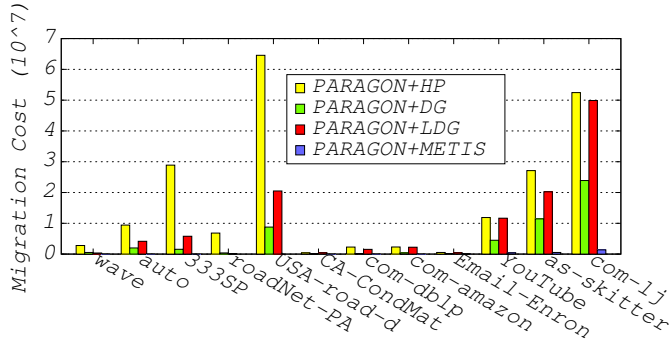


(a) Communication cost of the decompositions after being refined.

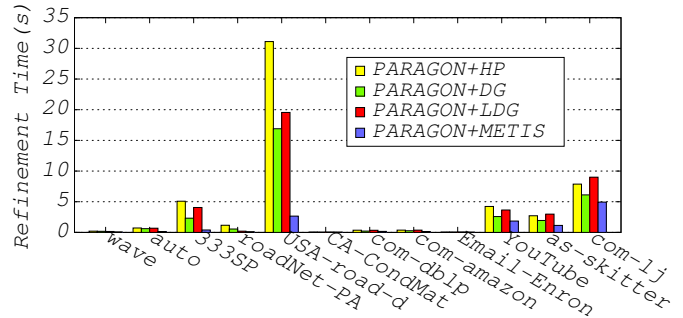


(b) Improvement achieved by PARAGON against the initial decomposition.

Figure 10: PARAGON’s sensitivity to varying initial decompositions in terms of the communication cost for a variety of graphs, which were initially partitioned by HP, DG, LDG, and METIS across cores of two 20-core compute nodes.



(a) Migration Cost



(b) Refinement Time

Figure 11: Overhead of the refinement on varying decompositions that were initially partitioned by HP, DG, LDG, and METIS across cores of two 20-core compute nodes.

## 7.2 Real-World Applications (BFS & SSSP)

**Configuration** This experiment evaluated PARAGON using BFS and SSSP on the YouTube, as-skitter, and com-lj datasets. Initially, the graphs were partitioned across cores of three compute nodes of two clusters using DG. Then, the decomposition was improved by PARAGON with the degree of refinement parallelism and the number of shuffle refinement times both set to 8. During the execution of BFS/SSSP, we grouped multiple (8 for YouTube and as-skitter dataset and 16 for com-lj dataset) messages sent by each MPI rank to the same destination into a single one.

**Resource Contention Modeling** To capture the impact of resource contention, we carried out a profiling experiment for BFS and SSSP with the 3 datasets on both clusters by increasing  $\lambda$  gradually from 0 to 1. Interestingly, we found that intra-node shared resource contention was more critical to the performance on PittMPICluster, while inter-node communication was the bottleneck on Gordon. This was probably caused by the differences in network topologies (flat vs hierarchical), core count per node (20 vs 16), memory bandwidth (65GB vs 85GB), and network bandwidth (56GB vs 8GB) between the two clusters, and that BFS/SSSP had to compete with other jobs running on Gordon for the network resource, while there was no contention on the network communication links on PittMPICluster. Hence, we fixed  $\lambda$  to be 1 on PittMPICluster and 0 on Gordon for the experiment.

**Job Execution Time (Tables 4 & 5)** Tables 4 and 5 show the overall execution time of BFS and SSSP with 15 randomly selected source vertices on the three datasets and the overhead of PARAGON. The execution time of a distributed graph computation is defined as:  $JET = \sum_{i=1}^n SET(i)$ , where  $n$  is the number of supersteps the job has, while  $SET(i)$  denotes the execution time of the  $i$ th super-

step and is defined as the  $i$ th superstep execution time of the slowest MPI rank. In the table, DG and METIS mean that BFS/SSSP was performed on the datasets without any repartitioning/refinement, PARMETIS is a state-of-the-art multi-level graph repartitioner [30], UNIPARAGON was a variant of PARAGON that assumes homogeneous and contention-free computing environment, and the numbers within the parentheses were the overhead of repartitioning/refining the decomposition computed by DG.

As expected, PARAGON beat DG, PARMETIS, and UNIPARAGON in all cases. Compared to DG, PARAGON reduced the execution time of BFS and SSSP on Gordon by up to 60% and 62%, respectively, and up to 83% and 78% on PittMPICluster, respectively. If we time the improvements by the number of MPI ranks (48 for Gordon and 64 for PittMPICluster), the improvements were more remarkable. Yet, the overhead PARAGON exerted (the sum refinement time and physical data migration time) was very small in comparison to the improvement it achieved and the job execution time. By comparing the results of UNIPARAGON with DG, we can conclude that PARAGON not only improved the mapping of the application communication pattern to the underlying hardware, but also the quality of the initial decomposition (edge-cut). Also, if we compare the execution time of BFS/SSSP on both clusters, we would find that the speedup PARAGON achieved by increasing the number of cores from 48 to 60 was much higher than that of DG. What we did not expect was that PARAGON with DG as its initial partitioner outperformed the gold standard, METIS, in 4 out of the 6 cases and was comparable to METIS in other cases.

**Communication Volume Breakdown (Figures 12 & 13)** To further confirm our observations, we also collected the total amount of data remotely exchanged per superstep by BFS and SSSP among cores of the same socket (intra-socket communication volume),

Table 4: BFS Job Execution Time (s)

Algorithm/Dataset	YouTube	as-skitter	com-lj
PittMPICluster			
DG	30	59	218
METIS	8.50	67	27
PARMETIS	29 (21.00)	59 (9.65)	185 (4.71)
UNI-PARAGON	25 (2.70)	27 (2.26)	159 (7.54)
PARAGON	8 (4.00)	10 (3.31)	40 (10.00)
Gordon			
DG	322	577	4319
UNI-PARAGON	264 (2.70)	350 (2.07)	3310 (6.98)
PARAGON	220 (3.83)	228 (2.96)	2586 (9.08)

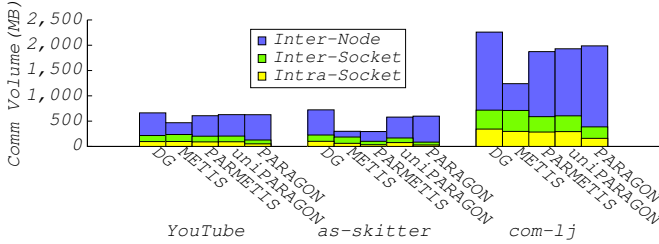


Figure 12: The breakdown of the accumulated communication volume across all supersteps for BFS on PittMPICluster.

among cores of the same compute node but belonging to different sockets (inter-socket communication volume), and among cores of different compute nodes (inter-node communication volume). Since we observed similar patterns for BFS and SSSP in all the cases, we only present the breakdown of the accumulated communication volume across all supersteps for BFS here.

As shown in Figures 12 (for PittMPICluster) and 13 (for Gordon), PARAGON and UNI-PARAGON have much lower remote communication volume than DG in all cases, and PARAGON has the lowest inter-node communication volume and highest intra-node (inter-socket & intra-socket) communication volume on Gordon (vice versa on PittMPICluster), which was expected given our choice for  $\lambda$ . It is worth mentioning that on PittMPICluster, intra-node data communication was the bottleneck. Another interesting thing was that in spite of its higher total communication volume when compared to METIS, PARMETIS, and UNI-PARAGON, PARAGON still outperformed them in most cases due to the reduced communication on critical components.

**Graph Dynamism (Figure 14)** To further validate the effectiveness of PARAGON in the presence of graph dynamism, we split the YouTube dataset (a collection of YouTube users and their friendship connections over a period of 225 days) into 5 snapshots with an interval of 45 days. Thus, snapshot  $S_i$  denotes the collection of YouTube users and their friendship connections appearing during the first  $45 * i$  days. We then ran BFS on snapshot  $S_1$  across three 20-core machines and injected vertices newly appeared in each snapshot to the system using DG whenever BFS finished its computation for every 15 randomly selected vertices. The injection also triggered the execution of PARAGON, UNI-PARAGON, and PARMETIS on the decomposition.

Figure 14 plots the BFS execution time for 15 randomly selected source vertices on each snapshot. As shown, both architecture-awareness and the capability to cope with graph dynamism were critical to achieve superior performance. This is especially true as the graph changes a lot from its original version: at snapshot  $S_5$ , PARAGON performed 90% better than DG, 85% better than METIS, 73% better than PARMETIS, and 89% better than UNI-PARAGON.

Table 5: SSSP Job Execution Time (s)

Algorithm/Dataset	YouTube	as-skitter	com-lj
PittMPICluster			
DG	2136	1823	5196
METIS	545	822	955
PARMETIS	1842 (19.00)	582 (9.28)	3268 (4.50)
UNI-PARAGON	1805 (2.45)	1031 (2.07)	3136 (6.98)
PARAGON	468 (3.88)	472 (3.14)	1549 (9.71)
Gordon			
DG	3436	7092	10732
UNI-PARAGON	3402 (2.76)	3355 (2.13)	7831 (9.75)
PARAGON	2838 (3.89)	2731 (2.97)	6841 (29.00)

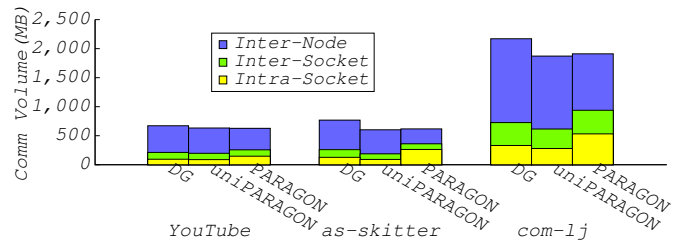


Figure 13: The breakdown of the accumulated communication volume across all supersteps for BFS on Gordon.

### 7.3 Billion-Edge Graph Scaling

**Configuration** In this experiment, we investigated the scalability of PARAGON as the graph scale increased. Towards this, we generated three additional datasets by sampling the edge list of the friendster dataset (3.6 billion edges). We denote the datasets generated as friendster- $p$ , where  $p$  was the probability that each edge was kept while sampling. Hence, friendster- $p$  would have around  $3.6 * p$  billion edges. Interestingly, the number of vertices remained almost unchanged in spite of the sampling. We ran the experiment on three compute nodes of PittMPICluster with the degree of refinement parallelism, the number of shuffle refinement times, and the message grouping size set to 10, 10, and 256, respectively.

**Results (Figures 15 & 16)** Figures 15 and 16 present the execution time of BFS with 15 randomly selected source vertices and the overhead of PARAGON at different graph scales. As shown, PARAGON not only led to lower job execution times, but also to lower speed in which the job execution time increased as the graph size increased. It should be noticed that PARAGON reduced the execution time of all machines ( $3 * 20$  cores) not just one. Also, the refinement time increased at a much slower rate (from 140s, to 236s, to 312s, and to 410s) than that of the graph size. The reason why we did not present the results of METIS or PARMETIS here was because they failed to (re)partition the graphs (even for the first dataset, of 0.9 billion edges).

## 8. RELATED WORK

Graph partitioning and repartitioning are receiving more and more attention in recent years due to the proliferation of large graph datasets. In this section, we categorize existing approaches of graph (re)partitioners into three types: (a) heavyweight, (b) lightweight, and (c) streaming, which are presented next.

**Heavyweight Graph (Re)Partitioning** Graph partitioning and repartitioning has been studied for decades (e.g., METIS [23], PARMETIS [30], Scotch [36], Chaco [7], and Zoltan [1]). These graph (re)partitioners are well-known for their capability of producing high-quality graph decompositions. However, they usually require full knowledge of the entire graph for (re)partitioning, making them scale poorly against

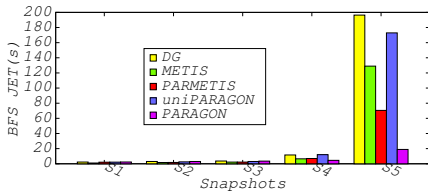


Figure 14: BFS JET with Graph Dynamism

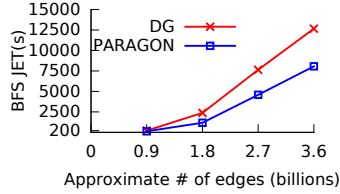


Figure 15: BFS JET vs Graph Size

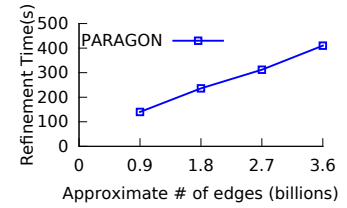


Figure 16: Refinement Time vs Graph Size

large graphs even if performed in parallel. Furthermore, they are all architecture-agnostic. Although [24], a METIS variant, considers the communication heterogeneity, it is a *sequential static graph partitioner*, which is inapplicable for massive graphs or dynamic graphs. Several recent works [48, 8] have been proposed to cope with the heterogeneity and dynamism. However, they are also too heavyweight for massive graphs because of the high communication volume they generate. As a consequence, they are not appropriate for online graph repartitioning in large-scale distributed graph computation. Furthermore, they disregard the issue of resource contention in multicore systems.

**Lightweight Graph Repartitioning** As a result of the shortcomings of heavyweight graph (re)partitioners, many lightweight graph repartitioners [37, 43, 26, 17, 45] have been proposed. They efficiently adapt the partitioning to changes by incrementally migrating vertices among partitions based on some heuristics (rather than repartitioning the entire graph). Nevertheless, they are not architecture-aware. Also, many of them assume uniform vertex weights and sizes, and some [43, 26] even assume uniform edge weights, which may not always be true.

In fact, work [17] is a Pregel-like graph computing engine, which migrates vertices based on runtime characteristics of the workload (i.e., # of message sent/received by each vertex and response time) instead of the graph structure (i.e., the distribution of vertex neighbors, edge weights, and vertex sizes). Paper [45] also presents a repartitioning system that migrates vertices on-the-fly based on some runtime statistics (i.e., the average compute and communication time of each superstep and the probability of a vertex becoming active in the next superstep).

Recently, a novel distributed graph partitioner, Sheep [22], has been proposed for large graphs. It is similar in spirit to METIS. That is, they both first reduce the original graph to a smaller tree or a sequence of smaller graphs, then do a partition of the tree or the smallest graph, and finally map the partitioning back to the original graph. In terms of partitioning time, Sheep outperforms both METIS and streaming partitioners. For partitioning quality, Sheep is competitive with METIS for a small number of partitions and is competitive with streaming graph partitioners for larger numbers of partitions. However, Sheep is unable to deal with both weighted and dynamic graphs, and it is architecture-agnostic.

**Streaming Graph Partitioning** Recently, a new family of graph partitioning heuristics, streaming graph partitioning [39, 11, 42], has been proposed for online graph partitioning. They are able to produce partitionings comparable to the heavyweight graph partitioner, METIS, within a relative short time. However, they are architecture-agnostic. Although [46] has presented a streaming graph partitioner with awareness of both compute and communication heterogeneity, it may lead to suboptimal performance in the presence of graph dynamism.

**Vertex-Cut Graph Partitioning** Several vertex-cut graph partitioners [44, 31, 13] were also proposed to improve the performance of distributed graph computation. Vertex-cut solutions partition

the graph by assigning edges of the graph across partitions instead of vertices. It has been shown that vertex-cut solutions reduce the communications with respect to edge-cut ones, especially on power-law graphs. However, it also has to deal with the issue of communication heterogeneity and the issue of shared-resource contention, since vertices appearing in multiple partitions need to communicate with each other during the computation. Nevertheless, its discussion is beyond the scope of this paper.

**Overview of Related Work** Table 6 visually classifies the state-of-the-art graph (re)partitioners according to algorithm and graph properties. In terms of *algorithm properties*, we characterize each approach as to whether it (a) runs in parallel and (b) is architecture-aware (i.e., CPU heterogeneity, network cost non-uniformity, and resource contention). In terms of *graph properties*, we characterize each approach as to whether it can handle graphs with (a) dynamism, (b) weighted vertices (i.e., nonuniform computation), (c) weighted edges (i.e., nonuniform data communication), and (d) vertex sizes (i.e., nonuniform data sizes on each vertex).

## 9. CONCLUSIONS

In this paper, we presented PARAGON, a *parallel architecture-aware* graph partition refinement algorithm that bridges the mismatch between the application communication pattern and the underlying hardware topology. PARAGON achieves this by modifying a given decomposition according to the nonuniform network communication costs and consideration of the contentiousness of the underlying hardware. To further reduce its overhead, we made PARAGON itself architecture-aware. Compared to the state-of-the-art, PARAGON improved the quality of graph decompositions by up to 53%, achieved up to 5.9x speedups on real workloads, and successfully scaled up to a 3.6 billion-edge graph.

## 10. ACKNOWLEDGMENTS

We would like to thank Jack Lange, Albert DeFusco, Kim Wong, Mark Silvis, and the anonymous reviewers for their valuable help on the paper. This work was funded in part by NSF awards CBET-1250171 and OIA-1028162.

## 11. REFERENCES

- [1] <http://www.cs.sandia.gov/zoltan/>.
- [2] <http://snap.stanford.edu/data>.
- [3] C. Binnig, U. Çetintemel, A. Crotty, A. Galakatos, T. Kraska, E. Zamanian, and S. B. Zdonik. The End of Slow Networks: It's Time for a Redesign. *CoRR*, 2015.
- [4] A. Buluç and K. Madduri. Parallel Breadth-First Search on Distributed Memory Systems. *CoRR*, abs/1104.4518, 2011.
- [5] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud. Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis. In *ICPP*, 2009.
- [6] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. T. Heaphy, and L. A. Riesen. A repartitioning hypergraph model for dynamic load balancing. *J Parallel Distr Com*, 2009.
- [7] <http://www.sandia.gov/~bahendr/chaco.html>.

Table 6: State-of-the-art Graph (Re)Partitioners

Name/Reference	Algorithm Properties				Graph Properties			
	Parallel	Architecture-Aware			Dynamism	Weighted		Vertex Size
		CPU	Network	Contention		Vertex	Edge	
Graph Partitioners								
METIS [23]						✓	✓	
ICA3PP'08 [24]		✓	✓			✓	✓	
Chaco [7]						✓	✓	
DG/LDG [39]/Fennel [42]					Yes/No			
arXiv'13 [11]					✓			
TKDE'15 [46]		✓	✓		Yes/No	✓	✓	
SoCC'12 [8]			✓			✓	✓	
Sheep [22]	✓							
Graph Repartitioners								
PARMETIS [30]	✓				✓	✓	✓	✓
Zoltan [1]	✓				✓	✓	✓	✓
Scotch [36]					✓	✓	✓	✓
CatchW [37]	✓				✓	✓	✓	
xdgp [43]	✓				✓	✓		
Hermes [26]	✓				✓	✓		
Mizan [17]	✓				✓	✓	✓	
LogGP [45]	✓				✓	✓	✓	
ARAGON [48]			✓		✓	✓	✓	✓
PARAGON	✓		✓	✓	✓	✓	✓	✓

- [8] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *SoCC*, 2012.
- [9] <http://www.dis.uniroma1.it/challenge9>.
- [10] <http://www.cc.gatech.edu/dimacs10/>.
- [11] L. M. Erwan, L. Yizhong, and T. Gilles. (Re) partitioning for stream-enabled computation. *arXiv:1310.8211*, 2013.
- [12] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *DAC*, 1982.
- [13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, 2012.
- [14] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel computing*, 2000.
- [15] R. Hood, H. Jin, P. Mehrotra, J. Chang, J. Djomehri, S. Gavali, D. Jespersen, K. Taylor, and R. Biswas. Performance impact of resource contention in multicore systems. In *IPDPS*, 2010.
- [16] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda. Limic: Support for high-performance mpi intra-node communication on linux cluster. In *ICPP*, 2005.
- [17] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, 2013.
- [18] <http://konecni.uni-koblenz.de/networks/>.
- [19] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv:1408.2041*, 2014.
- [20] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *VLDB*, 2014.
- [21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.
- [22] D. Margo and M. Seltzer. A Scalable Distributed Graph Partitioner. *VLDB*, 2015.
- [23] <http://glaros.dtc.umn.edu/gkhome/metis/metis/overview>.
- [24] I. Moulitsas and G. Karypis. Architecture aware partitioning algorithms. In *ICA3PP*, 2008.
- [25] <http://mvapich.cse.ohio-state.edu/>.
- [26] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen. Hermes: Dynamic partitioning for distributed social network graph databases. In *EDBT*, 2015.
- [27] <http://www.open-mpi.org/>.
- [28] <https://portal.xsede.org/sdsc-gordon>.
- [29] <http://mvapich.cse.ohio-state.edu/benchmarks/>.
- [30] <http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview>.
- [31] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni. HDRF: Stream-Based Partitioning for Power-Law Graphs. 2015.
- [32] <http://core.sam.pitt.edu/MPIcluster>.
- [33] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *SC*, 2000.
- [34] K. Schloegel, G. Karypis, and V. Kumar. *Graph partitioning for high performance scientific simulations*. AHPCCRC, 2000.
- [35] C. Schulz. *Scalable parallel refinement of graph partitions*. PhD thesis, Karlsruhe Institute of Technology, May 2009.
- [36] <http://www.labri.u-bordeaux.fr/perso/pelegrin/scotch/>.
- [37] Z. Shang and J. X. Yu. Catch the wind: Graph workload balancing on cloud. In *ICDE*, 2013.
- [38] <http://staffweb.cms.gre.ac.uk/~wc06/partition/>.
- [39] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *SIGKDD*, 2012.
- [40] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda. RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In *PPoPP*, 2006.
- [41] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, 2011.
- [42] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *WSDM*, 2014.
- [43] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella. xdgp: A dynamic graph processing system with adaptive partitioning. *CoRR*, 2013.
- [44] C. Xie, L. Yan, W.-J. Li, and Z. Zhang. Distributed Power-law Graph Computing: Theoretical and Empirical Analysis. In *NIPS*. 2014.
- [45] N. Xu, L. Chen, and B. Cui. LogGP: a log-based dynamic graph partitioning method. *VLDB*, 2014.
- [46] N. Xu, B. Cui, L.-n. Chen, Z. Huang, and Y. Shao. Heterogeneous Environment Aware Streaming Graph Partitioning. *TKDE*, 2015.
- [47] C. Zhang, X. Yuan, and A. Srinivasan. Processor affinity and MPI performance on SMP-CMP clusters. In *IPDPSW*, 2010.
- [48] A. Zheng, A. Labrinidis, and P. K. Chrysanthis. Architecture-Aware Graph Repartitioning for Data-Intensive Scientific Computing. In *BigGraphs*, 2014.

# Query Workload-based RDF Graph Fragmentation and Allocation

Peng Peng<sup>1</sup>, Lei Zou<sup>1,3\*</sup>, Lei Chen<sup>2</sup>, Dongyan Zhao<sup>1,3</sup>

<sup>1</sup>*Peking University, China;*

<sup>2</sup>*Hong Kong University of Science and Technology, China;*

<sup>3</sup>*Key Laboratory of Computational Linguistics (PKU), Ministry of Education, China*  
 { pku09pp, zoulei, zhaodongyan}@pku.edu.cn, leichen@cse.ust.hk

## ABSTRACT

As the volume of the RDF data becomes increasingly large, it is essential for us to design a distributed database system to manage it. For distributed RDF data design, it is quite common to partition the RDF data into some parts, called *fragments*, which are then distributed. Thus, the distribution design consists of two steps: fragmentation and allocation. In this paper, we propose a method to explore the intrinsic similarities among the structures of queries in a workload for fragmentation and allocation, which aims to reduce the number of crossing matches and the communication cost during SPARQL query processing. Specifically, we mine and select some *frequent access patterns* to reflect the characteristics of the workload. Based on the selected frequent access patterns, we propose two fragmentation strategies, vertical and horizontal fragmentation strategies, to divide RDF graphs while meeting different kinds of query processing objectives. Vertical fragmentation is for better throughput and horizontal fragmentation is for better performance. After fragmentation, we discuss how to allocate these fragments to various sites. Finally, we discuss how to process a query based on the results of fragmentation and allocation. Extensive experiments confirm the superior performance of our proposed solutions.

## 1. INTRODUCTION

As a standard model for publishing and exchanging data on the Web, Resource Description Framework (RDF) has been widely used in various applications to expose, share, and connect pieces of data on the Web. In RDF, data is represented as triples of the form (subject, property, object). An RDF dataset can be naturally seen as a graph, where subjects and objects are vertices connected by named relationships (i.e., properties). SPARQL is a structured query language proposed by W3C to access RDF repository. As we know, answering a SPARQL query  $Q$  is equivalent to finding subgraph matches of query graph  $Q$  over an RDF graph  $G$  [31]. Figures 1 and 2 show an RDF graph and a set of SPARQL query graphs used as the running example in this paper.

As RDF repositories increase in size, evaluating SPARQL queries

\*corresponding author: zoulei@pku.edu.cn

is beyond the capacity of a single machine. For example, DBpedia, a project aiming to extract structured content from Wikipedia, consists of 2.46 billion RDF triples [4]; according to the W3C, the numbers of triples in some commercial RDF datasets have been more than 1 trillion [6]. The large-scale of RDF data volume increases the demand of designing the high performance distributed RDF database system.

In distributed database design, the first issue is “data fragmentation and allocation” [18]. We need to divide an RDF graph into several parts, called *fragments*, and then distribute them among sites. One important issue during data fragmentation and allocation in a distributed system is how to reduce the communication cost between different fragments during distributed query evaluation (assuming different fragments are resident at different sites). To minimize the communication cost, many existing graph fragmentation strategies maximize the global goal (such as min-cut [12]). However, evaluating a SPARQL query is a subgraph (homomorphism) match problem. The subgraph match computation often does not involve all vertices in graph  $G$ , and the communication cost of subgraph match computation depends on not only the RDF graph but also the query graph. In other words, subgraph match computation exhibits strong locality. There is no direct relation between minimizing the communication cost (in subgraph match computation) and maximizing the global goal. Hence, we propose a *local pattern-based fragmentation* strategy in this paper, which can reduce the communication cost of subgraph match computation.

The intuition behind the local pattern-based fragmentation is as follows: if a query “satisfies” a local pattern and all its matches are in a single fragment, then the query can be evaluated on the single fragment and no communication cost is needed to answering the query. The key issue in local pattern-based fragmentation is how to define the “local patterns”. Different from the existing methods, we consider the query workload-driven “local pattern” definition.

### 1.1 Why Query Workload Matters ?

The workload-driven distributed data fragmentation has been well studied in relational databases [18]. However, few RDF data fragmentation proposals consider the query workload except for [8, 6]. We will review these related papers in Section 9. Here, we discuss why the query workloads is important for RDF data fragmentation.

We study one real SPAQRL query workload, the DBpedia query workload, which records 8,151,238 SPARQL queries issued in 14 days of 2012<sup>1</sup>. For this workload, if we set the minimum support threshold as 0.1% of the total number of queries, we mine 163 frequent subgraph patterns. The most surprising is that 97% query graphs are isomorphic to one of the 163 frequent subgraph pat-

<sup>1</sup><http://aksw.org/Projects/DBPSB.html>

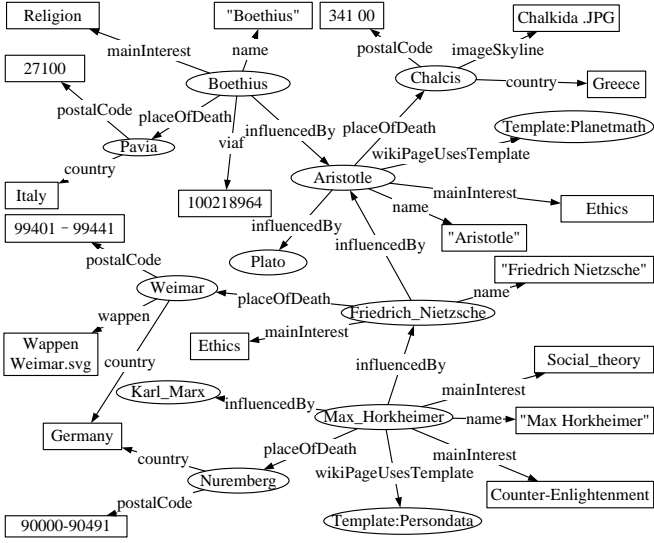


Figure 1: Example RDF Graph

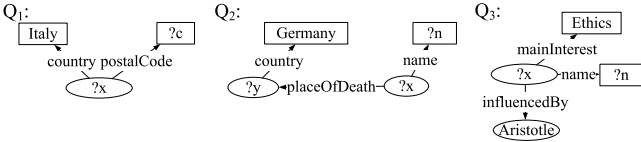


Figure 2: Example SPARQL Query Graphs

terns. Thus, if we use these frequent subgraph patterns as the basic fragmentation units, 97% SPARQL queries do not lead to communication cost, since their matches are resident at one fragment.

## 1.2 Our Solution

According to the above motivation, we propose a workload-driven data fragmentation for distributed RDF graph systems. Specifically, we first mine frequent subgraph patterns, named *frequent access patterns*, in the query workload. We treat these frequent access patterns as the *implicit* schemas for the underlying RDF data. Then, we propose two fragmentation strategies based on these implicit schemas. We study the following technical issues in this paper.

**Frequent Access Pattern Selection.** Given a frequent access pattern, we build a fragment by collecting all its matches in the RDF graph. In this way, we can reduce the communication cost (i.e., improve query performance) if a SPARQL query satisfies the frequent access pattern. However, if we simply select all frequent access patterns as the implicit schemas, it may lead to expensive space cost due to the data replication, since different frequent access patterns may involve share the same edges. In other words, we have a tradeoff between performance gain and space cost during selecting frequent access patterns. We formalize the frequent access pattern selection problem (Section 4.1) and prove that it is a NP-hard problem (Theorem 1). Thus, we propose a heuristic algorithm which can guarantee the data integrity and the approximation ratio (Theorem 2). This algorithm also achieves the good performance (See experiments in Section 8).

**Vertical and Horizontal Fragmentation.** Based on the selected frequent access patterns (i.e., implicit schemas), we design two

fragmentation strategies, i.e, vertical and horizontal fragmentation. These two fragmentation strategies are adaptive to different query processing objectives. The objective of vertical fragmentation strategy is to improve the query throughput, and requires that all structures involved by one frequent access pattern should be placed to the same fragment. Instead, the horizontal fragmentation strategy distributes the structures involved by one frequent access pattern among different fragments to maximize the parallelism of query evaluation, namely, reducing the query response time for a single query. To perform the horizontal fragmentation over RDF graphs, we extend the concept of “minterm predicate” in [18] to “structural minterm predicate” (see Section 5.2), which consider the structures of both RDF graphs and workloads. Different applications have different requirements, so we provide customizable options that can be used for different RDF graphs and SPARQL query workloads.

**Query Decomposition.** As we know, the query decomposition always depends on the fragmentation. In traditional vertical and horizontal fragmentation in RDBMS and XML, the query decomposition is unique, since there is no overlap between different fragments. As mentioned before, there are some data replications in our fragmentation strategies for RDF graphs. Thus, we may have multiple decomposition results for a query. A cost model driven selection is proposed in this paper.

The contributions of this paper can be summarized as follows:

- We analyze the characteristics of the real SPARQL query workload and use the intrinsic similarities of queries in the workload to mine and select some frequent access patterns for distributed RDF data design. Although we prove that the problem of frequent access pattern selection is NP-hard, we propose a heuristic method to achieve the good performance.
- Based on the above scheme, we propose two fragmentation strategies, vertical and horizontal fragmentation, to divide the RDF graph into many fragments and a cost-aware allocation algorithm to distribute fragments among sites. The two fragmentation strategies provide customizable options that are adaptive to different applications.
- We propose a cost-aware query optimization method to decompose a SPARQL query and generate a distributed execution plan. With the decomposition results and execution plan, we can efficiently evaluate the SPARQL query.
- We do experiments over both real and synthetic RDF datasets and SPARQL query workloads to verify our methods.

## 2. PRELIMINARIES

In this section, we review the terminologies used in this paper and formally define the problem to be addressed.

### 2.1 RDF and SPARQL

RDF data can be represented as a graph according to the following definition.

**DEFINITION 1. (RDF Graph)** An RDF graph is denoted as  $G = \{V(G), E(G), L\}$ , where (1)  $V(G)$  is a set of vertices that correspond to all subjects and objects in RDF data; (2)  $E(G) \subseteq V(G) \times V(G)$  is a set of directed edges that correspond to all triples in RDF data; and (3)  $L$  is a set of edge labels. For each edge  $e \in E(G)$ , its edge label is its corresponding property.

Similarly, a SPARQL query can also be represented as a query graph  $Q$ . For simplicity, we ignore FILTER statements in SPARQL syntax in this paper.

**DEFINITION 2. (SPARQL Query)** A SPARQL query is denoted as  $Q = \{V(Q), E(Q), L'\}$ , where (1)  $V(Q) \subseteq V(G) \cup V_{var}$  is a set of vertices, where  $V(G)$  denotes vertices in RDF graph  $G$  and  $V_{var}$  is a set of variables; (2)  $E(Q) \subseteq V(Q) \times V(Q)$  is a set of edges in  $Q$ ; and (3)  $L'$  is also a set of edge labels, and each edge  $e$  in  $E(Q)$  either has an edge label in  $L$  (i.e., property) or the edge label is a variable.

In this paper, we assume that  $Q$  is a connected graph; otherwise, all connected components of  $Q$  are considered separately. Given a SPARQL query  $Q$  over RDF graph  $G$ , a SPARQL match is a subgraph of  $G$  that is homomorphic to  $Q$  [31]. Thus, answering a SPARQL query is equivalent to finding all subgraph matches of  $Q$  over RDF graph  $G$ . The set of all matches for  $Q$  over  $G$  is denoted as  $\llbracket Q \rrbracket_G$ .

In this work, we study a query workload-driven fragmentation. A query workload  $Q = \{Q_1, Q_2, \dots, Q_q\}$  is a set of queries that users input in a given period.

## 2.2 Fragmentation & Allocation

In this paper, we study an efficient distributed SPARQL query engine. There are many issues related to distributed database system design, but, the focus of this work is “data fragmentation and allocation” for RDF repository. We formalize two important problems as follows.

**DEFINITION 3. (Fragmentation)** Given an RDF graph  $G$ , a fragmentation  $\mathcal{F}$  of  $G$  is a set of graphs  $\mathcal{F} = \{F_1, \dots, F_n\}$  such that: (1) each  $F_i$  is a subgraph of  $G$  and called as a fragment of RDF graph  $G$ ; (2)  $E(F_1) \cup \dots \cup E(F_n) = E(G)$ ; and (3)  $V(F_1) \cup \dots \cup V(F_n) = V(G)$ , where  $E(F_i)$  and  $V(F_i)$  denote the edges and vertices in  $F_i$  ( $i = 1, \dots, n$ ).

In our work, we allow the overlaps between different fragments. Given a fragmentation  $\mathcal{F}$ , the next issue is how to distribute these fragments among different sites (i.e., computing nodes). This is called *allocation*.

**DEFINITION 4. (Allocation)** Given a fragmentation  $\mathcal{F} = \{F_1, \dots, F_n\}$  over an RDF graph  $G$  and a set of sites  $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$  (usually  $m < n$ ), an allocation  $\mathcal{A} = \{A_1, \dots, A_m\}$  of fragments in  $\mathcal{F}$  to  $\mathcal{S}$  is a partitioning of  $\mathcal{F}$  such that (1)  $A_j \subseteq \mathcal{F}$ , where  $1 \leq j \leq m$ ; (2)  $A_{j_1} \cap A_{j_2} = \emptyset$ , where  $1 \leq j_1 \neq j_2 \leq m$ ; (3)  $A_1 \cup \dots \cup A_m = \mathcal{F}$ ; and (4) All fragments in  $A_j$  are stored at site  $S_j$ , where  $1 \leq j \leq m$ .

Given an RDF graph  $G$ , a query workload  $Q$  and a distributed system consisting of sites  $\mathcal{S}$ , the goal of this paper is to first decompose  $G$  into a fragmentation  $\mathcal{F}$  and then finding the allocation  $\mathcal{A}$  of  $\mathcal{F}$  to  $\mathcal{S}$ .

## 3. OVERVIEW

This paper studies a SPARQL query workload-driven data fragmentation and allocation problem. Some observations on the real query workload tell us that some RDF properties have few access frequencies. For example, few users input queries contain the properties like *imageSkyline* and *wikiPageUsesTemplate* in Figure 1. As well, the classical distributed database design suggests a “80/20” rule, meaning the active “20%” of query patterns account for “80%” of the total query input [24]. Therefore, we divide the whole RDF repository into two parts: “hot graph” and “cold graph” as follows.

**DEFINITION 5. (Infrequent and Frequent Property)** Given a query workload  $Q = \{Q_1, \dots, Q_n\}$ , if a property  $p$  occurs in less than  $\theta$  queries in  $Q$ , where  $\theta$  is an user specified parameter,  $p$  is an infrequent property; otherwise,  $p$  is a frequent property.

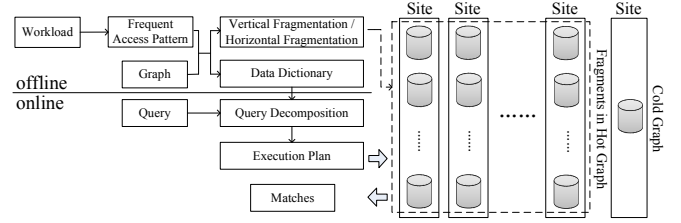


Figure 3: System Architecture

**DEFINITION 6. (Hot and Cold Graphs)** Given an edge  $e = \overrightarrow{u_i u_j} \in E(G)$  with property  $p$ , if property  $p$  is a frequent property,  $e$  is a hot edge; otherwise,  $e$  is a cold edge.

Given an RDF graph  $G$ , it is divided into two parts: hot graph  $H$  and cold graph  $C$ , where  $H$  consists of all hot edges and  $C$  consists of all cold edges.

The goal of this work is how to partition “hot graph” to achieve performance improvement. We regard the cold graph as a “black block”. The cold graph does not overlap to the hot graph, since the cold graph contains different edges with different kinds of properties from the hot graph. Any existing approach can be utilized for the cold graph. We only consider the cold graph in the SPARQL query processing (Section 7), since some queries may involve “infrequent” properties. Moreover, both the cold graph and the hot graph may be disconnected.

Figure 3 illustrates our system architecture. In the offline phase, we mine the *frequent access patterns* (see Section 4) in the workload. Each frequent access pattern can correspond to one or more fragments. Generating a fragment from all matches of a frequent access pattern make many queries be answered efficiently without cross-fragments joins, while it may also replicate some hot edges and increase the space cost. Thus, we should select an appropriate subset of frequent access patterns to balance the efficiency and the space cost. Since we find out that selecting an appropriate set of patterns is a NP-hard problem (Section 4.1), we propose a heuristic pattern selection solution while guaranteeing both the data integrity and the approximation ratio. Based on these selected frequent access patterns, we study two different data fragmentation strategies, i.e., vertical and horizontal fragmentation (Section 5). The vertical fragmentation is to improve the query throughput, and the horizontal fragmentation is to reduce a single query’s response time. Fragments are distributed among different sites. Meanwhile, we maintain the metadata in a data dictionary.

In the online phase, we study how to decompose a query into several subqueries on different fragments and generate an efficient execution plan. A cost model for guiding decomposition is proposed (Section 7.2). Finally, we execute the plan and return the matches of the query.

## 4. FREQUENT ACCESS PATTERNS

As mentioned before, we believe that a query often contains some patterns in the previously issued queries, so we mine some patterns with high access frequencies and use these patterns as the fragmentation units. Then, if a query  $Q$  can be decomposed to some subgraphs isomorphic to the frequent access patterns,  $Q$  can be answered while avoiding some joins across multiple fragments.

Before we mine frequent access patterns, we first normalize the query graphs in the workload to avoid overfitting. For each SPARQL query, we remove all constants (strings and URIs) at subjects and

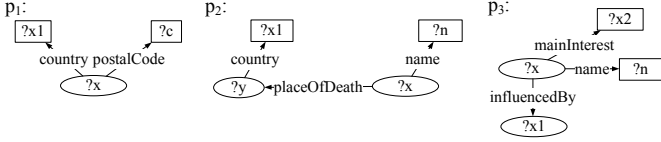


Figure 4: Example Frequent Access Patterns

objects and replace them with variables. The FILTER expressions are also removed. By doing this, we extract a general representation of a SPARQL query from the workload. Figure 4 shows the generalized query graphs of query graphs in Figure 2. We assume that the generalized query in Figure 4 graphs are also frequent access patterns.

To mine patterns with high access frequencies, we need to first count the number of queries in the workload where a pattern  $p$  is a subgraph. We define the *frequent access pattern usage value* to record the access frequencies of the frequent access patterns.

**DEFINITION 7. (Frequent Access Pattern Usage Value)** Given a SPARQL query  $Q$  and a frequent access pattern  $p$ , we associate a frequent access pattern usage value, denoted as  $use(Q, p)$ , and defined as follows:

$$use(Q, p) = \begin{cases} 1 & \text{if pattern } p \text{ is a subgraph of } Q \\ 0 & \text{otherwise} \end{cases}$$

Then, given a workload  $Q = \{Q_1, Q_2, \dots, Q_q\}$  and a pattern  $p$ , we define the *access frequency*,  $acc(p)$ , as the number of queries in  $Q$  where a pattern  $p$  is a subgraph.

$$acc(p) = \sum_{k=1}^q use(Q_k, p)$$

A pattern  $p$  is *frequent access pattern* if its access frequency is no less than a threshold,  $minSup$ .

The frequent access patterns can be easily generated by existing frequent graph mining algorithms [17]. Given a workload of SPARQL queries  $Q = \{Q_1, Q_2, \dots, Q_q\}$  in a given period, we denote the set of frequent access patterns that we find as  $P = \{p_1, p_2, \dots, p_x\}$ . In practice, the size of  $P$  is often limited. For example, if we set  $minSup$  as 0.1% of the total access frequency, there are only 163 frequent access patterns for DBPedia.

## 4.1 Frequent Access Pattern Selection

Obviously, it is not necessary to generate fragments from all frequent access patterns due to high space cost. For two similar frequent access patterns  $p$  and  $p'$ , if they are contained by similar queries of the workload, then selecting both  $p$  and  $p'$  for building fragments will not be able to provide more information than selecting one of  $p$  and  $p'$ . Hence, it is often sufficient to only select a subset of all frequent access patterns to generate fragments.

To select a subset of all frequent access patterns, there are two factors that we should consider.

1. (*Hitting the Whole Workload*) We should select frequent access patterns to hit the query workload as much as possible. This is because that when we select a frequent access pattern to generate a fragment, all queries isomorphic to this pattern can be answered directly, which improve the efficiency.
2. (*Satisfying the Storage Constraint*) The total storage of the system in real applications is limited, so selecting too many frequent access patterns is not desirable.

The above two factors contradict each other. Hitting the whole workload requires to select as many frequent access patterns as possible, while the storage constraint requires to select not too many frequent access patterns. There should be a tradeoff between the two factors.

In the following, we propose a cost model to combine these two factors for selecting a subset of all frequent access patterns.

### 4.1.1 Hitting the Whole Workload

If a fragment is generated from the graph induced by matches of a frequent access pattern, then evaluating all queries containing the pattern can be speeded up by using this fragment. The more queries a frequent access pattern hits, the more gains we obtain during query processing. Therefore, the *benefit* of selecting a frequent access pattern to generate its corresponding fragment should be defined based on the number of queries that the frequent access pattern hits.

In addition, if two similar frequent access patterns are contained by the same set of queries in the workload, it is probably wise to include only one of them. Generally speaking, among similar frequent access patterns contained by the same number of queries, it is often sufficient to materialize only the largest frequent access pattern. That is to say, if  $p'$ , a subgraph of  $p$ , is contained by the same set of queries as  $p$ ,  $p$  is more beneficial than  $p'$  to be selected as building fragments. This is because that if we select the larger pattern, a query is more probable to be decomposed to fewer number of subqueries during query processing. Fewer subqueries can avoid some distributed joins, which can improve the efficiency of query processing.

The above observation implies that larger frequent access patterns are more beneficial to be selected as building fragments. This above criterion on the selection of frequent access patterns is formally defined as *size-increasing benefit*.

**DEFINITION 8. (Size-increasing Benefit)** Given a frequent access pattern  $p$ , the benefit of selecting  $p$  for hitting the query  $Q$ ,  $Benefit(p, Q)$ , is denoted as follows.

$$Benefit(p, Q) = |E(p)| \times use(Q, p)$$

Furthermore, a query in the workload may contain multiple selected frequent access patterns. This means that the query can be decomposed into multiple sets of subqueries if we evaluate the query. Each set of subqueries can map to an execution plan. Since only one execution plan is finally selected to evaluate the query, a query in the workload should only be limited to contribute to the benefits of some particular frequent access patterns once. Based on this observation, we limit a query to only contribute the largest frequent access pattern that the query contains.

**DEFINITION 9. (Benefit of a Frequent Access Pattern Set)** Given a set of frequent access patterns  $P' \subseteq P$ , the benefit of selection of  $P'$  over the workload  $Q$  is the sum of the maximum benefit of its frequent access patterns over  $Q$ .

$$Benefit(P', Q) = \sum_{Q \in Q} \max_{p \in P'} \{Benefit(p, Q)\}$$

### 4.1.2 Satisfying the Storage Constraint

Furthermore, the total storage of the system in real applications is limited, so selecting too many frequent access patterns is not desirable. The selection of frequent access patterns should meet some constraints. When the size of all fragments is larger than the storage constraint, we cannot further select any more frequent



access patterns. We normalize the storage capacity of the system to a value  $SC$ . Then, we have the constraint as:

$$\sum_{p \in P'} |E(\llbracket p \rrbracket_G)| \leq SC$$

Here, we assume that  $SC$  is larger than the number of edges in the hot graph, so each hot edge can have at least one copy. This assumption guarantees the completeness of the RDF graph.

### 4.1.3 Combining the Two Factors

Then, our optimization objective is to maximize the benefit subject to the storage constraint. We can prove that this benefit function (Definition 9) is submodular as follows, so this problem is NP-hard.

**THEOREM 1.** *Finding a set of frequent access patterns with the largest benefit while subject to the storage constraint is NP-hard.*

**PROOF.** Here, we prove that the benefit function  $Benefit(P', Q) = \sum_{Q \in \mathcal{Q}} \max_{p \in P'} \{|E(p)| \times use(Q, p)\}$  is submodular. In other words, for every  $P_1 \subseteq P_2$  and a frequent access pattern  $p \notin P_2$ , we need to prove that  $\Delta_{Benefit}(p|P_1) \geq \Delta_{Benefit}(p|P_2)$ .

For pattern  $p$ , we assume that  $\mathcal{Q}'$  is the set of queries containing  $p$  in the workload. There are three kinds of queries in  $\mathcal{Q}'$ : the set  $\mathcal{Q}_1$  of queries not containing any patterns in  $P_2$ , the set  $\mathcal{Q}_2$  of queries containing patterns in  $(P_2 - P_1)$ , and the set  $\mathcal{Q}_3$  of queries only containing patterns in  $P_1$ .

Since any query in  $\mathcal{Q}_1$  and  $\mathcal{Q}_3$  does not concern patterns in  $(P_2 - P_1)$ ,  $Benefit(\{p\} \cup P_1, \mathcal{Q}_1 \cup \mathcal{Q}_3) = Benefit(\{p\} \cup P_2, \mathcal{Q}_1 \cup \mathcal{Q}_3)$ . Hence, the marginal gains of  $p$  for  $P_1$  and  $P_2$  over  $\mathcal{Q}_1$  and  $\mathcal{Q}_3$  are the same.

For  $\mathcal{Q}_2$ ,  $\Delta_{Benefit}(p|P_1) > \Delta_{Benefit}(p|P_2)$ , if there exist at least one query  $Q^*$  meeting all the two following conditions: 1) the largest pattern contained by  $Q^*$  over  $P_2$  is in  $(P_2 - P_1)$  and has larger size than  $p$ ; 2) the largest pattern contained by  $Q^*$  over  $P_1$  has smaller size than  $p$ . The above two conditions mean that  $p$  can only increase the benefit of  $P_1$  over  $\mathcal{Q}_2$  but not the benefit of  $P_2$  over  $\mathcal{Q}_2$ . Otherwise, for  $\mathcal{Q}_2$ ,  $\Delta_{Benefit}(p|P_1) = \Delta_{Benefit}(p|P_2)$ .

In conclusion,  $\Delta_{Benefit}(p|P_1) \geq \Delta_{Benefit}(p|P_2)$  and the function  $Benefit(P', Q)$  is submodular. Since the problem of maximizing submodular functions is NP-hard [3], the problem is NP-hard.  $\square$

### 4.1.4 Our Solution

As proved in Theorem 1, frequent access pattern selection is NP-complete problem. We propose a greedy algorithm as outlined in Algorithm 1. Note that, to guarantee data integrity of distributed RDF data fragmentation, each hot edge should be contained in at least one fragment. Hence, we initialize a pattern of one edge for each frequent property and compute out its corresponding fragment (Line 3-6).

After we select all patterns with one edge, we enumerate all feasible frequent access pattern sets containing one pattern of more than one edge. Let  $P_1$  be a feasible set of cardinality one that has the largest benefit (Line 7). Then, we iteratively select one of the remaining frequent access patterns  $p'$  to maximize the value of  $\frac{Benefit(\{p'\} \cup P_1, \mathcal{Q}) - Benefit(P_1, \mathcal{Q})}{|E(\llbracket p' \rrbracket_G)|}$  until we meet the storage constraint or cannot find a frequent access pattern to increase the benefit (Line 8-14). Let  $P_2$  be the solution obtained in the iterative phase. Finally, the algorithm outputs  $P' \cup P_1$  if  $Benefit(P' \cup P_1, \mathcal{Q}) \geq Benefit(P' \cup P_2, \mathcal{Q})$  and  $P' \cup P_2$  otherwise (Line 15-17).

**THEOREM 2.** *Algorithm 1 obtains a set of frequent access patterns of benefit at least  $\min\{\frac{1}{(\max_{p \in P} |E(p)|)}, \frac{1}{2}(1 - \frac{1}{e})\}$  times the value of an optimal solution.*

**PROOF.** There are two parts in Algorithm 1: initialization and greedy selection of frequent access patterns.

---

### Algorithm 1: Frequent Access Pattern Selection Algorithm

---

**Input:** A set of frequent access patterns  $P = \{p_1, p_2, \dots, p_x\}$   
**Output:** A set  $P' \subseteq P$  to generate fragments

- 1  $P' \leftarrow \emptyset$ ;
- 2  $TotalSize \leftarrow 0$ ;
- 3 **for** each  $p \in P$  and  $p$  has only one edge **do**
- 4      $P' \leftarrow P' \cup \{p\}$ ;
- 5      $P \leftarrow P - \{p\}$ ;
- 6      $TotalSize \leftarrow TotalSize + |E(\llbracket p \rrbracket_G)|$ ;
- 7  $P_1 \leftarrow \operatorname{argmax}_{\{ \frac{Benefit(\{p_i\}, \mathcal{Q})}{|E(\llbracket p_i \rrbracket_G)|} : p_i \in P, |E(\llbracket p_i \rrbracket_G)| + TotalSize \leq SC \wedge |E(p_i)| > 1 \}}$ ;
- 8  $P_2 \leftarrow \emptyset$ ;
- 9  $TotalSize' \leftarrow 0$ ;
- 10 **while**  $TotalSize' \leq SC - TotalSize$  **do**
- 11     Find the frequent access pattern  $p' \in P - P'$  with the largest additional value of  $\frac{Benefit(\{p'\} \cup P_1, \mathcal{Q}) - Benefit(P_1, \mathcal{Q})}{|E(\llbracket p' \rrbracket_G)|}$ ;
- 12      $P_2 \leftarrow P_2 \cup \{p'\}$ ;
- 13      $P \leftarrow P - \{p'\}$ ;
- 14      $TotalSize' \leftarrow TotalSize' + |E(\llbracket p' \rrbracket_G)|$ ;
- 15 **if**  $Benefit(P' \cup P_1, \mathcal{Q}) \geq Benefit(P' \cup P_2, \mathcal{Q})$  **then**
- 16     Return  $P' \cup P_1$ ;
- 17 Return  $P' \cup P_2$ ;

---

For initialization (Line 3-6 in Algorithm 1), all selected patterns only contain one edge, so  $|E(p)| = 1$ . Therefore, the benefit of patterns only having one edge of a frequent property is  $\sum_{Q \in \mathcal{Q}} \max_{p \in P'} \{1 \times use(Q, p)\}$ . Since the hot edges hit almost all queries in the workload,  $\sum_{Q \in \mathcal{Q}} \max_{p \in P'} \{1 \times use(Q, p)\}$  is approximately equal to the size of the workload,  $|\mathcal{Q}|$ . On the other hand, in the worst case, the optimal solution is that all queries in the workload contain the largest frequent access pattern. Then, the benefit of the optimal solution is  $\sum_{Q \in \mathcal{Q}} \{|E(p_{max})| \times use(Q, p)\}$ , where  $p_{max}$  is the frequent pattern with the largest size. Hence, the benefit of the selected patterns in the initial phase is at least  $\frac{1}{(\max_{p \in P} |E(p)|)}$  of the optimal benefit.

For the phase of greedily selecting frequent access patterns (Line 7-14 in Algorithm 1), since the problem of selecting the optimal set of frequent access patterns is a problem of maximizing a submodular set function subject to a knapsack constraint as discussed in Theorem 1, we directly apply the greedy algorithm in [11] to iteratively select frequent access patterns. [11] proves that the worst-case performance guarantee of the greedy algorithm is  $\frac{1}{2}(1 - \frac{1}{e})$ , so the benefit of the selected patterns in this phase is at least  $\frac{1}{2}(1 - \frac{1}{e})$  of the optimal benefit.

In summary, the final performance guarantee of our algorithm is  $\min\{\frac{1}{(\max_{p \in P} |E(p)|)}, \frac{1}{2}(1 - \frac{1}{e})\}$ .  $\square$

## 5. FRAGMENTATION

In this section, we present two fragmentation strategies: vertical and horizontal.

### 5.1 Vertical Fragmentation

For vertical fragmentation, we put matches homomorphic to the same frequent access pattern into the same fragment. Because a query graph often only contains a few frequent access patterns and matches of one frequent access pattern are put together, other irrelevant fragments can be filtered out during query evaluation and only sites stored relevant fragments need to be accessed to find matches. Filtering out irrelevant fragments can improve the query performance. Furthermore, sites not storing relevant fragments can be used to evaluate other queries in parallel, which improves the total throughput of the system. In summary, the vertical fragmentation strategy utilizes the locality of SPARQL queries to improve

both query response time and throughput. Experimental results in Section 8 also confirm the above argument.

Given a frequent access pattern  $p$ , it can then be transformed into a SPARQL query, resulting in a vertical fragment of the RDF graph. We use the results  $\llbracket p \rrbracket_G$  of a selection operation based on  $p$  to generate a vertical fragment. All vertical fragments generated from our selected frequent access patterns construct a vertical fragmentation. Given a set of frequent access patterns  $P$ , we formally define its corresponding vertical fragmentation over an RDF graph  $G$  as follows.

**DEFINITION 10. (Vertical Fragmentation)** Given an RDF graph  $G$  and a frequent access pattern  $p$ , a vertical fragment  $F$  generated from  $p$  is defined as  $F = \{V(F), E(F), L''\}$ , where (1)  $V(F) \subseteq V(G)$  is the set of vertices occurring in  $\llbracket p \rrbracket_G$ ; (2)  $E(F) \subseteq E(G)$  is the set of edges occurring in  $\llbracket p \rrbracket_G$ ; and (3)  $L'' \subseteq L$  is the set of edge labels occurring in  $\llbracket p \rrbracket_G$ .

Then, given a set of frequent access patterns  $P = \{p_1, p_2, \dots, p_x\}$ , the corresponding vertical fragmentation is  $\mathcal{F} = \{F_i | 0 \leq i \leq x \text{ and } F_i \text{ is the vertical fragment generated from } p_i\}$

**EXAMPLE 1.** Given the frequent access pattern  $p_3$  in Figure 4, Figure 5 shows the corresponding vertical fragment.

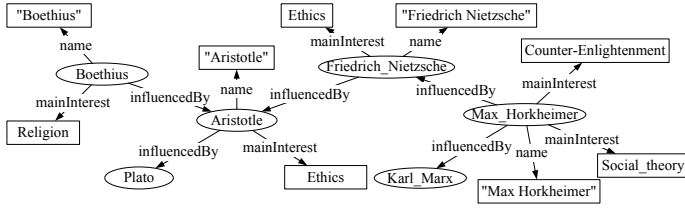


Figure 5: Example Vertical Fragment

## 5.2 Horizontal Fragmentation

For horizontal fragmentation, we put matches of one frequent access pattern into the different fragments and distribute them among different sites. Then, a query may involve many fragments and each fragment has a few matches. The size of a fragment is often much smaller than the size of the whole data, so finding matches of a query over a fragment explores smaller search space than finding matches over the whole data. If the fragments involved by a query are allocated to different sites, then each site finds a few matches over some fragments with the smaller size than the whole data. This strategy is to utilize the parallelism of clusters of sites to reduce the query response time. The above argument is also confirmed by the experimental results in Section 8.

In this section, we extend the concepts of *simple predicate* and *minterm predicate* originally developed for relational systems [18] to divide the RDF graph horizontally.

### 5.2.1 Structural Minterm Predicate

First, we define the structural simple predicate. Each structural simple predicate corresponds to a frequent access pattern with a single (in)equality. Given a frequent access pattern  $p$  with variables set  $\{var_1, var_2, \dots, var_n\}$ , a structural simple predicate  $sp$  defined on  $D$  has the following form.

$$sp : p(var_i) \theta \text{ Value}$$

where  $\theta \in \{=, \neq\}$  and *Value* is a constant constraint for  $var_i$  chosen from a query containing  $p$  in  $Q$ .

**EXAMPLE 2.** Let us consider the query graph  $Q_3$  in Figure 2 and its corresponding frequent access pattern  $p_3$  in Figure 4. We can generate four structural simple predicates: (1).  $sp_1 : p_3(?x1) = Aristotle$ ; (2).  $sp_2 : p_3(?x1) \neq Aristotle$ ; (3).  $sp_3 : p_3(?x2) = Ethics$ ; (4).  $sp_4 : p_3(?x2) \neq Ethics$ .

Then, we define the structural minterm predicate as the conjunction of structural simple predicates of the same frequent access pattern. We can obtain all structural minterm predicates by enumerating all possible combinations of structural simple predicates. Given a set of structural simple predicates  $SP = \{sp_1, sp_2, \dots, sp_y\}$  for frequent access pattern  $p$ , the set of structural minterm predicates  $M = \{mp_1, mp_2, \dots, mp_z\}$  for  $p$  is defined as follows.

$$M = \{mp_i | \bigwedge_{sp_k \in SP} sp_k^*, 1 \leq k \leq y\}$$

where  $sp_k^* = sp_k$  or  $sp_k^* = \neg sp_k$ . So each structural simple predicate can occur in a structural minterm predicate either in its natural form or its negated form.

Similar to the frequent access pattern, we can also define the *structural minterm predicate usage value* and *access frequency* to record the access frequency of a structural minterm predicate. We can prune the minterm predicates with small access frequencies.

**DEFINITION 11. (Structural Minterm Predicate Usage Value)** Given a SPARQL query  $Q$  and a structural minterm predicate  $mp$ , we associate a structural minterm predicate usage value, denoted as  $use(Q, mp)$ , and defined as follows:

$$use(Q, mp) = \begin{cases} 1 & \text{if predicate } mp \text{ is a subgraph of } Q \\ 0 & \text{otherwise} \end{cases}$$

Then, given a set of SPARQL queries  $Q = \{Q_1, Q_2, \dots, Q_q\}$ , we define the *access frequency* of a structural minterm predicate  $mp$  as follows.

$$acc(mp) = \sum_{k=1}^{k=q} use(Q_k, mp)$$

In practice, there may exist many minterm predicates. It is too expensive to enumerate all minterm predicates. Therefore, we prune some minterm predicates with too small access frequencies.

Given a structural minterm predicate  $mp$ , it can then be transformed into SPARQL queries, resulting in a horizontal fragment of the RDF graph. We use the results  $\llbracket mp \rrbracket_G$  of a selection operation based on  $mp$  to generate a horizontal fragment. All horizontal fragments generated from the structural minterm predicates that we obtain construct a horizontal fragmentation. Given a set of minterm predicates  $M$ , we formally define its corresponding horizontal fragmentation over an RDF graph  $G$  as follows.

**DEFINITION 12. (Horizontal Fragmentation)** Given an RDF graph  $G$  and a structural minterm predicate  $mp$ , a horizontal fragment  $F$  generated from  $mp$  is defined as  $F = \{V(F), E(F), L''\}$ , where (1)  $V(F) \subseteq V(G)$  is the set of vertices occurring in  $\llbracket mp \rrbracket_G$ ; (2)  $E(F) \subseteq E(G)$  is the set of edges occurring in  $\llbracket mp \rrbracket_G$ ; and (3)  $L'' \subseteq L$  is the set of edge labels occurring in  $\llbracket mp \rrbracket_G$ .

Then, given a set of structural minterm predicates  $M = \{mp_1, mp_2, \dots, mp_y\}$ , the corresponding horizontal fragmentation is  $\mathcal{F} = \{F_i | 0 \leq i \leq y \text{ and } F_i \text{ is the vertical horizontal generated from } mp_i\}$

**EXAMPLE 3.** Given the structural simple predicates in Example 2, we can get all structural minterm predicates from frequent access pattern  $p_3$  as follows: (1).  $mp_1 : p_3(?x0) = Aristotle \wedge p_3(?x1) = Ethics$ ; (2)  $mp_2 : p_3(?x0) = Aristotle \wedge p_3(?x1) \neq$

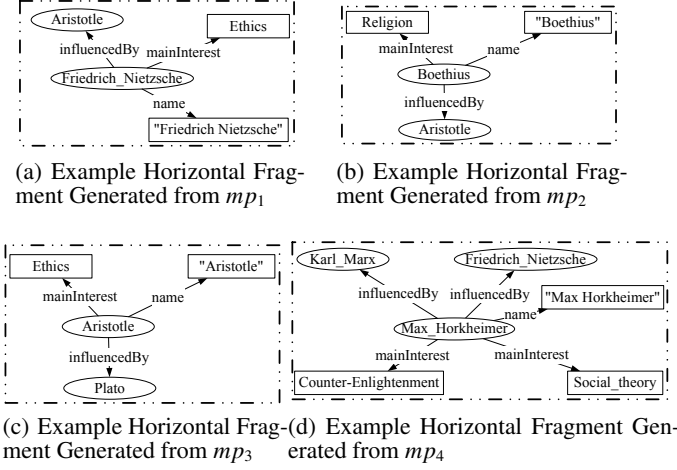


Figure 6: Example Horizontal Fragments

Ethics; (3).  $mp_3 : p_3(?x0) \neq Aristotle \wedge p_3(?x1) = Ethics$ ; (4).  $mp_4 : p_3(?x0) \neq Aristotle \wedge p_3(?x1) \neq Ethics$ .

Figure 6 shows all horizontal fragments generated from the above structural minterm predicates.

## 6. ALLOCATION

After fragmenting the RDF graph, the next step is to allocate all fragments on several sites. In real applications, some frequent access patterns or structural minterm predicates are usually accessed together, so their corresponding fragments should be placed in one site to further avoid the cross-fragments joins. There is a need for some measures evaluating precisely the notion of “togetherness”. This measure is the affinity of fragments, which indicates how closely related the fragments are.

We define *fragment affinity metric* to measure the togetherness between two frequent access patterns or structural minterm predicates as follows:

DEFINITION 13. (**Fragment Affinity Metric**) The fragment affinity metric between two fragments  $F$  and  $F'$  with respect to the workload  $Q = \{Q_1, Q_2, \dots, Q_q\}$  is defined as follows

- $aff(F, F') = \sum_{k=1}^q use(Q_k, p) \times use(Q_k, p')$ , if  $F$  and  $F'$  are vertical fragments generated from frequent access patterns  $p$  and  $p'$ ;
- $aff(F, F') = \sum_{k=1}^q use(Q_k, mp) \times use(Q_k, mp')$ , if  $F$  and  $F'$  are horizontal fragments generated from structural minterm predicates  $mp$  and  $mp'$ ;

Based on the fragment affinity metric, we can show how closely related the fragments are. If the affinity metric of two fragments is large, it means that these two fragments are often involved by the same query. Some fragments are so related that they should be placed together to reduce the number of cross-sites joins. Here, we group all fragments into some clusters. The result of clustering corresponds to an allocation  $\mathcal{A}$ , and each cluster corresponds to an element of  $\mathcal{A}$ , which means that all fragments in the cluster are placed into the same site.

There are many clustering algorithms to cluster all fragments and we need to select one of them. In this paper, we extend a graph clustering algorithm, PNN [5], to cluster all fragments into an allocation  $\mathcal{A} = \{A_1, A_2, \dots, A_m\}$ . All fragments of the same cluster are put into one site.

First, we build the *allocation graph* as follows.

DEFINITION 14. (**Allocation Graph**) Given a fragmentation  $\mathcal{F} = \{F_1, F_2, \dots, F_n\}$ , the corresponding allocation graph  $AG = \{V(AG), E(AG), f_w\}$  is defined as follows:

- $V(AG)$  is a set of vertices that map to all fragments;
- $E(AG)$  is a set of undirected edges that  $\overline{vv'} \in E(AG)$  if and only if the fragment affinity metric between the corresponding fragments of  $v$  and  $v'$  is larger than 0;
- $f_w$  is a weight function  $f_w : E(AG) \rightarrow \mathbb{N}^+$ . If  $v$  and  $v'$  correspond to fragments  $F$  and  $F'$ ,  $f_w(\overline{vv'}) = aff(F, F')$ .

Then, the allocation problem is equivalent to cluster all fragments in  $m$  clusters, and all fragments in a cluster are connected in  $AG$ . We define the *density* of a cluster  $A_i$  in  $AG$  to rate the quality of  $A_i$  as follows.

$$\delta(A_i) = \frac{\sum_{v_i \in A_i \wedge v_j \in A_i \wedge \overline{v_i v_j} \in E(AG)} f_w(\overline{v_i v_j})}{\binom{|A_i|}{2}}$$

where  $\sum_{v_i \in A_i \wedge v_j \in A_i \wedge \overline{v_i v_j} \in E(AG)} f_w(\overline{v_i v_j})$  is the sum of weights of all edges in  $A_i$  and  $\binom{|A_i|}{2}$  is the maximum possible number of edges.

The objective of our allocation algorithm is to search for  $m$  sub-graphs of  $AG$  that have the highest densities. Unfortunately, this problem is NP-complete [20], so we propose a heuristic solution as Algorithm 2. Algorithm 2 is a variant of PNN and picks the locally optimal choice of merging two vertices in  $AG$  at each step. Because our objective function can guarantee the locally optimal choice is also the optimal choice for the overall solution, Algorithm 2 can find out the optimal clustering result of  $AG$ .

Generally speaking, we initialize a cluster for each fragment. Then, we repeatedly pick the two clusters (singletons or larger) that have the highest weight value to be merged. The weight between two clusters are the density value of merging them. Such merging is iterated until the size of the allocation graph has been reduced to  $m$ .

---

### Algorithm 2: Allocation Algorithm

---

**Input:** The allocation graph  $AG$  and the preset threshold  $\theta$

**Output:** An allocation  $\mathcal{A} = \{A_1, A_2, \dots, A_m\}$

- 1 for each vertex  $v_i$  in  $V(AG)$  do
  - 2   |  $A_i \leftarrow \{v_i\}$ ;
  - 3 Find the edge  $e_{max}$  with the highest weight in  $E(AG)$ ;
  - 4 Initialize  $AG'$  that is the same to  $AG$ ;
  - 5 while  $|V(AG')| \neq m$  do
  - 6   | Generating  $AG'$  from  $AG$  by merging  $e_{max} = \overline{A_i A_j}$  to  $A_{ij}$ ;
  - 7   | for each  $A_k$  adjacent to  $A_{ij}$  in  $E(AG')$  do
  - 8   |   |  $f_w(\overline{A_k A_{ij}}) \leftarrow \frac{\sum_{v_i \in A_k \wedge (v_j \in A_i \vee v_j \in A_j) \wedge \overline{v_i v_j} \in E(AG)} f_w(\overline{v_i v_j})}{\binom{|A_k|}{2}}$
  - 9   | Find the edge  $e_{max}$  with the highest weight in  $E(AG')$ ;
- 

## 7. DISTRIBUTED QUERY PROCESSING

In this section, we discuss how to process a SPARQL query. For query processing, the metadata is necessary and we introduce how to maintain the metadata in a data dictionary in Section 7.1. Then, we discuss how to decompose a query into some subqueries in Section 7.2. Last, we discuss how to produce a distributed execution plan and execute all subqueries based on the plan in Section 7.3.

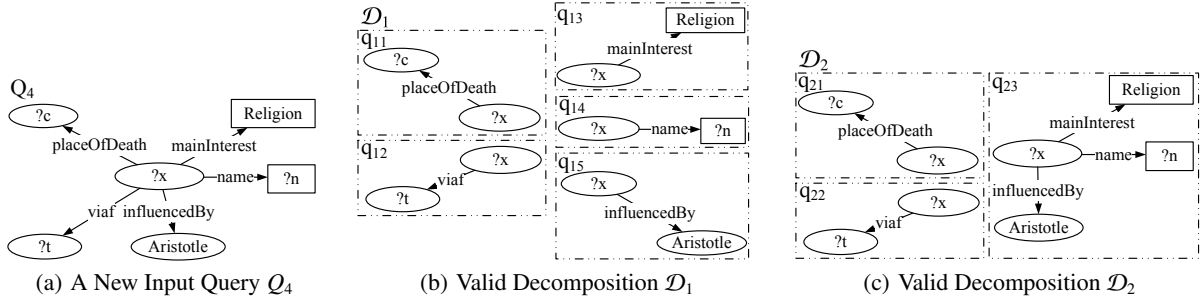


Figure 7: A New Input Query and Its Example Valid Decompositions

## 7.1 Data Dictionary

After fragmentation and allocation, the results of fragmentation and allocation need to be stored and maintained by the system. This information is necessary during distributed query processing. This information is stored in a data dictionary. The data dictionary stores a global statistics file generated at fragmentation and allocation time. It contains the following information: fragment definitions, their sizes, site mappings, access frequencies and so on.

Since each fragment corresponds to a frequent access pattern or a structural minterm predicate, the data dictionary uses the frequent access pattern with/without constraints as the representative of a fragment. Each frequent access pattern with/without constraints corresponds to a fragment and is associated with all statistics of the fragment. The data dictionary need to fast retrieve all frequent access patterns with/without constraints to determine the relevant frequent access pattern for a query.

We build a hash table to achieve the above objective. We first use the DFS coding [26] to translates frequent access patterns into sequences. With the DFS code of a frequent access pattern, we can map any frequent access pattern to an integer by hashing its canonical label. Then, we use the hash table to locate frequent access patterns and retrieve the statistics of their corresponding fragments

## 7.2 Query Decomposition

When users input a query  $Q$ , the system first uses the data dictionary to determine which fragments are involved in the query and decomposes the query into some subqueries on fragments.

Given a query  $Q$ , a decomposition of  $Q$  is a set of subqueries  $\mathcal{D} = \{q_1, q_2, \dots, q_t\}$  such that (1) each  $q_i$  is a subgraph of  $Q$  and  $q_i$  maps to a frequent access pattern or structural minterm predicate; (2)  $V(q_1) \cup \dots \cup V(q_t) = V(Q)$ ; and (3)  $E(q_1) \cup \dots \cup E(q_t) = E(Q) \wedge \forall i \neq j, E(q_i) \cap E(q_j) = \emptyset$ .

Since we partition the RDF graph based on the frequent access patterns, we also decompose the query based on the frequent access patterns. In other words, we decompose the query into subqueries that are homomorphic to frequent access patterns. If a query involves infrequent properties that cannot be decomposed into subqueries homomorphic to any frequent access patterns, then each connected subgraph of the query that only contains infrequent properties corresponds to a subquery. We define the *valid* decomposition as follows.

**DEFINITION 15. (Valid Decomposition)** Given a SPARQL query  $Q$ , a valid decomposition  $\mathcal{D} = \{q_1, q_2, \dots, q_t\}$  of  $Q$  should meet the following constraint: if  $q_i$  ( $1 \leq i \leq t$ ) is not homomorphic to any frequent access patterns, all edges in  $q_i$  should be cold edges.

There exist at least one valid decompositions. A possible decomposition is the decomposition of all subqueries of a single edge.

Because we select all frequent access patterns of one edge, the decomposition of all subqueries of a single edge is valid. Besides the valid decomposition, there may also exist some other valid decompositions. Hence, we propose a cost-model driven selection and the best valid decomposition is the valid decomposition with the smallest cost.

Here, we assume that the cost of a decomposition is the cost of joining all matches of the subqueries in  $\mathcal{D}$  and each pair of subqueries' matches can join together. The assumption is the worst case, so that we can quantify the worst-case performance. Then, we define the cost of a decomposition as follows.

$$\text{cost}(\mathcal{D}) = \prod_{q_i \in \mathcal{D}} \text{card}(q_i)$$

where  $\text{card}(q_i)$  is the number of matches for  $q_i$ , which can be estimated by looking up the data dictionary.

**EXAMPLE 4.** Assume that a user inputs a new query  $Q_4$  as shown in Figure 7(a). Given frequent access patterns in Figure 4, there can be two valid decompositions  $\mathcal{D}_1$  and  $\mathcal{D}_2$  as shown in Figures 7(b) and 7(c). For vertical fragmentation,  $q_{23}$  in  $\mathcal{D}_2$  is evaluated on the vertical fragment of  $p_3$  (Figure 5); for horizontal fragmentation,  $q_{23}$  is evaluated on the horizontal fragment of  $mp_2$  (Figure 6(b)).

Whether in vertical or in horizontal fragmentation, it is obvious that  $\mathcal{D}_2$  has fewer subqueries than  $\mathcal{D}_1$  and  $\text{card}(q_{23}) < \text{card}(q_{13}) \times \text{card}(q_{14}) \times \text{card}(q_{15})$ . Hence,  $\text{cost}(\mathcal{D}_2)$  is smaller than  $\text{cost}(\mathcal{D}_1)$ , and  $\mathcal{D}_2$  is more of a priority as the final decomposition.

Based on the above definitions, we propose the query decomposition algorithm as Algorithm 3. Because the SPARQL query graphs in real applications usually contain 10 or fewer edges, we can use a brute-force implementation to enumerate all possible decompositions and find the decomposition with the smallest cost.

---

### Algorithm 3: Query Decomposition Algorithm

---

**Input:** A query  $Q$   
**Output:** A valid decomposition  $\mathcal{D} = \{q_1, q_2, \dots, q_t\}$  of query  $Q$

- 1  $MinCost \leftarrow +\infty$ ;
- 2 Initialize  $\mathcal{D}$  as the decomposition of all subqueries of a single edge;
- 3 **for** each possible valid decomposition  $\mathcal{D}' = \{q_1, \dots, q_t\}$  **do**
- 4      $CurrentCost \leftarrow 1$ ;
- 5     **for** each query  $q_i$  in  $\mathcal{D}'$  **do**
- 6         Estimate the number of results for  $q_i$  as  $\text{card}(q_i)$  based on the data dictionary;
- 7          $CurrentCost \leftarrow CurrentCost \times \text{card}(q_i)$
- 8     **if**  $MinCost > CurrentCost$  **then**
- 9          $\mathcal{D} \leftarrow \mathcal{D}'$ ;
- 10          $MinCost \leftarrow CurrentCost$ ;
- 11 **Return**  $\mathcal{D}$ ;

---

### 7.3 Query Optimization and Execution

After decomposing the query, the next step is to find an execution plan for the query which is close to optimal. In this section, we discuss the major optimization issue of finding execution plan, which deals with the join ordering of subqueries. We extend the algorithm of System-R [2] to find the optimal execution plan for distributed SPARQL queries. The algorithm is described in Algorithm 4.

Generally speaking, Algorithm 4 is a variant of System-R style dynamic programming algorithm. It firstly generates the best execution plan of  $n - 1$  subqueries, and then join the matches of  $n - 1$  subqueries with the matches of  $n$ -th subquery. The cost of an execution plan can also be estimated based on the number of subqueries' results, which is stored in the data dictionary.

Finally, each subquery is executed in the corresponding sites in parallel. The optimization of each subquery uses the existing methods in centralized RDF database systems. After the matches of all subqueries are generated, we join them together according to the optimal execution plan.

---

#### Algorithm 4: Query Optimization Algorithm

---

**Input:** A decomposition  $\mathcal{D} = \{q_1, q_2, \dots, q_t\}$  of query  $Q$   
**Output:** An execution plan  $(\dots((q_{i1} \bowtie q_{i2}) \bowtie q_{i3}) \bowtie \dots \bowtie q_{it})$

- 1 **for** each two subqueries  $(q_i)$  and  $(q_j)$  where  $1 \leq i \neq j \leq t$  **do**
- 2     Initialize an execution plan  $q_i \bowtie q_j$  and estimate its cost;
- 3     Store all execution plans and their costs in a table  $T_2$ ;
- 4 **for**  $i = 3$  **to**  $t$  **do**
- 5     **for** each execution plan  $pl_j$  in  $T_{i-1}$  **do**
- 6         **for** each subquery  $q_k$  that is not contained by  $pl_j$  **do**
- 7             Build execution plan  $pl_j \bowtie q_k$  and estimate its cost;
- 8             Store this execution plan and its costs in a table  $T_i$ ;
- 9         **for** each two plans  $pl_j$  and  $pl_k$  in  $T_i$  **do**
- 10             **if**  $pl_j$  and  $pl_k$  map to the same set of subqueries **then**
- 11                 Eliminate one of  $pl_j$  and  $pl_k$  that has the larger cost;
- 12 **Return** the execution plan with the minimum cost;

---

## 8. EXPERIMENTAL EVALUATION

We conducted extensive experiments to test the effectiveness of our proposed techniques on a real dataset, DBPedia, and a synthetic dataset, WatDiv. In this section, we report the setting of test data and various performance results.

### 8.1 Setting

**DBPedia.** DBPedia<sup>2</sup> is an RDF dataset extracted from Wikipedia. The DBPedia contains 163,977,110 triples. We use the DBpedia SPARQL query-log as the workload. This workload contains queries posed to the official DBpedia SPARQL endpoint in 14 days of 2012. After removing some queries that cannot be handled, there are 8,151,238 queries in the workload.

**WatDiv.** WatDiv [1] is a benchmark that enable diversified stress testing of RDF data management systems. In WatDiv, instances of the same type can have the different sets of attributes. For testing our methods, we generate five datasets varying sizes from 50 million to 250 million triples. By default, we use the RDF dataset with 100 million triples. In addition, WatDiv can generate a workload by instantiating some templates with actual RDF terms from the dataset. WatDiv provides 20 templates to generate test queries. We use these benchmark templates to generate a workload with 2000 test queries.

We conduct all experiments on a cluster of 10 machines running Linux, each of which has one CPU with four cores of 3.06GHz. Each site has 16GB memory and 150GB disk storage. We select one of these sites as a control site. At each site, we install gStore

<sup>2</sup><http://km.aifb.kit.edu/projects/btc-2012/dbpedia/>

[31] to find matches. We use MPICH-3.0.4 running on C++ to join the results generated by subqueries.

For fair performance comparison, we use gStore and MPICH-3.0.4 to re-implement two recent distributed RDF fragmentation strategies. The first one is SHAPE [14], which defines a vertex and its neighbors as a triple group and assigns the triple groups according to the value of its center vertices. There are many different kinds of triple groups in [14] and we use the subject-object-based triple groups in this paper. The second one is WARP [8]. WARP first uses METIS [12] to divide the RDF graph into fragments. Then, it replicates all matches of a query pattern that cross two fragments in one fragment. We use all frequent access patterns to extend the fragments in WARP.

### 8.2 Parameter Setting

Our frequent access patterns selection method uses a parameter: *minSup*. In this subsection, we discuss how to set up *minSup* to optimize query processing. Note that, since the numbers of query templates and queries per query template in WatDiv are specified by users, the parameters can also be determined beforehand. Thus, we only discuss how to set the parameters for DBPedia.

Given a workload  $Q$ , we set the support threshold, *minSup*, to find patterns whose access frequencies are larger than *minSup*. It is clear that the smaller *minSup* is, the larger number of frequent access patterns there are. More frequent access patterns mean that a query in the workload may have a higher possibility to contain some frequent access patterns.

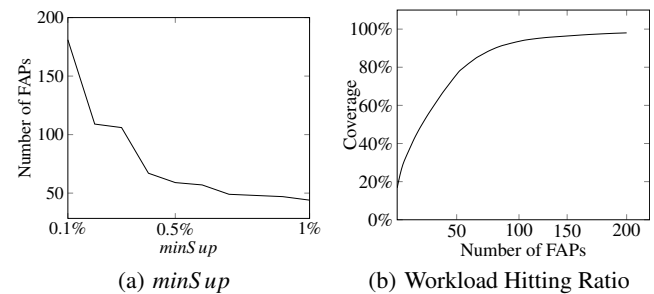


Figure 8: Effect of Frequent Access Patterns

Figure 8(a) shows the impact of *minSup*. As *minSup* increases, the number of frequent access patterns (FAPs) decreases. Hence, when we set *minSup* as 0.1% of the total number of queries in the workload, there are 163 frequent access patterns for DBPedia. When *minSup* is 1% of the total number of queries, the number of frequent access patterns is reduced to 44 for DBPedia. Furthermore, fewer frequent access patterns means that fewer queries in the workload are hit, as shown in Figure 8(b).

Even if we set *minSup* as 0.1% of the total number of queries, the number of frequent access patterns is not large. Hence, in the following, we set *minSup* as 0.1% of the total number of queries for DBPedia by default.

### 8.3 Throughput

In this experiment, we test the throughput of different fragmentation strategies. We sample 1% of all queries in the workload and measure the throughput in queries per minute. Figure 9 shows the number of queries answered in one minute of different fragmentation strategies.

For SHAPE and WARP, each query concerns all fragments, so queries are still processed sequentially. Since WARP is more balanced than SHAPE, the throughput of WARP is a little better than SHAPE. WARP can handle about 32 and 82 queries in one minute

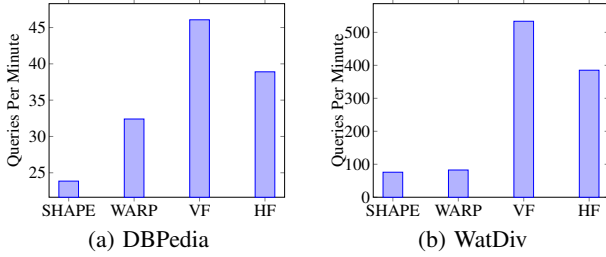


Figure 9: Throughput Comparison

for DBPedia and WatDiv, while SHAPE can handle 24 and 75 queries.

For the vertical fragmentation strategy (VF), since a query often only contains a few frequent access patterns, it only involves a few fragments. Two queries involving different fragments can be evaluated in parallel. Hence, about 46 queries and 533 queries can be answered in one minute for DBPedia and WatDiv, respectively. For the horizontal fragmentation strategy (HF), each frequent access pattern specified by the query may map to many structural minterm predicates and the corresponding fragments of these structural minterm predicates may be allocated to different sites. Hence, the throughput of the horizontal fragmentation strategy is a little worse than the vertical fragmentation strategy, and 38 and 385 queries can be answered in one minute for DBPedia and WatDiv.

## 8.4 Response Time

In this experiment, we test the query performance of different fragmentation strategies. We also sample 1% of all queries in the workload and compute the average query response time of a query. Figure 10 shows the performance results.

SHAPE and WARP partition the RDF graph into some subgraphs, and distributes these subgraphs among different sites. The query should be processed in many sites in parallel. Hence, SHAPE is less balanced and sometime need cross-fragment joins, so SHAPE needs about 2.5 and 0.79 seconds to answer a query for DBPedia and WatDiv, while WARP takes 1.8 and 0.72 seconds.

For the vertical fragmentation strategy, only relevant fragments are searched for matches and the search space is reduced. Therefore, a query can be answered in about 0.8 seconds for DBPedia and 0.3 seconds for WatDiv. For the horizontal fragmentation strategy, we can filter out all irrelevant fragments mapping to the structural minterm predicates not specified by the query, which can further reduce the search space. Hence, a query can be answered with about 0.6 seconds for DBPedia and 0.15 seconds for WatDiv.

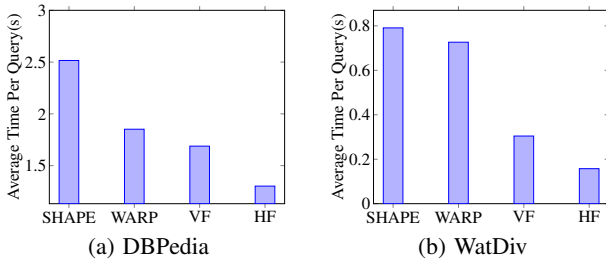


Figure 10: Performance Comparison

## 8.5 Scalability Test

In this experiment, we investigate the impact of dataset size on our fragmentation strategies. We generate five WatDiv datasets varying the from 50 million to 250 million triples to test our strate-

gies. Figure 11 shows the results. Generally speaking, as the size of RDF datasets gets larger, the average response times of one query increase and the numbers of queries answered in one minute decrease accordingly. However, the rates of increase and decrease are slow, and we can say that the query performance and throughput are scalable with RDF graph size on the datasets.

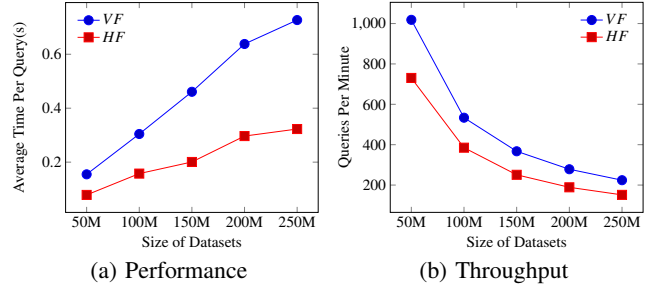


Figure 11: Varying Size of Datasets

## 8.6 Redundancy

Table 1 shows the redundancy ratio of the number of edges in all generated fragments to the total number of edges in the original RDF graph for each fragmentation strategy. For SHAPE, if a fragment contains a vertex with high degree, all adjacent edges of the high degree vertex are introduced. Most of these introduced edges are redundant, and cause the redundancy ratios of SHAPE nearly 3 for DBPedia and 1.74 for WatDiv. WARP divides the RDF graph while minimizing the edge cut, so the number of edges crossing two fragments for WARP is smaller than the number for SHAPE. Therefore, the redundancy ratio of WARP is smaller. Note that, WatDiv is much denser than DBPedia, so the minimum cut-set for WatDiv contains a higher proportion of edges. Hence, the redundancy ratio of WatDiv is 1.54, but the ratio of DBPedia is only 1.01.

	DBPedia	WatDiv
SHAPE	2.99	1.74
WARP	1.01	1.54
VF	1.38	1.04
HF	1.42	1.06

Table 1: Redundancy (Ratio to original dataset)

Our fragmentation strategies find and materialize some frequent access patterns (or structural minterm predicates). As discussed in Section 8.2, the number of frequent access patterns is limited. Hence, the redundancy ratios of our fragmentation strategies are limited. Note that, the horizontal strategy has a little larger redundancy ratio than the vertical fragmentation strategy. This is because that different structural minterm predicates derived from the same frequent access patterns share some common triple patterns. These common triple patterns may cause more redundant edges.

## 8.7 Offline Performance

Table 2 shows the data partitioning and loading time of the datasets for different fragmentation strategies. Although SHAPE has an almost perfect uniform distribution, its redundancy ratio is too large and each fragment contains too many redundant edges. Hence, loading fragments in SHAPE also takes much time. WARP uses METIS [12]. Since DBPedia is sparse (i.e.  $|E(G)|/|V(G)| \approx 1$ ), METIS can guarantee that there are a few redundant edges and all fragments have a nearly uniform distribution. Then, WARP has less loading time than SHAPE. However, for WatDiv, the data graph is dense (i.e.  $|E(G)|/|V(G)| \gg 1$ ), so the fragmentation result of

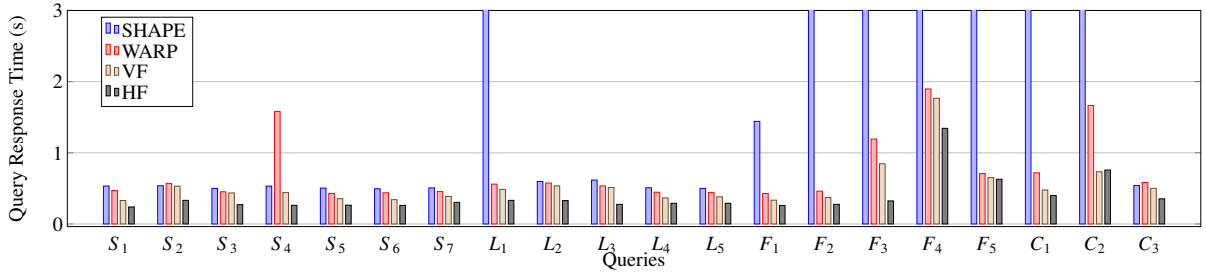


Figure 12: Query Performance of Benchmark Queries

METIS is unbalanced. Then, WARP takes more loading time than SHAPE to load the largest fragments.

Since nearly half of all edges for DBPedia are infrequent edges, loading the cold graph of DBPedia is the bottleneck in our fragmentation strategies. However, in WatDiv, there are not so many infrequent edges. Then, the loading time of our fragmentation strategies for WatDiv is more acceptable. Note that, because the structural minterm predicates are derived from the frequent access patterns, the cold graphs for the vertical and horizontal fragmentation strategies are the same. Thus, the loading times for the vertical and horizontal fragmentation strategies are the same.

Strategies	DBPedia			WatDiv		
	Partitioning	Loading	Total	Partitioning	Loading	Total
SHAPE	41	30	71	20	19	49
WARP	43	28	71	33	46	79
VF	50	97	147	31	28	59
HF	58	97	139	34	28	62

Table 2: Partitioning and Loading Time (in min)

## 8.8 Experiments for Benchmark Queries

In this experiment, we compare our methods with other fragmentation strategies on benchmark queries provided by WatDiv. There are 20 benchmark queries in WatDiv, and these queries can be classified into 4 structural categories: linear (L), star (S), snowflake (F) and complex (C). Figure 12 shows the performance of different approaches. Generally speaking, we find out that our methods outperforms other two methods in most cases. This is because that each benchmark query can be decomposed into some frequent access patterns or structural minterm predicates. Hence, our fragmentation strategies can filter out many irrelevant fragments. In contrast, SHAPE and WARP always concern all fragments, and SHAPE further needs some cross-fragment joins for complex queries.

Let us look deeper into Figure 12 and analyze each individual fragmentation strategy. SHAPE has to involve all fragments for any queries, so its performance is always worse than our fragmentation strategies. In particular, for star queries ( $S_1$  to  $S_7$ ), the difference between the query response times of SHAPE and our fragmentation strategies is not very large, because the subject-object-based triple groups that we use can guarantee that there is no intermediate result and all star queries can be answered at each fragment locally. However, for other shapes of queries, SHAPE has to decompose the queries and do cross-fragment joins to merge the intermediate results. Then, the performance of SHAPE decreases greatly. Especially for the unselective queries ( $L_1$ ,  $F_1$ ,  $F_2$ ,  $F_3$ ,  $F_4$ ,  $F_5$ ,  $C_1$  and  $C_2$ ), the performance of SHAPE is an order of magnitude worse than our fragmentation strategies.

Since WARP also use patterns to replicate triples for avoiding cross-fragment joins in complex queries, WARP has better performance than SHAPE in most case. However, WARP still always concerns all fragments in all sites for any kind of queries. The

search space of WARP for a query is higher than our fragmentation strategies. Thus, our fragmentation strategies always result in better performance. Especially for the query of very complex structure ( $C_2$ ), our fragmentation strategies can filter out many irrelevant fragments, which can result in much smaller search space than WARP. Hence, for  $C_2$ , our strategies is twice as fast as WARP.

Since all benchmark queries are generated from instantiating benchmark templates with actual RDF terms, these benchmark queries always correspond to a limited number of minterm predicates. Hence, the horizontal fragmentation is always faster than the vertical fragmentation.

## 9. RELATED WORK

For both the general graph and the RDF graph, as the graph size grows beyond the capability of a single machine, many works [6, 8, 9, 10, 29, 14, 15, 7, 23, 12, 30, 22, 25] have been proposed about graph fragmentation and allocation. We can divide all these methods into two categories: global goal-oriented graph fragmentation methods and local pattern-based graph fragmentation methods.

**Global Goal-Oriented Graph Fragmentation.** For this kind of methods [12, 9, 30, 22, 16], they divide  $G$  into several fragments while maximizing some goal function. They first transform a large graph into a small graph; then, apply some graph partitioning algorithms on the small graph; finally, the partitions on the small graph are projected back to the original graph. These methods often apply some existing methods (such as KL [13]) directly on the transformed graph in the second step. If we track the transforming step, the partitions on the small graph can be easily projected back to the original graphs in the third step. Hence, the largest difference among different graph coarsening-based methods is how to coarsen the original graph into a small graph.

In particular, METIS [12] uses the maximal matching to coarsen the graph. A matching of a graph is a set of edges that no two edges share an endpoint. A maximal matching of a graph is a matching to which no more edges can be added and remain a matching. GraphPartition [9] directly uses METIS in the RDF graph. WARP [8] uses some frequent structures in workload to further extend the results of GraphPartition. EAGRE [30] coarsens the RDF graph by using the entity concept in RDF data. It considers an entity to be a subject and its complete description. By grouping the entities of the same class, an RDF graph can be compressed as a compressed RDF entity graph. MLP [22] designs a method to coarsen the graph by label propagation. Vertices with the same label after the label propagation are coarsened to a vertex in the coarsened graph. Sheep [16] transform the graph into a elimination tree via a distributed map-reduce operation, and then partition this tree while reducing communication volume. Tomaszuk et. al. [21] briefly survey how to apply existing graph fragmentation solutions from the theory of graphs to RDF graphs.

Global goal-oriented graph fragmentation methods assume that if there are few edges crossing different fragments, the communi-

cation cost is little. If an application involves nearly all vertices in the graph, few cross-fragments edges indeed result in little communication. A typical application suitable for graph coarsening-based methods is PageRank.

In some applications, one static fragmentation cannot fit all. Hence, Sedge [28] maintains many fragmentations with different crossing edges, while Shang et. al. [19] move some vertices of one fragment to another fragment during graph computing according to the workload. Yan et. al. [27] propose an indexing scheme based on fragmentation to help query engine fast locate the instances.

**Local Pattern-based Graph Fragmentation.** For this kind of methods [10, 29, 14, 15, 7, 23, 25], they first find certain patterns as the fragmentation units to cover the whole graph; then, they distribute these patterns into sites. The local pattern-based methods mainly differ in their definitions of the fragmentation unit.

HadoopRDF [10] groups triples with the same property together and each group corresponds to a fragmentation unit. Then, they store all fragmentation units over HDFS. Yang et. al. [29] define some special query patterns, and subgraphs of a pattern are considered as a fragmentation unit. Lee et. al. [14, 15] define the fragmentation unit as a vertex and its neighbors, which they call a triple group. The triple groups are distributed based on some heuristic rules. For each vertex, SketchCluster [23] identifies the set of labeled vertices reachable within its one-hop neighborhood as its features and employs the KModes algorithm to group related vertices based on the features. Partout [6] extends the concepts of minterm predicates in relational database systems, and uses the results of minterm predicates as the fragmentation units. TriAD [7] uses METIS [12] to divide the RDF graph into many partitions. Then, each result partition is considered as a unit and distributed among different sites based on a hash function. PathPartitioning [25] uses paths in RDF graphs as fragmentation units.

Local pattern-based graph fragmentation methods assume that some real applications only concerns a part of the whole graph. If an application only concerns the vertices of some certain patterns, these methods only access the relevant fragments and reduce the communication cost across fragments. A typical example application is subgraph homomorphism checking.

## 10. CONCLUSION

In this paper, we discuss how to manage the large RDF graph in a distributed environment. First, we mine and select some frequent access patterns to partition the RDF graph into many smaller fragments. Then, we propose an allocation algorithm to distribute all fragments over different sites. Last, we discuss how process the query based on the results of fragmentation and allocation. Extensive experiments verify our approaches.

**Acknowledgement.** This was supported by 863 project under Grant No. 2015AA015402, NSFC under Grant No. 61532010, 61370055, 61272344 and 61303073. Lei Chen's work is supported in part by the Hong Kong RGC Project N HKUST637/13, National Grand Fundamental Research 973 Program of China under Grant 2014CB340303, NSFC Grant No. 61328202, NSFC Guang Dong Grant No. U1301253, Microsoft Research Asia Gift Grant and Google Faculty Award 2013

## 11. REFERENCES

- [1] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In *ISWC*, pages 197–212, 2014.
- [2] M. M. Astrahan, H. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1:97–137, 1976.
- [3] L. Bordeaux, Y. Hamadi, and P. Kohli. *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press, 2014.
- [4] DBpedia. <http://dbpedia.org/about>.
- [5] P. Fränti, O. Virmajoki, and V. Hautamäki. Fast PNN-based Clustering Using K-nearest Neighbor Graph. In *ICDM*, pages 525–528, 2003.
- [6] L. Galarraga, K. Hose, and R. Schenkel. Partout: A Distributed Engine for Efficient RDF Processing. In *WWW (Companion Volume)*, pages 267–268, 2014.
- [7] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In *SIGMOD Conference*, pages 289–300, 2014.
- [8] K. Hose and R. Schenkel. WARP: Workload-aware Replication and Partitioning for RDF. In *ICDE Workshops*, pages 1–6, 2013.
- [9] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11):1123–1134, 2011.
- [10] M. F. Husain, J. P. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Trans. Knowl. Data Eng.*, 23(9):1312–1327, 2011.
- [11] R. K. Iyer and J. A. Birmes. Submodular Optimization with Submodular Cover and Submodular Knapsack Constraints. *CoRR*, abs/1311.2106, 2013.
- [12] G. Karypis and V. Kumar. Analysis of Multilevel Graph Partitioning. In *SC*, 1995.
- [13] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Technical Journal*, 49(2), 1970.
- [14] K. Lee and L. Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *PVLDB*, 6(14):1894–1905, 2013.
- [15] K. Lee, L. Liu, Y. Tang, Q. Zhang, and Y. Zhou. Efficient and customizable data partitioning framework for distributed big RDF data processing in the cloud. In *IEEE CLOUD*, pages 327–334, 2013.
- [16] D. W. Margo and M. I. Seltzer. A Scalable Distributed Graph Partitioner. *PVLDB*, 8(12):1478–1489, 2015.
- [17] S. Nijssen and J. N. Kok. The Gaston Tool for Frequent Subgraph Mining. *Electr. Notes Theor. Comput. Sci.*, 127(1):77–87, 2005.
- [18] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.
- [19] Z. Shang and J. X. Yu. Catch the Wind: Graph Workload Balancing on Cloud. In *ICDE*, pages 553–564, 2013.
- [20] J. Sîma and S. E. Schaeffer. On the NP-Completeness of Some Graph Cluster Measures. *CoRR*, abs/cs/0506100, 2005.
- [21] D. Tomaszuk, L. Skonieczny, and D. Wood. RDF Graph Partitions: A Brief Survey. In *BDAS*, pages 256–264, 2015.
- [22] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to Partition a Billion-node Graph. In *ICDE*, pages 568–579, 2014.
- [23] Y. Wang, S. Parthasarathy, and P. Sadayappan. Stratification Driven Placement of Complex Data: A Framework for Distributed Data Analytics. In *ICDE*, pages 709–720, 2013.
- [24] G. Wiederhold. *Database Design, Second Edition*. McGraw-Hill, 1983.
- [25] B. Wu, Y. Zhou, P. Yuan, L. Liu, and H. Jin. Scalable SPARQL Querying using Path Partitioning. In *ICDE*, pages 795–806, 2015.
- [26] X. Yan, P. S. Yu, and J. Han. Graph Indexing: A Frequent Structure-based Approach. In *SIGMOD Conference*, pages 335–346, 2004.
- [27] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan. Efficient Indices Using Graph Partitioning in RDF Triple Stores. In *ICDE*, pages 1263–1266, 2009.
- [28] S. Yang, X. Yan, B. Zong, and A. Khan. Towards Effective Partition Management for Large Graphs. In *SIGMOD Conference*, pages 517–528, 2012.
- [29] T. Yang, J. Chen, X. Wang, Y. Chen, and X. Du. Efficient SPARQL Query Evaluation via Automatic Data Partitioning. In *DASFAA (2)*, pages 244–258, 2013.
- [30] X. Zhang, L. Chen, Y. Tong, and M. Wang. EAGRE: Towards Scalable I/O Efficient SPARQL Query Evaluation on the Cloud. In *ICDE*, pages 565–576, 2013.
- [31] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao. gStore: A Graph-based SPARQL Query Engine. *VLDB J.*, 23(4):565–590, 2014.



# Efficient Query Processing Using the Earth Mover's Distance in Video Databases

Merih Seran Uysal, Christian Beecks, Daniel Sabinasz, Jochen Schmücking, Thomas Seidl

Data Management and Exploration Group  
RWTH Aachen University,  
Germany

{uysal, beecks, sabinasz, schmuecking, seidl}@cs.rwth-aachen.de

## ABSTRACT

The rapid increase in generation and dissemination of online video data has recently raised the demand on efficient and effective query processing techniques in large video databases. In this paper, we first introduce a novel compact video representation model to achieve high effectiveness, and then propose to alleviate computational time complexity of the well-known *Earth Mover's Distance* by introducing a filter approximation analyzing earth flows locally and restricting the number of flows globally, ensuring *completeness*. Moreover, extensive experimental evaluation performed on high dimensional real world datasets points out high efficiency and effectiveness of the proposals, significantly reducing the number of Earth Mover's Distance computations and outperforming the state of the art by up to two orders of magnitude with respect to selectivity and query processing time.

## Categories and Subject Descriptors

H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval; H.2.4 [Database Management]: Systems—*Multimedia databases*

## General Terms

Theory, Performance, Experimentation

## Keywords

Earth Mover's Distance, Lower Bound, Filter Distance, Efficient Query Processing

## 1. INTRODUCTION

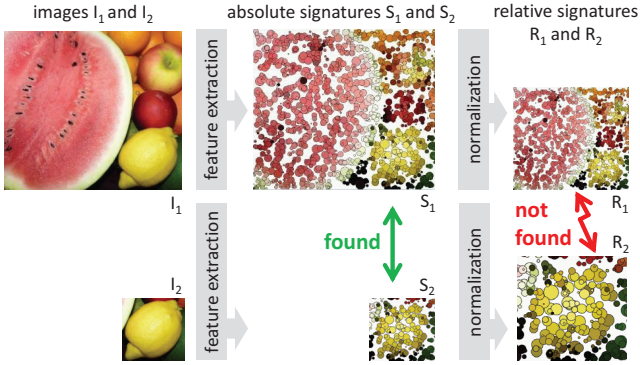
With increasing ubiquity of the internet and rich diversity of multimedia capture devices and social networking and data sharing web sites, recent years have witnessed an explosion in generation and collection of multimedia data, in particular videos. As reported in [26], 100 hours of video are uploaded to YouTube every minute, and over 6 billion

hours of video are watched each month on the same website. The resulting enormous amount of video data in the technological world today makes efficient and effective query processing indispensable for large video databases.

The Earth Mover's Distance (EMD) [15] denoting strong human perceptual similarity is proven to be a very effective distance-based similarity measure in various domains. The EMD determines the dissimilarity between two data objects by the minimum amount of work required to transform one feature representation into another one. Each data object, for example a video clip, can be represented by a *signature* denoting individual object-specific features, or by a *histogram* consisting of shared features in the feature space where histograms expose a special case of signatures. Signatures which are also referred to as *adaptive binning* or *individual binning* can be used to represent a wide spectrum of data types, such as uncertain [3], medical [2], probabilistic [25], and multimedia data [15, 24, 21, 22], as well as events [19] and molecules [6]. The major advantage of the utilization of the signatures is the high quality of content approximation coupled with similarity search and query processing in various type of databases.

A signature is basically defined as a set of features, also found as *representatives* in the literature, in a feature space. Each feature is assigned a real-valued weight denoting the number of features related to that corresponding feature. This is carried out by first extracting the features and then clustering them by using a clustering algorithm, such as k-means algorithm. The resulting counters of the features in the clusters form a signature which we refer to as an *absolute signature* exhibiting individual total weights. Absolute signatures are appropriate for applications for which different characteristics and properties of data are of high importance, such as different image resolution or different video clip length. Many applications rely on an additional preprocessing step by which the absolute signatures are normalized, leading to *relative signatures* exposing a uniform total weight among all data objects. Below, we will immediately show the limitations caused by this normalization step, and how important the usage of absolute signatures is, particularly if partial similarity is involved. Overall, this paper aims at efficient similarity query processing for absolute signatures which is supported only little by existing work.

Relative signatures have often been utilized in numerous applications [15, 1, 24, 21], however, absolute signatures and similarity search using them are still unexplored. The diagnosis of various types of cancer or neurological diseases, such as Alzheimer's Disease [9] require absolute signatures



**Figure 1:** Given images  $I_1$  and  $I_2$ , the absolute signature  $S_2$  is found to be a part of the absolute signature  $S_1$  which consider individual absolute weights. However, a normalization step results in relative signatures  $R_1$  and  $R_2$  which are detected as non-similar.

in (bio)medical image classification and similarity search. Rough and coarse boundaries of a cell makes it difficult to determine if it is a cancer/tumor cell. Such biomedical images are commonly stored in fuzzy object databases where each image is partitioned in a specific number of shells where the assignment of a certain probability to each pixel is essential, i.e. the absolute number of pixels for each shell is important to store [20]. An extra normalization step after the feature extraction would lead to inappropriate fuzzy object representation of the cells and, thus, to irrelevant results. In biotechnology, the metabolite identification and quantification is an important task for which data normalization results in obscuring variation of data where the chemical properties cannot be preserved any more [10]. In addition, normalizing the metabolomic data affects its covariance structure which is undesired by the experts.

To contribute to the reader’s understanding, we illustrate the absolute and relative signatures in Figure 1. A common task in partial similarity search is to determine if a particular part of a given data object exists in the target dataset. Each signature comprises representatives visualized by circles and is based on the characteristic information of the presented image, such as color. Image  $I_1$  comprises various fruits including also a lemon, while the image  $I_2$  shows only a lemon where the user intends to determine if a lemon exists in the image  $I_1$ . When the absolute signatures  $S_1$  and  $S_2$  are considered,  $S_2$  is found to be similar to  $S_1$ , since it is detected as a part of  $S_1$ . However, if an additional normalization step is applied to attain the relative signatures  $R_1$  and  $R_2$ , they are detected as non-similar. A closer look reveals that the two images are evaluated as different images due to the utilization of the normalization step after the feature extraction, i.e. normalizing absolute signatures does not carry out the required partial similarity search task. Another example is the currently attractive and vital domain of video similarity search, in particular near-duplicate video detection, where a typical subclip video detection task [8], required for various purposes as copyright protection and management, entails the need to utilize absolute signatures so that a query subclip can be detected in a given video dataset. Hence, depending on the application, it is explicitly crucial to utilize absolute signatures for queries and datasets, as normalization does

not attain partial similarity search tasks formulated and demanded by the user.

Since the empiric time complexity of the EMD is super-cubic with respect to feature dimensionality, database community has devised research to propose efficient query processing techniques for the EMD [15, 4, 1, 24, 25, 21]. While existing efficiency improvement techniques for the EMD have been successfully utilized on relative signatures, nevertheless most of them have unfortunately the shortcoming that they can only be applied to fixed-binned relative signatures. Furthermore, they cannot be applied to absolute signatures denoting individual total weights which come up in numerous applications and domains, such as in computer vision [13, 4], multimedia databases [11], fuzzy object databases [20], and biotechnology [10]. While the lower-bounding technique IM-Sig (Independent Minimization for Signatures) [21] is proven to result in efficient results, it can only be applied to relative signatures, not to absolute signatures.

In this paper, we introduce a lower-bounding filter approximation technique  $IM-Sig^*$  which is applicable to both fixed-binned and adaptive-binned absolute and relative signatures. In particular, our approach computes the same filter distance as for IM-Sig on relative signatures, and on top of this, our proposal can also be applied to the absolute signatures, hence, filling the gap with respect to lower-bounding the EMD on absolute signatures. To this end, we focus on efficient query processing with the EMD on both relative and absolute signatures in order to introduce a comprehensive solution, which is carried out by analyzing earth flows locally and restricting the number of flows globally. In addition, we take the video domain as an example in this paper, however, it is noteworthy that our efficiency improvement technique can be applied to all domains where complex data objects need to be represented by relative or absolute signatures, as mentioned above. The main contributions of our paper are listed as follows:

- We introduce an adaptive-binning video representation model applicable to the EMD (Section 3).
- We propose an analytic solution  $IM-Sig^*$  for adaptive-binned signatures without any weight restriction (Section 5.1-5.2).
- We show the optimality of our solution leading to the lower-bounding property of the  $IM-Sig^*$ , ensuring exact query processing with the EMD (Section 5.3).
- We develop an algorithm for our proposal and analyze the computational time complexity (Section 5.4).
- Experiments on real world data show the efficiency and efficacy of our approach (Section 6).

## 2. RELATED WORK

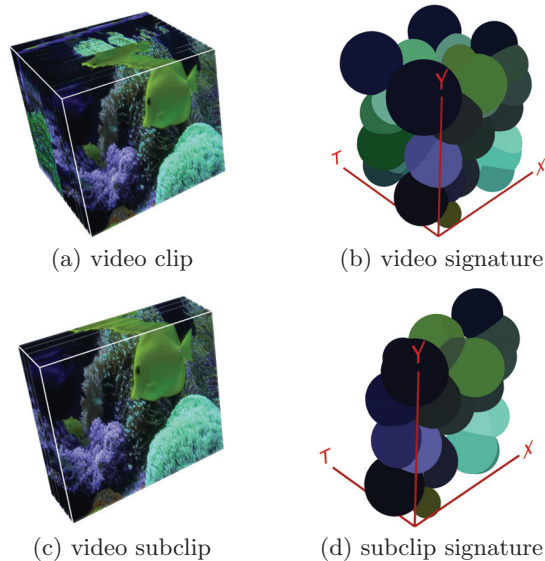
**Efficient Query Processing with the EMD.** Various efficient query processing techniques have been proposed for the EMD on histograms, i.e. fixed-binned signatures. [1] proposed to lower-bound the EMD via  $L_p$ -based distances and constraint relaxation. [24] developed dimensionality reduction techniques for the EMD where reduced cost matrices are utilized relying on the original cost matrix. Furthermore, [25] derived a lower bound of the EMD by utilizing the primal-dual theory in linear programming on top of  $B^+$ -trees. In addition, [16] proposed to lower-bound the EMD by

**Table 1: Overview of the lower bounds to the Earth Mover’s Distance regarding feature representations**

lower bounds to the EMD	signatures	
	adaptive binning	individual signature weight
$L_p$ -based [1]		
RedEMD [24]		
PrimalDual [25]		
Rubner [15]	✓	
IM-Sig [21]	✓	
Pemd [4]	✓	✓
IM-Sig*	✓	✓

projecting histograms on a vector and approximating their distance by a normal distribution. It is noteworthy that the limitation of all aforementioned approaches is that they are applicable to histograms sharing the same features in a feature space, not to adaptive-binned signatures denoting individual features per data object. As mentioned in the previous section, since histograms denote a special case of signatures by utilizing shared features instead of individual features per data object, it is of vital importance to propose comprehensive methods applicable to signatures. [15] proposed to lower-bound the EMD by computing the distance between mean signatures. Although the filter time is remarkably low, the efficiency of the query processing is hampered by the worse selectivity resulting in high refinement time. Moreover, a considerable limitation lies in the fact that it cannot be applied to absolute signatures, as will be presented in Section 4. Another efficiency improvement technique for signatures is proposed by [21], which is based on the relaxation of the target constraint of the EMD by local examination of each feature in the source signature. While this method indicates high efficiency improvement, it is nevertheless not applicable to absolute signatures with individual total weights. Furthermore, [4] proposed to lower-bound the EMD on signatures by computing the EMD values for projected signatures each of which comprises features projected on an individual dimension of the feature space. Note that the latter approach is applicable to both relative and absolute signatures. The overview of the applicability of existing lower-bounding methods and our approach IM-Sig\* with respect to feature representations is depicted in Table 1.

**Video Similarity Search Models.** Video similarity search in video databases has been a challenging research area where there have been numerous attempts to provide effective similarity search techniques. [18] (*vitri*) summarizes each video into a small number of clusters each of which includes similar frames. The similarity between two videos is determined by simply estimating the number of similar frames, neglecting the temporal information. [27] (*fras*) is another approach which is an improvement of [18], symbolizing video sequences on a frame basis. The limitation of both approaches lies in the determination of a threshold which is supposed to specify the similarity between any two frames. [8] (*vdt*) proposes to transform a video from a sequence of histograms denoting frames into a one-dimensional distance trajectory where the distance is determined via a reference point. This model is contingent upon frame elements which may lead to performance limitations regarding generation of



**Figure 2: Illustration of a video clip and a subclip with the visualization of their video signatures.**

video representations of a high number of frames, and the segmentation of linear segments requires the determination of a threshold regarding the distance between two consecutive frames, which is not robust to outliers. [7] (*bcs*) is another video clip representation model utilizing the principal component analysis in order to specify a bounded range of data projections along each coordinate axis. This method neglects temporal information causing a restriction, in particular for long video clips. Not least, all aforementioned approaches propose to represent each frame only by a histogram in RGB color space.

### 3. NOVEL VIDEO REPRESENTATION

In this section, we propose our novel video model *Video Signature (vis)* which utilizes the determination of individual features and related weights denoting the number of assigned features, leading to a particular compact video clip representation. Unlike frame-based and sequence-based models [7, 8, 18, 27], our model is not contingent upon frames or keyframes, attaining great flexibility via exploiting any requested feature types, such as color, position, contrast, and coarseness. On top of this, the proposal implicitly takes the temporal information into consideration which does not require extra effort at all, since the temporal information is utilized as an individual dimension of the underlying feature space. Mathematically, let  $(\mathbb{F}, \delta)$  be a feature space where  $\mathbb{F}$  is a set of features coupled with a ground distance function  $\delta : \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{R}$ . Any video clip is represented by a finite set of features  $x_1, \dots, x_n \in \mathbb{F}$ . We refer to a *video signature* as a finite set of features (so-called representatives) each of which is assigned a non-negative real number corresponding to the number of features assigned to that representative. The formal definition is given below.

**DEFINITION 1 (VIDEO SIGNATURE).** *Given a feature space  $(\mathbb{F}, \delta)$ , a video signature  $V$  is defined as  $V : \mathbb{F} \rightarrow \mathbb{R}^{\geq 0}$ , subject to  $|R_V| < \infty$ , where  $R_V := \{x \in \mathbb{F} \mid V(x) > 0\} \subseteq \mathbb{F}$  denotes the set of representatives of  $V$ .*

According to the definition above, each representative contributes to the video representation by taking a positive real number  $V(x) \in \mathbb{R}^{\geq 0}$  as weight. Any video signature, hence, includes an individual set of representatives and weights which leads to an appropriate video representation.

The illustration of a video clip and a subclip with their corresponding video signature visualizations are depicted in Figure 2. The signatures comprise 50 representatives which are visualized as spheres based on position, color, texture, and temporal information. In the visualization, positional (X,Y) and temporal (T) dimensions are explicitly utilized to contribute to the reader’s understanding, where the size of each sphere refers to the weight of that representative. This figure epitomizes the video approximation and modeling with respect to subclip video detection: Obviously, the individual total weight of the absolute signature of the video clip (b) is greater than that for the subclip video (d) which facilitates the utilization of the EMD to determine the dissimilarity between them in order to solve the desired subclip detection task. If they were normalized, the desired task would not be solved, since the EMD would then determine *total similarity* between the two videos, which does not correspond to the user’s intention. In the upcoming sections, the term *signature* refers to a video signature. As will be presented in Section 6, modeling videos as *video signatures* highly contributes to effectiveness results. For the sake of simplicity, in the following sections, we utilize the class of non-negative signatures  $\mathbb{S}^+ := \{V|V \in \mathbb{R}^{\mathbb{F}} \wedge 0 < |R_V| < \infty \wedge \forall x \in \mathbb{F} : V(x) \in \mathbb{R}^{\geq 0}\}$  including video signatures whose representatives denote non-negative weights. In the following section, we present the EMD and the utilized filter-and-refine architecture.

#### 4. EARTH MOVER’S DISTANCE

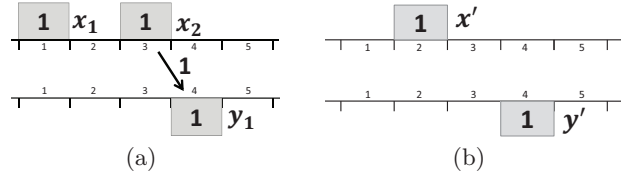
In this section, we present the well-known Earth Mover’s Distance which can be utilized in a filter-and-refine architecture in order to boost the query processing. Initially introduced in the computer vision domain, the Earth Mover’s Distance (EMD) [15] computes the dissimilarity between two signatures by transforming one signature into another one. The formal definition is given below.

**DEFINITION 2.** Let  $X, Y \in \mathbb{S}^+$  be two signatures over a feature space  $(\mathbb{F}, \delta)$  and  $\delta : \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{R}$  be a ground distance function. The Earth Mover’s Distance  $EMD : \mathbb{S}^+ \times \mathbb{S}^+ \rightarrow \mathbb{R}$  between  $X$  and  $Y$  is defined as a minimum-cost flow of all possible flows  $F = \{f|f : \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{R}\} = \mathbb{R}^{\mathbb{F} \times \mathbb{F}}$  as:

$$EMD(X, Y) = \min_{f \in F} \left\{ \frac{\sum_{x \in \mathbb{F}} \sum_{y \in \mathbb{F}} f(x, y) \cdot \delta(x, y)}{\min\left\{ \sum_{x \in \mathbb{F}} X(x), \sum_{y \in \mathbb{F}} Y(y) \right\}} \right\},$$

subject to constraints  $NC \wedge SC \wedge TAC \wedge FC$  with:  
 $NC: \forall x, y \in \mathbb{F} f(x, y) \geq 0, SC: \forall x \in \mathbb{F} \sum_{y \in \mathbb{F}} f(x, y) \leq X(x),$   
 $TAC: \forall y \in \mathbb{F} \sum_{x \in \mathbb{F}} f(x, y) \leq Y(y),$  and  
 $FC: \sum_{x \in \mathbb{F}} \sum_{y \in \mathbb{F}} f(x, y) = \min\left\{ \sum_{x \in \mathbb{F}} X(x), \sum_{y \in \mathbb{F}} Y(y) \right\}.$

The EMD is the minimum cost required to transform one signature into another one by guaranteeing the non-negativity (NC), source (SC), target (TAC), and total flow constraints (FC), as given above. Hence, the EMD denotes a linear optimization problem and can be solved by simplex

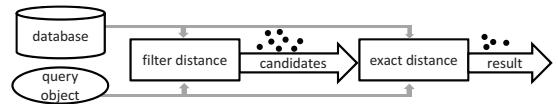


**Figure 3:** (a) The EMD between absolute signatures  $X$  and  $Y$  is  $EMD(X, Y) = 1$ . (b)  $Rubner(X, Y) = \delta(x', y') = 2 \not\leq EMD(X, Y)$  holds, i.e. Rubner filter is not a lower bound to the EMD on absolute signatures denoting individual total weights.

algorithms. Figure 3(a) illustrates an example EMD computation between two signatures  $X, Y$  where 1 unit earth is transferred from  $x_2 \in R_X$  with a distance of 1, resulting in the EMD value of  $1 \times 1 = 1$ . It is worth noting that the absolute signatures  $X$  and  $Y$  exhibit individual total weights of 2 and 1, respectively, where the total flow constraint FC guarantees that the minimum of the total weights is transferred from the source signature  $X$  to the target signature  $Y$ .

Given two signatures  $X, Y$  with total weights  $m_X, m_Y$ , and a norm-based ground distance function  $\delta$ , the Rubner filter distance [15] is defined as:  $\delta((\sum_{x \in R_X} X(x) \cdot x)/m_X, (\sum_{y \in R_Y} Y(y) \cdot y)/m_Y)$ , as depicted in Figure 3(b):  $x'$  and  $y'$  refer to weighted mean features of  $X$  and  $Y$  from (a), respectively. The Rubner filter computes  $\delta(x', y') = 2$  which, however, does not lower-bound the EMD. As illustrated by this example, the restriction of the Rubner filter is that it cannot be applied to absolute signatures.

**Filter-and-refine Architecture.** One of the efficiency improvement methods utilized in k-nearest-neighbor (k- $nn$ ) query processing is the filter-and-refine architecture model comprising filter and refinement steps [5, 17, 12], as summarized in Figure 4. In the filter step, a filter  $LB_d$  generates a set of candidates which is then refined in the refinement step by utilizing the exact distance  $d$  (here EMD). A filter ideally fulfills the following properties: First, its computation is attained more efficiently than for the exact distance computation (*efficiency*). Second,  $LB_d$  lower-bounds  $d$ , i.e. the final refined set includes all objects from the result set, guaranteeing no false dismissals, as it holds  $\forall x, y : LB_d(x, y) \leq d(x, y)$  (*completeness*). Third, the generated set of candidates is smaller if  $LB_d$  is tighter, leading to lower computation cost.



**Figure 4:** Multistep query processing

In this paper, we utilize a multistep approach which is proven to be optimal in the number of candidates [17]. After a ranking is generated by using a filter distance, it is processed as long as the filter distance does not exceed the exact distance of the current  $k^{th}$ -nearest neighbor where the result set and the  $k^{th}$ -nearest neighbor distance are continuously updated. After giving the EMD and filter-and-refine architecture, we below present our proposed technique IM-Sig\* applicable to both absolute and relative signatures.

## 5. LOWER-BOUNDING THE EMD ON SIGNATURES

In this section, we first deal with the shortcoming of the existing approach IM-Sig, and then present our proposed comprehensive lower-bounding technique IM-Sig\* on signatures, irrespective of any prior information about their total weights. Then, we theoretically show that our analytic solution is both feasible and optimal, which is thus a lower bound to the EMD on all type of signatures including absolute and relative signatures with individual and uniform total weights, respectively. We finally present the computational algorithm of our technique and its complexity analysis.

Our approach has two attractive advantages: First, it is applicable to both relative and absolute signatures, yielding high flexibility with respect to query processing and explicit user-driven tasks, such as subclip video detection, as mentioned before. Second, it is a generic solution regarding efficient query processing which is not restricted to the video domain, and, hence, can be applied to any other domain, such as multimedia, computer vision, and medicine.

### 5.1 IM-Sig\* and the Limitations of IM-Sig

The filter approximation technique Independent Minimization for Signatures (IM-Sig) [21], based on target constraint relaxation of the EMD, was originally proposed to lower-bound the EMD on signatures with uniform total weight. Below, we first give the formal definition of the comprehensive lower-bounding technique IM-Sig\*, irrespective of any prior information about the total weights of signatures, and then present the shortcoming of IM-Sig via illustrative examples.

**DEFINITION 3 (IM-SIG\* FILTER DISTANCE).** Let  $(\mathbb{F}, \delta)$  be a feature space with a distance function  $\delta$  and  $X, Y \in \mathbb{S}^+$  be two non-empty positive signatures with weights  $m_X = \sum_{x \in \mathbb{F}} X(x)$  and  $m_Y = \sum_{y \in \mathbb{F}} Y(y)$ . The comprehensive filter distance Independent Minimization for Signatures IM-Sig\* :  $\mathbb{S}^+ \times \mathbb{S}^+ \rightarrow \mathbb{R}^{\geq 0}$  between  $X$  and  $Y$  is defined as a minimization over all possible flows  $F = \{f | f : \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{R}\}$ :

$$IM-Sig^*(X, Y) = \min_{f \in F} \left\{ \sum_{x \in \mathbb{F}} \sum_{y \in \mathbb{F}} \frac{\delta(x, y)}{\min(m_X, m_Y)} f(x, y) \right\},$$

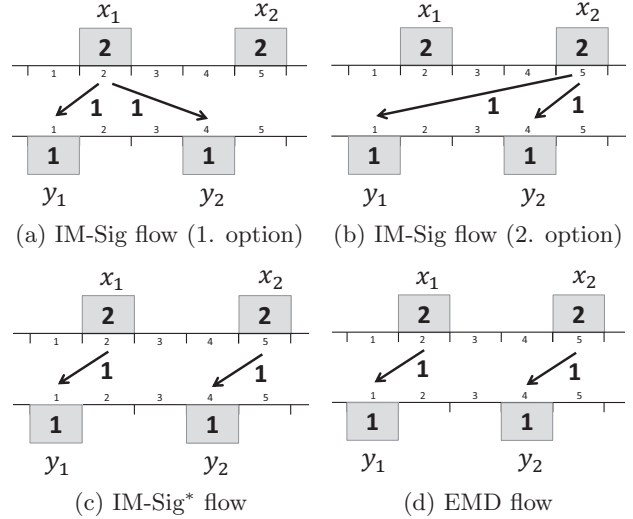
subject to constraints  $NC \wedge SC \wedge TC \wedge FC$  with:

$$NC: \forall x, y \in \mathbb{F} f(x, y) \geq 0, \quad SC: \forall x \in \mathbb{F} \sum_{y \in \mathbb{F}} f(x, y) \leq X(x),$$

$$TC: \forall x, y \in \mathbb{F} : f(x, y) \leq Y(y), \quad \text{and}$$

$$FC: \sum_{x \in \mathbb{F}} \sum_{y \in \mathbb{F}} f(x, y) = \min\left\{ \sum_{x \in \mathbb{F}} X(x), \sum_{y \in \mathbb{F}} Y(y) \right\}.$$

While the non-negativity (NC), source (SC), and total flow constraints (FC) remain unchanged for the EMD and IM-Sig\*, the target constraint (TC) of IM-Sig\* relaxes that for the EMD by allowing that any single incoming flow (instead of total incoming flows) may not exceed the target capacity. If the signatures exhibit uniform total weights, i.e. if they are relative signatures, the approach IM-Sig [21] can be applied. However, in other cases an appropriate solution is required for the computation of the filter approximation on any kind of signatures with relative or absolute weights. For all possible cases, we propose to compute the comprehensive IM-Sig\* by analyzing earth flows locally and restricting



**Figure 5: IM-Sig, IM-Sig\*, and EMD flows illustrated on two absolute signatures. Since IM-Sig is not defined on absolute signatures exhibiting individual total weights, there is neither a deterministic solution nor a computable minimum-cost flow (a-b). IM-Sig\* flow (c) computes an optimal flow which fulfills the minimum-cost property, lower-bounding the EMD (d) on both absolute and relative signatures.**

the number of flows globally. In other words, our approach IM-Sig\* computes the same flow as for IM-Sig for relative signatures and on top of this, IM-Sig\* can be applied to the absolute signatures to lower-bound the EMD.

In order to give the underlying basic notion and to clarify the difference between our approach IM-Sig\* and IM-Sig, we examine the illustrations given in Figure 5. Numbers 1-5 denote the positions in the 1-dimensional feature space, and the ground distance between any representatives with positions  $i$  and  $j$  is computed via  $|i - j|$ , such as  $\delta(x_2, y_2) = 1$ . The signatures  $X, Y$  are illustrated with 2 representatives each, where their weights are denoted in buckets. Since IM-Sig is not defined on absolute signatures, there does not exist a computable minimum-cost flow. Nonetheless, if we try to apply the naive IM-Sig algorithm to these absolute signatures denoting individual total weights, we face two main problems: First, there exists no deterministic solution since IM-Sig basically transfers earth from each representative  $x_i$  to its nearest neighbors in the target signature  $Y$ . Since there is no specific predefined order of the representatives for the earth transfer from  $X$  to  $Y$ , one can arbitrarily start with any representative. The first non-deterministic solution of IM-Sig (Fig. 5(a)) would transfer earth from  $x_1$  to its nearest neighbors  $y_1$  and  $y_2$ , resulting in  $IM-Sig(X, Y) = \frac{1}{2} \times (1 \cdot 1 + 1 \cdot 2) = 1.5$ . Another non-deterministic solution of IM-Sig (Fig. 5(b)) would transfer earth from  $x_2$  to its nearest neighbors  $y_2$  and  $y_1$ , resulting in  $IM-Sig(X, Y) = \frac{1}{2} \times (1 \cdot 1 + 1 \cdot 4) = 2.5$ . Second, the computed IM-Sig values do not necessarily yield optimal solutions, which can be inferred from the application of the optimal IM-Sig\* on the example (Fig. 5(c)): Our approach first ranks the representative pairs  $(x_i, y_j)$  with respect to their ground distance values in ascending order, and then the earth is

transferred from  $X$  to  $Y$  by taking this order into consideration, as well as all constraints given in Def. 3. Hence, we consider the permutation of  $((x_1, y_1), (x_2, y_2), (x_1, y_2), (x_2, y_1))$  with the ground distance values of 1,1,2, and 4. In this way, first 1 unit earth is transferred from  $x_1$  to  $y_1$  and then again 1 unit earth is transferred from  $x_2$  to  $y_2$  after which the optimal solution is attained, fulfilling the IM-Sig\* constraints:  $\text{IM-Sig}^*(X, Y) = \frac{1}{2} \times (1 \cdot 1 + 1 \cdot 1) = 1$ . In addition, the EMD (Fig. 5(d)) is computed as  $\text{EMD}(X, Y) = \frac{1}{2} \times (1 \cdot 1 + 1 \cdot 1) = 1$ , where we observe that the IM-Sig\* computes not only the minimum-cost flow but also the feasible solution, lower-bounding the EMD on these absolute signatures. As a result, this example illustrates that IM-Sig is unfortunately not applicable to absolute signatures with individual total weights and there is demand on novel efficient query processing techniques to solve the existing problem.

So far, we have seen the shortcomings of IM-Sig and deduce that there is need for new comprehensive lower-bounding techniques applicable to any kind of signatures without any restriction. Below, we present our novel analytic solution with respect to the computation of IM-Sig\*. For the definitions and theoretical analysis in the remainder of the paper, we assume that a feature space  $(\mathbb{F}, \delta)$  is given with a distance function  $\delta$ , and we refer to non-empty positive signatures  $X, Y \in \mathbb{S}^+$ .

## 5.2 Analytic Solution

In order to propose our novel analytic solution for IM-Sig\*, we first give the definitions of the local feasible set, extensive flow, and global feasible set which are required to define the IM-Sig\* flow, as will be given in Definition 7. The local feasible set of a representative  $x$  in the source signature  $X$  exhibits the greatest set of nearest neighbors in the target signature  $Y$ , where the total weight of its nearest neighbors may not exceed the capacity of  $x$ . The formal definition is given below.

**DEFINITION 4 (LOCAL FEASIBLE SET).** *Given two signatures  $X, Y \in \mathbb{S}^+$ , let  $(y_1, \dots, y_l)$  be a permutation of  $R_Y$  with respect to a feature  $x \in \mathbb{F}$  such that  $i \leq j \Rightarrow \delta(x, y_i) \leq \delta(x, y_j)$ . The local feasible set  $S_{X,Y}^x \subseteq R_Y$  is defined as the greatest set of nearest neighbors of  $x$  in  $R_Y$  whose total weight does not exceed  $X(x)$ :  $y_i \in S_{X,Y}^x \Leftrightarrow \sum_{j=1}^i Y(y_j) < X(x)$ . Let further  $k = |S_{X,Y}^x|$  be the greatest index in  $S_{X,Y}^x$ , then  $\hat{y}_x \notin S_{X,Y}^x$  is defined as the feature directly following  $y_k$  regarding the same permutation:*

$$\hat{y}_x = \begin{cases} y_{k+1} & \text{if } k < l = |R_Y| \\ y^* \in \mathbb{F} \setminus R_Y & \text{else} \end{cases}$$

Note that if  $|S_{X,Y}^x| = |R_Y|$ , i.e. the cardinality of the local feasible set of the representative  $x$  is the same as that for the representative set of the target signature  $Y$ ,  $\hat{y}_x \notin R_Y$  is an arbitrarily chosen feature in the feature space which does not belong to  $R_Y$ , since the capacity of  $x$  exceeds the total signature weight of the target signature, i.e.  $X(x) > m_Y$ .

The aim of the extensive flow  $f_e$  is to transfer earth from the source signature  $X$  to the target signature  $Y$  by filling up the nearest neighbors of each  $x \in R_X$  in the target signature so that for each  $x$  all the earth it owns is completely transferred to its nearest neighbors in the target signature. Note that the extensive flow does not take the individual total weights of the signatures into consideration. Techni-

cally, the local feasible set  $S_{X,Y}^x$  is utilized to define the flow whose definition is given as follows.

**DEFINITION 5 (EXTENSIVE FLOW).** *Given two signatures  $X, Y \in \mathbb{S}^+$ , let  $S_{X,Y}(x)$  be the local feasible set for any feature  $x \in \mathbb{F}$  (Def. 4). The extensive flow  $f_e : \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{R}$  is defined as follows:*

$$f_e(x, y) = \begin{cases} Y(y) & \text{if } y \in S_{X,Y}^x \\ X(x) - \sum_{y' \in S_{X,Y}^x} Y(y') & \text{if } y = \hat{y}_x \wedge \hat{y}_x \in R_Y \\ 0 & \text{else} \end{cases}$$

The extensive flow fulfills three constraints of IM-Sig\*, namely the non-negativity, source, and IM-Sig\* target constraint, which can be summarized in Corollary 1 as follows.

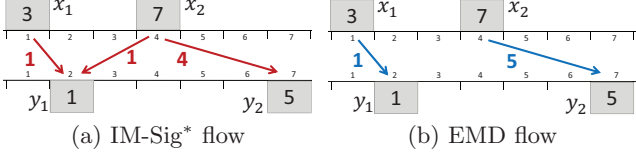
**COROLLARY 1.** *Given two signatures  $X, Y \in \mathbb{S}^+$ ,  $f_e$  fulfills the following constraints: non-negativity (NC):  $\forall x, y \in \mathbb{F} f_e(x, y) \geq 0$ , source (SC):  $\forall x \in \mathbb{F} \sum_{y \in \mathbb{F}} f_e(x, y) \leq X(x)$ , IM-Sig\* target (TC):  $\forall x, y \in \mathbb{F} f_e(x, y) \leq Y(y)$ .*

The corollary above directly follows from Def. 4 and Def. 5. After defining the local feasible set and extensive flow, we now define *global feasible set* which is required for IM-Sig\* flow. The global feasible set  $S_{X,Y}$  exhibits the greatest set including pairs  $(x_i, y_j)$  of representatives from both signatures, where the set includes all pairs for which the corresponding representative  $y_j$  receives a positive amount of earth from  $x_i$ . An important condition which needs to be fulfilled is that the pairs  $(x_i, y_j)$  are ranked according to their ground distance values in ascending order. Hence, the pairs are tracked at a *global* level, i.e. both signatures' representatives are taken into consideration, not only those of the target signature.

**DEFINITION 6 (GLOBAL FEASIBLE SET).** *Given two signatures  $X, Y \in \mathbb{S}^+$ , let  $p = ((x_1, y_1), \dots, (x_n, y_n))$  be a permutation of  $R_X \times R_Y$  such that  $i \leq j \Rightarrow \delta(x_i, y_i) \leq \delta(x_j, y_j)$ . The global feasible set  $S_{X,Y} \subseteq R_X \times R_Y$  is the greatest set satisfying  $(x_i, y_i) \in S_{X,Y} \Leftrightarrow \sum_{j=1}^i f_e(x_j, y_j) < \min(m_X, m_Y)$ , where  $k = |S_{X,Y}|$  is the greatest index in  $S_{X,Y}$  and  $(\hat{x}, \hat{y}) \notin S_{X,Y}$  is defined as  $(\hat{x}, \hat{y}) = (x_{k+1}, y_{k+1})$  which directly follows  $(x_k, y_k)$  regarding  $p$ .*

The global feasible set  $S_{X,Y}$  comprises all pairs from  $R_X \times R_Y$  sorted according to their ground distances in ascending order where the total amount of the flow coupled with such pairs may not exceed the minimum of the total weights of the signatures. The pair  $(\hat{x}, \hat{y})$  is concerned in denoting the last possible flow in the permutation  $p$  so that the total flow constraint is guaranteed.

Recall that our goal is to introduce a solution for any pair of signatures, including both relative and absolute total weights, and overcome current limitations. To this end, we explicate our proposed technique IM-Sig\* flow which transfers the minimum amount of total weights of given two signatures from the source signature  $X$  to the target signature  $Y$ . This is achieved by transferring earth by the utilization of the global feasible set which tracks the pairs of representatives allowing for appropriate flows with respect to non-negativity, source, and target constraints. IM-Sig\* flow additionally takes the total flow constraint into consideration which is significantly required to yield both feasible and optimal solution to the IM-Sig\* minimization problem. The formal definition of IM-Sig\* flow is given below.



**Figure 6: Illustration of the novel IM-Sig\* flow and the EMD flow on signatures.**

**DEFINITION 7 (IM-SIG\* FLOW).** Given two signatures  $X, Y \in \mathbb{S}^+$ , let  $S_{X,Y}$  be the global feasible set with  $m = \min(m_X, m_Y)$ . For any  $x, y \in \mathbb{F}$  IM-Sig\* flow is defined as:

$$f_S(x, y) = \begin{cases} f_e(x, y) & \text{if } (x, y) \in S_{X,Y} \\ m - \sum_{(x', y') \in S_{X,Y}} f_e(x', y') & \text{if } (x, y) = (\hat{x}, \hat{y}) \\ 0 & \text{else} \end{cases}$$

In order to contribute to the reader's understanding, we illustrate the novel comprehensive IM-Sig\* flow by means of an example in Figure 6(a). Numbers 1-7 expose the positions in the 1-dimensional feature space, and the ground distance between any representatives with positions  $i$  and  $j$  is computed via  $|i - j|$ . Two signatures  $X, Y$  are illustrated with 2 representatives each, where their weights are depicted in buckets. The required permutation is given as  $p = ((x_1, y_1), (x_2, y_1), (x_2, y_2), (x_1, y_2))$  with distances 1, 2, 3, 6, respectively. The global feasible set  $S_{X,Y} = \{(x_1, y_1), (x_2, y_1)\}$  is then determined as the greatest set whose total flow may not reach the minimum of the total weights of  $X, Y$ , i.e.  $1 + 1 < \min(10, 6) = 6$ . Thus, the pair  $(\hat{x}, \hat{y}) = (x_2, y_2)$  is the last element in the permutation which allows for flow with an amount of only 4 to fulfill the total flow constraint. Hence, IM-Sig\* is computed as  $\frac{1}{6} \times (1 \times 1 + 2 \times 1 + 3 \times 4) = 2.5$ . Not least, when compared with the EMD (Figure 6(b)) computed as  $\frac{1}{6} \times (1 \times 1 + 3 \times 5) = 2.66$ , it is obvious that the novel IM-Sig\* flow leads to a very tight lower bound to the EMD on absolute signatures.

So far, we have seen that the analytic solution IM-Sig\* flow can be introduced in order to boost the query processing with the EMD on signatures irrespective of their total weights. Below, we would like to show that our analytic solution is feasible and optimal with respect to the IM-Sig\* constraints which leads to the conclusion that the utilization of IM-Sig\* flow indeed leads to a lower bound to the EMD on any kind of signatures.

### 5.3 Theoretical Investigation

Now, we investigate our proposal with respect to its feasibility and optimality regarding IM-Sig\* constraints. First, we show that the proposed IM-Sig\* flow is a feasible flow fulfilling 4 constraints of non-negativity, source, IM-Sig\* target, and total flow, which is given by Theorem 1. Then, we show that IM-Sig\* flow is an optimal flow, i.e. there exists no other flow which results in lower overall cost than that for IM-Sig\* flow (Theorem 2). Finally, by the utilization of both theorems, we deduce that the utilization of the proposed IM-Sig\* flow leads to the lower bound to the EMD on signatures, irrespective of their total weights.

**Feasibility Analysis.** In order to show the feasibility of our approach, we consider the constraints in Def. 3 and show that these constraints are fulfilled.

**THEOREM 1 (FEASIBILITY OF IM-SIG\* FLOW).** Given signatures  $X, Y \in \mathbb{S}^+$  with total weights  $m_X = \sum_{x \in \mathbb{F}} X(x)$  and  $m_Y = \sum_{y \in \mathbb{F}} Y(y)$ , IM-Sig\* flow  $f_S$  fulfills the following constraints: non-negativity (NC):  $\forall x, y \in \mathbb{F} f_S(x, y) \geq 0$ , source (SC):  $\forall x \in \mathbb{F} \sum_{y \in \mathbb{F}} f_S(x, y) \leq X(x)$ , IM-Sig target (TC):  $\forall x, y \in \mathbb{F} f_S(x, y) \leq Y(y)$ , and total flow (FC):  $\sum_{x \in \mathbb{F}} \sum_{y \in \mathbb{F}} f_S(x, y) = \min(m_X, m_Y)$ .

**PROOF.** For the proof we consider each constraint given above and show that they are fulfilled regarding the definition of IM-Sig\* flow given in Def. 7. For any pair of features  $x, y$ , IM-Sig\* flow between these features does not exceed the extensive flow between them, i.e. given two signatures  $X, Y \in \mathbb{S}^+$ , for any  $x, y \in \mathbb{F}$  it holds:  $f_S(x, y) \leq f_e(x, y)$ , following from Def 5 and Def. 7. We denote this fact by the notation  $\otimes$  and use it below, where necessary. **NC:**  $\forall x, y \in \mathbb{F} : f_S(x, y) \geq 0$ . There exist three cases to examine: *Case 1:*  $(x, y) \notin S_{X,Y} \wedge (x, y) \neq (\hat{x}, \hat{y}) \Rightarrow f_S(x, y) = 0$ . *Case 2:*  $(x, y) \in S_{X,Y} \Rightarrow f_S(x, y) = f_e(x, y) \geq 0$ , by Cor. 1. *Case 3:*  $(x, y) = (\hat{x}, \hat{y})$ . Since  $\sum_{(x,y) \in S_{X,Y}} f_e(x, y) <$

$\min(m_X, m_Y)$  holds for any two signatures  $X$  and  $Y$ , we can write the following statement:

$$\sum_{(x', y') \in S_{X,Y}} f_e(x', y') < \min(m_X, m_Y) \Rightarrow \min(m_X, m_Y) - \sum_{(x', y') \in S_{X,Y}} f_e(x', y') > 0 \stackrel{\text{Def. 7}}{\Rightarrow} f_S(x, y) \geq 0. \quad \mathbf{SC:} \forall x \in \mathbb{F} : \sum_{y \in \mathbb{F}} f_S(x, y) \leq X(x). \quad \sum_{y \in \mathbb{F}} f_S(x, y) \stackrel{\otimes}{\leq} \sum_{y \in \mathbb{F}} f_e(x, y) \stackrel{\text{SC Cor.1}}{\leq}$$

$$X(x). \quad \mathbf{TC:} \forall x, y \in \mathbb{F} : f_S(x, y) \leq Y(y). \quad f_S(x, y) \stackrel{\otimes}{\leq} f_e(x, y) \stackrel{\text{TC Cor.1}}{\leq} Y(y).$$

$$\mathbf{FC:} \sum_{x \in \mathbb{F}} \sum_{y \in \mathbb{F}} f_S(x, y) = \min(m_X, m_Y). \quad \sum_{x \in \mathbb{F}} \sum_{y \in \mathbb{F}} f_S(x, y) = \sum_{(x,y) \in R_X \times R_Y} f_S(x, y) = \sum_{(x,y) \in S_{X,Y}} f_S(x, y) + f_S(\hat{x}, \hat{y}) \stackrel{\text{Def.7}}{=} \sum_{(x,y) \in S_{X,Y}} f_e(x, y) + \min(m_X, m_Y) - \sum_{(x', y') \in S_{X,Y}} f_e(x', y') = \min(m_X, m_Y). \quad \square$$

**Optimality Analysis.** To prove that IM-Sig\* with the proposed flow lower-bounds the EMD, we present that it yields the minimum overall cost among all possible flows by showing that any arbitrarily chosen feasible flow  $f$  results in a higher or equal overall cost than that for the IM-Sig\* flow.

**THEOREM 2 (OPTIMALITY OF IM-SIG\* FLOW).** Given signatures  $X, Y \in \mathbb{S}^+$  with  $m_X = \sum_{x \in \mathbb{F}} X(x)$ ,  $m_Y = \sum_{y \in \mathbb{F}} Y(y)$ , and set of all possible flows  $F$ ,  $f_S$  is the minimum-cost flow minimizing the overall cost with respect to IM-Sig\*:

$$f_S = \arg \min_{f \in F} \left\{ \sum_{x \in \mathbb{F}} \sum_{y \in \mathbb{F}} \frac{\delta(x, y)}{\min(m_X, m_Y)} f(x, y) \right\}.$$

**PROOF.** We show that any arbitrarily chosen flow  $f$  different from the proposed flow  $f_S$  does not lead to lower overall cost. Due to space limitations, we present the idea of the proof instead of giving all the theoretical details. Since  $f$  and  $f_S$  are feasible, the total amount of earth moved is  $\min(m_X, m_Y)$ . Let  $R_X = E \cup L \cup M$ , where  $E, L, M$  include features from which  $f_S$  transfers an equal, smaller, or greater amount of earth than  $f$ , respectively:

$$E := \{x \in R_X \mid \sum_{y \in R_Y} f_S(x, y) = \sum_{y \in R_Y} f(x, y)\}$$

$$L := \{x \in R_X \mid \sum_{y \in R_Y} f_S(x, y) < \sum_{y \in R_Y} f(x, y)\}$$

$$M := \{x \in R_X \mid \sum_{y \in R_Y} f_S(x, y) > \sum_{y \in R_Y} f(x, y)\}.$$

For any  $x \in R_X$  and any amount of earth  $m \leq X(x)$ , the minimum-cost local earth distribution regarding the target constraint is attained by transferring earth from  $x$  to its nearest neighbors  $y_1, \dots, y_l$  in  $R_Y$  where it holds  $1 \leq i \leq j \leq l \Rightarrow \delta(x, y_i) \leq \delta(x, y_j)$ , which can also be inferred from [21]. We refer this fact via the notation  $\star$  below, where required. Now, we consider 3 cases:

**Case 1:** For any  $x \in E$ , the amount of earth transferred by  $f_S$  is the same as that of  $f$ . By Def.7,  $f_S$  transfers earth from any  $x$  to its nearest neighbors in  $R_Y$ , and by  $(\star)$  it is guaranteed that it yields the lowest cost regarding any  $x \in E$ . Thus, we can conclude:

$$\sum_{x \in E} \sum_{y \in R_Y} \delta(x, y) \cdot f_S(x, y) \leq \sum_{x \in E} \sum_{y \in R_Y} \delta(x, y) \cdot f(x, y).$$

**Case 2:** For any  $x \in M$ , the total amount of earth  $f_S$  transfers from  $x$  exceeds that of  $f$ . We partition the amount of earth  $X(x)$  belonging to the feature  $x$  into two parts:  $m_{\bar{M}}(x)$  is the amount of earth which each flow transfers to the features in  $R_Y$ . The remaining  $X(x) - m_{\bar{M}}(x)$  amount of earth is then transferred only by  $f_S$  so that it totally transfers more earth than  $f$ . Regarding  $m_{\bar{M}}(x)$ , by  $(\star)$   $f_S$  attains the minimum cost by filling up its nearest-neighbors in  $R_Y$  (consideration of local distance order of  $y \in R_Y$ ). By Def.7,  $f_S$  only transfers earth regarding the pairs  $(x, y)$  with  $\delta(x, y) \leq \delta(\hat{x}, \hat{y})$  (consideration of the global distance order of  $(x, y) \in R_X \times R_Y$ ).

**Case 3:** For any  $x \in L$ , the total amount of earth  $f_S$  transfers from  $x$  is smaller than that of  $f$ . We partition the amount of earth  $X(x)$  belonging to the feature  $x$  into two parts:  $m_{\bar{L}}(x)$  is the amount of earth which each flow transfers to the features in  $R_Y$ . The remaining  $X(x) - m_{\bar{L}}(x)$  amount of earth is then transferred by only  $f$  so that it totally transfers more earth than  $f_S$ . Regarding  $m_{\bar{L}}(x)$ , by  $(\star)$   $f_S$  attains the minimum cost by filling up its nearest-neighbors in  $R_Y$  (consideration of local distance order of  $y \in R_Y$ ). We know  $f_S$  can only transfer earth regarding pairs  $(x, y)$  with  $\delta(x, y) \leq \delta(\hat{x}, \hat{y})$ , and it does not transfer all the earth from  $x$ . In addition, it fills up the features  $y \in R_Y$  regarding all pairs  $(x, y)$  with  $x \in L$  and  $\delta(x, y) < \delta(\hat{x}, \hat{y})$ . In best case,  $f$  distributes  $m_{\bar{L}}(x)$  amount of earth as  $f_S$  distributes, or  $f$  distributes it in another way. In the latter case, the last pair  $(x', y')$  used for the distribution by  $f$  satisfies  $\delta(\hat{x}, \hat{y}) \leq \delta(x', y')$ . Thus, the only place where  $f$  transfers the remaining  $X(x) - m_{\bar{L}}(x)$  amount of earth involves only the pairs  $(x, y)$  satisfying  $\delta(\hat{x}, \hat{y}) \leq \delta(x', y') \leq \delta(x, y)$  (consideration of the global distance order again).

By the facts elucidated in Case 2 and 3, and the constraint **FC**, it is concluded:  $\sum_{x \in M} \sum_{y \in R_Y} \delta(x, y) \cdot (f_S(x, y) - f(x, y)) \leq$

$$\sum_{x \in L} \sum_{y \in R_Y} \delta(x, y) \cdot (f(x, y) - f_S(x, y)).$$

As a result, after considering all the facts, we attain the final statement as follows:

$$\sum_{x \in R_X} \sum_{y \in R_Y} \delta(x, y) \cdot f_S(x, y) \leq \sum_{x \in R_X} \sum_{y \in R_Y} \delta(x, y) \cdot f(x, y),$$

which indicates that the overall cost induced by any feasible flow  $f$  does not lead to lower overall cost than that of  $f_S$ . In other words, the proposed flow  $f_S$  is proven to be the minimum-cost flow.  $\square$

As mentioned above, Theorem 2 states that IM-Sig\* flow is an optimal flow by showing that there exists no other flow

resulting in lower overall cost than that for IM-Sig\* flow.

**Lower-bounding the EMD with IM-Sig\*.** After presenting that IM-Sig\* flow is feasible and optimal, we below show the third significant theoretical result (Theorem 3) from which we deduce that the utilization of IM-Sig\* flow leads to the lower bound to the EMD on signatures, regardless of their total weights.

**THEOREM 3 (LOWER-BOUNDING EMD).** *Given any two signatures  $X, Y \in \mathbb{S}^+$  with total weights  $m_X, m_Y$ , it holds:*

$$IM-Sig^*(X, Y) \leq EMD(X, Y).$$

**PROOF.** By Theorem 1 and 2,  $f_S$  (Def. 7) is a feasible and minimum-cost flow regarding constraints of IM-Sig\*. Hence, there exists no other flow leading to smaller overall cost.  $\square$

Consequently, the theoretical results provide confirmatory evidence that the proposed IM-Sig\* flow can be utilized as a filter distance function lower-bounding the EMD on signatures, including both absolute and relative signatures. To this end, we can present a computational algorithm to compute IM-Sig\* between any signatures, given as below.

## 5.4 Computational Algorithm

---

**Algorithm 1:** IM-Sig\* computation

---

**input** : signatures  $X, Y$ , ground distance  $\delta$

**output**: IM-Sig\* between signatures  $X$  and  $Y$

---

```

1 cost = 0
2 construct minHeap for  $R_X \times R_Y$  regarding  $\delta$ 
3 minWeight =  $\min(m_X, m_Y)$ 
4 initialize sourceCap(x) =  $X(x)$  for each  $x \in R_X$ 
5 remainingEarth = minWeight
6 while remainingEarth > 0 do
7    $(x, y) = \text{minHeap.poll}()$ 
8   if sourceCap(x) > 0 then
9     if  $Y(y) \geq \text{remainingEarth}$  then
10      | earth =  $\min\{\text{remainingEarth}, \text{sourceCap}(x)\}$ 
11      else
12      | earth =  $\min\{\text{sourceCap}(x), Y(y)\}$ 
13      end
14      cost = cost +  $\delta(x, y) \cdot \text{earth}$ 
15      sourceCap(x) = sourceCap(x) - earth
16      remainingEarth = remainingEarth - earth
17   end
18 end
19 return cost/minWeight
```

---

**Algorithm.** The pseudo code of IM-Sig\* computation for both absolute and relative signatures with our proposed flow construction is depicted in Algorithm 1. After a min-heap is constructed over  $R_X \times R_Y$  with respect to distance values in ascending order (line 2), the algorithm extracts  $(x, y)$  with the smallest distance from the min-heap (line 7), until earth in an amount of the minimum weight of both signatures is transferred to  $Y$  totally (line 6). Each extracted pair from the min-heap refers to an element in the global feasible set  $S_{X,Y}$ , and the amount of earth transferred is determined by taking the remaining earth, current source capacity of  $x$ , and target capacity of  $y$  into consideration (lines 8-17).

**Complexity Analysis.** Assuming  $n = |R_X|, m = |R_Y|$ , the min-heap construction is performed in computation time



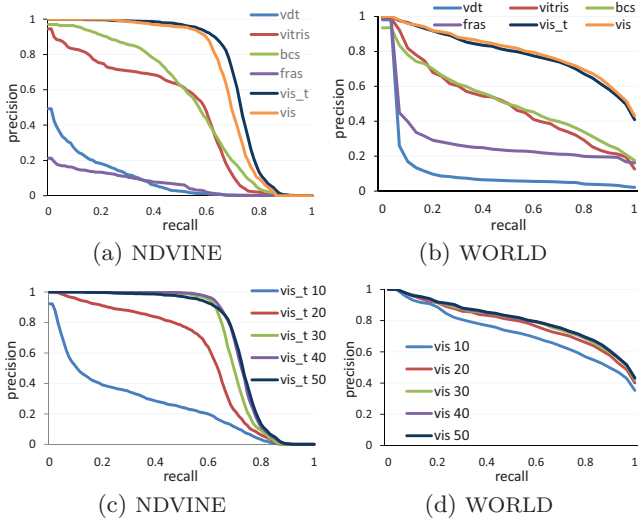


Figure 7: Precision-recall graphs.

complexity  $O(n \cdot m)$ . For each pair extraction from the min-heap, in worst case  $\log(n \cdot m)$  many steps are required to ensure the heap property again.

Note that for any given signatures  $X, Y$ , it holds that  $|S_{X,Y}| < |R_X \times R_Y|$ , i.e. the global feasible set  $S_{X,Y}$  which is the greatest set whose total flow may not reach the minimum of the total weights of both signatures, and hence, its cardinality is smaller than the number of all possible pairs of representatives in the signatures  $X$  and  $Y$ . Since the number of extractions from the min-heap is contingent on  $\min(m_X, m_Y)$  and is bounded by  $|S_{X,Y}| < |R_X \times R_Y|$ , only  $t$  many pair extractions are required with  $t < n \cdot m$ . Thus, the filter distance computation can be carried out in time complexity  $O(t \cdot \log(n \cdot m))$  with the proposed IM-Sig\* flow.

## 6. EXPERIMENTAL EVALUATION

**Experimental Setup.** Results presented in this section expose averages over a query workload of 50 where queries are randomly chosen. All methods are implemented in JAVA and evaluated on a single-core 2.3 GHz machine with Windows Server 2008 and 10 GB of main memory, without parallelization. While we utilize Manhattan distance as ground distance, our approach can, nevertheless, be combined with any ground distance function.

We use three real world datasets. First, we take the data in [14] of 3636 videos and generate approximately 100 near-duplicate copies for each video by altering brightness, contrast, playback speed, resolution, frame order, adding overlay text, borders, and modifying content by frame deletion, yielding a database (NDVINE) of 350,000 videos with 3636 ground truth categories. Second, we use WORLD consisting of 1020 videos we downloaded from *vine.co* (Vine) and *youtube.com*, sorted manually into 34 categories (such as soccer, beach, and forest) of videos which are determined as visually similar according to human perception. For effectiveness experiments, we use the aforementioned databases incorporating video category information. Third, we use the dataset from [23] including 250,000 public social videos of Vine, which we refer as PUBVID. We conduct efficiency experiments with the latter and NDVINE to attain appropri-

Table 2: Single distance computation time (ms)

Dim.	Distance					
	Fdbq	Fqdb	Fmax	Pemd	Rubner	EMD
4	0.0018	0.0023	0.0037	0.0068	0.0005	0.0266
8	0.0051	0.0052	0.0102	0.0070	0.0004	0.0331
16	0.0224	0.0229	0.0452	0.0137	0.0006	0.0858
32	0.1021	0.1035	0.2053	0.0287	0.0012	0.4784
64	0.4740	0.4801	0.9535	0.0651	0.0022	3.9208

ate evaluation regarding data cardinality. We generate video signatures of different dimensionalities as described in Section 3 with position, color, contrast, coarseness and temporal information, while for effectiveness experiments we generate 2 different types of video signatures to recognize the contribution of the temporal information to results: *vis\_t* and *vis* denote our novel video signature model with and without temporal information, respectively. In addition, since for any signatures  $X, Y$ ,  $\text{IM-Sig}^*(X, Y) \leq \max(\text{IM-Sig}^*(X, Y), \text{IM-Sig}^*(Y, X)) \leq \text{EMD}(X, Y)$  holds, we implement 3 variations of the algorithm from [17] to evaluate efficiency of query processing, and let *Fqdb*, *Fdbq*, *Fmax* refer to filter distance functions with our proposed flow computation where earth is transferred from query signatures to database signatures, from database signatures to query signatures, and where the maximum of both filter distances is utilized in multistep algorithm, respectively. Furthermore, in efficiency experiments we set  $k=100$  for  $k$ -nearest-neighbor query processing. The databases used here are available upon request.

**Effectiveness Experiments.** Figure 7 shows precision-recall graphs of our novel video signature models (*vis\_t*, *vis*), in comparison to four state-of-the-art methods: video triplets (*vitri*) [18], frame-sequence symbolization (*fras*) [27], bounded coordinate systems (*bcs*) [7], and video distance trajectories (*vdt*) [8]. Results summarized in Figure 7(a)-(b) provide confirmatory evidence that our model outperforms the state of the art for both near-duplicate detection (NDVINE) and visual similarity search (WORLD). Considering temporal dimension in the signature model yields better precision for NDVINE, since videos in the same category exhibit a similar temporal ordering. As illustrated in Figure 7(c)-(d), the precision of our models increases with higher signature dimensionality (10-50), as we expected.

**Efficiency Experiments.** It is noteworthy to remind again that our approach computes the same filter distance as for IM-Sig on relative signatures, and it corresponds to *Fdbq* in the experimental results presented in this section. First, we evaluate the influence of signature dimensionality on processing time of single distance computation on PUBVID dataset (Table 2). Note that *Pemd* and *Rubner* refer to the existing methods of projected EMD filter [4] and Rubner filter [15]. Since EMD can be computed in super-cubic time in signature dimensionality, it exhibits the highest values, while *Fmax*' performance is almost 2 times slower than *Fqdb* and *Fdbq*, corresponding to our expectation. Rubner shows the lowest time cost by only computing the ground distance among average signatures.

Figure 8 presents efficiency results of the state of the art (*Rubner*, *Pemd*) and our methods (*Fqdb*, *Fdbq*, *Fmax*). With increasing data cardinality, Rubner exhibits the highest time cost, directly followed by *Pemd*, which are substantially outperformed by our methods regarding selectivity and overall query time. In particular, *Fmax* computes 49.9

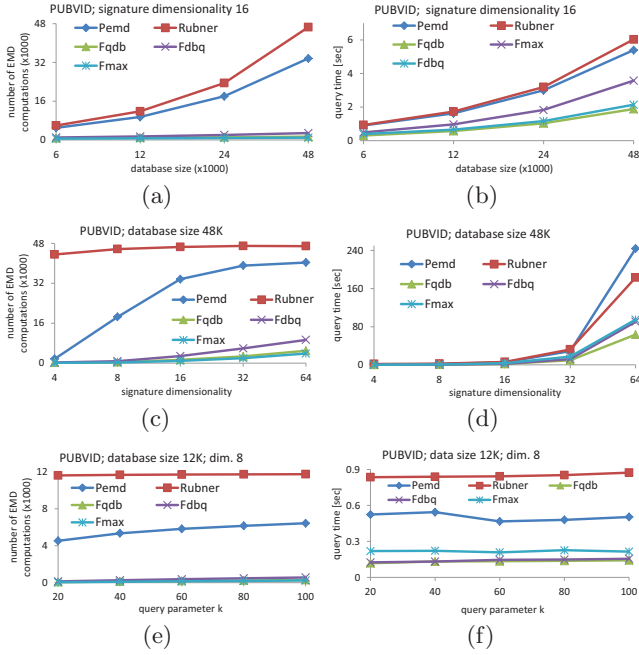


Figure 8: Selectivity and efficiency results with the state of the art.

and 36.1 times less EMD computations than Rubner and Pemd at data cardinality 48K, respectively. Another result matching our expectation is the constant behavior of Rubner with the worst selectivity, irrespective of dimensionality, while efficiency deterioration of Pemd is remarkable at a very high rate with increasing dimensionality, where, again, our methods outperform both existing approaches. The reason behind these observations is that Rubner simply utilizes distance between average signatures, and Pemd considers single EMD computations performed on each projected dimension, neglecting flow approximation, as given in Section 2. In contrast, our methods explicitly approximate the original EMD flow at a global level by tracking source and target capacity of representatives, and ensuring constraints given in Def. 3, attaining very high efficiency improvement. Furthermore, experiments analyzing the effect of query parameter  $k$  for  $k$ -nn query processing point out higher efficiency improvement of our proposals. Moreover, we conduct experiments to investigate the applicability to absolute signatures by using video subclip query signatures with varying total weights (0.1-1.0), while ensuring that the weight of any database video signature remains as 1 (Figure 9). Recall that Rubner is not a lower bound here, as illustrated in Section 4, and we observe considerably high selectivity difference between Pemd and our methods, confirming our methods' successful application on absolute signatures. In particular, Pemd refines 99% of all videos between query weights 0.1-0.4, while Fmax refines only at most 0.1%, performing 432 times less EMD computations than Pemd, attaining a selectivity improvement by two orders of magnitude. Note that query time results regarding Figure 9 are omitted, since they expose very similar behavior as those for the number of EMD computations.

After observing that our methods outperform the state of the art, we below perform extensive experiments for our

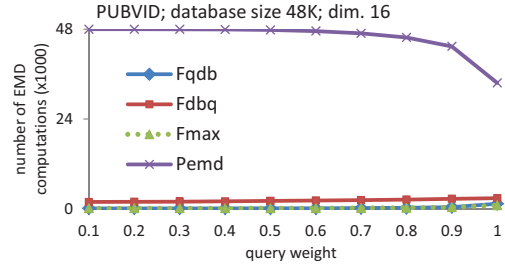


Figure 9: Results with the state of the art regarding selectivity vs. individual total weight of query signatures (absolute signatures).

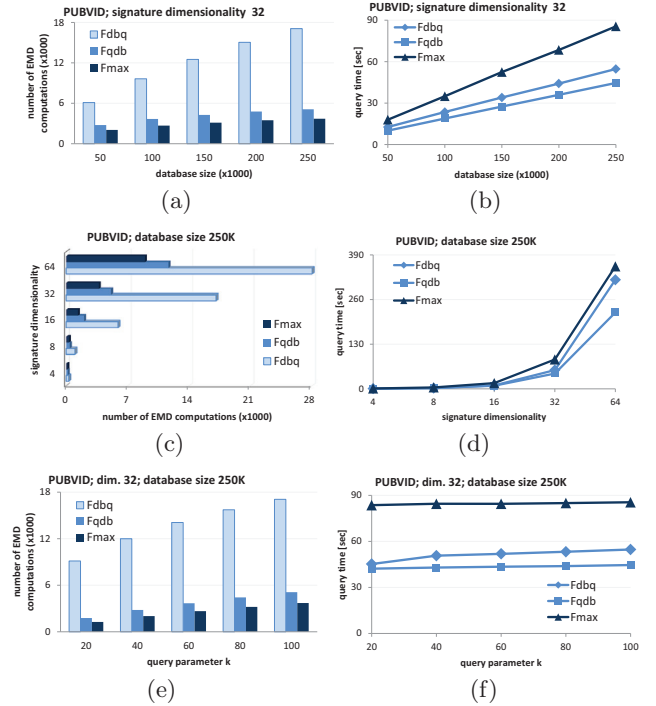


Figure 10: Results regarding selectivity and efficiency on PUBVID database.

methods, by varying signature dimensionality, database cardinality, and query parameter  $k$  for  $k$ -nn query processing.

Figure 10 summarizes the efficiency improvement achieved on PUBVID: Fmax indicates the best selectivity, in comparison to other proposed variations, exhibiting the lowest number of EMD computations with increasing data cardinality (50K-250K), and signature dimensionality (4-64), resulting from the computation of a tighter lower bound in the ranking phase of the multistep filter-and-refine algorithm [17]. Since Fmax shows higher filter time, its overall time cost is higher than that for the other two methods. Fqdb shows similar selectivity results, but its lower filter time cost than that for Fmax leads to the fact that Fqdb achieves the highest efficiency improvement regarding database size, dimensionality, and query parameter.

As depicted in Figure 11, evaluation results on NDVINE first indicate an almost constant selectivity behavior for Fqdb and Fmax regarding increasing data size, when compared to Fdbq. This can be elucidated by the intrinsic essence of

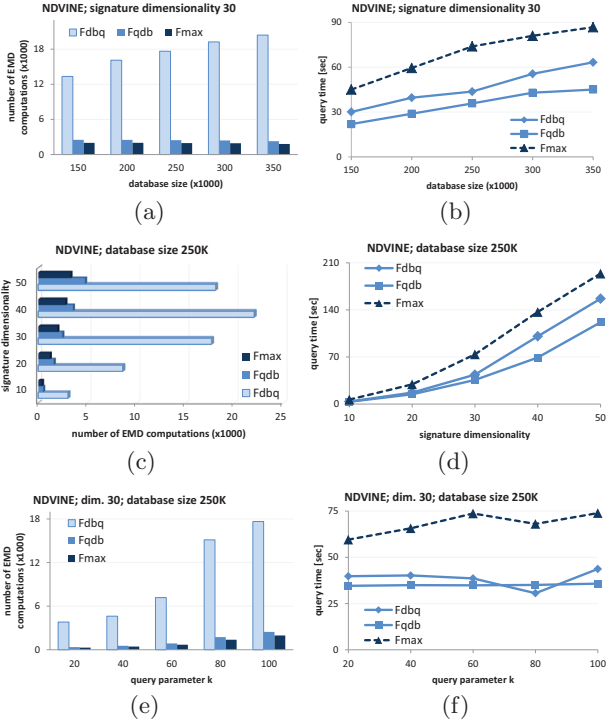


Figure 11: Results regarding selectivity and efficiency on NDVINE database.

this database: With increasing data size near-duplicates of videos emerging via various editing tasks can have similar distances to each other, which affects the filter step. Hence, the higher the data cardinality of near-duplicate video databases, it is more worth using Fqdb to attain an almost constant, low time cost, yielding a considerable advantage. Second, with increasing signature size at a constant data cardinality (250K), the number of promising objects passing Fdbq decreases after dimensionality 40, while Fmax and Fqdb show lower time cost, in particular 1.3 % of selectivity at dimensionality 50 for Fmax. Third, we observe that for all query parameters  $k$  (20-100), Fqdb and Fdbq show almost constant behavior for query time due to their lower filter time, matching our expectation.

For both databases, interestingly, Fdbq results in a higher number of EMD computations than for Fqdb. To expound this result, we perform experiments on another real world dataset which we omit due to space limitations, and recognize that selectivity results of Fdbq and Fqdb do not necessarily differ from each other at a high rate. Hence, the observed difference in selectivity and query time can be attributed to the feature distributions of these databases and the utilized queries, which cause Fqdb flow to be more similar to the EMD flow than that for Fdbq. A future research direction involves in further investigating this issue in detail.

Figure 12(a)-12(b) exhibit the effect of individual query signature weights on the number of EMD computations by fixing the total weight of any database video signature to 1. In particular for Fqdb, the number of EMD computations decreases with decreasing query weight, expounded by the fact that the smaller the query weight, the closer Fqdb approximates the EMD flow. To understand it in more detail, we analyze relative approximation error of Fqdb and Fdbq

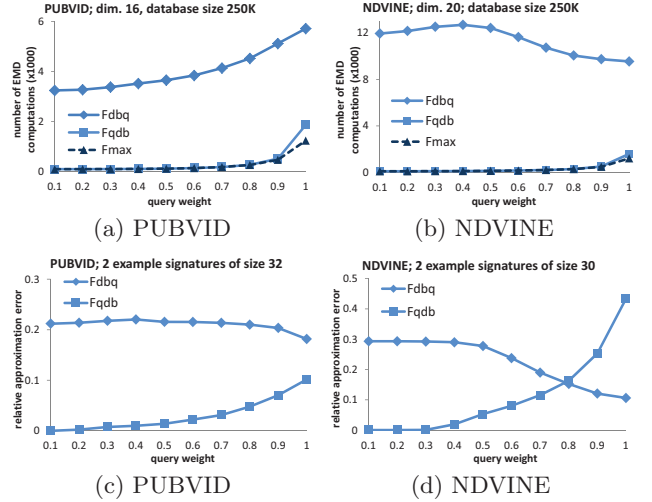


Figure 12: (a)-(b): Selectivity vs. individual total weight of query signatures. (c)-(d): Relative approximation error vs. individual total weight of query signatures for example signature pairs.

on example pairs of video signatures, summarized in Figure 12(c)-12(d). For smaller query weights, EMD distributes the weight of query representatives more locally among representatives of the database video, since capacities of target representatives are far greater than those for query representatives. Accordingly, since Fqdb distributes earth optimally for each query representative, its flow is similar to the EMD flow, allowing for a better approximation of the EMD than for Fdbq, which meets our expectation. Note that the flow approximation error increases especially after the query weight of 0.5 for both variations. Analogously, the higher the query weight, the better Fdbq approximates the EMD, as a higher query weight causes the EMD to distribute the weight of the database representatives more locally among those of query, resulting in a similar flow to that for Fdbq.

Figure 13 summarizes absolute filter and refinement distances for 10- $nn$  queries on an example from PUBVID at a data cardinality of 1000 and dimensionality 16. Unlike comprehensive overall results for PUBVID, we observe that Fqdb leads to a higher number of refinements (75 in ranking according to filter distance), while Fdbq and Fmax perform 50 and 43 EMD computations, respectively. A significant result gathered from these figures is Fmax leads to the lowest number of exact distance computations, irrespective of the fact how well the other variants approximate the EMD flow. All query time cost results depicted in aforementioned figures point out the advantage of Fqdb with respect to dimensionality, data cardinality, and query parameter  $k$ .

## 7. CONCLUSION

In this paper, we presented how efficient and effective query processing can be performed on high dimensional video databases. We introduced a new compact video representation model, and proposed to alleviate computational time complexity of the Earth Mover’s Distance (EMD) by a novel filter approximation guaranteeing completeness (no false dismissals). Furthermore, we presented both an extensive theoretical analysis of our techniques and a computational al-

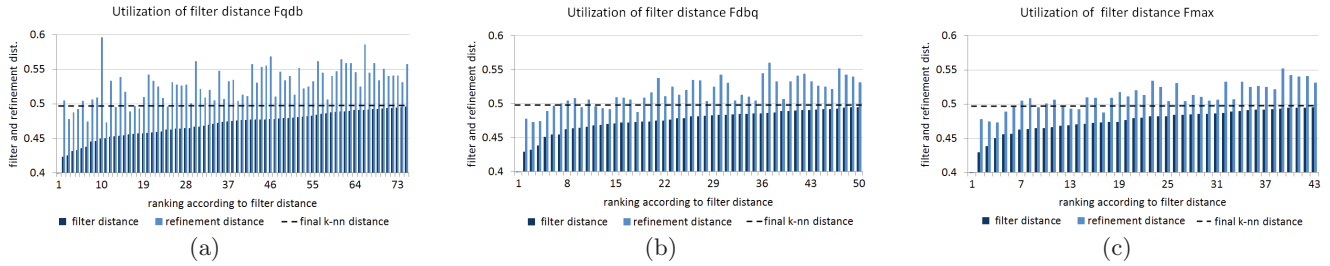


Figure 13: Filter and refinement distance values on an example from PUBVID dataset for 10-nn queries.

gorithm. Moreover, proposed techniques expose two vital advantages: First, they are applicable to both relative and absolute signatures exhibiting uniform and individual total weights, respectively, yielding high flexibility with respect to query processing and explicit user-driven tasks, such as sub-clip video detection. Second, they exhibit a comprehensive solution which is not restricted to the video domain, and, hence, can be applied to other domains, such as biotechnology and biomedicine. Extensive experimental evaluation on real world data indicates high efficiency, significantly reducing the number of EMD computations and outperforming the state of the art by up to two orders of magnitude regarding selectivity and query time. As future work, we plan to integrate our filter approximation in relational databases.

## 8. ACKNOWLEDGMENTS

This work is funded by DFG grant SE 1039/7-1.

## 9. REFERENCES

- [1] I. Assent, A. Wenning, and T. Seidl. Approximation techniques for indexing the earth mover’s distance in multimedia databases. In *ICDE*, page 11, 2006.
- [2] R. S. Chavez and T. F. Heatherton. Representational similarity of social and valence information in the medial pfc. *J. Cogn. Neuroscience*, 27(1):73–82, 2015.
- [3] R. Cheng, L. Chen, J. Chen, and X. Xie. Evaluating probability threshold k-nearest-neighbor queries over uncertain data. In *EDBT*, pages 672–683, 2009.
- [4] S. D. Cohen and L. J. Guibas. The earth mover’s distance: Lower bounds and invariance under translation. Technical report, Stanford Univ., 1997.
- [5] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. *SIGMOD*, 23(2):419–429, May 1994.
- [6] A. Hinneburg et al. Database support for 3d-protein data set analysis. In *SSDBM*, pages 161–170, 2003.
- [7] Z. Huang, H. T. Shen, J. Shao, X. Zhou, and B. Cui. Bounded coordinate system indexing for real-time video clip search. *Trans.I.S.*, 27(3):17:1–33, 2009.
- [8] Z. Huang, L. Wang et al. Online near-duplicate video clip detection and retrieval: An accurate and fast system. In *ICDE*, pages 1511–1514, 2009.
- [9] K. Kantarci. Molecular imaging of Alzheimer disease pathology. *AJNR*, 35:12–17, 2014.
- [10] M. Katajamaa and M. Oresic. Data processing for mass spectrometry-based metabolomics. *Journal of Chromatography A*, 1158(1–2):318–328, 2007.
- [11] C.-R. Kim and C.-W. Chung. A multi-step approach for partial similarity search in large image data using histogram intersection. *Inf. S.T.*, 45(4):203–215, 2003.
- [12] H.-P. Kriegel, P. Kröger, P. Kunath, and M. Renz. Generalizing the optimality of multi-step k-nearest neighbor query processing. In *SSTD*, p.75-92, 2007.
- [13] O. Pele and M. Werman. A linear time histogram metric for improved sift matching. In *ECCV*, pages 495–508, 2008.
- [14] M. Redi, N. OHare, R. Schifanella, M. Trevisiol, and A. Jaimes. 6 seconds of sound and vision: Creativity in micro-videos. In *CVPR*, pages 4272–4279, 2014.
- [15] Y. Rubner, C. Tomasi, and L. Guibas. A metric for distributions with applications to image databases. In *ICCV98*, pages 59–66, 1998.
- [16] B. E. Ruttenberg and A. K. Singh. Indexing the earth mover’s distance using normal distributions. *PVLDB*, 5(3):205–216, 2011.
- [17] T. Seidl and H. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, pages 154–165, 1998.
- [18] H. T. Shen, B. C. Ooi, and X. Zhou. Towards effective indexing for very large video sequence database. In *SIGMOD*, pages 730–741, 2005.
- [19] J. Strötgen, M. Gertz, and C. Junghans. An event-centric model for multilingual document similarity. In *ACM SIGIR*, pages 953–962, 2011.
- [20] D. Uskat, T. Emrich et al. Similarity search in fuzzy object databases. In *SSDBM*, pages 32:1–32:6, 2015.
- [21] M. S. Uysal, C. Beecks, J. Schmücking, and T. Seidl. Efficient filter approximation using the Earth Mover’s Distance in very large multimedia databases with feature signatures. In *CIKM*, pages 979–988, 2014.
- [22] M. S. Uysal, C. Beecks, J. Schmücking, and T. Seidl. Efficient Similarity Search in Scientific Databases with Feature Signatures. In *SSDBM*, p. 30:1–30:12, 2015.
- [23] B. Vandersmissen et al. The rise of mobile and social short-form video: an in-depth measurement study of vine. In *SoMus*, v. 1198, pages 1–10, 2014.
- [24] M. Wichterich et al. Efficient emd-based similarity search in multimedia databases via flexible dimensionality reduction. In *SIGMOD*, pages 199–212, 2008.
- [25] J. Xu, Z. Zhang et al. Efficient and effective similarity search over probabilistic data based on earth mover’s distance. *PVLDB*, 3(1):758–769, 2010.
- [26] YouTube. Statistics. Retrieved Sep. 1, 2015 from <https://www.youtube.com/yt/press/statistics.html>.
- [27] X. Zhou, X. Zhou, and H. T. Shen. Efficient similarity search by summarization in large video database. In *ADC*, pages 161–167, 2007.

# Probabilistic Threshold Indexing for Uncertain Strings

Sharma Thankachan  
Georgia Institute of  
Technology  
Georgia, USA  
thanks@csc.lsu.edu

Manish Patil  
Louisiana State University  
Louisiana, USA  
manish.m.patil@gmail.com

Rahul Shah  
Louisiana State University  
Louisiana, USA  
rahul@csc.lsu.edu

Sudip Biswas  
Louisiana State University  
Louisiana, USA  
sudipid@gmail.com

## ABSTRACT

Strings form a fundamental data type in computer systems. String searching has been extensively studied since the inception of computer science. Increasingly many applications have to deal with imprecise strings or strings with fuzzy information in them. String matching becomes a probabilistic event when a string contains uncertainty, i.e. each position of the string can have different probable characters with associated probability of occurrence for each character. Such uncertain strings are prevalent in various applications such as biological sequence data, event monitoring and automatic ECG annotations. We explore the problem of indexing uncertain strings to support efficient string searching. In this paper we consider two basic problems of string searching, namely substring searching and string listing. In substring searching, the task is to find the occurrences of a deterministic string in an uncertain string. We formulate the string listing problem for uncertain strings, where the objective is to output all the strings from a collection of strings, that contain probable occurrence of a deterministic query string. Indexing solution for both these problems are significantly more challenging for uncertain strings than for deterministic strings. Given a construction time probability value  $\tau$ , our indexes can be constructed in linear space and supports queries in near optimal time for arbitrary values of probability threshold parameter greater than  $\tau$ . To the best of our knowledge, this is the first indexing solution for searching in uncertain strings that achieves strong theoretical bound and supports arbitrary values of probability threshold parameter. We also propose an approximate substring search index that can answer substring search queries with an additive error in optimal time. We conduct experiments to evaluate the performance of our indexes.

## 1. INTRODUCTION

String indexing has been one of the key areas of computer science. Algorithms and data structures of string searching finds application in web searching, computational biology, natural language processing, cyber security, etc. The classical problem of string indexing is to preprocess a string such that query substring can be searched efficiently. Linear space data structures are known for this problem which can answer such queries in optimal  $O(m + occ)$  time, where  $m$  is the substring length and  $occ$  is the number of occurrences reported.

Growth of the internet, digital libraries, large genomic projects have contributed to enormous growth of data. As a consequence, noisy and uncertain data has become more prevalent. Uncertain data naturally arises in almost all applications due to unreliability of source, imprecise measurement, data loss, and artificial noise. For example sequence data in bioinformatics is often uncertain and probabilistic. Sensor networks and satellites inherently gather noisy information.

Existing research has focused mainly on the study of regular or deterministic string indexing. In this paper we explore the problem of indexing uncertain strings. We begin by describing the uncertain string model, possible world semantics and challenges of searching in uncertain strings.

Current literature models uncertain strings in two different ways: the string level model and the character level model. In string level model, we look at the probabilities and enumerate at whole string level, whereas character level model represents each position as a set of characters with associated probabilities. We focus on the character level model which arises more frequently in applications. Let  $S$  be an uncertain string of length  $n$ . Each character  $c$  at position  $i$  of  $S$  has an associated probability  $pr(c^i)$ . Probabilities at different positions may or may not contain correlation among them. Figure 1(a) shows an uncertain string  $S$  of length 5. Note that, the length of an uncertain string is the total number of positions in the string, which can be less than the total number of possible characters in the string. For example, in Figure 1(a), total number of characters in string  $s$  with nonzero probability is 9, but the total number of positions or string length is only 5.

"Possible world semantics" is a way to enumerate all the possible deterministic strings from an uncertain string. Based on possible world semantics, an uncertain string  $S$  of length  $n$  can generate a deterministic string  $w$  by choosing one possible character from each position and concatenating them in order. We call  $w$  as one of the possible world for  $S$ . Probability of occurrence of

$w = w_1 w_2 \dots w_n$  is the partial product  $pr(w_1^1) \times pr(w_2^2) \times \dots \times pr(w_n^n)$ . The number of possible worlds for  $S$  increases exponentially with  $n$ . Figure 1(b) illustrates all the possible worlds for the uncertain string  $S$  with their associated probability of occurrence.

A meaningful way of considering only a fraction of the possible worlds is based on a probability threshold value  $\tau$ . We consider a generated deterministic string  $w = w_1 w_2 \dots w_n$  as a valid occurrence with respect to  $\tau$ , only if it has probability of occurrence more than  $\tau$ . The probability threshold  $\tau$  effectively removes lower probability strings from consideration. Thus  $\tau$  plays an important role to avoid the exponential blowup of the number of generated deterministic strings under consideration.

Character	S[1]	S[2]	S[3]	S[4]	S[5]
a	.3	.6	0	.5	1
b	.4	0	0	0	0
c	0	.4	0	.5	0
d	.3	0	1	0	0

(a) Uncertain string  $S$

w	Prob(w)	w	Prob(w)	w	Prob(w)
w[1] aadaa	.09	w[5] badaa	.12	w[9] dadaa	.09
w[2] aaaca	.09	w[6] badca	.12	w[10] dadca	.09
w[3] acdaa	.06	w[7] badca	.08	w[11] dcdaa	.06
w[4] acdca	.06	w[8] badca	.08	w[12] dcdca	.06

(b) Possible worlds of  $S$

Figure 1: An uncertain string  $S$  of length 5 and its all possible worlds with probabilities.

Given an uncertain string  $S$  and a deterministic query substring  $p = p_1 \dots p_m$ , we say that  $p$  matched at position  $i$  of  $S$  with respect to threshold  $\tau$  if  $pr(p_1^i) \times \dots \times pr(p_m^{i+m-1}) \geq \tau$ . Note that,  $O(m + occ)$  is the theoretical lower bound for substring searching where  $m$  is the substring length and  $occ$  is the number of occurrence reported.

## 1.1 Formal problem definition

Our goal is to develop efficient indexing solution for searching in uncertain strings. In this paper, we discuss two basic uncertain string searching problems which are formally defined below.

**PROBLEM 1. Substring Searching:** Given an uncertain string  $S$  of length  $n$ , our task is to index  $S$  so that when a deterministic substring  $p$  and a probability threshold  $\tau$  come as a query, report all the starting positions of  $S$  where  $p$  is matched with probability of occurrence greater than  $\tau$ .

**PROBLEM 2. Uncertain String Listing:** Let  $\mathcal{D} = \{d_1, \dots, d_D\}$  be a collection of  $D$  uncertain strings of  $n$  positions in total. Our task is to index  $\mathcal{D}$  so that when a deterministic substring  $p$  and a probability threshold  $\tau$  come as a query, report all the strings  $d_j$  such that  $d_j$  contains atleast one occurrence of  $p$  with probability of occurrence greater than  $\tau$ .

Note that the string listing problem can be naively solved by running substring searching query in each of the uncertain string from the collection. However, this naive approach will take  $O(\sum_{d_i \in \mathcal{D}} \text{search time on } d_i)$  time which can be very inefficient if the actual number of documents containing the substring is small. Figure 2 illustrates an example for string listing. In this example, only the string  $d_1$  contains query substring "BF" with probability of occurrence greater than query threshold 0.1. Ideally, the query time should be proportionate to the actual number of documents reported as output. Uncertain strings considered in both these problems can contain correlation among string positions.

String collection  $\mathcal{D} = \{d_1, d_2, d_3\}$ :

$d_1[1]$	$d_1[2]$	$d_1[3]$	$d_2[1]$	$d_2[2]$	$d_2[3]$	$d_3[1]$	$d_3[2]$	$d_3[3]$
A.4	B.3	F.5	A.6	B.5	B.4	A.4	I.3	A.1
B.3	L.3	J.5	C.4	F.3	C.3	F.4	L.3	
F.3	F.3			J.2	E.2	P.2	P.3	
	J.1				F.1		T.3	

Output of string listing query (" $BF$ ", 0.1) on  $\mathcal{D}$  is:  $d_1$

Figure 2: String listing from an uncertain string collection  $\mathcal{D} = \{d_1, d_2, d_3\}$ .

## 1.2 Challenges in uncertain string searching

We summarize some challenges of searching in uncertain strings.

- An uncertain string of length  $n$  can have multiple characters at each position. As the length of an uncertain string increases, the number of possible worlds grows exponentially. This makes a naive technique that exhaustively enumerates all possible worlds infeasible.
- Since multiple substrings can be enumerated from the same starting position, care should be taken in substring searching to avoid possible duplication of reported positions.
- Enumerating all the possible sequences for arbitrary probability threshold  $\tau$  and indexing them requires massive space for large strings. Also note that, for a specific starting position in the string, the probability of occurrence of a substring can change arbitrarily (non-decreasing order) with increasing length, depending on the probability of the concatenated character. This makes it difficult to construct index that can support arbitrary probability threshold  $\tau$ .
- Correlated uncertainty among the string positions is not uncommon in applications. An index that handles correlation appeals to a wider range of applications. However, handling the correlation can be a bottleneck on space and time.

## 1.3 Related work

Although, searching over clean data has been widely researched, indexing uncertain data is relatively new. Below we briefly mention some of the previous works related to uncertain strings.

**Algorithmic approach:** Li et al. [19] tackled the substring searching problem where both the query substring and uncertain sequence comes as online query. They proposed a linear time and linear space dynamic programming approach to calculate the probability that a substring is contained in the uncertain string.

**Approximate substring matching:** Given as input a string  $p$ , a set of strings  $\{x_i | 1 \leq i \leq r\}$ , and an edit distance threshold  $k$ , the substring matching problem is to find all substrings  $s$  of  $x_i$  such that  $d(p, s) \leq k$ , where  $d(p, s)$  is the edit distance between  $p$  and  $s$ . This problem has been well studied on clean texts (see [22] for a survey). Most of the ideas to solve this problem is based on partitioning  $p$ . Tiangjian et al. [12] extended this problem for uncertain strings. Their index can handle strings of arbitrary lengths.

**Frequent itemset mining:** Some articles discuss the problem of frequent itemset mining in uncertain databases [6, 7, 3], where an itemset is called frequent if the probability of occurrence of the itemset is above a given threshold.

**Probabilistic database:** Several works [5, 26, 25] have developed indexing techniques for probabilistic databases, based on R-trees and inverted indices, for efficient execution of nearest neigh-

bor queries and probabilistic threshold queries. Dalvi et al. [8] proposed efficient evaluation method for arbitrary complex SQL queries in probabilistic database. Later on efficient index for ranked top- $k$  SQL query answering on a probabilistic database was proposed ([24, 18]). Kanagal et al. [16] developed efficient data structures and indexes for supporting inference and decision support queries over probabilistic databases containing correlation. They use a tree data structure named junction tree to represent the correlations in the probabilistic database over the tuple-existence or attribute-value random variables.

**Similarity joins:** A string similarity join finds all similar string pairs between two input string collections. Given two collections of uncertain strings  $R, S$ , and input  $(k, \tau)$ , the task is to find string pairs  $(r, s)$  between these collections such that  $Pr(ed(R, S) \leq k) > \tau$  i.e., probability of edit distance between  $R$  and  $S$  being at most  $k$  is more than probability threshold  $\tau$ . There are some works on string joins, e.g., [4, 13, 17], involving approximation, data cleaning, and noisy keyword search, which has been discussed in the probabilistic setting [15]. Patil et al. [23] introduced filtering techniques to give upper and (or) lower bound on  $Pr(ed(R, S) \leq k)$  and incorporate such techniques into an indexing scheme with reduced filtering overhead.

## 1.4 Our approach

Since uncertain string indexing is more complex than deterministic string indexing, a general solution for substring searching is challenging. However efficiency can be achieved by tailoring the data structure based on some key parameters, and use the data structure best suited for the purposed application. We consider the following parameters for our index design.

**Threshold parameter  $\tau_{min}$ :** The task of substring matching in uncertain string is to find all the probable occurrences, where the probable occurrence is determined by a query threshold parameter  $\tau$ . Although  $\tau$  can have any value between 0 to 1 at query time, real life applications usually prohibits arbitrary small value of  $\tau$ . For example, a monitoring system does not consider a sequence of events as a real threat if the associated probability is too low. We consider a threshold parameter  $\tau_{min}$ , which is a constant known at construction time, such that query  $\tau$  does not fall below  $\tau_{min}$ . Our index can be tailored based on  $\tau_{min}$  at construction time to suit specific application needs.

**Query substring length:** The query substring searched in the uncertain string can be of arbitrary length ranging from 1 to  $n$ . However, most often the query substrings are smaller than the indexed string. An example is a sensor system, collecting and indexing big amount of data to facilitate searching for interesting query patterns, which are smaller compared to the data stream. We show that more efficient indexing solution can be achieved based on query substring length.

**Correlation among string positions:** Probabilities at different positions in the uncertain string can possibly contain correlation among them. In this paper we consider character level uncertainly model, where a probability of occurrence of a character at a position can be correlated with occurrence of a character at a different position. We formally define the correlation model and show how correlation is handled in our indexes.

Our approach involves the use of suffix trees, suffix arrays and range maximum query data structure, which to the best of our knowledge, is the first use for uncertain string indexing. Succinct and compressed versions of these data structures are well known to have

good practical performance. Previous efforts to index uncertain strings mostly involved dynamic programming and lacked theoretical bound on query time. We also formulate the uncertain string listing problem. Practical motivation for this problem is given in Section 6. As mentioned before, for a specific starting position of an uncertain string, the probability of occurrence of a substring can change arbitrarily with increasing length depending on the probability of the concatenated character. We propose an approximate solution by discretizing the arbitrary probability changes with conjunction of a linking structure in the suffix tree.

## 1.5 Our contribution:

In this paper, we propose indexing solutions for substring searching in a single uncertain string, searching in a uncertain string collection, and approximate index for searching in uncertain strings. More specifically, we make the following contributions:

1. For the substring searching problem, we propose a linear space solution for indexing a given uncertain string  $S$  of length  $n$ , such that all the occurrences of a deterministic query string  $p$  with probability of occurrence greater than a query threshold  $\tau$  can be reported. We show that for frequent cases our index achieves optimal query time proportional to the substring length and output size. Our index can be designed to support arbitrary probability threshold  $\tau \geq \tau_{min}$ , where  $\tau_{min}$  is a constant known at index construction time.
2. For the uncertain string listing problem, given a collection of uncertain strings  $\mathcal{D} = \{d_1, \dots, d_D\}$  of total size  $n$ , we propose a linear space and near optimal time index for retrieving all uncertain strings that contain a deterministic query string  $p$  with probability of occurrence greater than a query threshold  $\tau$ . Our index supports queries for arbitrary  $\tau \geq \tau_{min}$ , where  $\tau_{min}$  is a constant known at construction time.
3. We propose an index for approximate substring searching, which can answer substring searching queries in uncertain strings for arbitrary  $\tau \geq \tau_{min}$  in optimal  $O(m + occ)$  time, where  $\tau_{min}$  is a constant known at construction time and  $\epsilon$  is the bound on desired additive error in the probability of a matched string, i.e. outputs can have probability of occurrence  $\geq \tau - \epsilon$ .

## 1.6 Outline

The rest of the paper is organized as follows. In section 2 we show some practical motivations for our indexes. In section 3 we give a formal definition of the problem, discuss some definitions related to uncertain strings and supporting tools used in our index. In section 4 we build a linear space index for answering a special form of uncertain strings where each position of the string has only one probabilistic character. In section 5 we introduce a linear space index to answer substring matching queries in general uncertain strings for variable threshold. Section 6 discusses searching in an uncertain string collection. In section 7, we discuss approximate string matching in uncertain strings. In section 8, we show the experimental evaluation of our indexes. Finally in section 9, we conclude the paper with a summary and future work direction.

## 2. MOTIVATION

Various domains such as bioinformatics, knowledge discovery for moving object database trajectories, web log analysis, text mining, sensor networks, data integration and activity recognition generates large amount of uncertain data. Below we show some practical motivation for our indexes.

**Biological sequence data:** Sequence data in bioinformatics is often uncertain and probabilistic. For instance, reads in shotgun sequencing are annotated with quality scores for each base. These quality scores can be understood as how certain a sequencing machine is about a base. Probabilities over long strings are also used to represent the distribution of SNPs or InDels (insertions and deletions) in the population of a species. Uncertainty can arise due to a number of factors in the high-throughput sequencing technologies. NC-IUB committee standardized incompletely specified bases in DNA to address this common presence of uncertainty [20]. Analyzing these uncertain sequences is important and more complicated than the traditional string matching problem.

We show an example uncertain string generated by aligning genomic sequence of Tree of At4g15440 from OrthologID and deterministic substring searching in the sequence. Figure 3 illustrates the example.

S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]	S[9]	S[10]	S[11]
P.1	S.7 F.3	F.1	P.1	Q.5 T.5	P.1	A.4 F.4 P.2	I.3 L.3 P.3 T.3	A.1	S.5 T.5	A.1

Figure 3: Example of an uncertain string  $S$  generated by aligning genomic sequence of the tree of At4g15440 from OrthologID.

Consider the uncertain string  $S$  of Figure 3. A sample query can be  $\{p = "AT", \tau = 0.4\}$ , which asks to find all the occurrences of string  $AT$  in  $S$  having probability of occurrence more than  $\tau = .4$ .  $AT$  can be matched starting at position 7 and starting at position 9. Probability of occurrence for starting position 7 is  $0.4 \times 0.3 = 0.12$  and for starting position 9 is  $1 \times 0.5 = 0.5$ . Thus position 9 should be reported to answer this query.

**Automatic ECG annotations:** In the Holter monitor application, for example, sensors attached to heart-disease patients send out ECG signals continuously to a computer through a wireless network ([9]). For each heartbeat, the annotation software gives a symbol such as N (Normal beat), L (Left bundle branch block beat), and R, etc. However, quite often, the ECG signal of each beat may have ambiguity, and a probability distribution on a few possibilities can be given. A doctor might be interested in locating a pattern such as  $\hat{A}IJNNAV\hat{A}$  indicating two normal beats followed by an atrial premature beat and then a premature ventricular contraction, in order to verify a specific diagnosis. The ECG signal sequence forms an uncertain string, which can be indexed to facilitate deterministic substrings searching.

**Event monitoring:** Substring matching over event streams is important in paradigm where continuously arriving events are matched. For example a RFID-based security monitoring system produces stream of events. Unfortunately RFID devices are error prone and associate probability with the gathered events. A sequence of events can represent security threat. The stream of probabilistic events can be modeled with uncertain string and can be indexed so that deterministic substring can be queried to detect security threats.

### 3. PRELIMINARIES

#### 3.1 Uncertain string and deterministic string

An uncertain string  $S = s_1 \dots s_n$  over alphabet  $\Sigma$  is a sequence of sets  $s_i, i = 1, \dots, n$ . Every  $s_i$  is a set of pairs of the form  $(c_j, pr(c_j^i))$ , where every  $c_j$  is a character in  $\Sigma$  and  $0 \leq pr(c_j^i) \leq 1$  is the probability of occurrence of  $c_j$  at position  $i$  in the string. Uncertain string length is the total number of positions in the string,

which can be less than the total number of characters in the string. Note that, summation of probability for all the characters at each position should be 1, i.e.  $\sum_j pr(c_j^i) = 1$ . Figure 3 shows an example of an uncertain string of length 11. A deterministic string has only one character at each position with probability 1. We can exclude the probability information for deterministic strings.

#### 3.2 Probability of occurrence of a substring in an uncertain string

Since each character in the uncertain string has an associated probability, a deterministic substring occurs in the uncertain string with a probability. Let  $S = s_1 \dots s_n$  is an uncertain string and  $p$  is a deterministic string. If the length of  $p$  is 1, then probability of occurrence of  $p$  at position  $i$  of  $S$  is the associated probability  $pr(p^i)$ . Probability of occurrence of a deterministic substring  $p = p_1 \dots p_k$ , starting at a position  $i$  in  $S$  is defined as the partial product  $pr(p_1^i) \times \dots \times pr(p_k^{i+k-1})$ . For example in Figure 3,  $SFPQ$  has probability of occurrence  $0.7 \times 1 \times 1 \times 0.5 = 0.35$  at position 2.

#### 3.3 Correlation among string positions

We say that character  $c_k$  at position  $i$  is correlated with character  $c_l$  at position  $j$ , if the probability of occurrence of  $c_k$  at position  $i$  is dependent on the probability of occurrence of  $c_l$  at position  $j$ . We use  $pr(c_k^i)^+$  to denote the probability of  $c_k^i$  when the correlated character is present, and  $pr(c_k^i)^-$  to denote the probability of  $c_k^i$  when the correlated character is absent. Let  $x_g \dots x_h$  be a the substring generated from an uncertain string.  $c_k^i, g \leq i \leq h$  is a character within the substring which is correlated with  $c_l^j$ . Depending on the position  $j$ , we have 2 cases:

**Case 1,  $g \leq j \leq h$  :** The correlated probability of  $(c_k^i)$  is expressed by  $(c_l^j \implies a, \neg c_l^j \implies b)$ , i.e. if  $c_l^j$  is taken as an occurrence, then  $pr(c_k^i) = pr(c_k^i)^+$ , otherwise  $pr(c_k^i) = pr(c_k^i)^-$ . We consider a simple example in Figure 4 to illustrate this. In this string,  $z^3$  is correlated with  $e^1$ . For the substring  $eqz$ ,  $pr(z^3) = .3$ , and for the substring  $fqz$ ,  $pr(z^3) = .4$ .

**Case 2,  $j < g$  or  $j > h$  :**  $c_l^j$  is not within the substring. In this case,  $pr(c_k^i) = pr(c_l^j) * pr(c_k^i)^+ + (1 - pr(c_l^j)) * pr(c_k^i)^+$ . In Figure 4, for substring  $qz$ ,  $pr(z^3) = .6 * .3 + .4 * .4$ .

S[1]	S[2]	S[3]
e: .6 f: .4	q: 1	z: $e^1 \implies .3, \neg e^1 \implies .4$

Figure 4: Uncertain string  $S$  with correlated characters.

#### 3.4 Suffix tree and generalized suffix tree

The suffix tree [28, 21] of a deterministic string  $t[1 \dots n]$  is a lexicographic arrangement of all these  $n$  suffixes in a compact trie structure of  $O(n)$  words space, where the  $i$ -th leftmost leaf represents the  $i$ -th lexicographically smallest suffix of  $t$ . For a node  $i$  (i.e., node with pre-order rank  $i$ ),  $path(i)$  represents the text obtained by concatenating all edge labels on the path from root to node  $i$  in a suffix tree. The locus node  $i_p$  of a string  $p$  is the node closest to the root such that the  $p$  is a prefix of  $path(i_p)$ . The suffix range of a string  $p$  is given by the maximal range  $[sp, ep]$  such that for  $sp \leq j \leq ep$ ,  $p$  is a prefix of (lexicographically)  $j$ -th



smallest suffix of  $t$ . Therefore,  $i_p$  is the lowest common ancestor of  $sp$ -th and  $ep$ -th leaves. Using suffix tree, the locus node as well as the suffix range of  $p$  can be computed in  $O(p)$  time, where  $p$  denotes the length of  $p$ . The suffix array  $A$  of  $t$  is defined to be an array of integers providing the starting positions of suffixes of  $S$  in lexicographical order. This means, an entry  $A[i]$  contains the position of  $i$ -th leaf of the suffix tree in  $t$ . For a collection of strings  $\mathcal{D} = \{d_1, \dots, d_D\}$ , let  $t = d_1 d_2 \dots d_D$  be the text obtained by concatenating all the strings in  $\mathcal{D}$ . Each string is assumed to end with a special character  $\$$ . The suffix tree of  $t$  is called the generalized suffix tree (GST) of  $\mathcal{D}$ .

#### 4. STRING MATCHING IN SPECIAL UNCERTAIN STRINGS

In this section, we construct index for a special form of uncertain string which is extended later. Special uncertain string is an uncertain string where each position has only one probable character with associated non-zero probability of occurrence. Special-uncertain string is defined more formally below.

**DEFINITION 1.** A special uncertain string  $X = x_1 \dots x_n$  over alphabet  $\Sigma$  is a sequence of pairs. Every  $x_i$  is a pair of the form  $(c_i, pr(c_i))$ , where every  $c_i$  is a character in  $\Sigma$  and  $0 < pr(c_i) \leq 1$  is the probability of occurrence of  $c_i$  at position  $i$  in the string.

Before we present an efficient index, we discuss a naive solution similar to deterministic substring searching.

##### 4.1 Simple index

Given a special uncertain string  $X = x_1 \dots x_n$ , construct the deterministic string  $t = c_1 \dots c_n$  where  $c_i$  is the character in  $x_i$ . We build a suffix tree over  $t$ . We build a suffix array  $A$  which maps each leaf of the suffix tree to its original position in  $t$ . We also build a successive multiplicative probability array  $C$ , where  $C[j] = \prod_{i=1}^j Pr(c_i^j)$ , for  $j = 1, \dots, n$ . For a substring  $x_i \dots x_{i+j}$ , probability of occurrence can be easily computed by  $C[i+j]/C[i-1]$ . Given an input  $(p, \tau)$ , we traverse the suffix tree for  $p$  and find the locus node and suffix range of  $p$  in  $O(m)$  time, where  $m$  is the length of  $p$ . Let the suffix range be  $[sp, ep]$ . According to the property of suffix tree, each leaf within the range  $[sp, ep]$  contains an occurrence of  $p$  in  $t$ . Original positions of the occurrence in  $t$  can be found using suffix array, i.e.,  $A[sp], \dots, A[ep]$ . However, each of these occurrence has an associated probability. We traverse each of the occurrence in the range  $A[sp], \dots, A[ep]$ . For an occurrence  $A[i]$ , we find the probability of occurrence by  $C[A[i] + m - 1]/C[A[i] - 1]$ . If the probability of occurrence is greater than  $\tau$ , we report the position  $A[i]$  as an output. Figure 5 illustrates this approach.

**Handling correlation:** Correlation is handled when we check for the probability of occurrence. If  $A[i]$  is a possible occurrence, then we need to consider any existing character within the substring  $x_i \dots x_{i+m-1}$ , that is correlated with another character. Let  $c_k$  is a character at position  $j$  within  $x_i \dots x_{i+m-1}$ , which is correlated with character  $c_{i'}^j$ , i.e. if  $c_{i'}^j$  is included in the substring, then  $pr(c_k^j) = pr(c_{i'}^j)^+$ , or else  $pr(c_k^j) = pr(c_{i'}^j)^-$ . To find the correct probability of  $(c_k^j)$ , if  $j'$  we check the  $j'$ -th position ( $j'$  depth character on the root to locus path in the suffix tree) of the substring. If the  $j'$ -th character is  $c_l$ , then  $C[A[i] + m - 1]/C[A[i] - 1]$  is the correct probability of occurrence for  $x_i \dots x_{i+m}$ . Otherwise,  $C[A[i] + m - 1]/C[A[i] - 1]$  contains the incorrect probability of  $c_k^j$ . Dividing  $C[A[i] + m - 1]/C[A[i] - 1]$  by  $pr(c_{i'}^j)^+$  and multiplying by  $pr(c_{i'}^j)^-$  gives the correct probability of occurrence

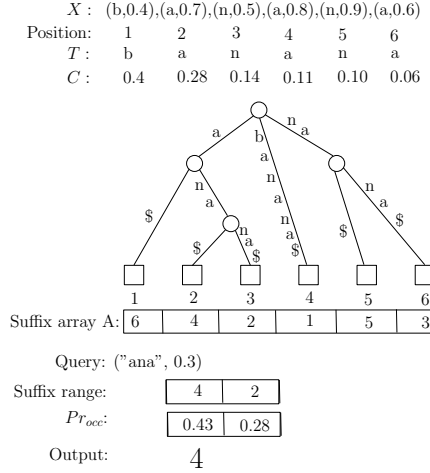


Figure 5: Simple index for special uncertain strings.

in this case. If  $c_l$  falls before or after the substring  $x_i \dots x_{i+m-1}$ ,  $pr(c_k^j) = pr(c_{i'}^j)^+ * pr(c_k^j)^+ + (1 - pr(c_{i'}^j)^+) * pr(c_k^j)^-$ . Dividing  $C[A[i] + m - 1]/C[A[i] - 1]$  by  $pr(c_{i'}^j)^+$  and multiplying by  $pr(c_{i'}^j)^-$  gives the correct probability of occurrence. Note that, we can identify and group all the characters with existing correlation, and search in the suffix tree in one scan for improved efficiency.

The main drawback in this approach is the query time. Within the suffix range  $[sp, ep]$ , possibly very few number of positions can qualify as output because of  $\tau$ . So spending time on each element of the range  $[sp, ep]$  is not justifiable.

##### 4.2 Efficient index:

Bottleneck of the simple index comes from traversing each element within the suffix range. For the efficient index, we iteratively retrieve the element with maximum probability of occurrence in the range in constant time. Whenever the next maximum probability of occurrence falls below  $\tau$ , we conclude our search. We use range maximum query (RMQ) data structure for our index which is briefly explained below.

**Range Maximum Query:** Let  $B$  be an array of integers of length  $n$ , a range maximum query (RMQ) asks for the position of the maximum value between two specified array indices  $[i, j]$ . i.e., the RMQ should return an index  $k$  such that  $i \leq k \leq j$  and  $B[k] \geq B[x]$  for all  $i \leq x \leq j$ . We use the result captured in following lemma for our purpose.

**LEMMA 1.** [10, 11] By maintaining a  $2n + o(n)$  bits structure, range maximum query (RMQ) can be answered in  $O(1)$  time (without accessing the array).

Every leaf of the suffix tree denotes a suffix position in the original text and a root to leaf path represents the suffix. For uncertain string, every character in this root to leaf path has an associated probability which is not stored in the suffix tree. Let  $y_j^i$ , for  $j = 1, \dots, n$  denote a deterministic substring which is the  $i$ -length prefix of the  $j$ -th suffix, i.e. the substring on the root to  $i$ -th leaf path. Let  $Y^i$  is the set of  $y_j^i$ , for  $j = 1, \dots, n$ .

For  $i = 1, \dots, n$ , we define  $C_i$  as the successive multiplicative probability array for the substrings of  $Y^i$ .  $j$ -th element of  $C_i$  is the successive multiplicative probability of the  $i$ -length prefix of the  $j$ -th suffix. More formally  $C_i[j] = \prod_{k=A[j]+i-1}^{A[j]} Pr(c_k^k) = C[A[j] + i - 1]/C[A[j] - 1]$  ( $1 \leq j \leq n$ ). For each  $C_i$  ( $i = 1, \dots, \log n$ ) we

use range maximum query data structure  $RMQ_i$  of  $n$  bits over  $C_i$  and discard the original array  $C_i$ . We convert  $C_i$  into an integer array by multiplying each element by a sufficiently large number and then build the  $RMQ_i$  structure over it. We obtain  $\log n$  number of such  $RMQ$  data structures resulting in total space of  $O(n \log n)$  bits or  $O(n)$  words. We also store the global successive multiplicative probability array  $C$ , where  $C[j] = \prod_{i=1}^j Pr(c_i^i)$ . Given a query  $(p, \tau)$ , idea is to use  $RMQ_i$  for iteratively retrieving maximum probability of occurrence elements in constant time each and validate using  $C$ . To maintain linear space, we can support query substring length of  $m = 0, \dots, \log n$  in this approach. Algorithm 1 illustrates the index construction phase for short substrings.

**Query answering for short substrings ( $m \leq \log n$ ):** Given an input  $(p, \tau)$ , we first retrieve the suffix range  $[l, r]$  in  $O(m)$  time using suffix tree, where  $m$  is the length of  $p$ . We can find the maximum probability occurrence of  $p$  in  $O(1)$  time by executing query  $RMQ_m(l, r)$ . Let  $max$  be the position of maximum probability occurrence and  $max' = A[max]$  be the the original position in  $t$ . We can find the corresponding probability of occurrence by  $C[max' + i - 1]/C[max' - 1]$ . If the probability is less than  $\tau$ , we conclude our search. If it is greater than  $\tau$ , we report  $max'$  as an output. For finding rest of the outputs, we recursively search in the ranges  $[l, max - 1]$  and  $[max + 1, r]$ . Since each call to  $RMQ_m(l, r)$  takes constant time, validating the probability of occurrence takes constant time, we spend  $O(1)$  time for each output. Total query time is optimal  $O(m + occ)$ . Algorithm 2 illustrates the query answering for short substrings. Note that, correlation is handled in similar way as described for the naive index, and we omit the details here.

---

**Algorithm 1: Special-Short-Substring-Index-Construction**

---

```

input : A special uncertain string  $X$ 
output: suffix tree, suffix array  $A$ , successive multiplicative
         probability array  $C$ ,  $RMQ_i$  ( $i = 1, \dots, \log n$ )
Build deterministic string  $t$  from  $X$ 
Build suffix tree over  $t$ 
Build suffix array  $A$  over  $t$ 
// Building successive multiplicative
  probability array
 $C[1] = Pr(c_1^1)$ 
for  $i = 2; i \leq n; i++$  do
  |  $C[i] = C[i-1] \times Pr(c_i^i)$ 
end
// Building  $C_i$  array for  $i = 1, \dots, \log n$ 
for  $i = 1; i \leq \log n; i++$  do
  for  $j = 1; j \leq n; j++$  do
    |  $C_i[j] = C[A[j] + i - 1]/C[A[j] - 1]$ 
    | // Handling correlated characters
    | for all character  $c_a^k$  in  $t[A[j] \dots t[A[j] + i - 1]$  that
    | are correlated with another character  $c_b^l$  do
    |   | if  $(A[j] \leq l \leq [A[j] + i - 1])$  and  $c_b^l$  is not within
    |   |  $t[A[j] \dots t[A[j] + i - 1])$ 
    |   |  $C_i[j] = C_i[j]/Pr(c_a^k)^+ * Pr(c_b^l)^-$ 
    |   | else
    |   |  $pr(c_a^k) = pr(c_b^l) * pr(c_a^k)^+ + (1 - pr(c_b^l)) * pr(c_a^k)^-$ 
    |   |  $C_i[j] = C_i[j]/Pr(c_a^k)^+ * Pr(c_a^k)$ 
    |   | end
    |   | end
    |   | end
  | end
  | end
  | Build  $RMQ_i$  over  $C_i$ 
end

```

---



---

**Algorithm 2: Special-Short-Substring-Query-Answering**

---

```

input : Query substring  $p$ , probability threshold  $\tau$ 
output: Occurrence positions of  $p$  in  $X$  with probability of
         occurrence greater than  $\tau$ 
 $m = length(p)$ 
call RecursiveRmq( $m, 1, n$ )
function RECURSIVERMQ( $i, l, r$ )  $\triangleright$  Recursive RMQ method
   $max = RMQ_m(l, r)$ 
   $max' = A[max]$ 
  if  $C[max' + i - 1]/C[max' - 1] > \tau$  then
    Output  $max'$ 
    Call RecursiveRmq( $m, l, max - 1$ )
    Call RecursiveRmq( $m, max + 1, r$ )
  end

```

---

**Query answering for long substrings ( $m > \log n$ ):** We use a blocking scheme for answering long query substrings ( $m > \log n$ ). Since exhaustively enumerating all possible substrings and storing the probabilities for each of them is infeasible, we only store selective probability values at construction time and compute the others at query time. We partition the entire suffix range of suffix array into different size blocks. More formally, for  $i = \log n, \dots, n$ , we divide the suffix range  $[1, n]$  of suffix array  $A[1, n]$  into  $O(n/i)$  number of blocks each of size  $i$ . Let  $B_i$  be the set of length  $i$  blocks, i.e.  $B_i = \{[A[1] \dots A[i]], [A[i+1] \dots A[2i]], \dots [A[n-i+1] \dots A[n]]\}$  and let  $B = \{B_{\log n}, \dots, B_n\}$ . For a suffix starting at  $A[j]$  and for  $B_i$ , we only consider the length  $i$  prefix of that suffix, i.e.  $A[j \dots j + i]$ . The idea is to store only the maximum probability value per block. For  $B_i, i = \log n, \dots, n$ , we define a probability array  $PB_i$  containing  $n/i$  elements.  $PB_i[j]$  is the maximum probability of occurrence of all the substrings of length  $i$  belonging to the  $j$ -th block of  $B_i$ . We build a range maximum query structure  $RMQ_i$  for  $PB_i$ .  $RMQ_i$  takes  $O(n/i)$  bits, total space is bounded by  $\sum_i O(n/i) = O(n \log n)$  bits or  $O(n)$  words.

For a query  $(p, \tau)$ , we first retrieve the suffix range  $[l, r]$ . This suffix range can spread over multiple blocks of  $B_m$ . We use  $RMQ_m$  to proceed to next step. Note that  $RMQ_m$  consists of  $N/m$  bits, corresponding to the  $N/m$  blocks of  $B_m$  in order. Our query proceeds by executing range maximum query in  $RMQ_m(l, r)$ , which will give us the index of the maximum probability element of string length  $m$  in that suffix range. Let the maximum probability element position in  $RMQ_m$  is  $max$  and the block containing this element is  $B_{max}$ . Using  $C$  array, we can find out if the probability of occurrence is greater than  $\tau$ . Note that, we only stored one maximum element from each block. If the maximum probability found is greater than  $\tau$ , we check all the other elements in that block in  $O(m)$  time. In the next step, we recursively query  $RMQ_m(l, max - 1)$  and  $RMQ_m(max + 1, r)$  to find out subsequent blocks. Whenever  $RMQ$  query for a range returns an element having probability less than  $\tau$ , we stop the recursion in that range. Number of blocks visited during query answering is equal to the number of outputs and inside each of those block we check  $m$  elements, obtaining total query time of  $O(m \times occ)$ .

In practical applications, query substrings are rarely longer than  $\log n$  length. Our index achieves optimal query time for substrings of length less than  $\log n$ . We show in the experimental section that on average our index achieves efficient query time proportional to substring length and number of outputs reported.

## 5. SUBSTRING MATCHING IN GENERAL UNCERTAIN STRING

In this section we construct index for general uncertain string based on the index of special uncertain string. The idea is to convert a general uncertain string into a special uncertain string, build the data structure similar to the previous section and carefully eliminate the duplicate answers. Below we show the steps of our solution in details.

### 5.1 Transforming general uncertain string

We employ the idea of Amihood et al [1] to transform general uncertain string into a special uncertain string. **Maximal factor** of an uncertain string is defined as follows.

**DEFINITION 2.** A *maximal factor* of a uncertain string  $S$  starting at location  $i$  with respect to a fixed probability threshold  $\tau_c$  is a string of maximal length that when aligned to location  $i$  has probability of occurrence at least  $\tau_c$ .

For example in figure 3, maximal factors of the uncertain string  $S$  at location 5 with respect to probability threshold 0.15 are "QPA", "QPF", "TPA", "TPF".

An uncertain string  $S$  can be transformed to a special uncertain string by concatenating all the maximal factors of  $S$  in order. Suffix tree built over the concatenated maximal factors can answer substring searching query for a fixed probability threshold  $\tau_c$ . But this method produces a special uncertain string of  $\Omega(n^2)$  length, which is practically infeasible. To reduce the special uncertain string length, Amihood et al. [1] employs further transformation to obtain a set of extended maximal factors. Total length of the extended maximal factors is bounded by  $O((\frac{1}{\tau_c})^2 n)$ .

**LEMMA 2.** Given a fixed probability threshold value  $\tau_c (0 < \tau_c \leq 1)$ , an uncertain string  $S$  can be transformed into a special uncertain string  $X$  of length  $O((\frac{1}{\tau_c})^2 n)$  such that any deterministic substring  $p$  of  $S$  having probability of occurrence greater than  $\tau_c$  is also a substring of  $X$ .

Simple suffix tree structure for answering query does not work for the concatenated extended maximal factors. A special form of suffix tree, namely property suffix tree is introduced by Amihood et al. [1]. Also substring searching in this method works only on a fixed probability threshold  $\tau_c$ . A naive way to support arbitrary probability threshold is to construct special uncertain string and property suffix tree index for all possible value of  $\tau_c$ , which is practically infeasible due to space usage.

We use the technique of lemma 2 to transform a given general uncertain string to a special uncertain string of length  $O((\frac{1}{\tau_{min}})^2 n)$  based on a probability threshold  $\tau_{min}$  known at construction time, and employ a different indexing scheme over it. Let  $X$  be the transformed special uncertain string. A running example is shown in Appendix B in the full version of this paper [27]. Following section elaborates the subsequent steps of the index construction.

### 5.2 Index construction on the transformed uncertain string

Our index construction is similar to the index of section 4. We need some additional components to eliminate duplication and position transformation.

Let  $N = |X|$  be the length of the special uncertain string  $X$ . Note that  $N = O((\frac{1}{\tau_{min}})^2 n) = O(n)$ , since  $\tau_{min}$  is a constant known in construction time. For transforming the positions of  $X$  into the original position in  $S$ , we store an array  $Pos$  of size  $N$ ,

where  $Pos[i]$ =position of the  $i$ -th character of  $X$  in the original string  $S$ . We construct the deterministic string  $t = c_1 \dots c_N$  where  $c_i$  is the character in  $X_i$ . We build a suffix tree over  $t$ . We build a suffix array  $A$  which maps each leaf of the suffix tree to its position in  $t$ . We also build a successive multiplicative probability array  $C$ , where  $C[j] = \prod_{i=1}^j Pr(c_i^i)$ , for  $1 \leq j \leq N$ . For a substring of length  $j$  starting at position  $i$ , probability of occurrence of the substring in  $X$  can be easily computed by  $C[i+j-1]/C[i-1]$ . For  $i = 1, \dots, n$ , we define  $C_i$  as the successive multiplicative probability array for substring length  $i$  i.e.  $C_i[j] = \prod_{k=A[j]}^{A[j]+i-1} Pr(c_k^k) = C[A[j] + i - 1]/C[A[j] - 1]$  ( $1 \leq j \leq n$ ). Appendix B of the full version [27] shows  $Pos$  array and  $C$  array after transformation of an uncertain string. Below we explain how duplicates may arise in outputs and how to eliminate them.

Possible duplicate positions in the output arises because of the general to special uncertain string transformation. Note that, distinct positions in  $X$  can correspond to the same position in the original uncertain string  $S$ , resulting in same position possibly reported multiple times. A key observation here is that for two different substrings of length  $m$ , if the locus nodes are different than the corresponding suffix ranges are disjoint. These disjoint suffix ranges collectively cover all the leaves of the suffix tree. For each such disjoint ranges, we need to store probability values for only the unique positions of  $S$ . Without loss of generality we store the value for leftmost unique position in each range.

For any node  $u$  in the suffix tree,  $depth(u)$  is the length of the concatenated edge labels from root to  $u$ . We define by  $L_i$  as the set of nodes  $u_i^j$  such that  $depth(u_i^j) \geq i$  and  $depth(parent(u_i^j)) \leq i$ . For  $L_i = u_i^1, \dots, u_i^k$ , we have a set of disjoint suffix ranges  $[sp_i^1, ep_i^1], \dots, [sp_i^k, ep_i^k]$ . A suffix range  $[sp_i^j, ep_i^j]$  can contain duplicate positions of  $S$ . Using the  $Pos$  array we can find the unique positions for each range and store only the values corresponding to the unique positions in  $C_i$ .

We use range maximum query data structure  $RMQ_i$  of  $n$  bits over  $C_i$  and discard the original array  $C_i$ . Note that,  $RMQ$  data structure can be built over an integer array. We convert  $C_i$  into an integer array by multiplying each element by a sufficiently large number and then build the  $RMQ_i$  structure over it. We obtain  $\log n$  number of such  $RMQ$  data structures resulting in total space of  $O(n \log n)$  bits or  $O(n)$  words. For long substrings ( $m > \log n$ ), we use the blocking data structure similar to section 4. Detailed construction phase is shown in Algorithm 3 of Appendix A in the full version of this paper [27].

### 5.3 Query answering

Query answering procedure is almost similar to the query answering procedure of section 4. Only difference being the transformation of position which is done using the  $Pos$  array. Detailed query answering Algorithm for short query substrings is included in Appendix A of the full version of this paper [27]. See Appendix B for an illustrative example of query answering.

### 5.4 Space complexity

For analyzing the space complexity, we consider each component of our index. Length of the special uncertain string  $X$  and deterministic string  $t$  are  $O(n)$ , where  $n$  is the number of positions in  $S$ . Suffix tree and suffix array each takes linear space. We store a successive probability array of size  $O(n)$ . We build probability array  $C_i$  for  $i = 1, \dots, \log n$ , where each  $C_i$  takes of  $O(n)$ . However we build  $RMQ_i$  of  $n$  bits over  $C_i$  and discard the original array  $C_i$ . We obtain  $\log n$  number of such  $RMQ$  data structures resulting in total space of  $O(n \log n)$  bits or  $O(n)$  words. For the blocking scheme, we build  $RMQ_i$  data structure for

$i = \log n, \dots, n$ .  $RMQ_i$  takes  $n/i$  bits, total space is  $\sum_i n/i = O(n \log n)$  bits or  $O(n)$  words. Since each component of our index takes linear space, total space taken by our index is  $O(n)$  words.

## 5.5 Proof of correctness

In this section we discuss the correctness of our indexing scheme.

**Substring conservation property of the transformation:** At first we show that any substring of  $S$  with probability of occurrence greater than query threshold  $\tau$  can be found in  $t$  as well. According to lemma 2, a substring having probability of occurrence greater than  $\tau_{min}$  in  $S$  is also a substring of the transformed special uncertain string  $X$ . Since query threshold value  $\tau$  is greater than  $\tau_{min}$ , and entire character string of  $X$  is same as the deterministic string  $t$ , a substring having probability of occurrence greater than query threshold  $\tau$  in  $S$  will be present in the deterministic string  $t$ .

**Complete set of occurrences are outputted:** For contradiction, we assume that an occurrence position  $z$  of substring  $p$  in  $S$  having probability of occurrence greater than  $\tau$  is not included in the output. From the aforementioned property,  $p$  is a substring of  $t$ . According to the property of suffix tree,  $z$  must be present in the suffix range  $[sp, ep]$  of  $p$ . Using  $RMQ$  structure, we report all the occurrence in  $[sp, ep]$  in their decreasing order of probability of occurrence value in  $S$  and stop when the probability of occurrence falls below  $\tau$ , which ensures inclusion of  $z$ .

**No incorrect occurrence appears in output:** An output  $z$  can be incorrect occurrence if it is not present in uncertain string  $S$  or its probability of occurrence is less than  $\tau$ . We query only the occurrences in the suffix range  $[sp, ep]$  of  $p$ , according to the property of suffix tree all of which are valid occurrences. We also validate the probability of occurrence for each of them using the successive multiplicative probability array  $C$ .

## 6. STRING LISTING FROM UNCERTAIN STRING COLLECTION

In this section we propose an indexing solution for problem 2. We are given a collection of  $D$  uncertain strings  $\mathcal{D} = \{d_1, \dots, d_D\}$  of  $n$  positions in total. Let  $i$  denotes the string identifier of string  $d_i$ . For a query  $(p, \tau)$ , we have to report all the uncertain string identifiers  $j$  such that  $d_j$  contains  $p$  with probability of occurrence more than  $\tau$ . In other words, we want to list the strings from a collection of a string, that are relevant to a deterministic query string based on probability parameter.

**Relevance metric:** For a deterministic string  $t$  and an uncertain string  $S$ , we define a relevance metric,  $Rel(S, t)$ . If  $t$  does not have any occurrence in  $S$ , then  $Rel(S, t)=0$ . If  $s$  has only one occurrence of  $t$ , then  $Rel(S, t)$  is the probability of occurrence of  $t$  in  $S$ . If  $s$  contains multiple occurrences of  $t$ , then  $Rel(S, t)$  is a function of the probability of occurrences of  $t$  in  $S$ . Depending on the application, various functions can be chosen as the appropriate relevance metric. A common relevance metric is the maximum probability of occurrence, which we denote by  $Rel(S, t)_{max}$ . The  $OR$  value of the probability of occurrences is another useful relevance metric. More formally, if a deterministic string  $t$  has nonzero probable occurrences at positions  $i_1, \dots, i_k$  of an uncertain string  $S$ , then we define the relevance metric of  $t$  in  $S$  as  $Rel(S, t)_{OR} = \sum_{j=i_1}^{i_k} pr(t_j) - \prod_{j=i_1}^{i_k} pr(t_j)$ , where  $pr(t_j)$  is the probability of occurrence of  $t$  in  $S$  at position  $j$ . Figure 6 shows an example.

**Practical motivation:** Uncertain string listing finds numerous practical motivation. Consider searching for a virus pattern in a collection of text files with fuzzy information. The objective is to

Uncertain string  $S$ :

$S[1]$	$S[2]$	$S[3]$	$S[4]$	$S[5]$	$S[6]$
A .4	B .3	A .5	A .6	B .5	A .4
B .3	L .3	F .5	B .4	F .3	C .3
F .3	F .3			J .2	E .2
	J .1				F .1

$$\begin{aligned}
 & Rel(S, "BFA")_{max} = .09 \\
 Rel(S, "BFA")_{OR} &= (.06 + .09 + .048) - (.06 * .09 * .048) \\
 &= .19786
 \end{aligned}$$

Figure 6: Relevance metric for string listing.

quarantine the files that contain the virus pattern. This problem can be modeled as a uncertain string listing problem, where the collection of text files is the uncertain string collection  $D$ , the virus pattern is the query pattern  $P$ , and  $\tau$  is the confidence of matching. Similarly, searching for a gene pattern in genomic sequences of different species can be solved using uncertain string listing data structure.

**The index:** As explained before, a naive search on each of the string will result in  $O(\sum_i \text{search time on } d_i)$  which can be much larger than the actual number of strings containing the string. Objective of our index is to spend only one search time and time proportional to the number of output strings. We construct a generalized suffix tree so that we have to search for the string only once. We concatenate  $d_1, \dots, d_D$  by a special symbol which is not contained in any of the document and obtain a concatenated general uncertain string  $S = d_1\$ \dots \$d_D$ . Next we use the transformation method described in previous section to obtain deterministic string  $t$ , construct suffix tree and suffix array for  $t$ . According to the property of suffix tree, the leaves under the locus of a query substring  $t$  contains all the occurrence positions of  $t$ . However, these leaves can possibly contain duplicate positions and multiple occurrence of the same document. In the query answering phase, duplicate outputs can arise because of the following two reasons:

1. Distinct positions in  $t$  can correspond to the same position in the original uncertain string  $S$
2. Distinct positions in  $S$  can correspond to the same string identifier  $d_j$  which should be reported only once

Duplicate elimination is important to keep the query time proportional to the number of output strings. At first we construct the successive multiplicative probability array  $C_i$  similar to the substring searching index, then show how to incorporate  $Rel(S, t)$  value for the multiple occurrences cases in the same document and duplicate elimination.

Let  $y_j^i$ , for  $j = 1, \dots, n$  denote a deterministic substring which is the  $i$ -length prefix of the  $j$ -th suffix, i.e. the substring on the root to  $i$ -th leaf path. Note that, multiple  $y_j^i$  can belong to the same locus node in the suffix tree. Let  $Y^i$  is the set of  $y_j^i$ , for  $j = 1, \dots, n$ . The  $i$ -depth locus nodes in the suffix tree constitutes disjoint partitions in  $Y^i$ . For  $i = 1, \dots, n$ , we define  $C_i$  as the successive multiplicative probability array for the substrings of  $Y^i$ .  $j$ -th element of  $C_i$  is the successive multiplicative probability of the  $i$ -length prefix of the  $j$ -th suffix. More formally  $C_i[j] = \prod_{k=A[j]}^{A[j]+i-1} Pr(c_k^k) = C[A[j] + i - 1] / C[A[j] - 1] (1 \leq j \leq n)$ .

The  $i$ -depth locus nodes in the suffix tree constitutes disjoint partitions in  $C_i$ . Let  $u$  be a  $i$ -depth locus node having suffix range  $[j \dots k]$  and root to  $u$  substring  $t$ . Then the partition  $C_i[j \dots k]$  belongs to  $u$ . For this partitions, we store only one occurrence of

a string  $d_j$  with the relevance metric value  $Rel(S, t)$ , and discard the other occurrences of  $d_j$  in that range. We build  $RMQ$  structure similar to section 5.

**Query answering:** We explain the query answering for short substrings. Blocking scheme described in previous section can be used for longer query substrings. Given an input  $(p, \tau)$ , we first retrieve the suffix range  $[l, r]$  in  $O(m)$  time using suffix tree, where  $m$  is the length of  $p$ . We can find the maximum relevant occurrence of  $p$  in  $O(1)$  time by executing query  $RMQ_m(l, r)$ . Let  $max$  be the position of maximum relevant occurrence and  $max' = A[max]$  be the original position in  $t$ . For relevance metric  $Rel(S, t)_{max}$ , we can find the corresponding probability of occurrence by  $C[max' + i - 1]/C[max' - 1]$ . In case of the other complex relevance metric, all the occurrences need to be considered to retrieve the actual value of  $Rel(S, t)$ . If the relevance metric is less than  $\tau$ , we conclude our search. If it is greater than  $\tau$ , we report  $max'$  as an output. For finding rest of the outputs, we recursively search in the ranges  $[l, max - 1]$  and  $[max + 1, r]$ . Each call to  $RMQ_m(l, r)$  takes constant time. For simpler relevance metrics (such as  $Rel(S, t)_{max}$ ), validating the relevance metric takes constant time. Total query time is optimal  $O(m + occ)$ . However, for more complex relevance metric, all the occurrences of  $t$  might need to be considered, query time will be proportionate to the total number of occurrences.

## 7. APPROXIMATE SUBSTRING SEARCHING

In this section we introduce an index for approximate substring matching in an uncertain string. As discussed previously, several challenges of uncertain string matching makes it harder to achieve optimal theoretical bound with linear space. We have proposed index for exact matching which performs near optimally in practical scenarios, but achieves theoretical optimal bound only for shorter query strings. To achieve optimal theoretical bounds for any query, we propose an approximate string matching solution. Our approximate string matching data structure answers queries with an additive error  $\epsilon$ , i.e. outputs can have probability of occurrence  $\geq \tau - \epsilon$ .

At first we begin by transforming the uncertain string  $S$  into a special uncertain string  $X$  of length  $N = O((\frac{1}{\tau_{min}})^2 n)$  using the technique of lemma 2 with respect to a probability threshold value  $\tau_{min}$ . We obtain a deterministic string  $t$  from  $X$  by concatenating the characters of  $X$ . We build a suffix tree for  $t$ . Note that, each leaf in the suffix tree has an associated probability of occurrence  $\geq \tau_{min}$  for the corresponding suffix. Given a query  $p$ , substring matching query for threshold  $\tau_{min}$  can now be answered by simply scanning the leafs in subtree of locus node  $i_p$ . We first describe the framework (based on Hon et. al. [14]) which supports a specific probability threshold  $\tau$  and then extend it for arbitrary  $\tau \geq \tau_{min}$ .

We begin by marking nodes in the suffix tree with positional information by associating  $Pos_{id} \in [1, n]$ . Here,  $Pos_{id}$  indicates the starting position in the original string  $S$ . A leaf node  $l$  is marked with a  $Pos_{id} = d$  if the suffix represented by  $l$  begins at position  $d$  in  $S$ . An internal node  $u$  is marked with  $d$  if it is the lowest common ancestor of two leaves marked with  $d$ . Notice that a node can be marked with multiple position ids. For each node  $u$  and each of its marked position id  $d$ , define a link to be a triplet  $(origin, target, Pos_{id})$ , where  $origin = u$ ,  $target$  is the lowest proper ancestor of  $u$  marked with  $d$ , and  $Pos_{id} = d$ . Two crucial properties of these links are listed below.

- Given a substring  $p$ , for each position  $d$  in  $S$  where  $p$  matches with probability  $\geq \tau_{min}$ , there is a unique link whose origin

is in the subtree of  $i_p$  and whose target is a proper ancestor of  $i_p$ ,  $i_p$  being the locus node of substring  $p$ .

- The total number of links is bounded by  $O(N)$ .

Thus, substring matching query with probability threshold  $\tau_{min}$  can now be answered by identifying/reporting the links that originate in the subtree of  $i_p$  and are targeted towards some ancestor of it. By referring to each node using its pre-order rank, we are interested in links that are stabbed by locus node  $i_p$ . Such queries can be answered in  $O(m + occ)$ , where  $|p| = m$  and  $occ$  is the number of answers to be reported (Please refer to [14] for more details).

As a first step towards answering queries for arbitrary  $\tau \geq \tau_{min}$ , we associate probability information along with each link. Thus each link is now a quadruple  $(origin, target, Pos_{id}, prob)$  where first three parameters remain same as described earlier and  $prob$  is the probability of  $prefix(u)$  matching uncertain string  $S$  at position  $Pos_{id} = d$ . It is evident that for substring  $p$  and arbitrary  $\tau \geq \tau_{min}$ , a link stabbed by locus node  $i_p$  with  $prob \geq \tau$  corresponds to an occurrence of  $p$  in  $S$  at position  $d$  with probability  $\geq \tau$ . However, a link stabbed by  $i_p$  with  $prob < \tau$  can still produce an outcome since  $prefix(i_p)$  contains additional characters not included in  $p$ , which may be responsible for matching probability to drop below  $\tau$ . Even though we are interested only in approximate matching this observation leads up the next step towards the solution. We partition each link  $(origin = u, target = v, Pos_{id} = d, prob)$  into multiple links  $(or_1 = u, tr_1, d, prob_1)$ ,  $(or_2 = tr_1, tr_2, d, prob_2)$ ,  $\dots$ ,  $(or_k = tr_{k-1}, tr_k = v, d, prob_k)$  such that  $prob_j - prob_{j-1} \leq \epsilon$  for  $2 \leq j \leq k$ . Here  $or_2, \dots, or_k$  may not refer to the actual node in the suffix tree, rather it can be considered as a dummy node inserted in-between an edge in suffix tree. In essence, we move along the path from node  $u = or_1$  towards its ancestors one character at a time till the probability difference is bounded by  $\epsilon$  i.e., till we reach node  $tr_1$ . The process then repeats with  $tr_1$  as the origin node and so on till we reach the node  $v$ . It can be seen that the total number of links can now be bounded by  $O(N/\epsilon)$ . In order to answer a substring matching query with threshold  $\tau \geq \tau_{min}$ , we need to retrieve all the links stabbed by  $i_p$  with  $prob \geq \tau$ . Occurrence of substring  $p$  in  $S$  corresponding to each such link is then guaranteed to have its matching probability at-least  $\tau - \epsilon$  due to the way links are generated (for any link with  $(u, v)$  as origin and target probability of  $prefix(v)$  matching in  $S$  can be more than that of  $prefix(v)$  only by  $\epsilon$  at the most).

## 8. EXPERIMENTAL EVALUATION

In this section we evaluate the performance of our substring searching and string listing index. We use a collection of query substrings and observe the effect of varying the key parameters. Our experiments show that, for short query substrings, uncertain string length does not affect the query performance. For long query substrings, our index fails to achieve optimal query time. However this does not deteriorate the average query time by big margin, since the probability of match also decreases significantly as substring gets longer. Index construction time is proportional to uncertain string size and probability threshold parameter  $\tau_{min}$ .

We have implemented the proposed indexing scheme in C++. The experiments are performed on a 64-bit machine with an Intel Core i5 CPU 3.33GHz processor and 8GB RAM running Ubuntu. We present experiments along with analysis of performance.

### 8.1 Dataset

We use a synthetic datasets obtained from their real counterparts. We use a concatenated protein sequence of mouse and human (alphabet size  $|\Sigma| = 22$ ), and break it arbitrarily into shorter strings.

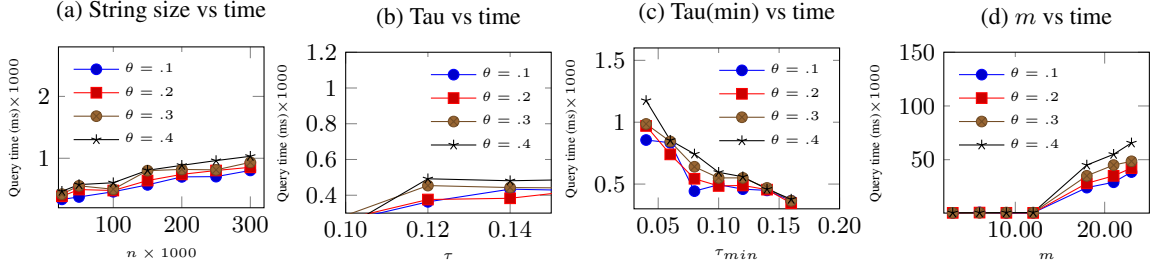


Figure 7: Substring searching query time for different string lengths ( $n$ ), query threshold value  $\tau$ , construction time threshold parameter  $\tau_{min}$  and query substring length  $m$ .

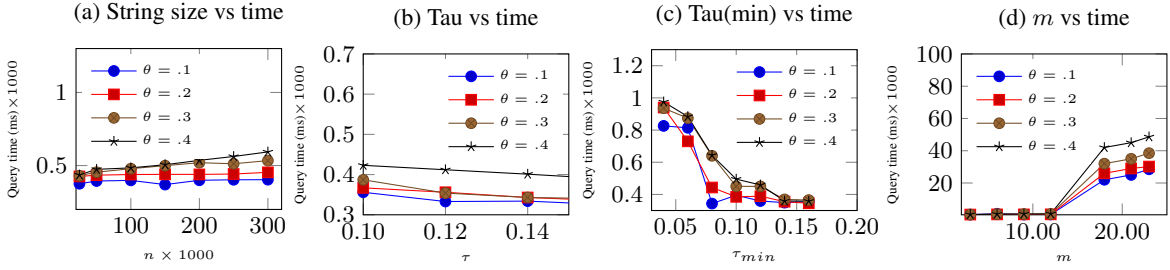


Figure 8: String listing query time for different string lengths ( $n$ ), query threshold value  $\tau$ , construction time threshold parameter  $\tau_{min}$  and query substring length  $m$ .

For each string  $s$  in the dataset we first obtain a set  $A(s)$  of strings that are within edit distance 4 to  $s$ . Then a character-level probabilistic string  $S$  for string  $s$  is generated such that, for a position  $i$ , the pdf of  $S[i]$  is based on the normalized frequencies of the letters in the  $i$ -th position of all the strings in  $A(s)$ . We denote by  $\theta$  the fraction of uncertain characters in the string.  $\theta$  is varied between 0.1 to 0.5 to generate strings with different degree of uncertainty. The string length distributions in this dataset roughly follows a normal distribution in the range of  $[20, 45]$ . The average number of choices that each probabilistic character  $S[i]$  may have is set to 5.

## 8.2 Query time for different string lengths ( $n$ ) and fraction of uncertainty ( $\theta$ )

We evaluate the query time for different string lengths  $n$ , ranging from  $2K$  to  $300K$  and  $\theta$  ranging from 0.1 to 0.5. Figure 7(a) and Figure 8(a), shows the query times for substring searching and string listing. Note that,  $n$  is number of positions in the uncertain string where each position can have multiple characters. We take the average time for query lengths of 10,100,500,1000. We use  $\tau_{min} = 0.1$  and query threshold  $\tau = 0.2$ . As shown in the figures, query times does not show much irregularity in performance when the length of string goes high. This is because for shorter query length, our index achieves optimal query time. Although for longer queries, our index achieves  $O(m \times occ)$  time, longer query strings probability of occurrence gets low as string grows longer resulting in less number of outputs. However when fraction of uncertainty( $\theta$ ) increases in the string, performance shows slight decrease as query time increases slightly. This is because longer query strings are more probable to match with strings with high level of uncertainty.

## 8.3 Query time for different $\tau$ and fraction of uncertainty ( $\theta$ )

In Figure 7(b) and Figure 8(b), we show the average query times for string matching and string listing for probability threshold  $\tau =$

0.04, 0.06, 0.08, 0.1, 0.12 for fixed  $\tau_{min} = 0.1$ . In terms of performance, query time increases with decreasing  $\tau$ . This is because more matching is probable for smaller  $\tau$ . Larger  $\tau$  reduces the output size, effectively reducing the query time as well.

## 8.4 Query time for different $\tau_{min}$ and fraction of uncertainty ( $\theta$ )

In Figure 7(c) and Figure 8(c), we show the average query times for string matching and string listing for probability threshold  $\tau_{min} =$  0.04, 0.06, 0.08, 0.1, 0.12 which shows slight impact of  $\tau_{min}$  over query time.

## 8.5 Query time for different substring lengths ( $m$ ) and fraction of uncertainty ( $\theta$ )

In figure 7(d) and figure Figure 8(d), we show the average query times for string matching and string listing. As it can be seen long pattern length drastically increases the query time.

## 8.6 Construction time for different string lengths and fraction of uncertainty ( $\theta$ )

Figure 9(a) shows the index construction times for uncertain string length  $n$  ranging from  $2K$  to  $300K$ . We can see that the construction time is proportional to the string length  $n$ . Increasing uncertainty factor  $\theta$  also impacts the construction time as more permutation is possible with increasing uncertain positions. Figure 9(b) shows the impact of  $\theta$  on construction time.

## 8.7 Space usage

Theoretical bound for our index is  $O(n)$ . However, this bound can have hidden multiplicative constant. Here we elaborate more on the actual space used for our index.

For our indexes, we construct the regular string  $t$  of length  $N = O((\frac{1}{\tau_{min}})^2 n)$  by concatenating all the extended maximal factors based on threshold  $\tau_{min}$ . We do not store the string  $t$  in our index.

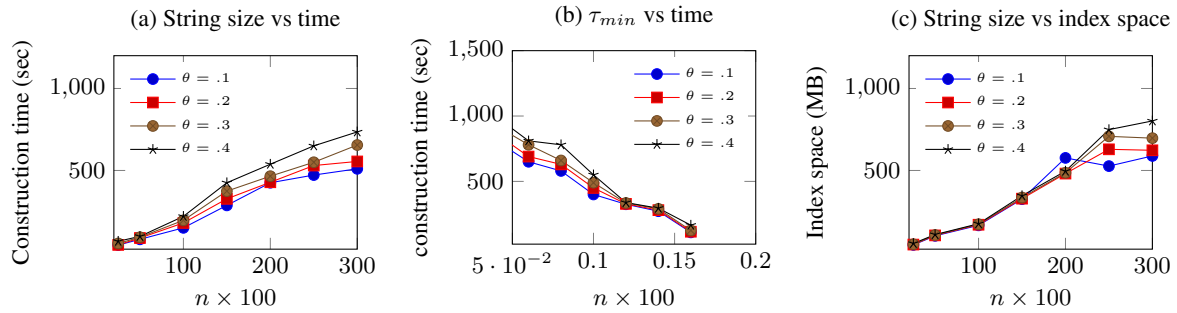


Figure 9: Construction time and index space for different string lengths ( $n$ ) and probability threshold  $\tau_{min} = .1$

We built RMQ structures  $RMQ_i$  for  $i = 1, \dots, \log n$  which takes  $O(N \log n)$  bits. The practical space usage of RMQ is usually very small with hidden multiplicative constant of  $2 - 3$ . So the average space usage of our RMQ structure in total can be stated as  $3N$  words. For a query string  $p$ , we find the suffix range of  $p$  in the concatenated extended maximum factor string  $t$ . For this purpose, instead of using Generalized Suffix Tree(GST), we use its space efficient version i.e., a compressed suffix array (CSA) of  $t$ . There are many versions of CSA's available in literature. For our purpose we use the one in [2] that occupies  $N \log \sigma + o(N \log \sigma) + O(N)$  bits space and retrieves the suffix range of query string  $p$  in  $O(p)$  time. In practice, this structure takes about  $2.5N$  words space. We also store an array  $D$  of size  $N$  storing the partial probabilities, which takes approximately  $4N$  bytes of space. Finally  $Pos$  array is used for position transformation, taking  $N$  words space. Summing up all the space usage, our index takes approximately  $3N + 2.5N + 4N + N = 10.5N = (\frac{1}{\tau_{min}})^2 10.5n$ . Figure 9(c) shows the space usage for different string length( $n$ ) and  $\theta$ .

## 9. CONCLUSIONS

In this paper we presented indexing framework for searching in uncertain strings. We tackled the problem of searching a deterministic substring in uncertain string and proposed both exact and approximate solution. We also formulated the uncertain string listing problem and proposed index for string listing from a uncertain string collection. Our indexes can support arbitrary values of probability threshold parameter. Uncertain string searching is still largely an unexplored area. Constructing more efficient index, variations of the string searching problem satisfying diverse query constraints are some interesting future work direction.

## 10. REFERENCES

- [1] A. Amir, E. Chencinski, C. S. Iliopoulos, T. Kopelowitz, and H. Zhang. Property matching and weighted matching. *Theor. Comput. Sci.*, 395(2-3):298–310, 2008.
- [2] D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. In *ESA*, pages 748–759, 2011.
- [3] T. Bernecker, H. Kriegel, M. Renz, F. Verhein, and A. Züfle. Probabilistic frequent pattern growth for itemset mining in uncertain databases. In *Scientific and Statistical Database Management - 24th International Conference, SSDBM 2012, Chania, Crete, Greece, June 25-27, 2012. Proceedings*, pages 38–55, 2012.
- [4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 5, 2006.
- [5] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 876–887. VLDB Endowment, 2004.
- [6] C. K. Chui and B. Kao. A decremental approach for mining frequent itemsets from uncertain data. In *Advances in Knowledge Discovery and Data Mining, 12th Pacific-Asia Conference, PAKDD 2008, Osaka, Japan, May 20-23, 2008 Proceedings*, pages 64–75, 2008.
- [7] C. K. Chui, B. Kao, and E. Hung. Mining frequent itemsets from uncertain data. In *Advances in Knowledge Discovery and Data Mining, 11th Pacific-Asia Conference, PAKDD 2007, Nanjing, China, May 22-25, 2007, Proceedings*, pages 47–58, 2007.
- [8] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, 2007.
- [9] S. Dash, K. Chon, S. Lu, and E. Raeder. Automatic real time detection of atrial fibrillation. *Annals of biomedical engineering*, 37(9):1701–1709, 2009.
- [10] J. Fischer and V. Heun. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *ESCAPE*, pages 459–470, 2007.
- [11] J. Fischer, V. Heun, and H. M. Stühler. Practical Entropy-Bounded Schemes for  $O(1)$ -Range Minimum Queries. In *IEEE DCC*, pages 272–281, 2008.
- [12] T. Ge and Z. Li. Approximate substring matching over uncertain strings. *PVLDB*, 4(11):772–782, 2011.
- [13] 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, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 491–500, 2001.
- [14] W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on*, pages 713–722. IEEE, 2009.
- [15] J. Jests, F. Li, Z. Yan, and K. Yi. Probabilistic string similarity joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 327–338, 2010.
- [16] B. Kanagal and A. Deshpande. Indexing correlated probabilistic databases. In *Proceedings of the 2009 ACM*

- SIGMOD International Conference on Management of data*, pages 455–468. ACM, 2009.
- [17] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 257–266, 2008.
- [18] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. *The VLDB Journal—The International Journal on Very Large Data Bases*, 20(2):249–275, 2011.
- [19] Y. Li, J. Bailey, L. Kulik, and J. Pei. Efficient matching of substrings in uncertain sequences. In *Proceedings of the 2014 SIAM International Conference on Data Mining, Philadelphia, Pennsylvania, USA, April 24-26, 2014*, pages 767–775, 2014.
- [20] D. M. Lilley, R. M. Clegg, S. Diekmann, N. C. Seeman, E. Von Kitzing, and P. J. Hagerman. Nomenclature committee of the international union of biochemistry and molecular biology (nc- iubmb) a nomenclature of junctions and branchpoints in nucleic acids recommendations 1994. *European Journal of Biochemistry*, s. *FEBS J*, 230(1):1–2, 1996.
- [21] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.
- [22] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.
- [23] M. Patil and R. Shah. Similarity joins for uncertain strings. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1471–1482. ACM, 2014.
- [24] C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 886–895. IEEE, 2007.
- [25] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. Hambrusch. Indexing uncertain categorical data. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 616–625. IEEE, 2007.
- [26] Y. Tao, R. Cheng, X. Xiao, W. K. Ngai, B. Kao, and S. Prabhakar. Indexing multi-dimensional uncertain data with arbitrary probability density functions. In *Proceedings of the 31st international conference on Very large data bases*, pages 922–933. VLDB Endowment, 2005.
- [27] S. Thankachan, M. Patil, R. Shah, and S. Biswas. Probabilistic threshold indexing for uncertain strings full version. <http://csc.lsu.edu/~sbiswas/papers/uncertainIndexingFullpaper.pdf>.
- [28] P. Weiner. Linear pattern matching algorithms. In *SWAT (FOCS)*, pages 1–11, 1973.



# Context-aware Event Stream Analytics

Olga Poppe\*, Chuan Lei\*\*, Elke A. Rundensteiner\* and Dan Dougherty\*

\*Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609, USA

\*\*NEC Labs America, 10080 N Wolfe Rd, Cupertino, CA 95014, USA

poppe|rundenst|dd@cs.wpi.edu, chuan@nec-labs.com

## ABSTRACT

Complex event processing is a popular technology for continuously monitoring high-volume event streams from health care to traffic management to detect complex compositions of events. These event compositions signify critical “application contexts” from hygiene violations to traffic accidents. Certain event queries are only appropriate in particular contexts. Yet state-of-the-art streaming engines tend to execute all event queries continuously regardless of the current application context. This wastes tremendous processing resources and thus leads to delayed reactions to critical situations. We have developed the first context-aware event processing solution, called CAESAR, which features the following key innovations. (1) The CAESAR model supports application contexts as first class citizens and associates appropriate event queries with them. (2) The CAESAR optimizer employs context-aware optimization strategies including context window push-down strategy and query workload sharing among overlapping contexts. (3) The CAESAR infrastructure allows for lightweight event query suspension and activation driven by context windows. Our experimental study utilizing both the Linear Road stream benchmark as well as real-world data sets demonstrates that the context-aware event stream analytics consistently outperforms the state-of-the-art strategies by factor of 8 on average.

## 1. INTRODUCTION

Complex Event Processing (CEP) has emerged as a prominent technology for supporting applications from financial fraud [30] to health care [32]. Traditionally, CEP systems consume event streams produced by smart digital devices like sensors and mobile phones and *continuously* evaluate the query workload to monitor the input event streams.

In many stream-based applications, events convey particular *application contexts* such that the system reaction to an event may significantly vary depending on the current context. Therefore, some event queries may only need to be executed under certain circumstances while others can

be safely suspended. The following examples highlight the challenges and opportunities of context-aware event stream processing that have been overlooked in the prior research.

**Motivating Example.** Traffic has both a huge economic and environmental impact on our daily lives. Drivers traveling the 10-worst U.S. traffic corridors annually spend an average of 140 hours idling in traffic [2]. Due to pollution and noise, congestion in the USA’s 83 largest urban areas in 2010 led to a related public health cost of \$18 billion [3]. Further, road traffic injuries caused an estimated 1.24 million deaths worldwide in 2010 [4].

An intelligent traffic control center could reduce these crippling impacts. The center receives vehicle position reports, analyzes them, infers the current situation in the monitored road segments and reacts instantaneously to ensure safe and smooth traffic flow. Early detection and prompt reaction to critical situations are eminently important. They prevent time and fuel waste, reduce pollution, avoid property damage and in some cases even save human lives.

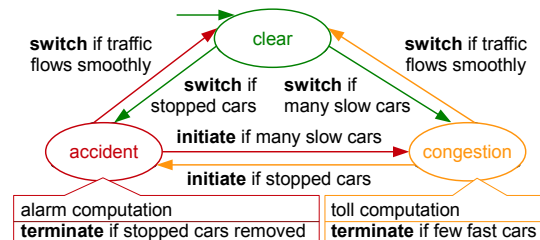


Figure 1: CAESAR model of traffic management

System reaction to a position report should thus be modulated depending on the *current situation on the road* (here referred to as context<sup>1</sup>). Indeed, if an *accident* is detected, all vehicles downstream should be warned and possibly alternative routes should be suggested (Figure 1). If a road segment becomes *congested*, drivers may be charged toll to discourage them from driving to control smooth traffic flow. If a road segment is *clear*, none of the above actions should take place. Clearly, *current application contexts* must be rapidly detected and continuously maintained to determine appropriate reactions of the system at all times.

Conditions implying an *application context* can be complex. They are specified on both the event streams and the current contexts. For example, if over 50 cars per minute

<sup>1</sup>Here we utilize the term *application context* to refer to the state of a sub-network such as *accident*, *congestion*, etc. We deliberately avoid using the notion *state* since it is a too overloaded term in the CEP literature.

move with an average speed less than 40 mph and the current context is no *congestion* then the *context deriving query* updates the context to *congestion* for this road segment. To save resources and thus to ensure prompt system responsiveness, such complex context detection should happen once. Its results must be available on-time and shared among all queries that belong to the detected context. In other words, *context processing* queries are *dependent* on the results of *context deriving* queries and a mechanism ensuring their correct execution must be employed.

The system responsiveness can be substantially improved by exploiting the optimization opportunities enabled by the application contexts. (1) Only those event queries that are relevant in the current contexts should be executed. All irrelevant computations should be suspended. (2) Workloads of overlapping contexts should be shared. Furthermore, application contexts break the application semantics into modules that facilitate the modular development and runtime maintenance of an event stream processing application.

**Challenges.** To enable such event stream processing applications, the following challenges must be tackled:

*Context-aware specification model.* As motivated above, event stream processing applications need to express rich semantics. In particular, they have to specify application contexts as first class citizens and enable linkage of appropriate event queries to their respective context. Furthermore, this model must be in a convenient human-readable format to facilitate on-the-fly reconfiguration, easy maintenance and avoid fatal specification mistakes.

*Context-exploiting optimization techniques.* To meet the demanding latency constraints of time-critical applications, this powerful context-aware application model must be translated into an efficient physical query plan. This query plan must be optimized by exploiting the optimization opportunities enabled by context-aware event stream analytics. This is complicated by the fact that the duration of a context is unknown at compile time and potentially unbounded.

*Context-driven execution infrastructure.* An efficient runtime execution infrastructure is required to support multiple concurrent contexts. To ensure correct query execution, the inter-dependencies between complex context deriving and context processing queries must be taken into account.

**State-of-the-Art.** The challenges described above have so far not been addressed in a comprehensive fashion.

Since the duration of a context varies, state-of-the-art window semantics such as fixed-length tumbling and sliding windows [22, 8] are inadequate to model the proposed notion of a context. Classical predicate windows [15] have variable duration. However, conditions leading to an application context can be rather complex and thus resource-consuming, worse yet they can be dependent on the previous contexts (Figure 1). Since predicate windows are independent from each other, they fail to express context windows.

While some event query languages (e.g., CQL [10], SASE [5, 34]) could be used to hard-code the equivalent of a context construct by queries that detect the context bounds. However, this approach is cumbersome and error-prone – requiring the careful specification of multiple complex inter-dependent event queries [25]. Furthermore, no optimization techniques have been developed to exploit the benefits of context-awareness such as suspension of irrelevant event queries nor the sharing workloads of overlapping contexts.

Business models [16, 28] focus on powerful modeling con-

structs to capture the semantics of processes and in that sense express application contexts. However, these models, targeting business process specification, were not designed for event stream processing. Thus, they neglect its core peculiarities such as the event-driven nature of context detection achieving high performance analytics and the importance of temporal windows and their processing techniques.

**The Proposed CAESAR Approach.** In [25], we formally defined the first context-aware event query processing model for which we now design the Context-Aware Event Stream Analytics in Real time system, CAESAR for short.

Our CAESAR model supports *context windows* as first-class citizens and associates appropriate event queries with each context window. Event queries that process events within a context are called *context processing queries*. Event queries that derive a context are called *context deriving queries*. Both types of queries operate within *context windows*, a new class of event query window we define.

To achieve near real-time system responsiveness, the CAESAR model is transformed into a stream query plan composed of context-aware operators of the *CAESAR algebra*. This algebra serves as foundation for the CAESAR optimizer. The optimizer exploits the notion of context windows to avoid unnecessary computations by suspending those operators which are irrelevant to the current context. Furthermore, the optimizer saves computations by sharing workloads of overlapping context windows. Finally, we built the *CAESAR runtime infrastructure* for correct yet efficient execution of inter-dependent context-aware event queries.

**Contributions** can be summarized as follows:

1) We introduce a new notion of windows, called *context windows*, to enable context-aware event query processing critical to modeling event-based systems. The proposed human-readable context-aware CAESAR model significantly simplifies the specification of rich event-driven application semantics by explicit support of context windows<sup>2</sup>. It also opens new multi-query optimization opportunities by associating appropriate event queries with each context.

2) We define the *CAESAR algebra* for our context-aware event query processing. The *CAESAR optimizer* pushes the context windows down to suspend the execution of irrelevant operators. Furthermore, we propose the context window grouping algorithm that exploits the sharing opportunities from workloads of overlapping context windows.

3) We built the *CAESAR runtime execution infrastructure* that guarantees correct and efficient execution of inter-dependent context deriving and context processing queries.

4) We evaluate the performance of the CAESAR system and its optimization strategies using the Linear Road stream benchmark [9] as well as the real world data set [26]. Our CAESAR system performs on average 8-fold faster than the context-independent solution for a wide range of cases.

**Outline.** We start with preliminaries in Section 2 and introduce the CAESAR model in Section 3. We present our algebraic execution paradigm in Section 4 and its optimization techniques in Section 5. Section 6 is devoted to the runtime execution infrastructure. We conduct the performance study in Section 7. Related work is discussed in Section 8, and Section 9 concludes the article.

<sup>2</sup>Visual editor for the CAESAR model and its evaluation, out of the scope of this article, are subjects for future research. In [25] we compare our model to a set of CQL event queries.

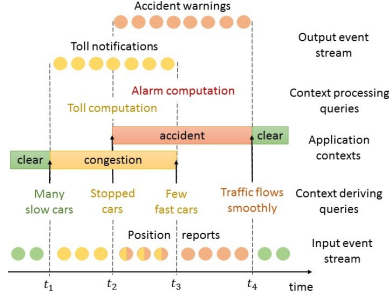


Figure 2: Key concepts of the CAESAR model

## 2. PRELIMINARIES

**Time.** Time is represented by a linearly ordered set of time points  $(\mathbb{T}, \leq)$ , where  $\mathbb{T} \subseteq \mathbb{Q}^+$  and  $\mathbb{Q}^+$  denotes the set of non-negative rational numbers. The set of time intervals is  $\mathbb{TI} = \{[start, end] \mid start \in \mathbb{T}, end \in \mathbb{T}, start \leq end\}$ . For a time point  $t \in \mathbb{T}$  and an interval  $w \in \mathbb{TI}$  we say that  $t$  is within  $w$ , denoted  $t \sqsubseteq w$ , if  $w.start \leq t \leq w.end$ .

**Event.** An event is a message indicating that something of interest happens in the real world. Each event  $e$  belongs to a particular event type  $E$ , denoted  $e.type = E$ . An event type  $E$  is defined by a schema which specifies the set of event attributes and the domains of their values. An event  $e$  has an occurrence time  $e.time \in \mathbb{T}$  assigned by the event source. For example, a vehicle position report in [9] has the following attributes: expressway, direction, segment, car identifier etc. The values of these attributes are integers.

**Event Stream.** Events can be simple or complex. Simple events are sent by event producers (e.g., sensors) to event consumers (e.g., a traffic control center) on an input event stream  $I$  to be processed to derive higher-level complex events. The occurrence time of a complex event comprises the occurrence time of all events it was derived from [23].

## 3. CAESAR MODEL

### 3.1 Key Concepts of the CAESAR Model

**Application contexts** are real-world higher-order situations the duration of which is not known at their detection time and potentially unbounded. This differentiates contexts from events. The duration of a context is called a *context window*. For example, *congestion* is a higher-order situation in the traffic control application (Figure 2). Its bounds are detected based on position reports sent from cars in the same road segment in the same time period. As long as a road segment remains congested, the context window *congestion* is said to hold. Hence, the duration of a context window cannot be predetermined.

**Context deriving queries** associated with a particular context determine when this context should be terminated and when a particular other context is to be initiated based on events. For example, two context transitions are possible from the *congestion* context. If the number of cars reduces and they start moving at higher speed the system transitions into the *clear* context. If two cars stop in the same location and at the same time the *accident* context is activated.

**Context processing queries** correspond to the workload associated with a particular context, i.e., the analytics to be computed based on events received while the system remains in this context. For example, cars entering a con-

gested road segment are charged toll to discourage drivers from driving during rush hours.

### 3.2 Benefits of Context-Awareness Property

**Event Query Relevance.** At each point of time, a context window re-targets all efforts of the system to the current situation by activating only those event queries (both context deriving and context processing queries) which belong to one of the currently active application contexts. All other event queries are suspended as they are irrelevant within the current contexts. This saves both CPU and memory resources. For example, toll is charged only during *congestion* on a road. This query is neither relevant in the *clear* nor in the *accident* contexts. Thus, it is evaluated only during *congestion* and suspended in all other contexts.

**Event Query Simplification.** The concept of an application context provides event queries with *situational knowledge* that allows us to specify simpler event queries. For example, if the event query computing toll is evaluated only during the *congestion* context, the complex conditions that determine that there is a traffic jam on the road are already implied by the context. Thus, there is no need to repeatedly double-check them in each of the active workload queries.

**Context Derivation.** The task of context derivation is the dedicated responsibility of the context deriving queries. For example, once too many slow cars in a road segment are detected, the context *congestion* is activated. Thereafter, the query detecting *congestion* is no longer evaluated. Also, all event queries that are evaluated during *congestion* leverage the insight detected by the context deriving query rather than re-evaluating the *congestion* condition over and over at each individual event query level.

### 3.3 Context Window

*Definition 1. (Context type and Context window.)*

A context type is defined by a name  $c$  and a workload of context deriving queries  $Q_d^c$  and context processing queries  $Q_p^c$  which are appropriate in this context.

Let  $C$  be the set of context types. Then, a context window  $w_c$  is defined by a type  $c \in C$  and a duration  $(t_i, t_t) \in \mathbb{TI}$  where  $t_i$  is the time point when a query  $q_i \in Q_d^{c'}$  matched the event stream and thus  $w_c$  got initiated and  $t_t$  is the time point when a query  $q_t \in Q_d^c$  matched the event stream and thus  $w_c$  got terminated where  $c' \in C$ .

Context windows of different types may overlap. Indeed, there can be a *congestion* and an *accident* in the same road segment at the same time such that two sets of event queries handling both situations must be executed concurrently.

*Definition 2. (Context window relationships.)*

Context windows of type  $c_1$  and  $c_2$  are guaranteed to overlap if based on the predicates of the respective context deriving queries it can be determined that for each window of type  $c_1$  there is a window of type  $c_2$  with  $w_{c_1}.start \sqsubseteq w_{c_2}$ . If in addition  $w_{c_1}.end \sqsubseteq w_{c_2}$  can be determined, a window of type  $c_1$  is contained in a window of type  $c_2$ .

In general, the predicates of the context deriving queries can be analyzed to determine if they imply such conditions. For example, Figure 7 shows the predicates that determine the bounds of the context windows  $w_{c_1}$  and  $w_{c_2}$ . It is easy

to conclude that the windows overlap. CAESAR employs established approaches for predicate subsumption [14].

The same event query can be appropriate in several different application contexts. For example, accident detection happens in both the *clear* and the *congestion* contexts. In contrast to that, the event query detecting accident clearance is executed only in the *accident* context.

For simplicity, we have made two assumptions: (1) Event queries associated with different contexts are independent, meaning that they do not produce events that are consumed by event queries in other contexts. (2) Only one context window of the same type can hold at a time per road segment. If there are multiple accidents in a road segment the context window *accident* holds until all of them are cleared.

### 3.4 Context-aware Event Queries

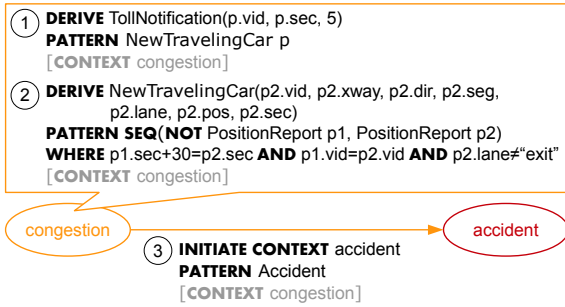


Figure 3: Context-aware event queries

The two application contexts, *congestion* and *accident*, are shown in Figure 3. Different event queries are appropriate within them. For compactness, only three of them within the *congestion* context are shown. Clauses in square brackets are optional since they are implied by the model. The CAESAR event query language grammar is defined in Figure 4.

**Definition 3. (Context-aware event queries.)**

A context-aware event query consists of several clauses.

Each clause performs one of the following tasks:

- Context initiation (INITIATE CONTEXT clause).
- Context switch (SWITCH CONTEXT clause).
- Context termination (TERMINATE CONTEXT clause).
- Complex event derivation (DERIVE clause).
- Event pattern matching (PATTERN clause).
- Event filtering (WHERE clause).
- Context window specification (CONTEXT clause).

Context deriving queries perform three actions: (1) initiate a new context window  $w_c$ , (2) terminate an existing context window  $w_c$ , or (3) switch from the current context window  $w_{c_1}$  into a new context window  $w_{c_2}$ .

Context initiation and termination can be used to express overlapping context windows. For example, *accident* and *congestion* may overlap. That is, query 3 initiates the context window *accident* when an accident is detected (Figure 3). However, query 3 does not terminate the context window *congestion*. The event queries that detect accidents are not shown for compactness.

In contrast, context switch expresses a sequence of two non-overlapping context windows. It corresponds to the termination of the previous context window  $w_{c_1}$  and the initiation of the new context window  $w_{c_2}$ . For example, the *clear* context overlaps neither *accident* nor *congestion* contexts.

Context processing queries analyze the stream of simple or complex events to derive higher-level knowledge in form of complex events. For example, query 2 detects the cars entering a congested road segment. These are vehicles which are not on an exit lane and for which there is no previous position report from the same road segment within 30 seconds. Query 1 derives toll notifications for such vehicles.

Both context deriving and context processing queries consume events that arrive during the context windows that these queries are associated with. Hence, both types of queries utilize event pattern matching and event filtering clauses which are commonly used in event queries [34, 23]. Section 4.1 defines when these clauses match.

<i>Query</i>	$:= \langle Window \rangle \mid \langle Retrieval \rangle$
<i>Window</i>	$:= (\text{INITIATE} \mid \text{SWITCH} \mid \text{TERMINATE})$ $\text{CONTEXT } Context$
<i>Retrieval</i>	$:= \langle Derive \rangle \langle Pattern \rangle \langle Where \rangle? \langle Context \rangle$
<i>Derive</i>	$:= \text{DERIVE } EventType ((Var.)? Attr, ?)^+$
<i>Pattern</i>	$:= \text{PATTERN } \langle Patt \rangle$
<i>Where</i>	$:= \text{WHERE } \langle Expr \rangle$
<i>Context</i>	$:= \text{CONTEXT } (Context, ?)^+$
<i>Patt</i>	$:= \text{NOT? } EventType Var? \mid \text{SEQ}(\langle Patt \rangle, ?)^+$
<i>Expr</i>	$:= Constant \mid Attr \mid \langle Expr \rangle \langle Op \rangle \langle Expr \rangle$
<i>Op</i>	$:= + \mid - \mid / \mid * \mid \% \mid = \mid \neq \mid > \mid \geq \mid < \mid \leq \mid \text{AND} \mid \text{OR}$

Figure 4: CAESAR event query language grammar

Putting the application contexts, transitions between them (Definitions 1 and 2) and context-aware event queries (Definition 3) together, we now define the CAESAR model.

**Definition 4. (CAESAR model.)** A CAESAR model is a tuple  $(I, O, C, c_d)$  where  $I$  and  $O$  are unbounded input and output event streams and  $C$  is a finite set of context types with the default context type  $c_d \in C$ .

While the goal of classical automata is to define a language, the CAESAR model is designed for context-aware event query execution. Thus, final contexts are omitted. The CAESAR model has a default context that holds when no other context does, e.g., at the system startup (the *clear* context in our example). The runtime processing of the model is defined in Section 4.1.

## 4. CAESAR ALGEBRA

The CAESAR model explicitly supports application contexts and the transition network to facilitate context-aware event query specification (Figure 3). However, at execution level an algebraic query plan tends to be easier to optimize than an automaton-based model [30].<sup>3</sup> We thus define the CAESAR algebra and the translation rules of the CAESAR model into an algebraic query plan.

### 4.1 CAESAR Operators

The CAESAR algebra consists of six operators. While event pattern, filter and projection are quite common for other stream algebras [30], [34], context initiation, termination and context window are unique operators of the CAESAR algebra. Context initiation and termination consume a stream  $I$  of events produced by other operators of the context deriving queries and the set of current context windows

<sup>3</sup>There are approaches to optimization and distribution of simpler automata than the CAESAR model however. We describe them in detail in Section 8.

$W$ . They update the set of the current context windows and return the updated set.

**Context initiation**  $CI_c$  starts a new context window  $w_c$ , adds it to the set of current context windows and removes the default context window  $w_{c_d}$  from the set, if there.

$CI_c(I, W) := \{W' \mid \text{if } w_c \in W \text{ then } W' = W. \text{ Otherwise } W' = W \cup w_c \text{ and if } w_{c_d} \in W \text{ then } W' = W' / w_{c_d} \text{ where } e \in I \text{ and } e.time = w_{c_d}.end = w_c.start\}$ .

**Context termination**  $CT_c$  ends the context window  $w_c$ , removes it from the set of current context windows, if the set becomes empty adds the default context window  $w_{c_d}$  to it.  $CT_c(I, W) := \{W' \mid \text{If } |W| > 1 \text{ then } W' = W / w_c \text{ else } W' = \{w_{c_d}\} \text{ where } e \in I \text{ and } e.time = w_c.end = w_{c_d}.start\}$ .

**Context window**  $CW_c$  consumes an event stream  $I$  and the set of all current context windows  $W$  and returns the stream of events that occur during the current context window  $w_c$ .  $CW_c(I, W) := \{e \mid w_c \in W, e \in I, e.time \sqsubseteq w_c\}$ .

**Filter**  $FI_\theta$  with a predicate  $\theta$  consumes an event stream  $I$  and returns a stream composed of all events that satisfy  $\theta$ .  $FI_\theta(I) := \{e \mid e \in I, e \text{ satisfies } \theta\}$ .

**Projection**  $PR_{A,E}$  with a set of attributes  $A$  and an event type  $E$  consumes an event stream  $I$ , restricts each input event to the set of attributes  $A$  and returns the stream of these restricted events of type  $E$ .

$PR_{A,E}(I) := \{e \mid e.type = E, e' \in I, e.a = e'.a \text{ for } a \in A\}$ .

**Pattern**  $P$  consumes an event stream  $I$  and constructs event sequences matched by the pattern  $P$ . For each event sequence, the operator outputs an event consisting of the attribute values of all events in the sequence. Let  $A$  be the set of all attributes of events of types  $E_1, \dots, E_n$  and  $1 \leq i \leq n$ . The pattern  $P$  is one of the following:

1) Event matching  $E$  returns input events of type  $E$ .

$E(I) := \{e \mid e \in I, e.type = E\}$ .

2) Sequence without negation  $SEQ_{(E_1, \dots, E_n)}$  constructs sequences of  $n$  events such that an  $i^{th}$  event is of type  $E_i$ .

$SEQ_{(E_1, \dots, E_n)}(I) := \{e \mid e_1, \dots, e_n \in I, e_1.type = E_1, \dots, e_n.type = E_n, e_1.time < \dots < e_n.time, e.a = e_i.a \text{ for } a \in A, e.time = [e_1.time, e_n.time]\}$ .

3) Sequence with negation  $SEQ_{(S_1, NOT E, S_2)}$  constructs event sequences  $SEQ_{S_1, S_2}$  without negation such that there is no event of type  $E$  between the sub-sequences constructed by  $SEQ_{S_1}$  and  $SEQ_{S_2}$ .

$SEQ_{(S_1, NOT E, S_2)}(I) := \{e \mid e_1, \dots, e_m \in SEQ_{S_1}, e_{m+1}, \dots, e_n \in SEQ_{S_2}, \nexists e' \in I \text{ with } e'.type = E, e_m.time < e'.time < e_{m+1}.time, e.a = e_i.a \text{ for } a \in A, e.time = [e_1.time, e_n.time]\}$ . A negated event can start or end an event sequence. In this case, temporal constraints must define the time interval within which the negated event may not occur [34].

## 4.2 Context-preserving Plan Generation

At compile time, the CAESAR model (Figure 3) is translated into an executable query plan that is then input to the CAESAR optimizer (Section 5). The CAESAR model translation happens in two phases (Figure 5). They are:

**Phase 1: CAESAR model to a query set.** First, the model is translated into a machine-readable query set. During this phase, contexts that are implied by the CAESAR model (the optional clauses in square brackets in Figure 3) become mandatory clauses of the CAESAR event queries. As a result, an event query that belongs to a context  $c$  has a mandatory clause  $CONTEXT c$ . For example, all queries in Figure 5 explicitly specify the context they belong to.

**Phase 2: Query set to a combined query plan.** Dur-

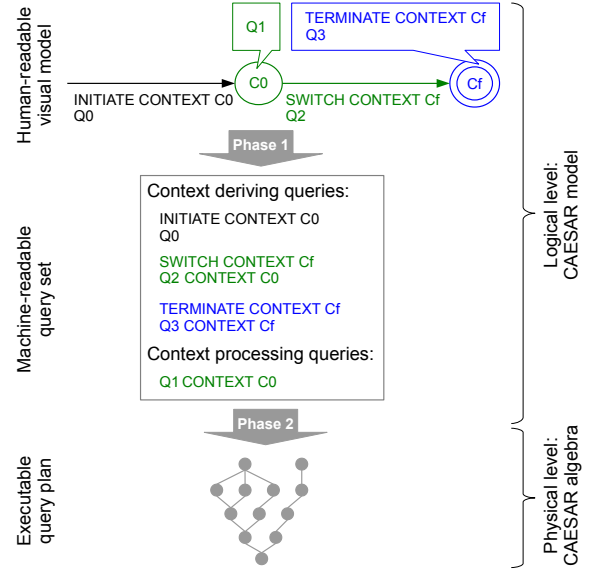


Figure 5: CAESAR model translation

Event query clause	Operator
INITIATE CONTEXT $c$	$CI_c$
SWITCH CONTEXT $c$	$CI_c, CT_{curr}$
TERMINATE CONTEXT $c$	$CT_c$
DERIVE $E(A)$	$PR_{A,E}$
PATTERN $P$	$P$
WHERE $\theta$	$FI_\theta$
CONTEXT $c$	$CW_c$

Table 1: Individual query plan construction

ing this phase, the machine-readable query set is translated into an executable query plan. This happens in two steps:

1) *Individual query plan construction.* Each event query is translated into a sequence of algebra operators such that each clause of the query corresponds to a set of operators as defined in Table 1.  $curr$  denotes the current context the context deriving query is associated with.

2) *Combined query plan construction.* Individual query plans are composed into a combined query plan such that if one query plan produces events which are consumed by another query plan then the output of the first plan is the input of the second plan. Since event queries in different contexts are independent (Section 3.3), all event queries in a combined query plan belong to the same context.

For example, queries 1 and 2 in Figure 3 are translated into the combined query plan in Figure 6(a). Query 1 is translated into the individual query plan consisting of operators 1–4. Query 2 corresponds to the individual query plan composed of operators 5–7. Since the first query plan produces complex events consumed by the second query plan, they are composed into a single combined query plan.

## 5. CAESAR OPTIMIZATION

### 5.1 CAESAR Optimization Problem Statement

*Definition 5.* Given a workload of context-aware event queries where each query is associated with an application

context. Our CAESAR optimization problem is to find an optimized query plan for all queries such that the CPU costs are minimized by suspending event queries that are irrelevant to the current application contexts and sharing the workload of overlapping context windows.

To avoid reinventing the wheel, we borrow the CPU cost estimation of event pattern construction from [24], and thus do not repeat here. Instead, we now discuss the cost of the context-specific operators. We maintain the information about current context windows in the *context bit vector*  $W$  with one bit for each context type. Since the number of possible context types for an application is predefined and constant, the size of the vector is also constant. The context initiation and termination operators update one bit and the time stamp of the context bit vector. Context windows look up one value in the vector to determine whether a context window of a certain type currently holds. In other words, the CPU cost of these operators is constant. Section 6 provides further implementation details.

## 5.2 Context Window Push Down

Since some operators of the CAESAR algebra are similar to other stream algebras, existing approaches, from operator reordering [24] to operator merging [30, 6], can be exploited by the CAESAR optimizer as well. For example, projections and filters can be executed in any order assuming that a projection that is pushed down below a filter discards no attributes accessed by the filter. Adjacent filters can be merged into a single filter by combining their predicates. However, these existing techniques are oblivious to the notion of contexts, and consequently do not avoid superfluous computations in the current contexts.

To avoid unnecessary computations when event queries are executed “out” of their respective context windows, we introduce the context window push-down strategy. Context window push-down can prevent the continuous execution of operators in the query plan. In other words, no event will be passed up by the context window operator if the current event stream does not qualify for the context window.

For instance, the two bottom most operators in Figure 6(a) are always executed regardless of the application contexts. Once the context window is pushed down to the bottom in Figure 6(b), it avoids the execution of all operators higher in the plan when they are irrelevant to the current contexts.

All event queries in a combined query plan belong to the same context (Section 4.2). By definition, a context window specifies the scope of its queries. Thus, pushing a context window down does not change the semantics of its queries. That is, context window push down strategy is correct.

Pushing a context window down seems similar to pushing a predicate or a traditional window down only at first sight. Differences are twofold:

- 1) Context windows *suspend* the entire query plan “above them” as long as the application is in different contexts. In contrast to that, a predicate or a traditional window is a filter on a stream that selects certain events to be passed through. It does not control the suspension of the above operators and keeps them in *busy waiting* state that wastes valuable resources and degrades system performance.
- 2) Our context-driven stream router directs *event stream portions during application contexts* to the appropriate event queries (Section 6.2). In contrast to that, predicates and traditional time constraints (e.g., event sequence within 4

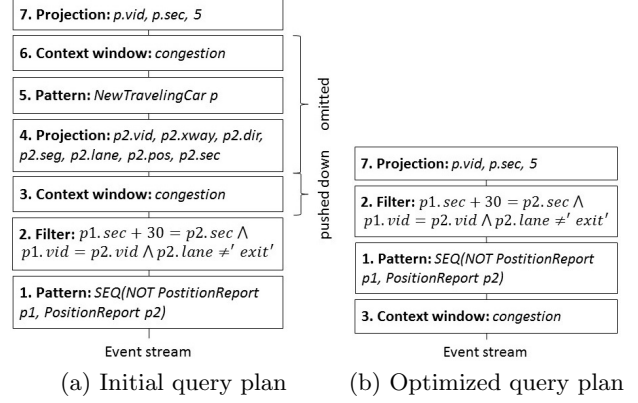


Figure 6: Query plans

hours [34]) typically filter events one by one at the *individual event level*. This is a resource consuming slow process.

Theorem 1 proves that pushing a context window down in a combined query plan leads to lower execution costs than placing it at any other position in the query plan.

**THEOREM 1.** *Given a query  $q$ , let  $P$  be the set of all possible query plans for  $q$ ,  $p \in P$  be a query plan for  $q$  and  $cost(p)$  be the cost of executing the plan  $p$ . With the context window pushed down, the new plan, denoted by  $p'$ , has cost  $cost(p')$ . Then  $\forall p \in P, p \neq p', cost(p') \leq cost(p)$ .*

**PROOF.** As described in Section 5.1, the cost of the context window operator is constant. That is, it adds constant cost to the overall execution costs of a query plan no matter its position in the query plan. The context window operator completely suspends the execution of all upstream operators while the context is not active. In that case, the cost of a query plan is reduced, i.e.,  $cost(p) < cost(p')$ . In an unlikely case when the context window happens to be always active, the costs of the query plans  $p$  and  $p'$  are equal.  $\square$

## 5.3 Context Workload Sharing

Inspired by traditional multi-query optimization, grouping the event queries enables computation sharing among these queries. We observe the opportunity that substantial computational savings can be achieved by executing only one instance of each context deriving query for each context. Without context window grouping, each context processing query has to run its respective context deriving queries separately so to determine its current context to ensure correct execution. If this context analytics is performed on an individual query level, significant computational resources are wasted and system responsiveness suffers.

Sharing query workloads of overlapping context windows is challenging for the following reasons: (1) The duration of context windows may vary and their bounds are unknown at compile time. So, we have to infer whether context windows overlap. (2) Different contexts may contain identical or similar event queries in their query workloads. How to share query workloads driven by contexts is crucial to achieve high performance execution. We now propose an efficient solution that addresses this problem by splitting and grouping overlapping context windows.

For overlapping context windows, a naive solution would be to merge these context windows to form a larger encompassing context window. Inside this large context window,

we can now analyze the associated event query workloads to optimize their executions. However, such solution could do more harm than good in some cases. For example, if all context windows were to be overlapping, then only one huge all encompassing context window would be formed as a result. This would forfeit the purpose of being context-aware. Consequently, redundant computations would be incurred.

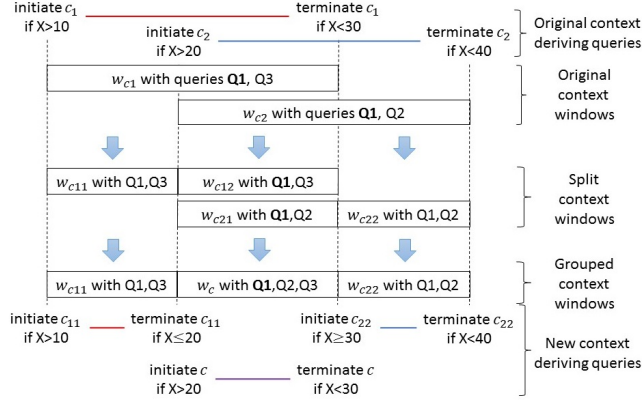


Figure 7: Context Window Grouping

We now propose an effective strategy for splitting the original user-defined *overlapping* context windows into finer granularity context windows and grouping them into *non-overlapping* context windows. For example, the context windows  $w_{c_1}$  and  $w_{c_2}$  in Figure 7 overlap. The window  $w_{c_1}$  is split into  $w_{11}$  and  $w_{12}$ , while the window  $w_{c_2}$  is split into  $w_{21}$  and  $w_{22}$ . Since the windows  $w_{12}$  and  $w_{21}$  cover the same time interval, the event queries associated with them are merged to form the workload of the *grouped context window*  $w$ . The context deriving queries are adjusted accordingly.

```

1 Input: Set  $W$  of user-defined context windows. A window is
   described by  $start, end$  and  $queries$ .
3 Output: Set  $G$  of grouped context windows
 $G = W.extractNonOverlappingWindows()$ 
5  $W = W.sortByStart()$ 
 $W = W.mergeIdenticalWindows()$ 
7  $Q = \emptyset$ 
while  $W.hasNextWindowBound()$ 
9    $next = W.getNextWindowBound()$ 
    $S = W.getStartingWindows(next)$ 
    $E = W.getEndingWindows(next)$ 
11  if  $Q.is Empty()$ 
13    then  $Q = S.queries$ 
   else new window  $w = (previous, next, Q)$ 
15     $G = G \cup w$ 
    $Q = Q - E.queries \cup S.queries$ 
17  end if
    $previous = next$ 
19 end while
for each  $w \in G$ 
21    $w.queries = w.queries.dropDuplicates()$ 
end for each
23 return  $G$ 

```

Listing 1: Context window grouping algorithm

Consider our context window grouping algorithm in Listing 1. It takes a set of user-defined context windows as input. Context windows which do not overlap any other window remain unchanged (line 4). The algorithm sorts the overlapping context windows in increasing order by start time (line 5). Even though the exact start time of context

windows is not known at compile time, the order of their beginning can be determined for overlapping context windows. For example, it is known at compile time that the window  $w_{c_2}$  in Figure 7 will start at the same time or later than the window  $w_{c_1}$ . If there are several identical context windows, the algorithm only keeps one by merging the workloads of identical windows (line 6). The core of the algorithm (lines 8-19) forms a new grouped context window for each time interval between two subsequent bounds of original context windows and associates the query workload with it that is appropriate during this time interval. For each grouped context window, the algorithm deletes duplicate event queries (lines 20-22) and returns the set of grouped context windows (line 23). Since several subsequent grouped context windows correspond to one original context window, an event query within a grouped context window may need access to its partial matches in the previous grouped context windows to ensure completeness of its results. In Section 6, we introduce a customized design (called *context history*) to ensure correct grouped context window execution.

The time complexity of the algorithm is  $O(n \log(n) * m)$  where  $n$  is the number of original user-defined context types that are sorted (line 5) and  $m$  is the number of predicates that have to be analyzed while comparing two context types.

Based on the newly produced non-overlapping context windows, we now further exploit traditional multi-query optimization (MQO) techniques [31, 27, 21] to produce an optimized shared query execution plan for each group of event queries. This opens opportunities to share the similar workload within a context which further saves computational costs and reduces query latency. MQO is an NP-Hard problem due to the exponential search space. Thus, the solutions [31, 27, 21] of MQO tend to be expensive.

Our context window grouping solution divides the event query workloads into smaller groups based on their time overlap. Hence, the search space for an optimal query plan within each group is substantially reduced compared to the global space. Any state-of-the-art MQO solution can leverage this idea to return an optimized query plan efficiently.

The search space for multi-query optimization is doubly exponential in the size of the queries ( $n$ ). The spectrum of possible multi-query groupings ranges from a separate group per each *individual* query (i.e., non-sharing) in the given query workload to a single group for *all queries*. The upper-bound for all possible multi-query groups corresponds to the number of distinct ways of assigning  $n$  event queries to one or more groups. The number that describes this value is the Bell number  $B_n$ , which represents the number of different groupings of a set of  $n$  elements. The Bell number is the sum of Stirling numbers. A Stirling number  $S(n, k)$  is the number of ways to partition  $n$  elements into  $k$  partitions [20]:

$$B_n = \sum_{k=1}^n S(n, k) = \sum_{k=1}^n \left( \frac{1}{k!} \sum_{j=1}^k (-1)^{k-j} \binom{n}{k} j^n \right)$$

By dividing our  $n$  queries first into  $m$  groups, we subsequently only need to optimize the small shared set one by one and thus reduce the search space to:

$$B'_n = \sum_{k'=1}^{n/m} S(n/m, k') = \sum_{k'=1}^{n/m} \left( \frac{1}{k'!} \sum_{j=1}^{k'} (-1)^{k'-j} \binom{n/m}{k'} j^{n/m} \right)$$

As confirmed by our experimental study in Section 7.2,

the CAESAR optimizer produces a context-aware query plan 2712 times faster than the state-of-the-art context-independent multi-query optimization approaches.

## 6. CAESAR EXECUTION INFRASTRUCTURE

### 6.1 Overview of the CAESAR Infrastructure

Figure 8 shows the CAESAR execution infrastructure. The boxes represent the system components.

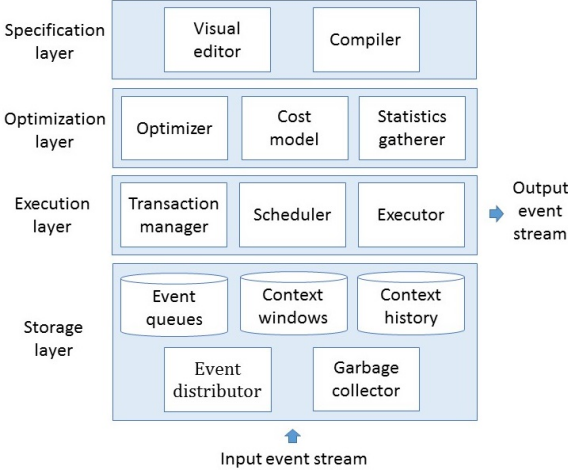


Figure 8: CAESAR infrastructure

**Specification Layer.** A CAESAR model is specified by the application designer using the visual CAESAR editor [25]. As explained in Section 4, we then translate it into an algebraic query plan.

**Optimization Layer.** As described in Section 5, the query plan is optimized using several context-aware optimization strategies to produce an execution plan.

**Execution Layer.** The optimized query plan is forwarded to the transaction manager that forms transactions. These transactions are submitted for execution by the scheduler that guarantees correctness. These components build the core of the CAESAR execution infrastructure. They are described in details in Section 6.2.

**Storage Layer.** The event distributor buffers the incoming events in the event queues. The current context windows and the context history are compactly maintained in-memory. The garbage collector ensures that only the values which are relevant to the current contexts are kept.

### 6.2 Core of the CAESAR Infrastructure

The core of the CAESAR execution infrastructure consists of the context derivation, context processing, context-aware stream routing and scheduling of these processes (Figure 9).

**Context Derivation.** For each stream partition (unidirectional road segment in the traffic management use case), we save which context windows currently hold in the context bit vector  $W$ . This vector  $W$  has a time stamp  $W.time$  and a one-bit entry for each context type, i.e.,  $W.size = |C|$ . The entries are sorted alphabetically by context names to allow for constant time access. The entry 1 (0) for a context  $c$  means that the context window  $w_c$  holds (does not hold) at the time  $W.time$ . Since context windows may overlap, multiple entries in the vector may be set to 1.

The context window vector is updated by the context deriving queries. Since *each* event in the stream can potentially

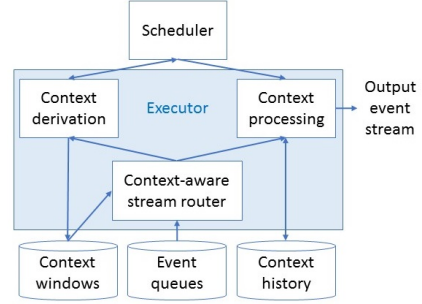


Figure 9: Core of the CAESAR infrastructure

update a context window, the context deriving queries processes all input events.  $W.time$  is the application time when the vector  $W$  was last updated. Since events arrive in-order by time stamps, only one most recent version of the context bit vector is kept.

**Context-aware Stream Routing.** Based on the context window vector, the system is aware of the currently active event query workloads. For each current context window  $w_c$ , it routes all its events to the query plan associated with the context  $c$ . Query plans of all currently inactive context windows do not receive any input. They are suspended to avoid busy waiting, i.e., waste of resources.

Context-aware stream routing is a light-weight process for the following reasons. First, the lookup of all vector entries set to 1 takes constant time. Second, this routing happens for stream batches (multiple subsequent events in the input event stream) rather than for single events.

**Context Processing.** The CAESAR model allows the application designer to specify the *scope* of event queries in terms of their context windows. When a user-defined context window ends, all event queries associated with it are suspended and thus will not produce new matches until they become activated again. Therefore, their partial matches, called *context history*, can be safely discarded.

When a user-defined context window  $w_c$  with its associated query workload  $Q^c$  is split into smaller non-overlapping context windows  $w_{c_1}$  and  $w_{c_2}$  partial matches of the queries  $Q^c$  are maintained across these newly grouped windows  $w_{c_1}$  and  $w_{c_2}$  to ensure correctness of these queries  $Q^c$ . Therefore, for each event query we save the grouped context windows across which the results of the query are kept. For example, the event query  $q_1$  in Figure 7 is executed during all 3 grouped context windows. However, when the third window begins, the partial results within the first window expire.

**Correct Context Management.** Context processing queries are dependent on the results of context deriving queries. Due to bursty input streams, network and processing delays context derivation might not happen on time. To avoid race conditions, these inter-dependencies must be taken into account to guarantee correct execution.

We define a *stream transaction* as a sequence of operations that are triggered by all input events with the same time stamp. The application time stamp of a transaction (and all its operations) coincides with the application time stamp of the triggering events. An algorithm for scheduling read and write operations on the shared context data is *correct* if conflicting operations<sup>4</sup> are processed sorted by time stamps.

<sup>4</sup>Two operations on the same value such that at least one of



While existing transaction schedulers can be deployed in the CAESAR system, we now describe a time-driven scheduler. For each time stamp  $t$ , our scheduler waits till the event distributor progress is larger than  $t$  and the context derivation for all transactions with time stamps smaller than  $t$  is completed. Then, the scheduler extracts all events with the time stamp  $t$  from the event queues, wraps their processing into transactions (one transaction per road segment for the traffic control use case) and submits them for execution.

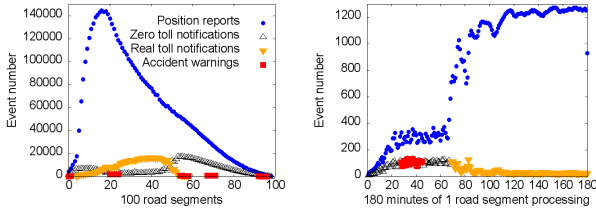
## 7. PERFORMANCE EVALUATION

### 7.1 Experimental Setup and Methodology

**Experimental Infrastructure.** We have implemented our CAESAR system in Java with JRE 1.7.0.25 running on Linux CentOS release 6.3 with 16-core 3.4GHz QEMU Virtual CPU and 48GB of RAM. We execute each experiment three times and report their average results here.

**Linear Road Benchmark.** We have chosen this benchmark [9] to evaluate the effectiveness of the CAESAR system for the following reasons: (1) it expresses a variety of application contexts such that the system reactions to an event depends on the current context, and (2) it is time critical since it poses tight latency constraint of 5 seconds.

**Event Queries.** We focused on a subset of event queries of the benchmark that based on input events derive toll notifications and accident warnings. The queries depicted in Figure 3 are simplified versions of the actual benchmark queries to illustrate the key concepts of our model in the paper only. We simulate low, average and high query workloads by replicating the event queries of the benchmark.



(a) Events per road segment (b) Events per minute

Figure 10: Event streams

**Event Streams.** Event distribution across road segments varies. Figure 10(a) shows the number of processed and derived events per segment of a randomly chosen unidirectional road<sup>7</sup>. There are more cars in some road segments than in others. In some road segments accidents and traffic jams happen more often than in others. Hence, more toll notifications and accident warnings are triggered for them.

Event distribution across time also varies. Event rate gradually increases during 3 hours of an experiment. Figure 10(b) shows the number of processed and derived events per minute by for a randomly chosen unidirectional road segment<sup>7</sup> and visualizes the application contexts. Accident warnings are derived only during accidents (minutes 30-50). The benchmark requires zero toll derivation during accidents and clear road conditions (minutes 0-70). During traffic jams, real toll is computed (minutes 70-180).

**Real Data Set.** In addition to the benchmark, we evaluate our system using the physical activity monitoring real them is a write are called conflicting operations.

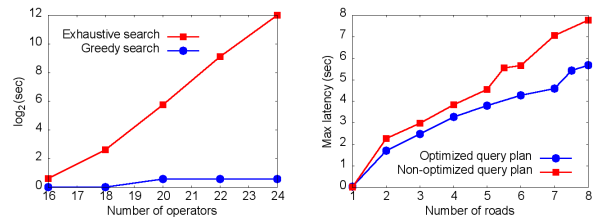
<sup>7</sup>We have observed a similar event distribution for other roads and roads segments.

data set (1.6GB) [26]. It contains physical activity reports from 14 people during 1 hour 15 minutes.

**Metrics.** We measure two metrics common for stream systems, namely *maximal latency* and *scalability*. Additionally, we measure the *win ratio* of context-aware over context independent event stream analytics in terms of CPU processing time. *Maximal latency* is the maximal time interval elapsed from the event arrival time (i.e., system time when a position report is generated) till the complex event derivation time (i.e., system time when a toll notification or an accident warning is derived based on this position report). As mentioned above, the benchmark restricts the query latency to be within 5 seconds. The *system scalability* is determined by the *L-factor*, the maximal number of roads that are processed without violating this constraint. The *win ratio* of context-aware over context-independent event stream analytics is computed as the maximal latency of context-independent processing divided by the maximal latency of context-aware processing of the same event query workload against the same input event stream.

**Methodology.** To show the efficiency of our context-aware query optimization, we compare it to the exhaustive search approach by varying the number of operators in a query plan. To measure the effectiveness of our optimized context-aware query plan, we compare the performance of our context-aware query execution plan to the state-of-the-art solution [34, 5]. To demonstrate the effectiveness of our context-aware workload sharing technique, we compare it to our default non-sharing solution. We conduct these comparisons by varying the following parameters: number of operators in a query plan, input event stream rate, number of event queries, data distribution in a context window, length of a context window, number of context windows, number of overlapping context windows, overlapping ratio of context windows, and the shared workload size. Since context windows are derived from the input events, context window related parameters can be varied only through input data manipulation. Thus, for the experiments on real data set we vary the number of event queries.

### 7.2 Efficiency of CAESAR Optimizer



(a) CAESAR optimizer (b) L-factor

Figure 11: CAESAR Optimization Techniques

In Figure 11(a), we vary the number of operators in a query plan and measure the CPU time required for the query plan search (logarithmic scale on Y-axis). We compare the context-independent (CI) exhaustive to the context-aware (CA) greedy query plan search. As confirmed by the search space analysis (Section 5.3), the processing time of the exhaustive search grows exponentially with the number of operators in a query plan. In contrast, the CPU time required for our context-aware search stays fairly constant while varying the query plan size. At size 24, CAESAR's optimizer is 2712-fold faster than the exhaustive search. This is due to

the fact that our context window push down and context grouping techniques substantially reduce the search space.

### 7.3 Efficiency of CAESAR Runtime

Next, we conduct a comprehensive evaluation to demonstrate the efficiency of our CAESAR solution. The baseline approach is the context-independent processing commonly used in most state-of-the-art solutions [34, 5, 32]. We first demonstrate the superiority of our CAESAR’s context-aware event processing compared to the state-of-the-art context-independent approach by strictly following the constraints of the benchmark. Then we evaluate the CAESAR’s context window sharing technique.

#### 7.3.1 Efficiency of Context-Aware Stream Analytics

**L-factor.** In Figure 11(b), we vary the input stream rate by increasing the number of roads and measure the maximal latency. We compare the latency of the optimized versus non-optimized query plan. As the figure shows, the optimized query plan processes at most 7 roads without violating the latency constraint of 5 seconds. In contrast, the non-optimized query plan can process at most 5 roads under this constraint. Intuitively, our context-aware optimization approach successfully avoids the unnecessary computations by executing different queries only at appropriate time periods (contexts). On the other hand, the state-of-the-art solution suffers from executing all queries all the time.

Next, we compare continuous context-independent stream processing to context-aware solution when some of the involved event query workloads are appropriate only in certain critical contexts and can be suspended in other contexts. Unless stated otherwise, in Figures 12 and 13, we consider 3 roads (1.7GB) and assume that 2 critical non-overlapping context windows of length 3 minutes process 10 event queries each. These queries can be suspended in other contexts.

**Evaluating diverse context window distributions.** In Figure 13, we vary the number of event queries per context window and measure the maximal latency. We compare three setups, namely, uniform context window distribution versus their Poisson distribution with positive skew ( $\lambda$  is the first second) and with negative skew ( $\lambda$  is the last second). Context window bounds vary across different window distributions. The rest of the stream is identical for these setups.

As expected, when the context windows are at the end of the experiment where the stream rate is high, the maximal latency remains almost constant with the growing event query workload. This is explained by the fact that most queries are irrelevant for these contexts and thus are suspended. In contrast, if these windows are uniformly distributed or are at the beginning of the experiment when the stream rate is low, the maximal latency grows linearly with the number of queries. The maximal latency of 20 event queries with uniform context window distribution is 1.8-fold faster than with Poisson distribution with positive skew and 11-fold slower than with Poisson distribution with negative skew. Thus, to achieve fair results we consider uniform context window distribution in all following experiments.

**Scaling event query workload.** In Figure 12(a), we vary the number of event queries per context window and measure the maximal latency of context-aware versus context-independent event stream processing. The maximal latency grows linearly in the number of event queries. For an average workload of 10 event queries, we find that the context-aware

processing is 8-fold faster than the context-independent solution using the Linear Road benchmark data (LR). CAESAR achieves the same win using the Physical Activity Monitoring data set (PAM) and 20 event queries. Our system has a clear win in this case because the context-aware event stream analytics suspends those event queries which are irrelevant to the current context.

**Varying event stream rates.** In Figure 12(b), we vary the number of considered roads. We measure the maximal latency of context-aware versus context-independent processing. The maximal latency grows linearly with an increasing input stream rate (number of roads). For 7 roads, context-aware processing is 9-fold faster than the context independent solution. The results show that the CAESAR system is more robust to the event stream rate increase compared to the context-independent solution.

**Varying context window lengths.** In Figure 12(c), we vary the length of the context windows and measure the win ratio of the context-aware over context-independent processing. The numbers above the bars indicate the percentage of the input event stream covered by the context windows that allow suspension of complex event query workload. Given that CAESAR only keeps one context active at a time, the win ratio exceeds 3 if such context windows cover more than 80% of the stream. It becomes negligible (almost 1) when they cover less than 50% of the input event stream.

**Varying the number of context windows.** A similar trend can be observed while varying the number of context windows that allow suspension of irrelevant event queries (Figure 12(d)). Again, we measure the win ratio of the context-aware over context-independent stream processing. The numbers above the bars indicate the percentage of the input event stream covered by the context windows. Similarly, the win ratio exceeds 2 if the context windows cover more than 80% of the input event stream. It becomes negligible (almost 1) when they cover less than 50%.

#### 7.3.2 Efficiency of Context-Aware Workload Sharing

Next, we measure the effect of the shared workload processing of overlapping context windows (Figure 14). Unless stated otherwise, 30 windows of length 15 minutes each overlap by 10 minutes. Each of them processes 4 event queries.

**Varying the number of overlapping context windows.** In Figure 14(a), we vary the maximal number of overlapping context windows and measure the maximal latency of shared versus non-shared query processing. As expected, the larger the number of overlapping context windows are the more significant is the gain of the event query sharing. If 45 context windows overlap, the workload sharing strategy outperforms the default non-shared solution by factor of 10. The reason is that the CAESAR context window grouping technique exploits the sharing opportunities within overlapping context windows at a fine granularity level by splitting the overlapping context windows into non-overlapping parts and sharing event query processing within them.

**Varying the length of context window overlap.** In Figure 14(b), we vary the minimal length of context window overlap and measure the maximal latency of shared versus non-shared workload execution. The gain of sharing grows linearly with the length of overlap. If 30 context windows overlap by 15 minutes, our workload sharing strategy performs 6-fold faster than the non-shared solution. This is due to the fact that similar workloads can be shared for a longer

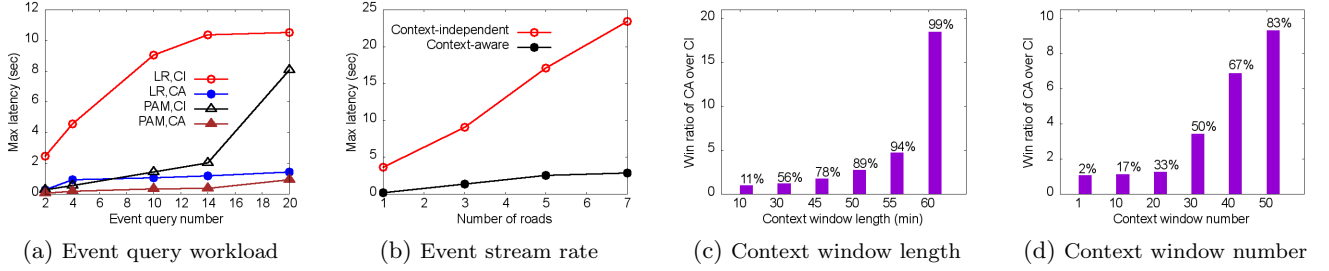


Figure 12: Context-aware event stream analytics

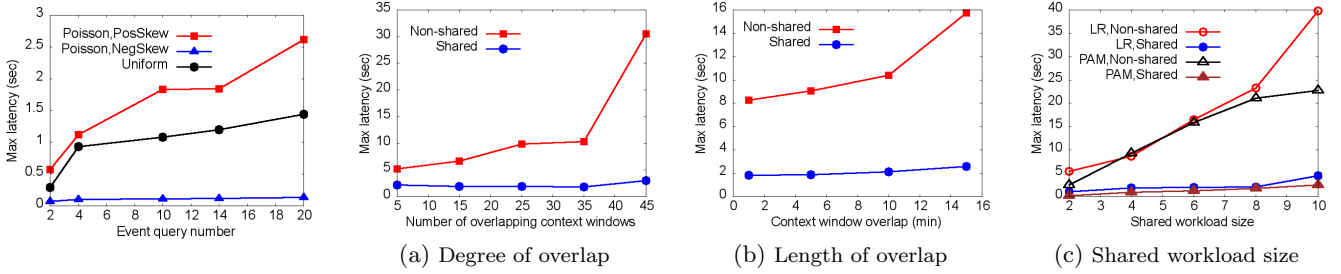


Figure 13: Context window distribution

Figure 14: Shared workload of overlapping context windows

time period, and hence more computational savings can be harvested from the overlapping part of the context windows.

**Shared workload size.** In Figure 14(c), we vary the number of event queries per context window and measure the maximal latency of shared versus non-shared event query processing. As expected, the more event queries can be shared the more significant is the gain of the event query sharing. If each context window contains 10 queries that can be shared with other context windows, the workload sharing strategy outperforms the default non-shared solution by factor of 9 using the Linear Road benchmark data (LR). A similar trend can be observed with the Physical Activity Monitoring data set (PAM).

## 8. RELATED WORK

**Context-aware Event Stream Models** [11, 33] propose a fuzzy ontology to support uncertainty in event queries. Context-aware event queries are rewritten into context-independent and processed in parallel on different stream partitions. Isoyama et al. [18] propose to allocate event queries to event processors so that the state of event processing (i.e., intermediate event query results) is efficiently managed. These ideas are orthogonal to our optimization techniques.

Hermosillo et al. [17] and Proton software prototype [1] feature a model similar to the CAESAR model. However, these approaches lack a formal definition of the event language, optimization techniques and experimental evaluation. The recent emergence of these approaches confirms the importance of the notion of context-aware event queries that has been formally defined by us in [25]. Our current work on CAESAR now is completed by the context-aware optimization strategies and their experimental evaluation.

**Event Stream Processing Automata** [34, 5, 13] continuously evaluate the *same* set of *single* event queries using traditional windows of *fixed* length. These automata capture single query processing states. Their runs corre-

spond to independent event query instances. In contrast to that, the CAESAR model expresses the semantics of the *whole stream-based application* rather than single isolated event queries. Our model captures application contexts of variable, statically unknown duration. Its runs correspond to interdependent processes traveling through application contexts, triggering appropriate context-aware event queries and incrementally maintaining their results.

**Event Query Languages**, like CQL [10] and SASE [5, 34], lack explicit support of application contexts. These contexts could possibly be hard-coded by queries deriving events that mark context bounds. However, this approach is cumbersome and error prone. It is neither modular nor human-readable. It requires tedious specification of multiple complex event queries – placing an unnecessary burden on the designer [25]. Inter-dependencies between context deriving and context processing queries would have to be taken into account to avoid re-computations, waste of valuable resources, delayed responsiveness and even incorrect results. Special optimization techniques would have to be developed to enable the benefits of context-awareness – as accomplished by our approach (Section 3.2). Furthermore, workload sharing among overlapping context windows has not been addressed in prior research.

**Event Query Optimization Techniques** [29] are often based on stream algebras [30, 34, 12]. Operators of these stream algebras work on *events*. In these approaches, *application contexts* are not supported as first-class citizens and thus they are not available for the operators. Hence, these approaches miss the event query optimization opportunities enabled by context-aware stream processing. Application contexts are first class citizens in our CAESAR model. They enable context-aware optimization techniques (Section 5).

**Business Process Models** [16, 28] explicitly support application contexts in a readable manner and allow to specify context-aware system reactions. However, these models target business process specification. They were not designed

for CEP and neglect its peculiarities. In particular, the event-driven nature of streaming applications and the importance of temporal aspect are not given enough attention. Indeed, context transitions in these models are triggered by conditions and process flow rather than by events [19]. Temporal constraints are specified on clocks rather than on event time stamps [7]. These models do not derive higher level knowledge in form of complex events.

## 9. CONCLUSIONS

The responsiveness of time-critical decision-making event stream processing applications can be substantially speed-up by evaluating only those event queries which are relevant to the current situation. Inspired by this observation, we propose the first context-aware event stream processing solution, called CAESAR, which is composed of the following key components: (1) To allow for human-readable context-aware event query specification, we propose the CAESAR model that visually captures the application contexts and allows the designer to associate appropriate event queries with each context. (2) To achieve prompt system responsiveness, the model is translated into a query plan composed of the context-aware operators of the CAESAR algebra we propose. This algebra serves as a foundation for the CAESAR optimizer that suspends those event queries which are irrelevant to the current application context and detects workload sharing opportunities of overlapping contexts. (3) We built the CAESAR runtime execution infrastructure that guarantees correct and efficient execution of inter-dependent context deriving and context processing queries. The context-aware processing is shown to perform 8-fold faster on average than the context-independent solution when using the Linear Road stream benchmark and real world data sets.

## ACKNOWLEDGEMENTS

This work was supported by NSF grants IIS 1018443 and IIS 1343620.

## 10. REFERENCES

- [1] Proton. <https://github.com/ishkin/Proton>, 2015. [Online; accessed 10-December-2015].
- [2] The Wall Street Journal. <http://www.wsj.com/articles/SB10001424052970203733504577024000381790904>, 2015. [Online; accessed 24-July-2015].
- [3] USA Today. [http://usatoday30.usatoday.com/news/nation/2011-05-25-traffic-pollution-premature-deaths-emissions\\_n.htm](http://usatoday30.usatoday.com/news/nation/2011-05-25-traffic-pollution-premature-deaths-emissions_n.htm), 2015. [Online; accessed 24-July-2015].
- [4] Wikipedia. [https://en.wikipedia.org/wiki/List\\_of\\_countries\\_by\\_traffic-related\\_death\\_rate](https://en.wikipedia.org/wiki/List_of_countries_by_traffic-related_death_rate), 2015. [Online; accessed 24-July-2015].
- [5] J. Agrawal et al. Efficient pattern matching over event streams. In *Proc. of Int. Conf. on Management of data*, SIGMOD '08, pages 147–160. ACM, 2008.
- [6] M. Akdere et al. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, Aug. 2008.
- [7] R. Alur et al. Automata for modeling real-time systems. In *Proc. of Int. Colloquium on Automata, Languages and Programming*, ICALP'90, pages 322–335. Springer, 1990.
- [8] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *Proc. of Int. Conf. on Very Large Data Bases - Volume 30*, VLDB '04, pages 336–347. VLDB Endowment, 2004.
- [9] A. Arasu et al. Linear road: A stream data management benchmark. In *Proc. of Int. Conf. on Very Large Data Bases*, volume 30 of *VLDB'04*, pages 480–491. VLDB Endowment, 2004.
- [10] A. Arasu et al. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
- [11] K. Cao et al. Context-aware distributed complex event processing method for event cloud in internet of things. *Journal of Advances in Information Science and Service Sciences*, 5(8):1212–1222, April 2013.
- [12] S. Chakravarthy et al. Integrating stream and complex event processing. In *Stream Data Processing: A Quality of Service Perspective*, volume 36 of *Advances in Database Systems*, pages 187–214. Springer US, 2009.
- [13] A. Demers et al. Cayuga: A general purpose event monitoring system. In *Proc. of Int. Conf. on Innovative Data Systems Research*, pages 411–422, 2007.
- [14] K. P. Eswaran et al. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.
- [15] T. M. Ghanem et al. Exploiting predicate-window semantics over data streams. *SIGMOD Rec.*, 35(1):3–8, Mar. 2006.
- [16] A. Grosskopf et al. *The Process: Business Process Modeling using BPMN*. Meghan Kiffer Press, 2009.
- [17] G. Hermosillo et al. Complex Event Processing for Context-Adaptive Business Processes. In *Belgium-Netherlands Software Evolution Seminar*, pages 19–24, Dec. 2009.
- [18] K. Isoyama et al. A scalable complex event processing system and evaluations of its performance. In *Proc. of Int. Conf. on Distributed Event-Based Systems*, pages 123–126, New York, NY, USA, 2012. ACM.
- [19] F. Joyce. *Programming Logic and Design, Comprehensive*. Thomson, 2008.
- [20] M. Klazar. Bell numbers, their relatives, and algebraic differential equations. *J. Comb. Theory, Ser. A*, 102(1):63–87, 2003.
- [21] W. Le et al. Scalable multi-query optimization for sparql. In *ICDE*, pages 666–677, 2012.
- [22] J. Li et al. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, Mar. 2005.
- [23] M. Liu et al. E-Cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *Proc. of Int. Conf. on Management of data*, SIGMOD'11, pages 889–900. ACM, 2011.
- [24] Y. Mei and S. Madden. ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events. In *Proc. of the SIGMOD Int. Conf. on Management of Data*, SIGMOD'09, pages 193–206. ACM, 2009.
- [25] O. Poppe et al. The HIT model: Workflow-aware event stream monitoring. In A. Hameurlain et al., editor, *Advanced data stream management and continuous query processing*, volume 8 of *Transactions on large-scale data and knowledge-centered systems*, pages 26–50. Springer, 2013.
- [26] A. Reiss et al. Creating and benchmarking a new dataset for physical activity monitoring. In *Proc. of Int. Conf. on Pervasive Technologies Related to Assistive Environments*, PETRA'12, pages 40:1–40:8. ACM, 2012.
- [27] P. Roy et al. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD*, pages 249–260, 2000.
- [28] N. Russell et al. On the suitability of UML 2.0 activity diagrams for business process modelling. In *Proc. Asia-Pacific Conf. on Conceptual Modelling*, volume 53 of *APCCM*, pages 95–104. Australian Computer Society, Inc., 2006.
- [29] S. Schneider et al. Tutorial: Stream Processing Optimizations. In *Proc. of Int. Conf. on Distributed Event-Based Systems*, DEBS'13, pages 249–258. ACM, 2013.
- [30] N. P. Schultz-Møller et al. Distributed Complex Event Processing with query rewriting. In *Proc. of Int. Conf. on Distributed Event-Based Systems*, DEBS '09, pages 1–12. ACM, 2009.
- [31] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.
- [32] D. Wang, E. A. Rundensteiner, and R. T. Ellison, III. Active complex event processing over event streams. *Proc. VLDB Endow.*, 4(10):634–645, July 2011.
- [33] Y. Wang et al. Context-aware complex event processing for event cloud in internet of things. In *Proc. of Int. Conf. on Wireless Communications and Signal Processing*, pages 1–6. IEEE, 2012.
- [34] E. Wu et al. High-performance Complex Event Processing over streams. In *Proc. of Int. Conf. on Management of data*, SIGMOD '06, pages 407–418. ACM, 2006.

# Who Cares about Others' Privacy: Personalized Anonymization of Moving Object Trajectories

Despina Kopanaki  
Dept. of Informatics  
University of Piraeus, Greece  
dkopanak@unipi.gr

Vasilis Theodossopoulos  
Dept. of Informatics  
University of Piraeus, Greece  
bill.theodossopoulos@gmail.com

Nikos Pelekis  
Dept. of Statistics and Insurance Sc.  
University of Piraeus, Greece  
npelekis@unipi.gr

Ioannis Kopanakis  
Dept. of Business Administration  
Tech. Educational Institute of Crete, Greece  
i.kopanakis@teicrete.gr

Yannis Theodoridis  
Dept. of Informatics  
University of Piraeus, Greece  
ythead@unipi.gr

## ABSTRACT

The preservation of privacy when publishing spatiotemporal traces of mobile humans is a field that is receiving growing attention. However, while more and more services offer personalized privacy options to their users, few trajectory anonymization algorithms are able to handle personalization effectively, without incurring unnecessary information distortion. In this paper, we study the problem of *Personalized (K,Δ)-anonymity*, which builds upon the model of  $(k, \delta)$ -anonymity, while allowing users to have their own individual privacy and service quality requirements. First, we propose efficient modifications to state-of-the-art  $(k, \delta)$ -anonymization algorithms by introducing a novel technique built upon users' personalized privacy settings. This way, we avoid over-anonymization and we decrease information distortion. In addition, we utilize dataset-aware trajectory segmentation in order to further reduce information distortion. We also study the novel problem of *Bounded Personalized (K,Δ)-anonymity*, where the algorithm gets as input an upper bound the information distortion being accepted, and introduce a solution to this problem by editing the  $(k, \delta)$  requirements of the highest demanding trajectories. Our extensive experimental study over real life trajectories shows the effectiveness of the proposed techniques.

## Keywords

Moving objects databases; Trajectories;  $k$ -anonymity; Personalization; Uncertainty; Segmentation; Distortion.

## 1. INTRODUCTION

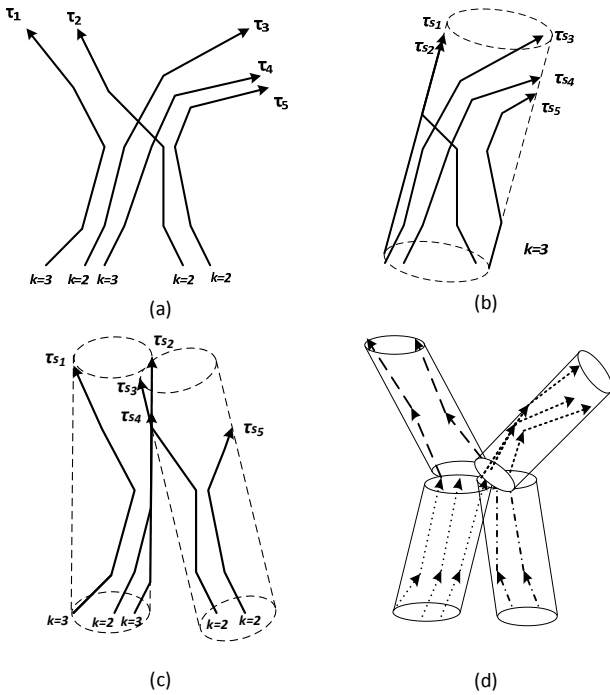
With the rapid development of information and communication technologies, the advent of mobile computing and the increasing popularity of location-aware services, the volume of mobility data

© 2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

gathered daily by service providers has exploded. It is safe to predict that this trend will continue in the near future. Publishing such information allows researchers to analyze humans' trajectories and extract behavioral patterns from them, in order to support decision-making.

However, publishing datasets consisting of humans' trajectories creates threats regarding the privacy of the individuals involved. This occurs when the spatiotemporal traces that users leave behind are combined with other publicly available information, which can reveal their identity, as well as other sensitive information about them (place of residence, sexual orientation, religious or political beliefs, etc.). Thus, it becomes necessary to develop methods providing privacy-preservation in mobility data publishing, where a sanitized version of the original dataset is published while the maximum possible data utility is maintained. A number of anonymization methods have been proposed so far, with most of them adopting the concept of *k-anonymity*, the fundamental principle which states that every entry of a published database should be indistinguishable from at least  $k-1$  other entries. For example, trajectories are grouped into clusters of at least  $k$  members and published as cylindrical volumes which 'conceal' the individual trajectories [1][2], points of trajectories are suppressed so that adversaries with partial knowledge of a trajectory cannot identify a specific one amongst at least  $k-1$  others [13], and so on.

Figure 1(a) illustrates an example dataset consisting of 5 trajectories, where each trajectory is associated with its own  $k$ -anonymity requirement, whereas Figure 1(b) illustrates the anonymization provided by W4M [2], a state-of-the-art  $(k, \delta)$ -anonymity algorithm, assuming (a universal)  $k = 3$  requirement (i.e., the maximum of the particular requirements). Clearly, the result fails to maintain the trend of the original data. However, if we could have taken into account the specific users' privacy preferences (i.e., the different  $k$ 's in Figure 1(a)), two clusters instead of one would have been created, as illustrated in Figure 1(c); *this is the first objective of this paper*. Moreover, if we could have performed an appropriate segmentation of trajectories in sub-trajectories before the anonymization process, the distortion would be even less, as illustrated in Figure 1(d); *this is the second objective of this paper*. (In this example, the  $\delta$ - parameter effect is not discussed but it is similar to that of  $k$ .)



**Figure 1: (a) a set of five trajectories along with the anonymization result provided by (b) universal  $k$ ; (c) personalized  $k_i$ ; (d) segmentation and personalized  $k_i$ .**

As revealed by the previous example, an important drawback of most of the existing anonymization methods is that they are based on universal (e.g.  $k = 3$ ) rather than user-defined privacy requirements. This lack of personalization may lead to unnecessary anonymization and data utility loss for users whose privacy requirements are overvalued and to inadequate anonymization and violation of privacy for users whose requirements are undervalued. Towards this goal, state-of-the-art techniques, such as [1][2] can be extended to use a user-specific privacy threshold  $k$  and uncertainty diameter  $\delta$ . On the other hand [9] offers personalization by introducing trajectory-specific privacy requirements, however it ignores service quality, as it will be discussed in Section 2. In contrast, the method we propose uses trajectory-specific values to determine each user's specific privacy level and service quality requirements, therefore reducing data utility loss and improving service quality.

An additional shortcoming of the existing anonymization methods that are based on clustering is that they function at the trajectory level. As a result, when dealing with trajectories that are on the whole very different to each other, though they maintain some similar parts, these algorithms fail to recognize this situation and either assign such trajectories to different clusters (i.e.,  $k$ -anonymity sets) or assign them into the same cluster after considerable spatiotemporal translation. This failure to recognize and make use of similarities between parts of trajectories is counter-intuitive and increases the overall distortion. In our proposal, we deal with this problem by utilizing trajectory segmentation in order to discover similar sub-trajectories and use those as the basis of our clustering process.

In this paper, we present the so-called *Who-Cares-about-Others'-Privacy* (WCOP) suite of methods for publishing spatiotemporal trajectory data using personalized  $(K, \Delta)$ -anonymity, extending the concept of  $(k, \delta)$ -anonymity as introduced in [1][2], where, for

each user  $u_i$ ,  $k_i$  dictates the required privacy level of the specific user and  $\delta_i$  functions as a service quality threshold. In addition, we adopt a privacy-aware trajectory segmentation phase, during which trajectories are partitioned into sub-trajectories. This phase allows the clustering algorithm to discover similarities between the trajectories and assign the respective partitions into clusters, the members of which require the least necessary editing to fulfill  $(K, \Delta)$ -anonymity, thus keeping distortion as low as possible. Finally, we present an approach aiming at controlling the information loss caused by the anonymization. The most *demanding* trajectories, i.e., the ones corresponding to users who require to be hidden among a large number of other users (hence, high  $k$ ) in a small region (thus, low  $\delta$ ), are edited in order to be made less demanding, thus decreasing anonymization distortion.

Summarizing, in this paper, we make the following contributions:

- We propose WCOP-CT, an algorithm that extends [1][2] for spatiotemporal trajectory data publication based on the assumption that users have different privacy preferences.
- We extend WCOP-CT to WCOP-SA, by incorporating a trajectory segmentation phase aiming to facilitate the discovery of patterns shared between parts of trajectories, thus decreasing the distortion caused during the anonymization process.
- We propose WCOP-B, an algorithm able to control the level of the anonymization distortion by data assessment and requirements relaxation.
- Finally, we conduct a comprehensive set of experiments over a real trajectory dataset, in order to evaluate our approach.

The rest of the paper is structured as follows: Section 2 presents related work. Section 3 formulates the two problems to be addressed: *Personalized  $(K, \Delta)$ -anonymity* and *Bounded Personalized  $(K, \Delta)$ -anonymity*, respectively. Sections 4 and 5 provide effective solutions to the above two problems, respectively. Our experimental study is presented in Section 6. Finally, Section 7 concludes the paper.

## 2. RELATED WORK

The methods proposed so far in order to tackle the issue of privacy-preserving mobility data publishing mostly adopt the principle of  $k$ -anonymity, which was originally proposed for relational databases [12]. In the context of mobility data, trajectories of moving objects are time ordered sequences of  $(p, t)$  pairs, where  $p$  denotes the place a moving object was located at recorded time  $t$ , usually assuming linear interpolation between consecutively recorded locations. Such a trajectory dataset is considered  $k$ -anonymized if each trajectory is indistinguishable from at least  $k-1$  other trajectories. Given the complicated nature of spatiotemporal data and the dependence of consecutive points in a trajectory, attributes  $(p, t)$  are considered both sensitive and quasi-identifiers at the same time. Under this setting, methods similar to those used for relational data can be employed to achieve anonymization.

Hoh and Gruteser's method [6] is an example of data perturbation with a goal of decreasing an adversary's certainty of correctly identifying a user. To do that, the so-called Path Perturbation algorithm creates fake intersection points between couples of non-intersecting trajectories if they are close enough. The crossing points must be generated within a specific time-window and within a user-specified radius, which indicates the maximum allowable perturbation and desired degree of privacy. Terrovitis

and Mamoulis [13] proposed an approach that uses suppression. Trajectories are modeled as sequences of locations where users made transactions and an adversary is assumed to have partial knowledge of users' visited locations and their relative order, therefore an incomplete projection of the dataset. Based on this assumption the algorithm seeks to eliminate the minimum amount of locations from trajectories so that the remaining trajectories are  $k$ -anonymous w.r.t. an adversary's partial knowledge. Always Walk with Others (AWO) [11] is a generalization-based approach, which transforms trajectories into series of anonymized regions, while assuming adversary's partial or full knowledge of a trajectory. To achieve anonymity, the algorithm creates groups with representative trajectories and then iteratively adds to them their closest trajectories until they consist of  $k$  members. After that,  $k$  points from each anonymized region are randomly selected and connected to points similarly generated in adjacent regions in order to form  $k$  new trajectories. Monreale et al. [10] propose  $k$ -anonymization using spatial generalization of trajectories. In particular, their method finds characteristic points of trajectories and applies spatial clustering to them. The centroids of those clusters are then used for Voronoi tessellation of the area covered in the dataset dividing it into cells with at least  $k$  trajectories. Trajectories are formed by segments linking those cells.

Never Walk Alone (NWA) [1] and its extension Wait For Me (W4M) [2], proposed by Abul et al., follow a clustering-based approach which takes advantage of the inherent uncertainty of a moving object's location introducing the concept of  $(k, \delta)$ -anonymity. An object's location at a given time is not a point, but a disk of radius  $\delta$ , and the object could be anywhere inside that, so a trajectory is not a polyline, but a cylinder consisting of consecutive such disks. To achieve  $k$ -anonymity, each trajectory is assigned to a group of at least  $k-1$  other trajectories using a greedy clustering algorithm. Then, the trajectories of each cluster are spatially translated so that they will all lie entirely within the same cylinder (uncertainty area) of radius  $\delta/2$ . W4M is a variant of NWA that uses the time-tolerant EDR distance function [4] during the clustering phase in order to overcome the limitations of Euclidean distance. Moreover, W4M performs spatio-temporal instead of spatial translation to the trajectories. All the aforementioned approaches offer no degree of personalization since they assume that all users share the same privacy level  $k$  which is application-determined.

The most related to our work is the one proposed by MahdaviFar et al. [9]. It introduces the idea of non-uniform privacy requirements, where each trajectory is associated with its own privacy level indicating the number of trajectories it should be indistinguishable from. Trajectories are first divided into groups depending on their privacy level. Clusters are then created by randomly selecting a centroid and adding to the cluster the trajectories nearest to it if their EDR distance is lower than a threshold, until the maximum privacy requirement within the cluster is satisfied. If the requirements are not satisfied, groups with lower privacy levels are progressively searched for trajectories to be added to the cluster, until all the privacy requirements are met. Finally, the trajectories of each cluster are anonymized using a matching point algorithm that generates an anonymized trajectory as the cluster's representative. While this approach offers a greater degree of personalization than others, it still leads to a compulsory trade-off between privacy and quality for each user. If a trajectory has a high privacy requirement  $k$ , it will very likely be part of a large cluster, thus suffering from increased information loss and low data utility, since the user cannot set a 'quality' requirement.

### 3. PROBLEM FORMULATION

In this section, we present the formal background and definition of the *Personalized (K,Δ)-anonymity* problem in two variations. We assume that the trajectory  $\tau$  of a moving object is a polyline in 3-dimensional space represented as a sequence of time-stamped locations:  $(p_1, t_1), (p_2, t_2), \dots, (p_n, t_n), t_1 < t_2 < \dots < t_n$ . During the non-recorded time periods  $(t_i, t_{i+1})$ , we assume linear interpolation, i.e., the object moves along a straight line from  $p_i$  to  $p_{i+1}$  with a constant speed.

Following the definition adopted by [1], an uncertain trajectory buffer is defined as a cylindrical volume of diameter  $\delta$  centered at an object's expected trajectory. Formally:

**Definition 1 (uncertain trajectory):** Given a trajectory  $\tau$  defined in  $[t_1, t_n]$  and an uncertainty threshold  $\delta$ ,  $\tau^\delta$  is the uncertain counterpart of trajectory  $\tau$ , defined as follows: for each 3-dimensional point  $(p, t)$  in  $\tau$ , its uncertainty area is the horizontal disk centered at  $(p, t)$  with diameter  $\delta$ . The trajectory volume of  $\tau^\delta$ , denoted by  $Vol(\tau^\delta)$  is the union of all such disks for every  $t \in [t_1, t_n]$ . A possible motion curve of  $\tau$  is any continuous function  $f_{PMC}^\tau: Time \rightarrow \mathbb{R}^2$  defined on the interval  $[t_1, t_n]$ , such that for any  $t \in [t_1, t_n]$ , the 3-dimensional point  $(f_{PMC}^\tau(t), t)$  lies inside the uncertainty area at time  $t$ . ■

**Definition 2 (co-localized trajectories):** Two trajectories  $\tau_1, \tau_2$ , both defined in  $[t_1, t_n]$ , are considered co-localized w.r.t.  $\delta$ , if for each point  $(p_1, t)$  in  $\tau_1$  and  $(p_2, t)$  in  $\tau_2$ ,  $t \in [t_1, t_n]$ , it holds that the Euclidean distance  $d(p_1, p_2) \leq \delta$ ; we write  $Coloc_\delta(\tau_1, \tau_2)$  omitting the time interval  $[t_1, t_n]$ . ■

**Definition 3 ((k,δ)-anonymous set of trajectories):** Given a set of trajectories  $S$ , an uncertainty threshold  $\delta$ , and an anonymity threshold  $k$ ,  $S$  is  $(k, \delta)$ -anonymous iff  $|S| \geq k$  and  $Coloc_\delta(\tau_i, \tau_j)$  for each  $\tau_i, \tau_j \in S$ . ■

A dataset  $D$  of moving object trajectories is considered  $(k, \delta)$ -anonymous if each of its members belongs to a  $(k, \delta)$ -anonymity set. If  $D$  does not meet this requirement, then it should be transformed into a sanitized version, called  $D_s$ , which satisfies the aforementioned condition. Hence:

**Definition 4 ((k,δ)-anonymity):** Given a dataset  $D$  of moving object trajectories, an uncertainty threshold  $\delta$ , and an anonymity threshold  $k$ ,  $(k, \delta)$ -anonymity is satisfied by transforming  $D$  to  $D_s$ , such that for each trajectory  $\tau_s \in D_s$ , there exists a  $(k, \delta)$ -anonymity set  $S \subseteq D_s$ ,  $\tau_s \in S$ , and the distortion between  $D$  and  $D_s$  is minimal. ■

One of the possible approaches to transform a dataset to its sanitized version is the spatiotemporal translation of the trajectory points. Distortion usually measures the difference between the original and the sanitized data. A trajectory's distortion is defined as the sum of its point-wise distances to its sanitized version. In case the trajectory is an outlier, thus removed from the anonymized dataset, the distortion is proportional to the number of the distorted moving points of the original trajectory. The total distortion caused by sanitizing the entire database is defined as the aggregation of its individual trajectories' distortion.

**Definition 5 (trajectory distortion due to translation):** Given a trajectory  $\tau \in D$  defined in  $[t_1, t_n]$  and its sanitized version  $\tau_s \in D_s$ , the translation distortion (TD) over  $\tau$  due to its translation into  $\tau_s$  is defined as:

$$TD(\tau, \tau_s) = \begin{cases} \sum_{t \in [t_1, t_n]} d(\tau[t], \tau_s[t]) & |\tau^s| > 0 \\ |\tau| \cdot \Omega & |\tau^s| = 0 \end{cases} \quad (1)$$

where  $|\tau|, |\tau^s|$  indicate the size (i.e., number of points) of the original and the sanitized trajectory, respectively, and  $\Omega$  is a constant that penalizes distorted moving points. Summing for all trajectories, the total translation distortion over a trajectory dataset  $D$  due to its translation into  $D_s$  is defined as:

$$TTD(D, D_s) = \sum_{\tau \in D} T D(\tau, \tau_s) \quad (2)$$

■

Regarding  $\Omega$ , in our experiments it corresponds to the maximum translation occurred during the anonymization process.

The problem introduced in this paper is that of achieving anonymity of a trajectory dataset, where each trajectory prescribes its own  $(k_i, \delta_i)$  values, while keeping the total distortion as low as possible. The problem is formulated as follows:

**Problem 1 (Personalized  $(K, \Delta)$ -anonymity problem):** Given a dataset  $D$  of moving object trajectories,  $D = \{\tau_1, \dots, \tau_n\}$ , along with their respective anonymity preferences  $(k_i, \delta_i)$ , and a trash size threshold  $trash_{max}$ , the Personalized  $(K, \Delta)$ -anonymity problem, where  $K = \{k_1, \dots, k_n\}$  and  $\Delta = \{\delta_1, \dots, \delta_n\}$ , is to find an anonymized version of  $D$ ,  $D_s = \{\tau_{s_1}, \dots, \tau_{s_m}\}$ ,  $0 \leq n-m \leq trash_{max}$ , where  $\tau_{s_i}$  is a  $(k_i, \delta_i)$ -anonymity version of  $\tau_i$  and the total distortion,  $TTD(D, D_s)$ , is minimal. ■

In the above definition, please note that  $m \leq n$ , i.e., the cardinality of the output dataset  $D_s$  may be lower than that of the input dataset  $D$ . This is due to the fact that during the anonymization process some of the original trajectories may be moved to the trash bin, i.e., they are completely removed.

A comment on Problem 1 is that the distortion caused in the original dataset due to anonymization is not controlled since it has to do with the nature of the trajectories, as well as the values of their anonymity preferences  $(k_i, \delta_i)$ . Therefore, a natural variation of the above problem is that of anonymizing a database of trajectories of moving objects where each object has its own  $(k_i, \delta_i)$  values, while keeping a control over the overall distortion. This problem is formulated as follows:

**Problem 2 (Bounded Personalized  $(K, \Delta)$ -anonymity problem):** Given a dataset  $D$  of moving object trajectories,  $D = \{\tau_1, \dots, \tau_n\}$ , along with their respective anonymity preferences  $(k_i, \delta_i)$ , a trash size threshold  $trash_{max}$ , and a distortion threshold  $distort_{max}$ , the Bounded Personalized  $(K, \Delta)$ -anonymity problem, where  $K = \{k_1, \dots, k_n\}$  and  $\Delta = \{\delta_1, \dots, \delta_n\}$ , is to find an anonymized version of  $D$ ,  $D_s = \{\tau_{s_1}, \dots, \tau_{s_m}\}$ ,  $0 \leq n-m \leq trash_{max}$  where  $\tau_{s_i}$  is a  $(k_i, \delta_i)$ -sanitized version of  $\tau_i$  and the total distortion,  $Distortion(D, D_s) \leq distort_{max}$ . ■

The total distortion of the dataset when compared to its sanitized version,  $Distortion(D, D_s)$ , is a formula consisting of two factors, i.e. the distortion from the translation during the anonymization step,  $TTD(D, D_s)$ , which is calculated according to Eq.(2) along with the distortion caused from the editing phase,  $TE(D)$  (to be introduced in Section 5).

A special case is when the solution  $D_s$  of Problem 1 also makes a solution to Problem 2, formally:  $Distortion(D, D_s) \leq distort_{max}$ ; in such case, nothing extra has to be done in order to provide a solution to Problem 2. In the general case, however, where  $Distortion(D, D_s) > distort_{max}$ , Problem 2 can be solved by relaxing the  $(k_i, \delta_i)$  constraints of those trajectories that are most responsible for the distortion caused; we call them, the most demanding ones, and a formulation of the demandingness of a trajectory will be defined in Section 5.

## 4. PERSONALIZED $(K, \Delta)$ -ANONYMITY

In this section, we present a suite of methods for publishing spatiotemporal trajectory data using the personalized  $(K, \Delta)$ -anonymity. In particular, Section 4.1 describes baseline solutions for providing personalized anonymization w.r.t. different users' preferences. In section 4.2, we present an approach that provides personalized  $(K, \Delta)$ -anonymity based on trajectory segmentation.

### 4.1 Baseline Solutions

Given a trajectory dataset  $D$  along with personalized privacy requirements  $(k_i, \delta_i)$  for each trajectory  $\tau_i$ , a baseline solution to Problem 1 consists of exploiting a state-of-the-art  $(k, \delta)$ -anonymity algorithm, such NWA [1] or W4M [2], using a single, universal value for each  $k$  and  $\delta$ . In order to satisfy every user's privacy requirement, it is the maximum among all  $k_i$  and the minimum among all  $\delta_i$  that are assigned to the universal  $k$  and  $\delta$  variables, respectively. The following algorithm, being the naïve version of our *Who-Cares-about-Others'-Privacy* (WCOP) suite of methods, illustrates this solution. As already discussed, Function  $k\text{-}\delta\text{-anonymity}(\cdot)$  in Line 3 of the algorithm corresponds to a state-of-the-art  $(k, \delta)$ -anonymity algorithm, such as NWA or W4M.

---

#### Algorithm 1. WCOP-NV

---

**Input:** (1) a trajectory dataset,  $D = \{(\tau_1, k_1, \delta_1), \dots, (\tau_n, k_n, \delta_n)\}$ ; (2) a trash size threshold,  $trash_{max}$ , (3) a maximum cluster radius threshold,  $radius_{max}$

**Output:** A sanitized trajectory dataset,  $D_s = \{\tau_{s_1}, \dots, \tau_{s_m}\}$

1.  $max_k \leftarrow \max_i \{k_i\}$
  2.  $min_\delta \leftarrow \min_i \{\delta_i\}$
  3.  $D_s \leftarrow k\text{-}\delta\text{-anonymity}(D, max_k, min_\delta, trash_{max}, radius_{max})$
  4. **return**  $D_s$
- 

In order to improve this very crude attempt of satisfying personalized  $(K, \Delta)$  values, we propose an alternative approach directly using the user-specific privacy requirements and being based on clustering and translation, called WCOP-CT (where 'CT' stands for Clustering and Translation). WCOP-CT follows the general structure of W4M, consisting of two phases: a *greedy clustering phase*, which has been shown in [1] to have the best effectiveness/efficiency ratio, followed by a *spatiotemporal translation phase*, which uses Edit Distance on Real sequence (EDR) distance function [4] in order to modify each cluster to make it an anonymity set. During the first phase, a pivot trajectory is randomly selected and a cluster is formed around it by its  $k-1$  unvisited closest neighbors. Then the unvisited trajectory that is farthest away from previous pivots is selected as a new pivot, and the process is repeated until clusters satisfying certain criteria. During the second phase, each cluster formed in the previous phase is transformed into a  $(k, \delta)$ -anonymity set.

The input of WCOP-CT algorithm is a dataset  $D$  consisting of  $n$  trajectories  $\tau_i$  along with their personalized privacy requirements  $(k_i, \delta_i)$ , a  $trash_{max}$  value that bounds the size of trash, which contains the outliers suppressed during the clustering process (phase 1 of the algorithm) in order to improve the quality of the final output and the maximum allowable cluster radius,  $radius_{max}$ . The output of the algorithm is the personalized  $(K, \Delta)$ -anonymized dataset  $D_s$ .



---

**Algorithm 2.** WCOP-CT

---

**Input:** (1) a trajectory dataset,  $D = \{(\tau_1, k_1, \delta_1), \dots, (\tau_n, k_n, \delta_n)\}$ ; (2) a trash size threshold,  $trash_{max}$ , (3) a maximum cluster radius threshold,  $radius_{max}$

**Output:** A sanitized trajectory dataset,  $D_s = \{\tau_{s_1}, \dots, \tau_{s_m}\}$

```
1.  $D_s \leftarrow \emptyset$ 
2.  $\gamma \leftarrow WCOP\text{-Clustering}(D, trash_{max}, radius_{max})$ 
   /* Clustering phase */
3. for each cluster  $C \in \gamma$  do
4.    $C_s \leftarrow \emptyset$ 
5.    $\tau_c \leftarrow \text{pivot of } C; \delta_c \leftarrow \min_i\{\delta_i\}$  in  $C$ 
   /* Translation phase */
6.   for each  $\tau \in C$  do
7.      $\tau_s \leftarrow WCOP\text{-Translation}(\tau, \tau_c, \delta_c)$ 
8.      $C_s \leftarrow C_s \cup \{\tau_s\}$ 
9.   end for
10.   $D_s \leftarrow D_s \cup C_s$ 
11. end for
12. return  $D_s$ 
```

---

In detail, WCOP-CT works as follows: the operation *WCOP-Clustering* (line 2) extracts a set of clusters  $\gamma$  from the original dataset. Then, for each cluster (represented by the corresponding pivot trajectory), the algorithm defines its own  $\delta$  value, which is the  $\min_i\{\delta_i\}$  among its members (lines 3-5). All trajectories contained on the current cluster are translated by the *WCOP-Translation* operation (lines 6-7). The procedure is repeated until all trajectories of all clusters are anonymized.

The main difference between the proposed WCOP-CT and, e.g. W4M is that, whereas in W4M a pivot is selected and then invariably grouped along with its  $k-1$  closest neighbors in order to form a cluster, in WCOP-CT each cluster has its own, non-fixed  $k$  value. It can easily be seen that this approach results in clusters of non-fixed size ranging between 2 and  $\max_i\{k_i\}$ . In the same spirit, the spatiotemporal editing phase of WCOP-CT, called *WCOP-Translation* in Algorithm 2, differs to that of W4M in that there is no universal  $\delta$  applied to all clusters, but each cluster is edited based on its own  $\delta_c$  value, which is the  $\min_i\{\delta_i\}$  among its members.

The core of WCOP-CT, i.e., the clustering step under the name *WCOP-Clustering*( $D, trash_{max}, radius_{max}$ ) in Algorithm 2 above, is listed in Algorithm 3 below. *WCOP-Clustering* follows the general structure of the respective algorithm of W4M and Greedy Clustering, where W4M is based on. After forming clusters, each of them is separately processed and transformed into a  $(k, \delta)$ -anonymity set, with  $(k, \delta)$  being values specific to the cluster. Here we follow the approach proposed in [2], which achieves that by using the cluster's pivot as reference and editing the other trajectories so that they are co-located with it (see Section 3) and also have the same number of points as the pivot. The difference with our method is that each cluster uses its own  $\delta$  value for co-localization instead of a universal value.

In detail, *WCOP-Clustering* iteratively selects pivot trajectories to function as centers of clusters, with pivots being selected at random from amongst the available active trajectories (Line 4). A pivot's  $(k, \delta)$  values serve as the initial  $(k, \delta)$  requirements of its candidate cluster (Line 6). The algorithm then successively adds to the candidate cluster the nearest unvisited neighbor of the pivot and updates the cluster's  $k$  and  $\delta$ , until the cluster's size is enough to satisfy its  $k$  requirement, which equals to the maximum  $k_i$  value among its members (Lines 7-11).

---

**Algorithm 3.** WCOP-Clustering

---

**Input:** (1) a trajectory dataset,  $D = \{(\tau_1, k_1, \delta_1), \dots, (\tau_n, k_n, \delta_n)\}$ ; (2) a trash size threshold,  $trash_{max}$ , (3) a maximum cluster radius threshold,  $radius_{max}$

**Output:** A set of clusters  $\gamma$

```
1. repeat
2.   Active  $\leftarrow D$ ; Clustered  $\leftarrow \emptyset$ ; Pivots  $\leftarrow \emptyset$ ; Trash  $\leftarrow \emptyset$ 
3.   while Active  $\neq \emptyset$  do
4.      $\tau_p \leftarrow \text{random}(\tau) \mid \tau \in \text{Active}$ 
5.      $c_{\tau_p}.size \leftarrow 1$ 
6.      $c_{\tau_p}.k \leftarrow \tau_p.k; c_{\tau_p}.\delta \leftarrow \tau_p.\delta$ 
7.     while  $(c_{\tau_p}.k > c_{\tau_p}.size)$  do
8.        $c_{\tau_p} \leftarrow \{\tau_p\} \cup \{\text{NN of } \tau_p \in D - \text{Clustered}\}$ 
9.        $c_{\tau_p}.size \leftarrow c_{\tau_p}.size + 1$ 
10.       $c_{\tau_p}.k \leftarrow \max(c_{\tau_p}.k, \tau_{NN}.k)$ 
11.       $c_{\tau_p}.\delta \leftarrow \min(c_{\tau_p}.\delta, \tau_{NN}.\delta)$ 
12.    end while
13.    if  $\max_{\tau \in c_{\tau_p}} \text{Dist}(\tau_p, \tau) \leq radius_{max}$  then
14.      Active  $\leftarrow \text{Active} - c_{\tau_p}$ 
15.      Clustered  $\leftarrow \text{Clustered} \cup c_{\tau_p}$ 
16.      Pivots  $\leftarrow \text{Pivots} \cup \{\tau_p\}$ 
17.    else
18.      Active  $\leftarrow \text{Active} - \{\tau_p\}$ 
19.    end while
20.    for each  $\tau \in D - \text{Clustered}$  do
21.       $\tau_p \leftarrow \text{argmin}_{\tau' \in \text{Pivots} \mid c_{\tau'}.size \geq \tau.k - 1, c_{\tau'}.\delta \leq \tau.\delta} \text{Dist}(\tau', \tau)$ 
22.      if  $\text{Dist}(\tau_p, \tau) \leq radius_{max}$  then
23.         $c_{\tau_p} \leftarrow c_{\tau_p} \cup \{\tau\}$ 
24.      else
25.        Trash  $\leftarrow \text{Trash} \cup \{\tau\}$ 
26.      end for
27.      increase( $radius_{max}$ )
28.    until  $|\text{Trash}| \leq |Trash_{max}|$ 
29.    return  $\{c_{\tau_p} \mid \tau_p \in \text{Pivots}\}$ 
```

---

Once all possible clusters have been formed, the remaining unassigned trajectories are assigned to the cluster of their closest pivot, on condition that their  $k_i$  can be satisfied by the cluster's size (including themselves), their  $\delta_i$  are not smaller than the cluster's current  $\delta$ , and their addition will not increase the cluster's radius beyond  $radius_{max}$  (Lines 20-23). If a trajectory cannot be added to any cluster without violating a condition, it is moved to the trash (Line 25). If the solution found results in trash with size larger than the  $trash_{max}$  threshold, the  $radius_{max}$  constraint is relaxed and the process starts again from the beginning until a solution is achieved that satisfies the  $trash_{max}$  size requirement (Lines 27-28). As output, the algorithm returns only the clusters formed, excluding the suppressed trajectories implicitly (Line 29).

After the clusters have been defined, Algorithm 2 proceeds to the necessary spatiotemporal translation. Since trajectories may not be of the same size, the EDR time-tolerant distance function is responsible to minimize the necessary number of operations so as to make them indistinguishable. The goal of the *editing* operations (i.e. translate points towards pivot, remove deleted points, insert new points) that are performed to a trajectory is to make it more similar to the pivot. Algorithm 4 describes the translation procedure that is followed by WCOP-CT.

---

**Algorithm 4.** WCOP-Translation

---

**Input:** (1) a trajectory  $\tau$ , (2) cluster's pivot trajectory  $\tau_c$ , (3) cluster's uncertainty threshold  $\delta_c$

**Output:** Anonymized trajectory  $\tau_s$

1.  $edit \leftarrow EDR\_op\_sequence(\tau, \tau_c)$
  2.  $\tau_s \leftarrow \langle \rangle$
  3.  $i, j \leftarrow 1$
  4. **for** all  $op \in edit$  **do**
  5.     **if**  $op = remove(\tau_{c,j})$  **then**
  6.          $\tau_s.append(\langle random\_point\_in\_circle(\tau_{c,j}.x, \tau_{c,j}.y, \delta_c/2), \tau_{c,j}.t \rangle)$
  7.          $j \leftarrow j+1$
  8.     **else**
  9.         **if**  $op = match(\tau_i, \tau_{c,j})$  **then**
  10.              $\tau_s.append(\langle transl(\tau_i.x, \tau_i.y, \tau_{c,j}.x, \tau_{c,j}.y, \delta_c/2), \tau_{c,j}.t \rangle)$
  11.              $i \leftarrow i+1$
  12.              $j \leftarrow j+1$
  13.         **else**
  14.              $i \leftarrow i+1$
  15.     **end for**
  16. **return**  $\tau_s$
- 

As a first step, the algorithm reconstructs the sequence of the required operations (Line 1). In case of a point deletion from the pivot trajectory  $\tau_c$ , the algorithm instead of deleting, it creates a new point randomly inside the circle of radius  $\frac{\delta_c}{2}$  around the corresponding point in  $\tau_c$  (Lines 5-7). Recall that in our case each cluster has its own  $\delta_c$  equal to  $\min_i \{\delta_i\}$  among its members. Differently, when the deletion concerns trajectory  $\tau$ , the point is permanently removed (Lines 13-14). If a matching between  $\tau$  and  $\tau_c$  occurs, then operation *transl* is responsible to ensure that the distance between them will be equal or less than  $\frac{\delta_c}{2}$ . If the two points match w.r.t. the temporal dimension, then trajectory's point is transferred inside the circle to a point having the minimum distance translation from the original one. Else, the temporal coordinate value of the pivot's point is used and the point is spatially translated again inside the circle with radius equal to or less than  $\frac{\delta_c}{2}$  (Lines 9-12).

## 4.2 Personalized $(K, \Delta)$ -Anonymity using Trajectory Segmentation

In this section, we introduce a novel approach to the problem of *Personalized  $(K, \Delta)$ -anonymity*, which aims to improve upon the baseline solutions presented in the previous section by implementing trajectory segmentation, in order to increase clustering effectiveness and decrease distortion levels. A shortcoming of the two baseline solutions (WCOP-NV and WCOP-CT), which is common in all clustering methods, is that they use the trajectory as the smallest working unit. As a result, when two trajectories have some similar parts, but are significantly different on the whole, the algorithm is unable to discover and make use of those similar elements, leading to an overall increased distortion during clustering. In order to deal with this issue, our approach includes a trajectory segmentation phase, where trajectories are partitioned into sub-trajectories according to a set of privacy-aware criteria. It is these sub-trajectories that are then used as input for the anonymization stage of the algorithm that follows. While this segmentation incurs extra computational cost, it offers a distinct advantage in that it facilitates the discovery of patterns shared between parts of trajectories, which otherwise are significantly different on the whole.

The so-called WCOP-SA (where 'SA' stands for Segmenting and Anonymizing), which is presented in Algorithm 5 below, is the generic two-step method that we propose. Given a dataset  $D$  consisting of  $n$  trajectories  $\tau_i$  along with their personalized privacy requirements  $(k_i, \delta_i)$ , in the first step the algorithm applies a trajectory segmentation process to produce the respective dataset of partitioned sub-trajectories  $D_p$ , followed by the second step that anonymizes those sub-trajectories.

---

**Algorithm 5.** WCOP-SA

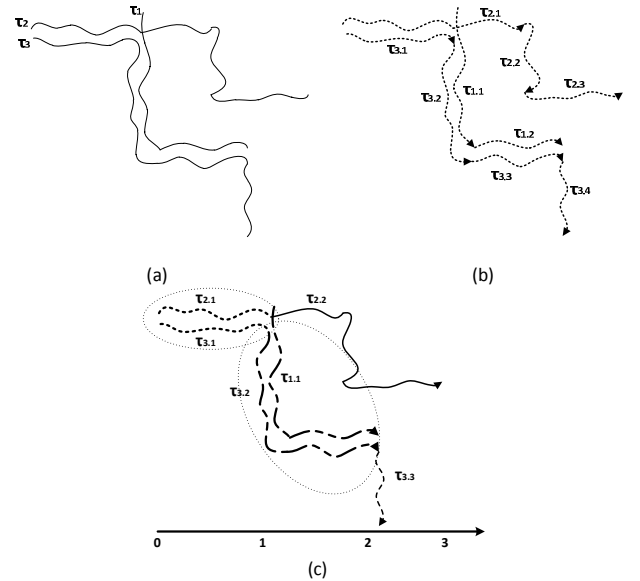
---

**Input:** (1) a trajectory dataset,  $D = \{(\tau_1, k_1, \delta_1), \dots, (\tau_n, k_n, \delta_n)\}$ ; (2) a trash size threshold,  $trash_{max}$ , (3) a maximum cluster radius threshold,  $radius_{max}$

**Output:** A sanitized trajectory dataset,  $D_s = \{\tau_{s_1}, \dots, \tau_{s_m}\}$ .

1.  $D_p \leftarrow WCOP\_Segmenting(D)$
  2.  $D_s \leftarrow WCOP\_Anonymizing(D_p, trash_{max}, radius_{max})$
  3. **return**  $D_s$
- 

An example of using WCOP-SA on a dataset partitioned using this method can be seen in Figure 1(d), where segments that are similar in terms of the number of their neighboring trajectories have been identified and grouped into sub-trajectories, which in turn have been assigned to appropriate clusters causing less spatiotemporal translation. WCOP-SA is by purpose generic, in that it does not strictly specify the algorithms used for segmentation and anonymization. Any algorithm of this kind can be used. However, in our experimental study (Section 6) we evaluate WCOP-SA using a trajectory-aware (Traclus [8]) vs. a neighborhood-aware segmentation algorithm (Convoys [7]) for the segmentation step, and WCOP-CT vs. WCOP-B (to be introduced in Section 5) for the anonymization step.



**Figure 2:** (a) a set of trajectories segmented by (b) Traclus and (c) Convoys partitioning-and-clustering algorithms.

Why Traclus vs. Convoys? Traclus [8] is a well-known and widely-used partitioning and clustering framework that performs density-based clustering on line segments aiming at discovering common sub-trajectories instead of grouping trajectories as a whole. During this process, trajectories are first partitioned on segmentation points representing significant changes of the trajectory's behavior (i.e. direction) by using the minimum description length principle.

Then, the directed line segments previously discovered are clustered with a variant of DBSCAN density-based clustering algorithm. However, the aforementioned approach does not properly incorporate the temporal dimension of trajectories as it spatially segments trajectories w.r.t their direction. A well-known temporal-awareness approach for clustering spatiotemporal trajectories is the one of Convoys [7]. It is a concept that uses different criteria for grouping the trajectories. A *convoy* is defined as a group of objects that has at least  $m$  objects, which are density-connected with respect to a distance threshold  $e$ , during  $k$  consecutive time-instants.

The difference between the two aforementioned approaches is illustrated in Figure 2. Let us assume three trajectories  $\tau_1, \tau_2, \tau_3$  (Figure 2(a)); when Traclus is applied for the segmentation of trajectories, the derived sub-trajectories (illustrated in Figure 2(b)) are constructed based on geometric parameters, actually, significant changes on their own direction. In contrast, Convoys performs segmentation by discovering neighboring trajectories that are moving together during a time period; as illustrated in Figure 2(c),  $\tau_2$  and  $\tau_3$  are moving together between  $t = 0$  and  $t = 1$  while  $\tau_1$  and  $\tau_3$  between  $t = 1$  and  $t = 2$ , thus two convoys are discovered, which are then used for the segmentation of the trajectories to 6 sub-trajectories.

## 5. BOUNDED PERSONALIZED $(K, \Delta)$ -ANONYMITY

In order to deal with the problem of *Bounded Personalized  $(K, \Delta)$ -anonymity* defined in Section 3, where the requirement is to keep anonymization distortion below a given threshold, we extend the methods presented in the previous sections by introducing dataset assessment and requirement relaxation. Since distortion is due to spatiotemporal translation, a naïve approach would be to anonymize a dataset once, identify the trajectories, which have undergone the most translation and edit them. However, a trajectory  $\tau$  might be translated not due to its own  $(k, \delta)$  values, but in order to be assigned to a cluster that includes very demanding members, i.e.,  $\tau$ 's neighbors. Therefore, in order to decrease the overall distortion of a dataset, we argue that the most demanding trajectories should be detected and edited.

By intuition, high values of  $k$  and low values of  $\delta$  make a trajectory demanding. As such, a metric for the demandingness of a trajectory is defined as follows:

**Definition 6 (dataset-aware trajectory demandingness):** Given a trajectory  $\tau \in D$  with privacy requirements  $(k, \delta)$ , its dataset-aware demandingness,  $d_{dem}(\tau, D) \rightarrow [0, 1]$ , is defined as:

$$d_{dem}(\tau, D) = w_1 * \frac{\tau.k}{k_{max}} + w_2 * \frac{\delta_{min}}{\tau.\delta} \quad (3)$$

where  $k_{max} > 1$  and  $\delta_{min} > 0$  correspond to the maximum  $k$  and minimum  $\delta$  values of trajectories in  $D$ , respectively, and  $(k, \delta)$  contribute to the overall value according to some weights,  $\sum w_i = 1$ .

■

Eq. (3) formulates the intuition that trajectory demandingness is proportional to  $k$  and reversely proportional to  $\delta$ .  $k_{max}$  and  $\delta_{min}$  are used for normalization purposes and the weights  $w_i$  are introduced in order for the importance of the two components to be controlled, according to the application scenario. For simplicity, in the rest of the paper, the two components are equally weighted, i.e.,  $w_1 = w_2 = 1/2$ .

As an example, consider a dataset  $D$  consisting of 50 trajectories where  $k_{max} = 50$  and  $\delta_{min} = 20$ . Table 1 below lists the top-5 demanding trajectories of the dataset, where their demandingness has been calculated according to Eq. (3). Now assume that the overall distortion caused by the sanitization of the dataset  $D$  to  $D_s$  exceeds  $distort_{max}$  threshold. The two most demanding trajectories, i.e.  $\tau_{21}, \tau_5$  could be edited in order to decrease the complexity of assigning them to a cluster. Trajectory  $\tau_{21}$  requires to be hidden among other 49 neighbors within an area of diameter 30 m. Similarly, trajectory  $\tau_5$ , less demanding, requires other 29 neighbors within an area of diameter 20 m.

**Table 1: An example of editing the most demanding trajectories**

$\tau_i$	$(k_i, \delta_i)$	$d_{dem}(\tau_i, D)$
$\tau_{21}$	(50,30)	0.83
$\tau_5$	(30,20)	0.8
$\tau_{47}$	(23,100)	0.33
$\tau_{15}$	(23,220)	0.27
$\tau_7$	(20,200)	0.25

How can we relax these requirements? Actually, we need a measure that indicates the degree of trajectory editing. For this purpose, we define *trajectory edit cost* as the ratio of the  $(k, \delta)$ -editing required for the particular trajectory over the  $(k, \delta)$ -editing required for the most demanding one. Note that for a trajectory  $\tau$ , its dataset-aware demandingness can be reduced by editing its  $k$  and/or  $\delta$  values.

**Definition 7 (trajectory edit cost):** Given a trajectory  $\tau \in D$  with dataset-aware demandingness,  $d_{dem}$ , defined over  $D$ , and a threshold trajectory  $\tau_{thres}$ , the edit cost of trajectory  $\tau$ ,  $0 \leq cost_{edit} \leq 1$ , is defined as:

$$cost_{edit}(\tau, D) = \begin{cases} \frac{d_{dem}(\tau, D) - d_{dem}(\tau_{thres}, D)}{\max_{\tau \in D} d_{dem}(\tau, D) - d_{dem}(\tau_{thres}, D)}, & \text{if } \max_{\tau \in D} d_{dem}(\tau, D) \neq d_{dem}(\tau_{thres}, D) \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

where  $\max_{\tau \in D} d_{dem}(D)$  is the maximum dataset-aware demandingness among the trajectories in  $D$ . ■

As a threshold trajectory,  $\tau_{thres}$ , we refer to the trajectory with the maximum acceptable demandingness in the ranking. All trajectories having a higher ranking (i.e. more demanding) are being edited. Back to Table 1, let us assume that the two most demanding trajectories need to be edited. Trajectory  $\tau_{47}$  will be the threshold trajectory  $\tau_{thres}$ . Thus, according to Eq. (4), the edit cost of  $\tau_{21}$  equals to 1 while for  $\tau_5$  it is equal to 0.94.

The distortion caused by an edited trajectory can be measured as the number of its points multiplied by the maximum dataset translation, multiplied by the trajectory's edit cost, which indicates the required editing degree compared to the maximally edited one.

**Definition 8 (trajectory distortion due to  $(k, \delta)$  editing):** Given an edited trajectory  $\tau \in D$ , the contribution of trajectory  $\tau$  to the overall distortion cost,  $distort$ , is defined as:

$$distort(\tau, D) = |\tau| \cdot \Omega \cdot cost_{edit}(\tau, D) \quad (5)$$

where  $|\tau|$  indicates the size of the trajectory (i.e. the number of its points) and  $\Omega$  is a constant that penalizes distortion. The overall

editing distortion,  $DE$ , over a trajectory dataset  $D$  due to trajectory editing, is defined as:

$$DE(D) = \sum_{\tau \in D} distort(\tau, D) \quad (6)$$

■

Regarding  $\Omega$ , as in Eq. (2), we define it to be the maximum translation occurred during the anonymization process.

**Definition 9 (dataset distortion):** The total distortion of a dataset  $D$  due to its anonymization to  $D_s$ , is defined as the sum of the total distortion due to translation,  $TTD$ , and the total distortion due to editing,  $DE$ :

$$Distortion(D, D_s) = TTD(D, D_s) + DE(D) \quad (7)$$

■

WCOP-B ('B' stands for Bounded), which is presented in Algorithm 6 below, shows the generic concept we propose to tackle the *Bounded Personalized (K,Δ)-anonymity problem*, as it was defined in Section 3. Note that, as a first step, a user is able to apply WCOP-CT on the original dataset. The output will be an anonymized dataset along with the distortion caused during the anonymization. The user is then capable to estimate the desired distortion thus determining  $distort_{max}$ .

---

#### Algorithm 6. WCOP-B

---

**Input:** (1) a trajectory dataset,  $D = \{(\tau_1, k_1, \delta_1), \dots, (\tau_n, k_n, \delta_n)\}$ ; (2) a trash size threshold,  $trash_{max}$ ; (3) a distortion threshold,  $distort_{max}$ ; (3) an amount of editable trajectories,  $step$ .

**Output:** A sanitized trajectory dataset,  $D_s = \{\tau_{s_1}, \dots, \tau_{s_m}\}$

```

1. for each  $\tau \in D$  do
2.   Calculate  $d_{dem}(\tau, D)$  // according to Eq. (3)
3. end for
4.  $edit_{size} \leftarrow step$ 
5. SortByDemandingness(D)
6. repeat
7.   ResetTrajectories(D)
8.   Edited  $\leftarrow \emptyset$ ; Trashed  $\leftarrow \emptyset$ ; editcount  $\leftarrow 0$ 
9.    $\tau \leftarrow$  highest scoring trajectory
10.   $\tau_{thres} \leftarrow \tau_{N-edit_{size}-1}$ 
    /*Editing phase */
11.  while editcount < editsize do
12.    Calculate costedit( $\tau, D$ ) // according to Eq. (4)
13.     $\tau.k \leftarrow \tau_{thres}.k$ 
14.     $\tau.\delta \leftarrow \tau_{thres}.\delta$ 
15.    Edited  $\leftarrow$  Edited  $\cup \{\tau\}$ 
16.    editcount  $\leftarrow$  editcount + 1
17.     $\tau \leftarrow$  next trajectory
18.  end while
    /* Anonymization phase */
19.   $D_s \leftarrow$  WCOP-CT(D, trashmax, radiusmax)
20.  Calculate Distortion(D,  $D_s$ ) // according to Eq. (7)
21.  editsize  $\leftarrow$  editsize + step
22. until (Distortion  $\leq$  distortmax || editsize  $\geq$  |D|)
23. return  $D_s$ 

```

---

Algorithm WCOP-B-Edit-and-Anonymization works as follows: With the trajectory database and a distortion threshold  $distort_{max}$  given as input, the trajectories are first assessed in order for the algorithm to calculate the demandingness of each trajectory according to Eq. (3) (Lines 1-3). Then, the trajectories are sorted according to their demandingness, to facilitate the steps that follow (line 5). Based on the demandingness scores previously calculated, the  $(k, \delta)$  values of the most demanding trajectories are edited, with  $edit_{size}$  determining the amount of editable trajectories (Lines 11-18). In more detail, the goal of the editing process that

follows is to edit the most expensive trajectories so that their editing score will become equal to the threshold trajectory's editing score. Starting with the highest-scoring trajectory and continuing until  $edit_{size}$  has been reached, the editing cost of each trajectory is calculated (lines 11-12). Trajectory's  $k$  value is then decreased to the corresponding value of the threshold trajectory (lines 13). Next, the trajectory's  $\delta$  is increased up to threshold trajectory's  $\delta$  value (Line 14). The trajectory is then marked as 'edited', the edit-counter is updated and the next-highest-ranking trajectory selected (Lines 15-17). After the editing phase is completed, the edited dataset  $D$  is given as input to the WCOP-CT algorithm, which produces an anonymized dataset  $D_s$  (Line 19). The total distortion of the dataset is then calculated according to Eq.(7) (Lines 20). If the total distortion is below the distortion threshold,  $distort_{max}$ , the algorithm ends and the anonymized dataset is given as output, otherwise the portion of the dataset that is marked for editing is increased (Line 21) and the editing - anonymization phase is repeated; this loop continues until either the distortion requirement is satisfied or the entire dataset has been edited (Line 22).

It is worth to note that the method is valid for datasets consisting of either whole trajectories or segmented sub-trajectories. Therefore, it is the same algorithm that can be used in combination with either WCOP-CT or WCOP-SA (see line 19 in Algorithm 6).

Since the distortion caused by the anonymization of a dataset is heavily dependent on the original data and the dataset's privacy / quality requirements, it is possible that there will be combinations of strict distortion requirements and very demanding datasets that prohibit the discovery of a solution.

## 6. EXPERIMENTAL STUDY

In this section, we evaluate the effectiveness of our WCOP suite of methods for addressing the *Personalized (K,Δ)-anonymity problem* and its *Bounded* variation, as defined in Section 3. Namely, our suite consists of four algorithms: WCOP-NV, WCOP-CT, and WCOP-SA that address the first problem and WCOP-B that addresses the second problem.

We describe the experimental settings in Section 6.1. We make a base comparison between all the proposed algorithms in Section 6.2, while in Section 6.3, we briefly discuss the effects of  $(k, \delta)$  parameter values. In Section 6.4, we examine the results of having first partitioned the trajectories of the dataset into sub-trajectories using dataset-aware criteria. In Section 6.5, we validate the results of using trajectory editing to relax demanding trajectories' requirements so as to decrease anonymization distortion.

### 6.1 Experimental Setting

In this experimental study we use a real dataset to evaluate the performance of the examined algorithms. In particular, we use a sample of GeoLife dataset [14] reporting the traces of a group of individuals monitored in Beijing, consisting of 238 trajectories.

The dataset used in our experiments is visualized in Figure 3 whereas in Table 2, we report the characteristics of the dataset, namely the number of objects – users, the number of trajectories,  $|D|$ , the total number of spatiotemporal points composing those trajectories, the derived average speed, the half-diagonal of the minimum bounding box of the entire space that the dataset is covering,  $radius(D)$ , and the duration of the dataset.



**Figure 3: the trajectory dataset used in the experimental study (portion of GeoLife dataset).**

**Table 2: Statistics of GeoLife dataset**

GeoLife	
# objects (users)	72
# trajectories, $ D $	238
# spatiotemporal points	343,129
avg. speed (in m/s)	6.36
half-diagonal of entire space, radius( $D$ ) (in m)	51,982
dataset duration (in days)	1,477

$\delta_{max}$  parameter is set to 3% of  $radius(D)$ .  $trash_{max}$ , i.e., the maximum number of trajectories that can be suppressed, is set to 10% of  $|D|$ .  $radius_{max}$  is set equal to  $radius(D)$ . Finally, the tolerance thresholds of the EDR,  $\Delta = \{dx, dy, dt\}$ , are set as heuristic functions of  $\delta_{max}$ :  $\Delta = \{10 * \delta_{max}, 10 * \delta_{max}, 10 * \delta_{max} / avg\_speed\}$ , where  $avg\_speed$  is the average speed of all the moving objects in the dataset.

The experiments were performed on an Intel Xeon 2 GHz processor with 4 Gb of RAM and all the proposed algorithms were implemented in C.

## 6.2 Base Comparison

In this section, a base comparison between the proposed approaches is presented, in order to prove the validity of our personalized approaches. The dataset is used for this experiment with randomly generated  $(k, \delta)$  requirements for each trajectory,  $k \in [2, 100]$ ,  $\delta \in [10, 1400]$ . WCOP-NV finds the  $\max\{k_i\}$ ,  $\min\{\delta_i\}$  values in the dataset and uses them for all trajectories, ignoring their individual requirements, essentially replicating the way W4M works. WCOP-CT does not take universal  $(k, \delta)$  values as input, instead it parses each trajectory's specific  $(k_i, \delta_i)$  requirements from the dataset and uses them throughout the process. Moreover, WCOP-SA algorithms first converts the dataset into a set of sub-trajectories and then WCOP-CT is applied in order to anonymize them w.r.t. user preferences. Finally, WCOP-B improves the overall distortion when the dataset is anonymized with WCOP-CT by editing the most demanding trajectories.

Table 3 displays the results from the experiment previously described. In particular, it lists a number of useful statistics, such as the number of the input (sub-)trajectories, the number of created clusters, the number of trajectories and the number of trajectory points that ended to the trash bin, the discernibility

metric [3], which measures the data quality of the anonymized trajectories, the number of created and deleted points on trajectories, the average spatial and temporal translation per trajectory, the total distortion according to Eq.(2) (for WCOP-B Eq. (7) is used), and the runtime.

In particular, *discernibility* [3] aims at measuring the quality of the sanitized data. Given a set of clusters  $C_s = \{C_{s_1}, \dots, C_{s_m}\}$  of  $D$  and the trash bin, *Trash*, it is defined as:

$$DM = \sum_{i=1}^n |C_{s_i}|^2 + |Trash| \cdot |D| \quad (6)$$

Lower values of discernibility imply that more data elements are becoming indistinguishable.

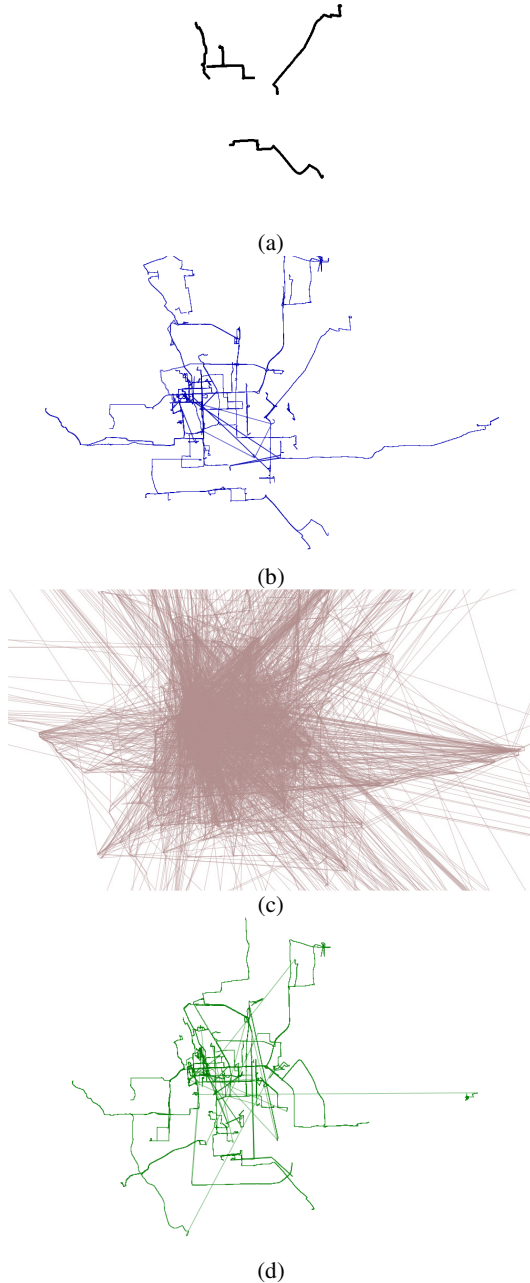
**Table 3: Comparison of WCOP-NV, WCOP-CT, WCOP-SA (Traclus & Convoys), WCOP-B anonymizing the GeoLife dataset with the same parameters ( $k_{max}=5$ ,  $\delta_{max}=250$ )**

Stat.	Algor.	WCOP-NV	WCOP-CT	WCOP-SA Traclus	WCOP-SA Convoys	WCOP-B
# (sub-)trajectories		238	238	17,717	272	238
# clusters		21	55	4,412	3	51
# trajectories moved to trash		17	6	83	2	4
# points moved to trash		25,103	9,189	3,634	2,731	5,706
discernibility ( $\times 10^6$ )		19.7	2.5	1,546	40.4	2.1
# created points		14,995	41,056	176,706	75,785	47,946
# deleted points		56,086	5,118	18,704	26,321	5,509
avg. spatial translation ( $\times 10^6$ )		453	4,612	48	1,638	525
avg. temporal translation ( $\times 10^6$ )		31,633	31,244	103	33,752	30,333
total distortion ( $\times 10^2$ )		10.5	8.6	2.5	9.9	8.2
runtime (seconds)		30	30	120	114	414

WCOP-NV causes greater values of distortion when compared to the other approaches. The minimum distortion and the maximum discernibility metric appear when the input is a set of sub-trajectories that are segmented with the use of Traclus algorithm, thus trajectories are assigned to clusters more effectively. Moreover, 7% of the trajectories and 7% of the trajectories' points are trashed when they are anonymized by using universal  $(k, \delta)$  privacy requirements, in contrast to WCOP-SA Traclus where the corresponding portions reaches 0.4% and 1% respectively. WCOP-B is able to decrease the overall distortion of the dataset by more than 20% when it is anonymized with WCOP-CT via editing the 6 most demanding trajectories (edit step is set to 1). Finally, runtime comparison shows that the approaches that anonymize sub-trajectories are slower than those that anonymize trajectories since a greater number of trajectories is processed. WCOP-B is even slower since every time that trajectories are edited it repeats the anonymization process until the distortion is lower than the threshold.

Based on the visualization of the aforementioned experiments as illustrated in Figure 4, we can conclude that the original trajectory dataset (see Figure 3) was better anonymized by WCOP-CT (Figure 4(b)) than by WCOP-NV (Figure 4(a)).

Clearly, WCOP-NV was not able to maintain the trend of the original trajectories. due to the reduced number of the created clusters. WCOP-CT and WCOP-SA-Convoy (Figure 4(d)) better preserved the pattern of the original trajectories. WCOP-SA-Traclus (Figure 4(c)) reports dense sanitized trajectories due to the segmentation of the original dataset that increased its size by 99%. Thus, we can argue that the result is expected.

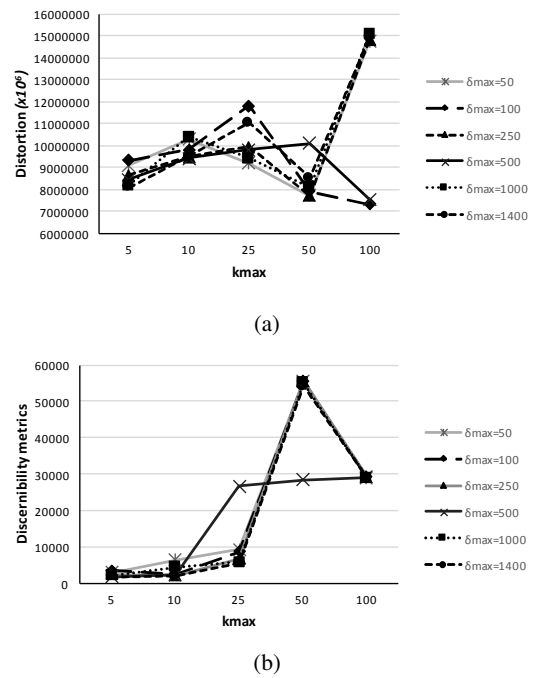


**Figure 4: Anonymized trajectories by (a) WCOP-NV; (b) WCOP-CT; (c) WCOP-SA-Traclus; (d) WCOP-SA-Convoy.**

### 6.3 The Effect of $(K, \Delta)$ Parameters

In this section, we examine the effects of using varying combinations of  $(K, \Delta)$  values with respect to the total information distortion caused by the anonymization. Each trajectory's  $(k, \delta)$  requirements are randomly generated, with  $k \in [2, k_{max}]$ ,  $\delta \in [10, \delta_{max}]$ . As mentioned above,  $\delta_{max}$  parameter has been set to 3% of the dataset bounding rectangle's radius. The  $k_{max}$  and  $\delta_{max}$  variables are varying on each iteration with  $k_{max}=\{5, 10, 25, 50, 100\}$  while  $\delta_{max}=\{50, 100, 250, 500, 1000, 1400\}$ .

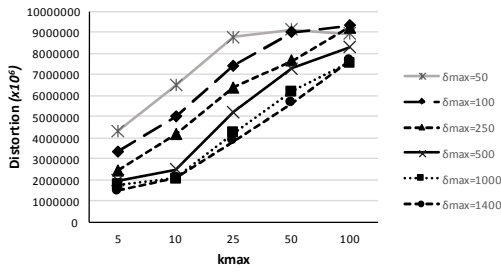
Focusing on WCOP-CT, Figures 5(a) and 5(b) provide a visual representation of the total distortion and the discernibility metric, respectively, for different combinations of  $(k_i, \delta_i)$ . It is clear that WCOP-CT is affected from the changes both in  $k_{max}$  and  $\delta_{max}$  parameters. However, there is a point in Figure 5(a) where while the distortion decreases with the increase of  $k_{max}$ , reaching the minimum values when  $k=50$ , a sudden increase appears when  $k=100$ . This is due to the fact that the number of the trash size increases up to a point that exceeds  $trash_{max}$ . When this occurs,  $radius_{max}$  is enlarged in order to cluster more trajectories thus the number of the removed trajectories is shrunk but trajectories are spatially translated even more. This trend is obvious at the overall distortion and the discernibility metric when  $k_{max}=25$  (Figures 5(a) and 5(b)).



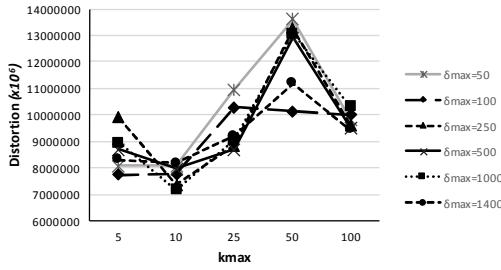
**Figure 5: WCOP-CT: (a) total distortion and (b) discernibility for different combinations of  $(k_{max}, \delta_{max})$ .**

### 6.4 The Effect of Trajectory Partitioning

In this section, we validate WCOP-SA algorithm. In particular, we compare the effects of using WCOP-CT with two different inputs of the GeoLife dataset, i.e. trajectories after being segmented into sub-trajectories using either Traclus [7] or Convoy [6]. Regarding  $k$  and  $\delta$ , they were again randomly generated with  $k \in [2, k_{max}]$  and  $\delta \in [10, \delta_{max}]$ .

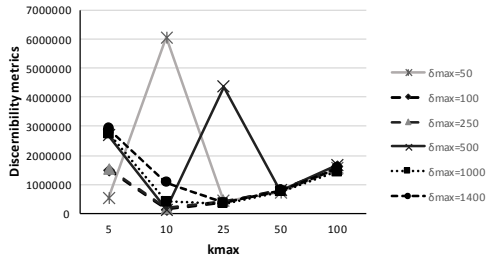


(a)

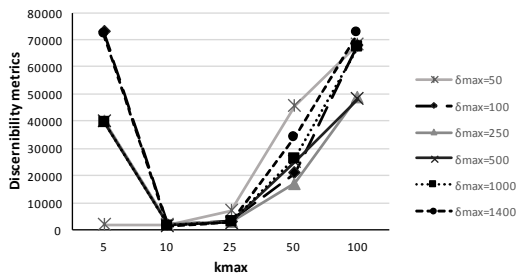


(b)

**Figure 6: total distortion for different combinations of ( $k_{max}$ ,  $\delta_{max}$ ) using (a) WCOP-SA-Traclus; (b) WCOP-SA-Convoys.**



(a)



(b)

**Figure 7: discernibility for different combinations of ( $k_{max}$ ,  $\delta_{max}$ ) using (a) WCOP-SA-Traclus; (b) WCOP-SA-Convoys.**

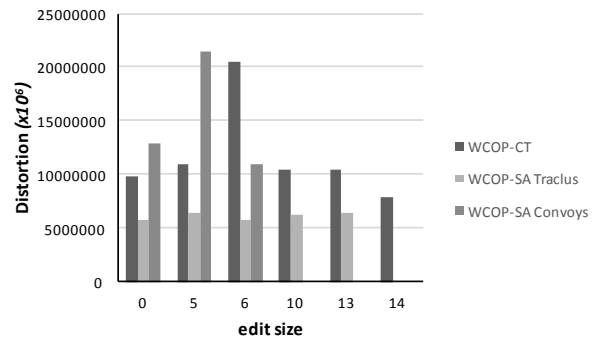
As we observe in the base comparison (see Table 3), partitioning the dataset into sub-trajectories results in very high discernibility, caused by the significantly higher number of clusters. Moreover, the segmentation of trajectories also appears to cause substantially decreased information distortion especially when they were partitioned with the Traclus algorithm. Figures 6(a) and 6(b) illustrate the total distortion in both approaches which rises as the value of  $k_{max}$  increases. Discernibility metrics in Figure 7(a) and 7(b) reports that the data quality is maintained either in lower or in higher values of  $k_{max}$ . It is worth noting that WCOP-SA with

Traclus manages to increase the average quality by 99% and decrease the average total distortion by 43% compared to the corresponding average values of WCOP-CT. Similarly, WCOP-SA when using the Convoys algorithm increases the average data quality by 31% and decreases the distortion by 2%.

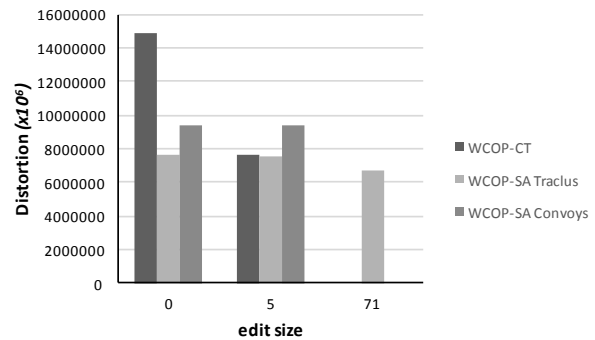
## 6.5 The Effect of Trajectory Editing

In the final part of our experimental study, we examine the effects of trajectory editing based on the algorithm outlined in Section 5. Two different versions of GeoLife dataset are applied in this set of experiments, using different ranges of randomly assigned ( $k, \delta$ ) values, i.e. [25, 500] and [100, 1400], in order to examine the effect of edit on the final result and how privacy requirements can influence it. Secondly, in order to examine the effects of trajectory editing on datasets of whole trajectories and on datasets consisting of segmented sub-trajectories, we apply WCOP-B in both types of data.

Figure 8(a) illustrates the effects of editing various numbers of trajectories for a dataset that corresponds to medium demanding users. In contrast, Figure 8(b) depicts the respective outcome but for much more demanding users. It is obvious that most of the approaches decreased 10% of their distortion by only editing the top-5 demanding trajectories apart from WCOP-SA-Traclus due to the increased number of sub-trajectories.



(a)



(b)

**Figure 8: WCOP-B: distortion for varying edit size values where (a)  $k_{max} = 25$  and  $d_{max} = 500$ ; (b)  $k_{max} = 100$  and  $d_{max} = 1400$ .**

It is not only that distortion changes in a non-monotone as edit size increases; we also observe that it can actually increase as edit size increases. This is due to the fact that each edited trajectory incurs a distortion penalty, which grows proportionally to the edit-size. However, the distribution of demanding trajectories across

the clusters and the distribution of  $(k, \delta)$  values in the dataset significantly influence the degree to which relaxing additional trajectories' requirements affect the clustering and anonymization phases. Therefore, higher percentage of edited trajectories does not guarantee decreased distortion, indicating that there exists an 'optimal' edit-size value, where distortion is the minimum possible.

## 7. CONCLUSIONS

In this paper, we proposed a novel approach for anonymizing trajectories called Personalized  $(K, \Delta)$ -Anonymity, which uses user-specific privacy requirements. Based on this framework, we have developed WCOP-CT algorithm, which takes advantage of user-specific  $(k, \delta)$  requirements in order to assign trajectories to clusters of minimal size, so as to avoid over-anonymization, increase data quality and decrease distortion. Expanding upon that framework, we made use of dataset-aware trajectory segmentation, in order to further improve our approach's effectiveness, by partitioning trajectories to sub-trajectories that are more easily assignable to clusters. Additionally, we examined the concept of Bounded  $(K, \Delta)$ -Anonymity, whereby there is a threshold to the acceptable distortion caused by the anonymization process, and proposed methods for trajectory assessment and editing by relaxing the requirements of the most demanding trajectories without editing the spatiotemporal data.

To show the effectiveness of our methods, we have performed experiments using the GeoLife dataset. Our personalized anonymity approach has been shown to significantly increase to the overall quality and to decrease of the total distortion of the anonymized datasets, while it has also been demonstrated that trajectory segmentation can improve data quality even further. Experimental results also show that our trajectory assessment and editing algorithms perform very well towards the goal of decreasing data distortion without altering the trajectories' spatiotemporal information itself.

Overall, we argue that we have provided a novel approach in mobility data anonymization. Using our WCOP suite of techniques, data analysts are able to preserve the quality of anonymized datasets taking advantage of user-specific privacy requirements combined with methods such as segmentation and trajectory editing. However, there is a number of points, such as sensitivity to  $(k, \delta)$  values distribution, replacement of greedy clustering with a more sophisticated clustering method, sensitivity to segmentation method and alternative trajectory assessment and editing methods, which deserve further study in order to expand and improve upon the framework presented here.

## 8. ACKNOWLEDGMENTS

The publication of this paper has been partly supported by the University of Piraeus Research Center.

## 9. REFERENCES

[1] Abul, O., Bonchi, F., Nanni, M. (2008). Never walk alone: Uncertainty for anonymity in moving objects databases. *In*

*Proceedings of International Conference on Data Engineering, ICDE*, pp. 376-385.

[2] Abul, O., Bonchi, F., Nanni, M. (2010). Anonymization of moving objects databases by clustering and perturbation. *Information Systems*, 35(8), pp. 884-910.

[3] Bayardo, R. J., Agrawal, R. (2005). Data privacy through optimal  $k$ -anonymization. *In Proceedings of the International Conference on Data Engineering*, pp. 217-228.

[4] Chen, L., Özsu, M. T., Oria, V. (2005). Robust and fast similarity search for moving object trajectories. *In Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 491-502.

[5] Chow, C. Y., Mokbel, M. F. (2011). Trajectory privacy in location-based services and data publication. *ACM SIGKDD Explorations Newsletter*, 13(1), pp. 19-29.

[6] Hoh, B., Gruteser, M. (2005). Protecting location privacy through path confusion. *In Proceedings of SecureComm*, pp. 194-205.

[7] Jeung, H., Yiu, M. L., Zhou, X., Jensen, C. S., Shen, H. T. (2008). Discovery of convoys in trajectory databases. *In Proceedings of the VLDB Endowment*, 1(1), pp. 1068-1080.

[8] Lee, J. G., Han, J., & Whang, K. Y. (2007). Trajectory clustering: a partition-and-group framework. *In Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 593-604.

[9] MahdaviFar, S., Abadi, M., Kahani, M., Mahdikhani, H. (2012). A clustering-based approach for personalized privacy preserving publication of moving object trajectory data. *In Proceedings of Network and System Security*, pp. 149-165.

[10] Monreale, A., Andrienko, G. L., Andrienko, N. V., Giannotti, F., Pedreschi, D., Rinzivillo, S., Wrobel, S. (2010). Movement data anonymity through generalization. *Transactions on Data Privacy*, 3(2), pp. 91-121.

[11] Nergiz, M. E., Atzori, M., Saygin, Y. (2008). Towards trajectory anonymization: a generalization-based approach. *In Proceedings of the ACM International Workshop on Security and Privacy in GIS and LBS, SIGSPATIAL* pp. 52-61.

[12] Sweeney, L (2002)  $k$ -anonymity: a model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge Based Systems*, 10(5), pp. 557-570.

[13] Terrovitis, M., Mamoulis, N. (2008). Privacy preservation in the publication of trajectories. *In Proceedings of International Conference on Mobile Data Management MDM*, pp. 65-72.

[14] Zheng, Y., Xie, X., Ma, W.-Y. (2010). GeoLife: A Collaborative Social Networking Service among User, Location and Trajectory. *IEEE Data Eng. Bull.*, 33(2), pp. 32-39.



# Identifying and Describing Streets of Interest

Dimitrios Skoutas  
IMIS, Athena R.C., Greece  
dskoutas@imis.athena-  
innovation.gr

Dimitris Sacharidis  
TU Wien, Austria  
dimitris@ec.tuwien.ac.at

Kostas Stamatoukos  
IMIS, Athena R.C., Greece  
kstamatoukos@imis.athena-  
innovation.gr

## ABSTRACT

The amount of crowdsourced geospatial content on the Web is constantly increasing, providing a wealth of information for a variety of location-based services and applications. This content can be analyzed to discover interesting locations in large urban environments which people choose for different purposes, such as for entertainment, shopping, business or culture. In this paper, we focus on the problem of identifying and describing Streets of Interest. Given the road network in a specified area, and a collection of geolocated Points of Interest and photos in this area, our goal is to identify the most interesting streets for a specified category or keyword set, and to allow their visual exploration by selecting a small and spatio-textually diverse set of relevant photos. We formally define the problem and we present efficient algorithms, based on spatio-textual indices and filter and refinement strategies. The proposed methods are evaluated experimentally regarding their effectiveness and efficiency, using three real-world datasets containing road networks, POIs and photos collected from several Web sources.

## 1. INTRODUCTION

A large amount of user-generated content is becoming available on the Web daily, with increasingly large portions of it being associated with geospatial information. Typical examples include maps of road networks and other spatial features available on OpenStreetMap and Wikimapia, information about *Points of Interest* (POIs) from Wikipedia and Foursquare, geotagged photos from Flickr and Panoramio. This creates a valuable resource for discovering and exploring locations and areas of interest, with numerous applications in location-based services, geomarketing, trip planning, and other domains. In this paper, we advocate the use of *street* as the elemental area of interest in modern cities, and tackle two complementary problems, *identifying* and *describing* them.

Regarding the first task, there has been substantial work in identifying a *single POI*, based on spatial and textual criteria. More precisely, spatio-textual similarity queries, aim at retrieving POIs that are both spatially close to a given location and textually relevant to a given set of keywords specifying an information need [9]. Additional metadata associated to the POIs, such as ratings, comments, “likes” or check-ins, can be considered to weigh the *import-*

*tance* of each POI when computing the ranking. The proximity of a POI to other relevant POIs has also been considered as a factor indicating importance [5]. Furthermore, in Location-Based Social Networks, information about social connectivity is also considered in determining the importance of POIs (and users) [4, 2].

A line of work more related to our first task deals with discovering a set of nearby and topically related POIs, that designate a *Region of Interest*. This is a more challenging problem, and proposals mainly differ in the way regions are formed. The majority of past research addresses the *maximum range sum problem*, where the region is defined as either a rectangle of fixed length and width [21, 24, 10], or a disk with fixed radius [10], and the objective is to find the one that maximizes an aggregate importance score on the topically relevant POIs it contains. Other works do not enforce a constraint on region shape or size, and rather implement density-based clustering of POIs [20, 19].

All aforementioned works assume that POIs are located in a Euclidean space. However, particularly in urban environments, it is often more realistic and useful to consider the underlying road network. Grouping together nearby POIs makes little sense if the actual travel distance between them is large, e.g., when they are located in opposite banks of a river. Surprisingly, little work is done in this frontier. In [7], the authors look for a connected subgraph of the road network that maximizes an aggregate score on the relevant POIs that are included, subject to a constraint on its total length.

Nonetheless, such an approach also has shortcomings. First, the fact that there is no control on the *subgraph type* may result in returning oddly-shaped regions that are hard to inspect and not particularly meaningful in a user exploration setting. Additionally, the approach favors *POI quantity* over density. More often than not, there exists a single popular street with a high density of POIs. Using the formulation of [7], such a street would be in the result but accompanied by several other smaller adjacent streets that happen to have at least one relevant POI. Similarly, looking for *connected components* may lead to discovering artificial links among important streets for the sole purpose of ensuring connectivity. Another limitation is that [7] assumes POIs are conveniently situated on the road network as vertices. In reality, however, the situation is much different. Figure 1(a) shows the map of a popular corner (Oxford Str. and Regent Str.) in the center of London, and also depicts various types of POIs. It should be apparent that there is no straightforward mapping of POIs to the road network vertices. Instead, it is more natural to “assign” POIs to edges (streets) but not necessarily in an exclusive manner. For example a POI (e.g., the clothing shop in Figure 1(a)) near the corner should be associated with both intersecting streets. Moreover, a POI (e.g., the photo shop in Figure 1(a)) farther from the streets but inside a corner building should also contribute to the importance of the main crossing streets.



Figure 1: Illustrative example for shopping streets in the center of London.

Motivated by these observations, we formulate the problem of *identifying Streets of Interest* (SOIs). Briefly, given some textual information (keywords, categories) the goal is to identify the streets (more accurately the street segments) that have a large density of relevant POIs around them. An example for the center of London is illustrated in Figure 1(b), where the top 20-SOIs are highlighted with red. Notice that the returned streets are not connected via non-interesting streets. Moreover, our ranking approach naturally allows for an exploratory search of the area. To efficiently retrieve a ranked list of SOIs, we propose an algorithm inspired from top- $k$  query processing that operates on top of spatio-textual indices.

Although very helpful, identifying the main SOIs for the user’s keywords is only the first step towards exploring a larger area. Typically, the user then needs to find more information and gain more insights about those results that are discovered and suggested. For this purpose, perhaps the easiest and most effective way is by providing visual information; thus, a valuable source is the numerous relevant photos that can be found in various Web sources, such as Flickr and Panoramio. The challenge that arises then is how to select a small set of results to present, in order to avoid overloading the user, especially when using a mobile device with limited resources in terms of bandwidth, screen size and battery, while still providing enough information. This can be achieved by *diversifying* the results to present, so that more and different information can be conveyed with fewer results.

To that end, the second task we address in this paper refers to the selection of a concise and spatio-textually diverse set of relevant photos for describing the discovered SOIs. We follow the general diversity principles from information retrieval [16, 8], and formulate a *spatio-textual SOI diversification* problem. In particular, we introduce spatial and textual measures of *relevance* and *diversity*, and seek to extract a small set of photos that act as an informative summary of a given SOI (see the example in Figure 1(c) for a 4-photo summary of Oxford Str). As this is a computationally hard problem, we turn into heuristic methods supported by appropriate spatio-textual indices.

The main contributions of our work can be summarized as follows:

- We formally define the top- $k$  SOI query, and we present an efficient algorithm for its evaluation.
- We present spatio-textual relevance and diversity criteria for selecting subsets of available photos to describe SOIs, and we propose an efficient approximation algorithm for their computation.
- We present the results of an experimental evaluation of our proposed methods, using real-world datasets containing road

networks, POIs and photos from several major Web sources, covering the areas of three different European capital cities.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 formally introduces the problem of finding  $k$ -SOIs, and presents an efficient algorithm for their computation. Section 4 presents measures for spatio-textual relevance and diversity, and an efficient approximation algorithm for selecting diversified subsets of relevant photos to describe SOIs. Finally, Section 5 presents the results of our experimental evaluation, while Section 6 concludes the paper.

## 2. RELATED WORK

This section reviews existing approaches for spatio-textual POI retrieval and diversification.

### 2.1 Ranking Points and Areas of Interest

Numerous works have focused on discovering and ranking points or areas of interest, based on various definitions and criteria. The main differences involve the following aspects: (a) whether the focus is on single POIs or whole areas, i.e. sets of POIs; (b) whether the problem involves nearby search around a given query location or rather browsing and exploration within a whole area; (c) whether the aim is to maximize the number or the total score (e.g. relevance or importance) of the POIs enclosed in the discovered area or to minimize some cost function (e.g. distance or travel time) on a set of POIs that suffice for covering the query keywords.

The majority of existing works focuses on the ranking of single POIs. Location-aware top- $k$  text retrieval queries have been studied in [11]. Given the user location and a set of keywords, this query returns the top- $k$  POIs ranked according to both their spatial proximity and their textual relevance to the query. For the efficient evaluation of such queries, a hybrid indexing approach was proposed, integrating the inverted file for text retrieval and the R-tree for spatial proximity querying. Further variations on spatio-textual queries and indexes have been extensively studied [9]. Top- $k$  spatial keyword queries have been studied also in [23], with distances being calculated on the road network instead of the Euclidean space. A different perspective for ranking POIs is taken in [5], where the importance of a POI takes into account the presence of other relevant nearby POIs.

Queries involving sets of spatio-textual objects have been investigated in [6, 27]. Given a set of keywords and, optionally, a user location, the goal is to identify sets of POIs that collectively satisfy the query keywords while minimizing the maximum distance or the sum of distances between each other and to the query.

More recently, other works have focused on discovering regions

of interest w.r.t. a specified category or set of keywords, where the importance of a region is determined based on the number or the total weight of relevant POIs it contains. In [19], density-based clustering is applied to identify regions with high concentration of POIs of certain categories, collected and integrated from several Web sources. A method for extracting scenic routes from geo-tagged photos uploaded on sites such as Flickr and Panoramio is presented in [3]. Discovering and recommending regions of interest based on user-generated data is also addressed in [20]. The quality of a recommended area is determined based on the portion of the contained POIs that can be visited within a given time budget. Other variations of queries for discovering interesting regions include the subject-oriented top- $k$  hot region query [21] and the maximizing range sum query [10]. The region is defined by a rectangle or circle with a maximum size constraint, and the goal is to maximize the score of the relevant POIs contained in it.

The most closely related work to our approach is [7], which proposes the length-constrained maximum-sum region query. Given a set of POIs in an area and a set of keywords, this query computes a region that does not exceed a given size constraint and that maximizes the score of the contained POIs that match the query keywords. The query assumes an underlying road network, in which the POIs are included as additional vertices, and the returned region has the form of a connected subgraph of this network with arbitrary shape. The problem is shown to be NP-hard, and approximation algorithms are proposed.

This problem is similar to our setting; in both cases, the goal is to discover interesting parts of a road network that are associated with large number of POIs relevant to a given set of keywords. However, in [7], the result is a single connected subgraph of the road network, maximizing the score of contained points, while our method returns a ranked list of streets that are not necessarily connected and are ordered according to their density w.r.t. POIs relevant to the query. Moreover, we additionally consider the problem of describing these discovered streets by means of a diversified set of photos, which is not addressed in [7] or any of the other works mentioned before.

Finally, in a different line of research, other works have applied probabilistic topic modeling on user-generated spatio-textual data and events to associate urban areas with topics and patterns of user mobility and behavior [18, 15].

## 2.2 Search Results Diversification

Information retrieval engines often try to improve the utility of the search results by taking into account not only their relevance to the user's query, but also their dissimilarity, offering thus a range of alternatives, which comes handy in situations where the true intent of the user is unknown or many highly similar objects exist. Stated in an abstract manner, the content-based diversification problem is to determine a set of objects that maximizes an objective function with two components, the relevance and the diversity.

While there exist many different formulations (refer to [16, 13] for classification), the most well-known is the MaxSum problem, where the goal is to maximize the weighted sum of two components, the total relevance of objects, and the sum of pairwise diversities among the objects. Similar to other diversification problems, MaxSum is NP-hard as it is related to the dispersion problem [22]. Therefore, various greedy heuristics are proposed. Typically, they incrementally construct the diversified result set by choosing at each step the object that maximizes a certain scoring function. The most well-known function is the *maximal marginal relevance (mmr)* [8]. An evaluation of various object scoring functions and different heuristics can be found in [26].

Since [8], several works addressed other diversification prob-

lems, such as taxonomy/classification-based diversification [1], [25] or multi-criteria diversification [12]. Another related work is the coverage problem [14], where the goal is to select a set of diverse objects that cover the entire database.

## 3. IDENTIFYING STREETS OF INTEREST

We first formulate the problem of identifying interesting streets, and then present our proposed approach.

### 3.1 Problem Definition

A road network is a directed graph  $G = (\mathcal{V}, \mathcal{L})$ , where the set of vertices  $\mathcal{V}$  contains street intersections or breakpoints in streets, and the set of links  $\mathcal{L}$  contains street segments (between intersections or breakpoints) represented as line segments. Each vertex  $v \in \mathcal{V}$  is associated with its coordinates  $(x_v, y_v)$ . The length  $len(\ell)$  of a segment  $\ell \in \mathcal{L}$  is computed as the Euclidean distance between its endpoints. We also consider the set of streets  $\mathcal{S}$ , where each street  $s \in \mathcal{S}$  comprises a set of consecutive segments (a simple path on  $G$ ). Each segment  $\ell \in \mathcal{L}$  belongs to a unique street  $s$ , and we denote this relationship by  $\ell \in s$ . The length  $len(s)$  of street  $s$  is the sum of the length of its segments.

Moreover, we define an additional data source  $\mathcal{P}$ , being a set of POIs. Each POI  $p \in \mathcal{P}$  is defined by a tuple  $p = \langle (x_p, y_p), \Psi_p \rangle$ , where  $(x_p, y_p)$  are the coordinates of the POI, and  $\Psi_p$  is a set of keywords describing this POI (e.g., keywords derived from its name, description, tags). The distance  $dist(p, \ell)$  of a POI  $p$  to a line segment  $\ell$  is defined as the minimum Euclidean distance between POI location  $(x_p, y_p)$  and any point on  $\ell$ . Accordingly, the distance of POI  $p$  to a street  $s$  is the minimum distance of  $p$  to any segment of  $s$ , i.e.,  $dist(p, s) = \min_{\ell \in s} dist(p, \ell)$ .

To measure the interest of a street segment w.r.t. a given set of keywords, we use the notion of *mass*, which refers to the number of relevant POIs that exist in its proximity. Then, we rank segments according to their *mass density*, to account also for the different lengths of each segment. We formally define these concepts below.

**DEFINITION 1 (SEGMENT MASS).** *For a given set of keywords  $\Psi$  and a distance threshold  $\epsilon$ , the mass of segment  $\ell$  is the number of POIs within distance  $\epsilon$  that contain at least one keyword from  $\Psi$ :*

$$mass(\ell | \Psi, \epsilon) = |\{p \in \mathcal{P} : dist(p, \ell) \leq \epsilon \ \& \ \Psi_p \cap \Psi \neq \emptyset\}|.$$

Note that this definition can be straightforwardly adapted in the case that POIs have different weights.

**DEFINITION 2 (SEGMENT INTEREST).** *The interest of segment  $\ell$  is its mass density, i.e., the ratio of  $\ell$ 's mass over the size of the area within distance  $\epsilon$  around  $\ell$ :*

$$int(\ell | \Psi, \epsilon) = \frac{mass(\ell | \Psi, \epsilon)}{2\epsilon len(\ell) + \pi\epsilon^2}.$$

Given this definition for the interest of a segment, there exist several alternatives for defining the interest of an entire street. Here, we use a simple definition, as stated below.

**DEFINITION 3 (STREET INTEREST).** *Given  $\Psi$  and  $\epsilon$ , the interest of a street  $s$  is the maximum interest among its segments, i.e.:*

$$int(s | \Psi, \epsilon) = \max_{\ell \in s} int(\ell | \Psi, \epsilon). \quad (1)$$

Based on these definitions, a  $k$ -SOI query returns the  $k$  most interesting streets.

**Problem 1. [ $k$ -SOI]** Assume a set of streets  $\mathcal{S}$ , forming a road network  $G$ , and a set of POIs  $\mathcal{P}$ . The  $k$ -Streets of Interest ( $k$ -SOI)

query  $q = \langle \Psi, k, \epsilon \rangle$ , where  $\Psi$  is a set of keywords,  $k$  a positive integer, and  $\epsilon$  a distance threshold, returns a set of  $k$  streets  $S^k$  such that for each  $s' \notin S^k$  it holds that  $\text{int}(s' | \Psi, \epsilon) \leq \min_{s \in S^k} \text{int}(s | \Psi, \epsilon)$ .

## 3.2 The SOI Algorithm

In what follows, we present the SOI (Streets Of Interest) algorithm. We assume a given query  $q = \langle \Psi, k, \epsilon \rangle$ ; for brevity,  $\Psi$  and  $\epsilon$  are omitted when it is clear from the context.

### 3.2.1 Methodology and Indices

The SOI algorithm for processing  $k$ -SOI queries operates in a manner reminiscent of top- $k$  processing algorithms [17]. It progressively examines segments of streets and POIs until it can establish that the  $k$ -SOI can be determined solely from the information already collected. Specifically, SOI maintains a *seen lower bound*  $LB_k$  on the interest of the  $k$  best streets encountered so far, and an *unseen upper bound*  $UB$  on the interest of any street for which no segment has been encountered yet. As more segments are considered,  $LB_k$  progressively increases, while  $UB$  progressively decreases. When  $LB_k$  becomes not smaller than  $UB$ , the examination stops, since the  $k$ -SOIs are those streets with interest not smaller than  $LB_k$ .

Under the aforementioned strategy, there are two issues to address: (1) how to compute the seen lower bound  $LB_k$  and the unseen upper bound  $UB$ , and (2) how to expedite the termination condition,  $LB_k \geq UB$ .

We address the former first, as its solution gives intuition about the latter. Based on the definition of street interest (Equation 1), we can compute bounds on the interest of a street  $s$  by considering directly the segments. The following lemma suggests a method to compute  $LB_k$  and  $UB$ ; their exact definition is presented later.

LEMMA 1. Consider a subset of segments  $\mathcal{L}_{seen} \subseteq \mathcal{L}$ . Then, for a street  $s$  it holds that:

$$\begin{aligned} \text{int}(s) &\geq \max_{\ell \in s \cap \mathcal{L}_{seen}} \text{int}(\ell), & \text{if } s \cap \mathcal{L}_{seen} \neq \emptyset \\ \text{int}(s) &\leq \max_{\ell \in \mathcal{L} \setminus \mathcal{L}_{seen}} \text{int}(\ell), & \text{if } s \cap \mathcal{L}_{seen} = \emptyset. \end{aligned}$$

PROOF. For the first case observe that  $s \cap \mathcal{L}_{seen} \subseteq s$ , and thus  $\max_{\ell \in s \cap \mathcal{L}_{seen}} \text{int}(\ell) \leq \max_{\ell \in s} \text{int}(\ell) = \text{int}(s)$ . For the second case observe that  $\mathcal{L} \setminus \mathcal{L}_{seen} \supseteq s$ , and thus  $\max_{\ell \in \mathcal{L} \setminus \mathcal{L}_{seen}} \text{int}(\ell) \geq \max_{\ell \in s} \text{int}(\ell) = \text{int}(s)$ .  $\square$

Consider a seen street  $s$ , meaning that one of its segments has been encountered, i.e.,  $s \cap \mathcal{L}_{seen} \neq \emptyset$ . The first case of Lemma 1 implies that a lower bound on the interest of  $s$  can be extracted from the largest interest of its seen segments, or, more practically, from the largest lower bound on the interest of any of its seen segments.

On the other hand, consider an unseen street  $s$ , i.e.,  $s \cap \mathcal{L}_{seen} = \emptyset$ . The second case of Lemma 1 implies that an upper bound on the interest of  $s$  can be extracted from the largest possible interest among unseen segments, or, more practically, from an upper bound on the largest possible interest among unseen segments.

In other words, Lemma 1 directly addresses the former issue. However, it also suggests how to address the latter. Suppose we have encountered a set of segments  $\mathcal{L}_{seen}$ . What segments should it contain so as to increase the chances of satisfying the termination condition? To obtain a high seen lower bound  $LB_k$ , the first case of Lemma 1 suggests putting in  $\mathcal{L}_{seen}$  segments with high interest. Moreover, to obtain a small unseen upper bound  $UB$ , the second case of Lemma 1 suggests leaving out from  $\mathcal{L}_{seen}$  segments with

low interest. Therefore, the algorithm should try to visit segments with large interest first.

As it is impractical to directly retrieve segments by interest (that would require precomputation for all possible  $k$ -SOI queries, i.e., for arbitrary  $\epsilon, \Psi$ ), we need a way to identify promising segments having large interest. To this end, we employ the following data structures.

- A *spatial grid index* with arbitrary cell size storing all POIs. Within each cell  $c$ , there is a *local inverted index* on the set of keywords among the cell POIs. The entry for keyword  $\psi$  is a list of POIs sorted increasingly on POI id.
- A *global inverted index* on the set of all keywords. The entry for keyword  $\psi$  is a list of  $\langle c, \text{numPOIs} \rangle$  entries sorted decreasingly on  $\text{numPOIs}$ , which is the number of POIs within cell  $c$  that contain keyword  $\psi$ .
- A *cell-to-segment map* that stores for each grid cell the segments that pass through it. At query time when  $\epsilon$  is known, the map is augmented to contain for each cell all segments that are within distance  $\epsilon$ . We denote the augmented list for cell  $c$  as  $\mathcal{L}_\epsilon(c)$ .
- A *segment-to-cell map* that stores for each segment the grid cells that it intersects. At query time when  $\epsilon$  is known, the map is augmented to contain for each segment all cells that are within distance  $\epsilon$ . We denote the augmented list for segment  $\ell$  as  $\mathcal{C}_\epsilon(\ell)$ .
- A *list of segments* sorted increasingly on their length.

Note that since street segments and POIs are relatively static, these data structures can be created and maintained offline.

### 3.2.2 Algorithm Description

In what follows, we discuss the case of a query specifying a single keyword, i.e.,  $\Psi = \{\psi\}$ . Intuitively, we look for segments that have large *mass* and small *len*. Thus, given the above data structures, we look for segments that (1) are close (within distance  $\epsilon$ ) to cells with large number of relevant POIs (satisfying  $\psi$ ), (2) are close to many cells, and (3) have small *len*. The first two factors combined contribute to the *mass*, while the third directly to *len*. Therefore, the algorithm considers segments according to the following three ranked source lists constructed, in part, at query time.

$SL_1$ : Contains all cells sorted decreasingly on the number of POIs with keyword  $\psi$ . This is essentially the list of the global inverted index for keyword  $\psi$ .

$SL_2$ : Contains all segments sorted decreasingly on the number of cells within distance  $\epsilon$  to them (the cells each segment intersects when enlarged by  $\epsilon$ ).

$SL_3$ : Contains all segments, sorted increasingly on their length.

The algorithm proceeds iteratively, considering in each iteration either the next cell from  $SL_1$  or the next segment from  $SL_2$  or  $SL_3$ . Each source list can be accessed in a round robin fashion; the correctness of our method is not affected by the access strategy. In practice, we alternate between  $SL_1$  and  $SL_3$ , trying to balance the number of segments considered from each source; each cell access results in the access of multiple segments, while each segment access causes the visit of multiple cells. We only access segments via the second source  $SL_2$  in the case that a few segments with a large number of neighboring cells exist.

During the processing of  $k$ -SOI, a segment can be in three possible states. Initially, a segment is *unseen*, meaning that the algorithm

has not considered it via any source. An efficient algorithm would leave many segments in the unseen phase. Then, when a segment is first retrieved, it is put into the *partial* state, meaning that some, but not all, POIs near it that satisfy the query have been accounted for. For each partial segment  $\ell$ , we maintain two pieces of information: (1) the count  $mass^-(\ell)$  of relevant POIs seen so far that satisfy the query, and (2) a list  $toVisit$  indicating which neighboring cells (and consequently their POIs) to visit. Based on these, we can compute a lower bound on  $\ell$ 's interest as:

$$int(\ell) \geq int^-(\ell) = \frac{mass^-(\ell)}{2\epsilon len(\ell) + \pi\epsilon^2}.$$

Once all relevant POIs have been processed (equivalently, all neighboring cells have been visited), the segment is in the *final* state, where its exact interest is known.

Algorithm 1 presents the pseudocode for the SOI algorithm. During initialization, SOI prepares the three source lists. Particularly, it builds  $SL_1$  by examining the global inverted index (lines 1–3); for the simple case of one keyword  $\psi$ ,  $SL_1$  is essentially the inverted list  $I[\psi]$ . Moreover, source list  $SL_3$  corresponds to the list of segments, while  $SL_2$  is extracted from the augmented segment-to-cell map (lines 4–7).

Then, SOI proceeds to the main filtering phase (lines 8–24), where segments from the source lists are examined until the termination condition  $LB_k \geq UB$  holds; initially  $LB_k$  and  $UB$  are set to zero and infinity, respectively (line 9).

Assume that cell  $c$  via source list  $SL_1$  is to be accessed (lines 11–15). For this cell we examine the segments that are within distance  $\epsilon$  (lines 13–14), and for each segment we determine the number of POIs with keyword  $\psi$  that are within distance  $\epsilon$  so as to update their interest score given the contents of cell  $c$ . Specifically, we employ the cell-to-segment map to determine all segments  $\mathcal{L}_\epsilon(c)$  that are within distance  $\epsilon$  to cell  $c$  (line 13). For each such segment  $\ell$ , we invoke the procedure `UpdateInterest` (line 14), whose pseudocode is also depicted. After the procedure returns, we set the next source list to consider according to round robin (line 15).

Procedure `UpdateInterest` first checks whether cell  $c$  has been visited for  $\ell$  and immediately returns if so. Otherwise, it removes  $c$  from the  $toVisit$  list. Then it visits the local inverted index of cell  $c$  and retrieves the list for keyword  $\psi$ . For each POI  $p$  in the list, `UpdateInterest` checks whether it is within distance  $\epsilon$  to segment  $\ell$ , and if true increments  $mass^-(\ell)$  by one.

Now, assume that segment  $\ell$  is to be accessed (lines 16–21), either via source list  $SL_3$  or  $SL_2$ . Its exact interest will be determined, changing its state to final. Using the segment-to-cell map, we visit sequentially all neighboring cells of  $\ell$ , and invoke procedure `UpdateInterest` (lines 18–19). A cell visited during some segment access, may be again visited due to another segment's access or via a direct cell access via source list  $SL_1$ . The next step is to properly set the next source list to consider (lines 20–21).

After each access, cell or segment, algorithm SOI checks whether the termination condition applies. It first computes an upper bound  $UB$  on the interest of any *unseen* segment (line 22). Let  $top(SL_1)$ ,  $top(SL_2)$ ,  $top(SL_3)$  denote the top items in the corresponding sources lists that are to be accessed next. Due to the second case of Lemma 1, it computes the unseen interest upper bound as:

$$UB = \frac{top(SL_1) \cdot top(SL_2)}{2\epsilon top(SL_3) + \pi\epsilon^2}.$$

SOI also computes a lower bound  $LB_k$  on the interest of the  $k$ -SOIs based on the first case of Lemma 1 (lines 23–24). It maintains all seen segments in a ranked list  $\mathcal{L}_{seen}$  sorted decreasingly on the interest lower bound  $int^-(\cdot)$ . Let  $\mathcal{L}_{seen}[i]$  denote the first  $i$  items

---

### Algorithm 1: Algorithm SOI

---

**Input:** network  $G$ , streets  $S$ , query  $q = \langle \Psi, k, \epsilon \rangle$   
**Output:**  $k$ -SOIs  $S^k$   
 $\triangleright$  build source list  $SL_1$   
1 **foreach** cell  $c$  that has an entry in  $I[\psi]$  for some  $\psi \in \Psi$  **do**  
2    $|\mathcal{P}_\Psi(c)| \leftarrow \min\{|\mathcal{P}_c|, \sum_{\psi \in \Psi} I[\psi][c]\}$   
3   insert entry  $\langle c, |\mathcal{P}_\Psi(c)| \rangle$  in  $SL_1$   
 $\triangleright$  build source lists  $SL_3, SL_2$   
4 **foreach** segment  $\ell$  **do**  
5   insert entry  $\langle \ell, len(\ell) \rangle$  in  $SL_3$   
6    $|\mathcal{C}_\epsilon(\ell)| \leftarrow \{c \in \mathcal{C} \mid dist(c, \ell) \leq \epsilon\}$   
7   insert entry  $\langle \ell, |\mathcal{C}_\epsilon(\ell)| \rangle$  in  $SL_2$   
 $\triangleright$  filtering phase  
8  $SL \leftarrow SL_1$   $\triangleright$  next source list  
9  $LB_k \leftarrow 0; UB \leftarrow \infty$   
10 **while**  $UB > LB_k$  **do**  
11   **if**  $SL = SL_1$  **then**  
12      $c \leftarrow pop(SL)$   $\triangleright$  retrieve cell  
13     **foreach**  $\ell \in \mathcal{L}_\epsilon(c)$  **do**  $\triangleright \ell$  within distance  $\epsilon$  to  $c$   
14        $\lfloor UpdateInterest(\ell, c, \Psi)$   
15        $SL \leftarrow SL_2$   $\triangleright$  set next source list  
16   **else**  
17      $\ell \leftarrow pop(SL)$   $\triangleright$  retrieve segment  
18     **foreach**  $c \in \mathcal{C}_\epsilon(\ell)$  **do**  $\triangleright \ell$  within distance  $\epsilon$  to  $c$   
19        $\lfloor UpdateInterest(\ell, c, \Psi)$   
20       **if**  $SL = SL_2$  **then**  $SL \leftarrow SL_3$   $\triangleright$  set next source list  
21       **else**  $SL \leftarrow SL_1$   
22      $UB \leftarrow \frac{top(SL_1) \cdot top(SL_2)}{2\epsilon top(SL_3) + \pi\epsilon^2}$   
23      $\mu \leftarrow \min_i : |\{s \mid \exists \ell \in \mathcal{L}_{seen}[i], \ell \in s\}| = k$   
24      $LB_k \leftarrow int^-(\ell_\mu)$   
 $\triangleright$  refinement phase  
25 **foreach**  $\ell \in \mathcal{L}_{seen}$  **do**  
26   **foreach**  $c \in \mathcal{C}_\epsilon(\ell)$  **do**  $\triangleright \ell$  within distance  $\epsilon$  to  $c$   
27      $\lfloor UpdateInterest(\ell, c, \Psi)$   
28 **return**  $S^k \leftarrow$  extract  $k$ -SOIs from  $\mathcal{L}_{seen}$

---

### Procedure `UpdateInterest`( $\ell, c, \Psi$ )

---

1 **if**  $c \notin \ell.toVisit$  **then return**  $\triangleright c$  is already visited for  $\ell$   
2 remove  $c$  from  $\ell.toVisit$   
 $\triangleright$  traverse lists  $c.I(\psi), \forall \psi \in \Psi$  synchronously  
3 **foreach**  $p \in \bigcup_{\psi \in \Psi} c.I(\psi)$  **do**  
4    $\lfloor mass^-(\ell) \leftarrow mass^-(\ell) + 1$

---

in the list. Then,  $LB_k$  is set to the interest lower bound of the  $\mu$ -th ranked segment  $\ell_\mu$  provided that  $\mu$  is the smallest index such that the segments of  $\mathcal{L}_{seen}[\mu]$  belong to  $k$  distinct streets, i.e.,

$$LB_k = int^-(\ell_\mu), \text{ for } \mu = \min_i : |\{s \mid \exists \ell \in \mathcal{L}_{seen}[i], \ell \in s\}| = k.$$

The accesses on source lists stop as soon as  $UB \leq LB_k$ . At that point, it is guaranteed that the result to  $k$ -SOI can be extracted from the segments in  $\mathcal{L}_{seen}$ . To identify the streets with the top- $k$  interest, a refinement phase begins (lines 25–28). The exact interest of each segment in  $\mathcal{L}_{seen}$  is computed by invoking `UpdateInterest` as necessary. The extraction of the streets with the highest interest, i.e., the  $k$ -SOI, is then straightforward.

A final note concerns the case of multiple keywords  $\Psi$  in the query. The SOI algorithm changes in only two places. The first is when source list  $SL_1$  is built. It is necessary to account for POIs that have any keyword among those in  $\Psi$ . SOI looks within the global inverted index, for each entry  $I[\psi][c]$  corresponding to the entry for cell  $c$  in the list for keyword  $\psi$ . This entry contains the count of POIs in cell  $c$  that have keyword  $\psi$ . Adding these counts for all keywords provides an upper bound to the number of POIs within  $c$  that have any keyword among  $\Psi$ . The minimum of this number and the total number  $|\mathcal{P}_c|$  of POIs in the cell (line 2) is then inserted into  $SL_1$ . The second change is in the `UpdateInterest` procedure. To compute for segment  $\ell$  the exact number of POIs within cell  $c$  that satisfy  $\Psi$ , lists  $c.I[\psi]$  for each  $\psi \in \Psi$  are tra-

versed in parallel; recall that the lists are sorted by POI id. For each encountered POI, the mass of  $\ell$  is incremented.

## 4. DESCRIBING STREETS OF INTEREST

Having identified the  $k$ -SOIs in the road network, the next step is to provide summarized information to describe them. Section 4.1 formalizes the problem, while Section 4.2 describes our solution.

### 4.1 Problem Definition

We begin by formalizing the problem, and then present details about the measures considered.

#### 4.1.1 Problem Statement

In the following, we assume an additional data source  $\mathcal{R}$ , being a set of geo-tagged photos. Each photo  $r \in \mathcal{R}$  is defined by a tuple  $r = \langle (x_r, y_r), \Psi_r \rangle$ , specifying its location and a set of keywords (its tags); the distance of a photo to a segment or a street is defined as in the case of a POI.

To describe a SOI, we exploit its related photos. For each street  $s$ , these are the photos that are located within distance  $\epsilon$ , i.e.  $\mathcal{R}_s = \{r \in \mathcal{R} : \text{dist}(r, s) \leq \epsilon\}$ . However, the size of  $\mathcal{R}_s$  can typically be quite large. Thus, the problem is to select a relatively small subset of  $k$  photos ( $k \ll |\mathcal{R}_s|$ ) to present as an overview for the street  $s$ . To avoid redundancy and repetition, we formulate the problem as a MaxSum diversification problem, where a subset of items is selected from a set in such a way as to maximize both their relevance to a given need and their pairwise dissimilarity. More formally, the problem can be defined as a bi-criteria optimization problem, aiming at optimizing an objective function  $\mathcal{F}$  that comprises a *relevance* component and a *diversity* component [16, 26].

Let  $R^k$  be a subset of  $\mathcal{R}_s$  of size  $k$ , and let  $\text{rel}(R^k)$  and  $\text{div}(R^k)$  be two functions that measure, respectively, the relevance and the diversity of the contents of  $R^k$ . Then, the problem is to select among all possible subsets  $R^k$  the one that maximizes the function  $\mathcal{F}$  defined as follows:

$$\mathcal{F}(R^k) = (1 - \lambda) \cdot \text{rel}(R^k) + \lambda \cdot \text{div}(R^k) \quad (2)$$

where  $\lambda \in [0, 1]$  is a parameter determining the tradeoff between relevance ( $\lambda = 0$ ) and diversity ( $\lambda = 1$ ).

**Problem 2. [SOI Diversification]** Given a street  $s$  with an associated set of photos  $\mathcal{R}_s$ , where each photo has a geolocation and a set of keywords, select a subset  $R^k$  of  $\mathcal{R}_s$  containing  $k$  photos such that the objective function  $\mathcal{F}$  is maximized, i.e.:

$$R^k = \arg \max_{R \subseteq \mathcal{R}_s, |R|=k} \mathcal{F}(R) \quad (3)$$

We proceed to define the functions  $\text{rel}(R^k)$  and  $\text{div}(R^k)$  for our problem. Note that the value of  $k$  throughout this section refers to the number of photos describing a street, and is thus unrelated to the value of  $k$  in Section 3 which refers to the number of SOIs.

#### 4.1.2 Spatio-Textual Relevance and Diversity

The function  $\mathcal{F}$  described above provides a generic criterion for selecting a subset of items that are both relevant and diverse w.r.t. a given query. In our context, this needs to take into account both the spatial and the textual description of a street. In particular, the *spatial aspect* of a street  $s$  is determined by the locations of its associated photos  $\mathcal{R}_s$ . The *textual aspect* of  $s$  is captured by a keyword frequency vector  $\Phi_s$ , which describes the strength of each keyword associated with  $s$ ; we denote as  $\Psi_s$  the set of keywords with non-zero frequency in  $\Phi_s$ . Note that there are many ways

to derive the keyword frequency vector of a street; for example, we can extract it directly from a textual description, or from the keywords of its neighboring POIs and/or photos.

The relevance and diversity functions of a set  $R^k$  of photos should thus capture both aspects. Assuming a weight parameter  $0 \leq w \leq 1$  between the two aspects, we define:

$$\text{rel}(R^k) = \frac{w}{k} \sum_{r \in R^k} \text{spatial\_rel}(r) + \frac{1-w}{k} \sum_{r \in R^k} \text{textual\_rel}(r) \quad (4)$$

and

$$\begin{aligned} \text{div}(R^k) = & \frac{2w}{k(k-1)} \sum_{r, r' \in R^k} \text{spatial\_div}(r, r') \\ & + \frac{2(1-w)}{k(k-1)} \sum_{r, r' \in R^k} \text{textual\_div}(r, r'). \end{aligned} \quad (5)$$

Notice that the relevance of set  $R^k$  is defined as the sum of the spatial and textual relevance of each photo  $r \in R^k$ , whereas the diversity of  $R^k$  is the sum of the pairwise spatial and textual diversity over all photo pairs  $r, r' \in R^k$ . To balance the different number of summands in  $\text{rel}(R^k)$  and  $\text{div}(R^k)$ , we normalize them using the fractions  $\frac{1}{k}$  and  $\frac{2}{k(k-1)}$ , respectively. Next, we define the four functions that account for spatio-textual relevance and diversity.

**Spatial relevance and diversity.** For a point query, the spatial distance of an item to the query point would typically constitute a natural way to measure relevance. However, in our case, ranking the photos of a street according to their distance from it does not generally provide an indicative criterion for judging relevance. Thus, we select instead a different criterion, based on spatial coverage. The intuition is that high density of photos in an area can be considered as an indication of ‘‘importance’’; thus, the selection of photos should be biased towards those areas. Accordingly, we define the spatial relevance of a photo based on the number of other photos contained in its neighborhood. Furthermore, we divide this number to the total number of photos associated with the street, in order to obtain a normalized value in the range  $[0, 1]$ .

**DEFINITION 4 (SPATIAL RELEVANCE).** Assuming a radius  $\rho$  for the neighborhood of a photo  $r$ , we define the spatial relevance of  $r$  w.r.t. the street  $s$  as:

$$\text{spatial\_rel}(r) = \frac{|\{r' \in \mathcal{R}_s : \text{dist}(r, r') \leq \rho\}|}{|\mathcal{R}_s|}. \quad (6)$$

The spatial diversity of a pair of photos  $r, r'$  can be defined by means of their spatial distance. For normalization, we divide the distance of the pair with  $\text{maxD}(s)$ , which is the largest possible distance between any two photos associated with  $s$ . Value  $\text{maxD}(s)$  is computed as the length of the diagonal of the minimum bounding rectangle, extended with a buffer of size  $\epsilon$ , that encloses  $s$ . Thus:

**DEFINITION 5 (SPATIAL DIVERSITY).** We define the spatial diversity of two photos  $r, r'$  associated with street  $s$  as:

$$\text{spatial\_div}(r, r') = \frac{\text{dist}(r, r')}{\text{maxD}(s)}. \quad (7)$$

**Textual relevance and diversity.** The textual relevance of a photo  $r$  measures the similarity of its textual description  $\Psi_r$  to the textual aspect of street  $s$  as captured by its keyword frequency vector  $\Phi_s$ .

DEFINITION 6 (TEXTUAL RELEVANCE). *The textual relevance of a photo  $r$  to street  $s$  is defined as:*

$$\text{textual\_rel}(r) = \frac{\sum_{\psi \in \Psi_r} \Phi_s(\psi)}{\|\Phi_s\|_1}, \quad (8)$$

where  $\|\Phi_s\|_1 = \sum_{\psi \in \Psi_s} \Phi_s(\psi)$  is a normalization term.

Finally, we define textual diversity by means of the Jaccard distance.

DEFINITION 7 (TEXTUAL DIVERSITY). *We define the textual diversity of two photos  $r, r'$  as the Jaccard distance of their sets of keywords, i.e.:*

$$\text{textual\_div}(r, r') = 1 - \frac{|\Psi_r \cap \Psi_{r'}|}{|\Psi_r \cup \Psi_{r'}|}. \quad (9)$$

## 4.2 The ST\_Rel+Div Algorithm

Next, we present the ST\_Rel+Div (Spatio-Textual Relevance and Diversity) algorithm. We first describe the methodology and index used, and then we present how the algorithm selects the diversified subset of photos more efficiently.

### 4.2.1 Methodology and Indices

The ST\_Rel+Div algorithm follows the standard greedy methodology for solving MaxSum diversification problems. It builds the diversified set  $R^k$  iteratively, selecting at each step the photo that maximizes the maximum marginal relevance function  $mmr$  (see Section 2). Suppose that set  $R$  has been constructed, where  $|R| < k$ . Then, the photo from  $\mathcal{R}_s \setminus R$  to be inserted next is the one that maximizes the function:

$$\text{mmr}(r) = (1 - \lambda) \cdot \text{rel}(r) + \frac{\lambda}{k - 1} \cdot \sum_{r' \in R} \text{div}(r, r'). \quad (10)$$

Functions  $\text{rel}(r)$  and  $\text{div}(r)$  are as defined in Section 4.1.2 taking into account the spatial and textual aspects.

The main issue with applying this heuristic methodology in our context is the computational complexity of  $mmr$ . A naïve implementation would compute a large number of spatial and textual photo-to-street relevances and photo-to-photo diversities. In particular, computing  $mmr$  at each iteration, requires  $O(|\mathcal{R}_s|)$  computations for its first component, and  $O(|R||\mathcal{R}_s|)$  for the second. Even though some of these computations need not be repeated across iterations, the total computational cost can be prohibitive, especially when the set  $\mathcal{R}_s$  is large.

Consequently, the goal of ST\_Rel+Div is to efficiently evaluate each component of the objective function  $mmr$  towards retrieving the best candidate at each iteration. For this purpose, we construct an index as described in the following. We use an index structure that combines a spatial grid with inverted indices in each cell. Each cell  $c_{i,j}$  in the grid has side length  $\frac{\ell}{2}$ , and contains the following information:

- a list of the photos in the cell, denoted as  $c_{i,j}.\mathcal{R}$
- an inverted index  $c_{i,j}.I$ , where the terms are the keywords appearing in the photos in this cell, and each postings list  $c_{i,j}.I[\psi]$  contains those photos that have the keyword  $\psi$  (we denote as  $c_{i,j}.\Psi$  the set of keywords present in  $c_{i,j}.I$ )
- the maximum ( $c_{i,j}.\psi_{max}$ ) and minimum ( $c_{i,j}.\psi_{min}$ ) number of keywords for the photos in this cell.

Next, we show how this index is used to derive lower and upper bounds for each of the components of the objective function  $mmr$ . Note that the described index, although similar to the grid index used in Section 3.2, is distinct, indexing a different dataset, i.e. the set of photos instead of POIs.

### 4.2.2 Computing Bounds

Using the index, we can derive, for any of the photos within a cell, upper and lower bounds for each of the components of the  $mmr$  function. The ST\_Rel+Div algorithm exploits these bounds while computing the  $mmr$  function in order to iterate over the cells instead of individual photos and to prune the search space more quickly, identifying the next candidate that optimizes the  $mmr$  criterion. In particular, we need to derive lower and upper bounds for the following: (a) spatial and textual relevance of a cell to a street and (b) spatial and textual diversity of a cell to a photo. We elaborate on each below.

**Cell spatial relevance.** Consider a cell  $c_{i,j}$ . The spatial relevance of a photo  $r$  w.r.t. street  $s$  is defined according to Equation 6. Moreover, recall that the length of each side of a cell in the spatial grid is  $\frac{\ell}{2}$ . Hence, each photo  $r \in c_{i,j}.\mathcal{R}$  covers at least all other photos in the same cell and at most all photos that are no more than two cells away. Accordingly, we derive the following lower and upper bounds for the cell-to-street spatial relevance:

$$\text{spatial\_rel}^-(c_{i,j}) = \frac{|c_{i,j}.\mathcal{R}|}{|\mathcal{R}_s|}, \quad (11)$$

$$\text{spatial\_rel}^+(c_{i,j}) = \frac{\sum_{\Delta i, \Delta j \in [-2, 2]} |c_{i+\Delta i, j+\Delta j}.\mathcal{R}|}{|\mathcal{R}_s|}. \quad (12)$$

**Cell textual relevance.** Consider a cell  $c$ . The textual relevance of a photo  $r \in c.\mathcal{R}$  w.r.t. street  $s$  is defined according to Equation 8. We seek to construct keyword sets  $\Psi^-(c|s), \Psi^+(c|s)$ , which are subsets of  $c.\Psi$ , such that when they substitute set  $\Psi_r$  in Equation 8, the obtained values bound the textual relevance of any photo  $r \in c.\mathcal{R}$ . In other words, we obtain the following bounds of  $\text{textual\_rel}(r)$ :

$$\text{textual\_rel}^-(c) = \frac{\sum_{\psi \in \Psi^-(c|s)} \Phi_s(\psi)}{\|\Phi_s\|_1}, \quad (13)$$

$$\text{textual\_rel}^+(c) = \frac{\sum_{\psi \in \Psi^+(c|s)} \Phi_s(\psi)}{\|\Phi_s\|_1}. \quad (14)$$

We next describe how to build the keyword sets  $\Psi^-(c|s), \Psi^+(c|s)$ . Based on the information stored in the index, each photo  $r \in P(c)$  may contain at least  $c.\psi_{min}$  and at most  $c.\psi_{max}$  keywords. Hence, sets  $\Psi^-(c|s), \Psi^+(c|s)$  should obey these cardinality constraints.

For set  $\Psi^-(c|s)$ , we should choose the fewest possible keywords from  $c.\Psi$ , i.e.,  $c.\psi_{min}$ , and make sure they have as low frequencies in  $\Phi_s$  as possible. Therefore, we select up to  $c.\psi_{min}$  keywords from  $c.\Psi$  that do not appear in  $\Psi_s$ , and, if necessary (so as to satisfy the minimum cardinality constraint), we additionally select keywords from  $c.\Psi$  with the lowest frequencies in  $\Phi_s$ .

On the other hand, for set  $\Psi^+(c|s)$ , we select up to  $c.\psi_{max}$  keywords from  $c.\Psi$  that also appear in  $\Psi_s$ , and if necessary (so as to satisfy the minimum cardinality constraint), we arbitrarily choose additional keywords from  $c.\Psi$ .

**Cell-to-photo spatial diversity.** Assume a grid cell  $c$  and a photo  $r$ . The lower and upper bounds of the spatial diversity between  $r$  and any photo  $r' \in c.\mathcal{R}$  are determined by respective bounds on the distance of  $r$  and cell  $c$ . Therefore, we have:

$$spatial\_div^-(c, r) = \frac{mindist(r, c)}{maxD(s)}, \quad (15)$$

$$spatial\_div^+(c, r) = \frac{maxdist(r, c)}{maxD(s)}, \quad (16)$$

where functions  $mindist(r, c)$  and  $maxdist(r, c)$  return the minimum and maximum distance, respectively, between  $r$  and any point within cell  $c$ .

**Cell-to-photo textual diversity.** Assume a grid cell  $c$  and a photo  $r$ . We determine the lower and upper bounds of the textual diversity between  $r$  and any other photo  $r' \in c.\mathcal{R}$ . We follow a similar rationale as for the cell textual relevance, and construct keyword sets  $\Psi^+(c|r)$ ,  $\Psi^-(c|r)$  so that when they substitute  $\Psi_{r'}$  in Equation 9 they provide a lower and an upper bound, respectively, on the textual diversity of any photo  $r' \in c.\mathcal{R}$ .

For the lower bound for the textual diversity, we construct set  $\Psi^+(c|r)$  maximizing the common keywords with  $\Psi_r$ . Specifically, we insert in  $\Psi^+(c|r)$  up to  $c.\psi_{max}$  common keywords, and if necessary, we additionally insert keywords from  $c.\Psi$  until we obtain at least  $c.\psi_{max}$  keywords in  $\Psi^+(c|r)$ . Therefore, the textual diversity of any  $r' \in c.\mathcal{R}$  is lower bounded by:

$$textual\_div^-(c, r) = 1 - \frac{|\Psi^+(c|r) \cap \Psi_r|}{|\Psi^+(c|r) \cup \Psi_r|} = \begin{cases} 1 - \frac{|c.\Psi \cap \Psi_r|}{|\Psi_r| + c.\psi_{min} - |c.\Psi \cap \Psi_r|}, & \text{if } |c.\Psi \cap \Psi_r| < c.\psi_{min} \\ 1 - \frac{\min(|c.\Psi \cap \Psi_r|, c.\psi_{max})}{|\Psi_r|}, & \text{if } |c.\Psi \cap \Psi_r| \geq c.\psi_{min} \end{cases} \quad (17)$$

For the upper bound for the textual diversity, we construct set  $\Psi^-(c|r)$  minimizing the common keywords with  $\Psi_r$ . Specifically, we insert in  $\Psi^-(c|r)$  up to  $c.\psi_{min}$  keywords from  $c.\Psi$  that are not in  $\Psi_r$ , and if necessary, we insert additional keywords from  $c.\Psi$  until we obtain at least  $c.\psi_{min}$  keywords. Therefore, the textual diversity of any  $r' \in c.\mathcal{R}$  is upper bounded by:

$$textual\_div^+(c, r) = 1 - \frac{|\Psi^-(c|r) \cap \Psi_r|}{|\Psi^-(c|r) \cup \Psi_r|} = \begin{cases} 1 - \frac{c.\psi_{min} - |c.\Psi \setminus \Psi_r|}{|r.\Psi| + |c.\Psi \setminus \Psi_r|}, & \text{if } |c.\Psi \setminus \Psi_r| < c.\psi_{min} \\ 1, & \text{if } |c.\Psi \setminus \Psi_r| \geq c.\psi_{min} \end{cases} \quad (18)$$

### 4.2.3 Algorithm Description

Algorithm 2 shows the pseudocode for the `ST_Rel+Div` algorithm, which incrementally builds the result set  $R^k$  by adding, at each step, the next best photo, denoted as  $next\_r$ , that maximizes the objective function  $mmr$  defined in Equation 10. However, the distinguishing factor of `ST_Rel+Div`, compared to a naïve algorithm that directly computes the  $mmr$  function for each photo, is that instead of evaluating individual photos, it first considers entire grid cells. Specifically, at each step, in the filtering phase (lines 4–9), it iterates over the cells and computes for each cell the lower and upper bound of the objective function  $mmr$  (lines 5–7), by applying the corresponding bounds presented in 4.2.2.

Then, any cell having an upper bound that is lower than the lower bound of another cell is discarded (line 9). The remaining cells are organized in a priority queue ordered descending on their  $mmr$  upper bound. Only the photos belonging to the remaining cells are

### Algorithm 2: Algorithm `ST_Rel+Div`

---

**Input:** Set of all relevant photos  $\mathcal{R}_\ell$ , integer  $k$ , the `ST_Rel+Div` index  $\mathcal{I}$   
**Output:** Diversified subset  $R^k$

```

1  $R^k \leftarrow \emptyset$ 
2  $C \leftarrow$  the grid cells in  $\mathcal{I}$ 
   $\triangleright$  select next candidate
3 while  $|R^k| < k$  do
   $\triangleright$  filtering phase
  4  $B_{min}, B_{max} \leftarrow \emptyset$   $\triangleright$  maps to store cell bounds
  5 foreach cell  $c \in C$  do
  6    $B_{min}(c) \leftarrow mmr(c)^-$   $\triangleright$  using bounds in
  7    $B_{max}(c) \leftarrow mmr(c)^+$   $\triangleright$  Section 4.2.2
  8  $mmr\_min \leftarrow \max_{c \in C} B_{min}(c)$ 
  9  $C \leftarrow \{c : B_{max}(c) \geq mmr\_min\}$   $\triangleright$  list of candidate cells
   $\triangleright$  refinement phase
  10 while  $C \neq \emptyset$  do
  11    $c \leftarrow top(C)$   $\triangleright$  visit next cell with largest  $B_{max}(c)$ 
  12   foreach  $r \in c.\mathcal{R}$  do
  13      $v \leftarrow mmr(r)$   $\triangleright$  compute exact value
  14     if  $v > mmr\_min$  then
  15        $mmr\_min \leftarrow v$   $\triangleright$  refine bound
  16        $C \leftarrow C \setminus \{c : B_{max}(c) < mmr\_min\}$ 
  17        $next\_r \leftarrow r$ 
  18  $R^k \leftarrow R^k \cup next\_r$   $\triangleright$  add next best photo
  19 return  $R^k$ 

```

---

**Table 1: Datasets used in the evaluation.**

Dataset	Num of segm.	Min segm. length (m)	Max segm. length (m)	Num of POIs
London	113,885	0.93	5,834.71	2,114,264
Berlin	47,755	0.06	6,312.96	797,244
Vienna	22,211	1.35	9,913.42	408,712

processed in the refinement phase (lines 10–17). For each examined photo, its exact value for the objective function is calculated (line 13). During this process, if the upper bound of a cell is lower than the value computed for an examined photo, this cell is also discarded (lines 14–17). The process continues until the priority queue contains no cells.

## 5. EXPERIMENTAL EVALUATION

We have conducted an experimental evaluation using real-world data comprising road networks, POIs and photos. The datasets cover the areas of three European capital cities, London, Berlin and Vienna, and were collected from various Web sources, in particular: (a) road networks from OpenStreetMap, (b) POIs from DBpedia, OpenStreetMap, Wikimapia and Foursquare, and (c) photos from Flickr and Panoramio. Table 1 presents statistics about the datasets. All algorithms were implemented in Java and experiments were run on a machine with an Intel Core i7 2400 CPU and 8GB RAM.

The primary focus of this paper is to propose efficient algorithms for the tasks of identifying and describing SOIs, as defined in Sections 3.1 and 4.1. A detailed *performance* study is presented in Section 5.2. Nonetheless, it is also very important to gauge the *effectiveness* of our methods in achieving their goals. Therefore, in Section 5.1, we present the results of an empirical study of both tasks, identification and description.

### 5.1 Effectiveness Study

The goal of this section is to highlight the effectiveness of our methods in identifying and describing streets of interest.

#### 5.1.1 Identifying Streets of Interest

We focus on a particular SOI retrieval scenario: determine streets in Berlin that are interesting for “shopping”. As ground truth, we





Figure 2: Main shopping sites in Berlin and their most important streets.

Table 2: Comparison of identified top SOIs for “shops” in Berlin.

Top-10 SOIs	Source #1	Source #2
1. <b>Neue Schönhauser Straße</b>	<b>Tauentzienstraße</b>	Kurfürstendamm
2. Rosenthaler Straße	Fasanenstraße	<b>Tauentzienstraße</b>
3. Mäusetunnel	<b>Friedrichstraße</b>	<b>Potsdamer Platz</b>
4. <b>Münzstraße</b>	<b>Alte/Neue Schönhauser Straße</b>	<b>Friedrichstraße</b>
5. <b>Potsdamer Platz Arkaden</b>	<b>Münzstraße</b>	<b>Alte/Neue Schönhauser Straße</b>
6. <b>Friedrichstraße</b>		
7. Mulackstraße		
8. <b>Alte Schönhauser Straße</b>		
9. Weinmeisterstraße		
10. <b>Tauentzienstraße</b>		

assume two authoritative Web sources about top shopping destinations in Berlin<sup>1 2</sup>. Each source provided a (non-ranked) list of 5 streets, displayed in the two last columns in Table 2.

In our SOI algorithm, we set the query parameters to  $\Psi=\{\text{“shop”}\}$ ,  $k = 10$ ,  $\epsilon = 0.0005^\circ \approx 55m$ , i.e., looking for the top 10 streets that have a large concentration of “shop”-related POIs within 55 meters from them. The ranked list is shown in the first column of Table 2. Streets that are common among the 10-SOIs and the two sources are shown in bold. For both sources, we retrieve 4 out of 5 streets; therefore, our method has a recall (at rank 10) of 0.8.

A closer inspection of the results though, suggests that our method has actually better recall (and precision). All streets included in Table 2 belong to one of four main shopping sites in Berlin, near Alte/Neue Schönhauser Straße, near Kurfürstendamm, near Friedrichstraße, and near Potsdamer Platz. Figure 2 illustrates the first three areas on the map (the last one is a public square that has no important adjacent shopping streets) and highlights the streets included in Table 2. Green color indicates that the street is in the 10-SOIs and in at least one of the source lists (true positives); orange means it is in the 10-SOIs but not in any source (false positives); blue indicates it is in a source but not in the 10-SOIs (false negatives).

It should be apparent that all orange streets near Alte/Neue Schönhauser Straße are actually valid results as they are adjacent to the main streets in the respective area, and further have lots of little shops in their vicinity. Similarly, the orange street near Friedrichstraße is also valid as it is a pedestrian underground tunnel with shops. Regarding the blue streets in the Kurfürstendamm site (which is analogous to Champs-Élysées in Paris), we should note that Kurfürstendamm appears in the 20-SOIs. The reason they are ranked lower is that in their vicinity the density of shops is lower, as they essentially house big luxury brands. This could be addressed by exploiting additional metadata to assign different weights to the POIs.

### 5.1.2 Describing Streets of Interest

To study the effectiveness of our method in selecting appropriate photos for describing a particular SOI, we focus on the popular Oxford Street in London and seek for 3 photos. Since deriving the ground truth from the collected crowd-sourced data is not possible, we choose to empirically test the result of our method against simpler techniques that select photos based on spatial, textual information or their combination, and consider relevance, diversity, or their combination. More precisely, we compare our method  $ST\_Rel+Div$  that combines spatio-textual relevance and diversity, against 8 other techniques depicted in Table 4. Symbols S and T denote, respectively, that spatial and textual information is considered, while Rel and Div indicate, respectively, that relevance and diversity are taken into account; our method considers all factors and information, hence its name  $ST\_Rel+Div$ .

For a visual inspection of the results, Figure 3 shows the 3-photo summary of Oxford Street, according to methods S\_Rel, T\_Rel and  $ST\_Rel+Div$ , respectively. In the first method, all selected photos are located outside HMV, an entertainment retailing company, and are in fact near-duplicates. The reason for this seems to be that the particular location attracts a large number of photos, due to the release of popular movies, music albums and other similar events, thus creating a high density spot. For the second method, all results are photos from a particular demonstration that took place along Oxford Street. This bias was introduced due to the high frequency of the corresponding tags, thus resulting in a higher rank for photos having those tags. On the other hand, observe that in our method (Figure 3(c)) the result comprises different kinds of photos, achieving both high relevance and diversity: one photo outside HMV, another from the aforementioned demonstration, and a third photo showing a view of the street undergoing construction work.

For a quantifiable measure of effectiveness, we use the objective function of Equation 2 ( $\lambda = 0.5$ ,  $w = 0.5$ ), that provides a balanced score reflecting the relevance and diversity of both spatial and textual information included in the photo summary. For the top SOI in the considered cities, Table 3 presents the scores achieved by each method; the value is normalized with respect to that of the

<sup>1</sup><http://www.tripadvisor.com/Travel-g187323-s405/Berlin:Germany:Shopping.html>

<sup>2</sup><http://www.globalblue.com/destinations/germany/berlin/top-five-shopping-streets-in-berlin>

**Table 3: Objective scores (Equation 2 after normalization).**

Method	London	Berlin	Vienna
S_Rel	0.831	0.726	0.508
S_Div	0.923	0.982	0.961
S_Rel+Div	<i>0.982</i>	<i>0.953</i>	<i>0.911</i>
T_Rel	0.708	0.367	0.219
T_Div	0.831	0.811	0.895
T_Rel+Div	0.949	0.848	0.919
ST_Rel	0.776	0.367	0.279
ST_Div	0.913	<i>0.986</i>	<i>0.961</i>
ST_Rel+Div	<b>1.000</b>	<b>1.000</b>	<b>1.000</b>

ST\_Rel+Div method. In all cities, our method achieves the highest normalized score (shown with bold), often by a large margin (up to 4.5x). It is worth mentioning that there is no clear runner-up, as S\_Rel+Div is second best for London, while ST\_Div is for Berlin and Vienna (second highest values shown with italics).

## 5.2 Performance Study

We next evaluate the efficiency of the proposed methods.

### 5.2.1 Identifying Streets of Interest

**Methods.** First, we evaluate the performance of our proposed SOI algorithm for solving  $k$ -SOI. To the best of our knowledge, no approaches have been proposed in previous works for the specific problem addressed (see Problem 1). Hence, we compare the performance of SOI to a baseline implementation, denoted as BL. Specifically, BL uses only the spatial grid index to efficiently compute the interest of every segment, and then determines the  $k$ -SOIs.

**Parameters.** Throughout our experiments, we set the distance threshold  $\epsilon$  to a fixed value ( $\epsilon = 0.0005^\circ \approx 55m$ ). We study the effect of the number  $k$  of SOIs requested, and the number  $|\Psi|$  of keywords in the query. To construct the keyword set, we select the first  $|\Psi|$  keywords among  $\{\textit{religion, education, food, services}\}$ . The resulting accumulated number of relevant POIs is shown in Table 4. In each experiment, we vary one parameter while setting the other to its default value ( $k = 50, |\Psi| = 3$ ).

**Table 4: Relevant POIs according to  $|\Psi|$ .**

Dataset	$ \Psi  = 1$	$ \Psi  = 2$	$ \Psi  = 3$	$ \Psi  = 4$
London	10,445	32,682	113,211	202,127
Berlin	1,969	10,506	47,950	78,310
Vienna	1,678	7,660	25,695	41,484

**Metrics.** The objective of our evaluation is to analyze the performance of SOI compared to BL. Therefore, we measure the total execution time of both methods. For SOI we further break it down into time spent during list construction, filtering, and refinement.

**Results.** Figure 4 presents the evaluation results on the three datasets for all settings considered. Note that the bars for SOI are divided into the time spent in each of the three phases discussed.

The value of  $k$  has a small effect on all algorithms. Particularly for SOI the execution time slightly increases with  $k$ . SOI outperforms BL by a factor around 2.1–3.2 for London, 1.6–2.1 for Berlin, and 1.1–2.5 for Vienna. An important observation is that our method is more efficient for larger datasets, such as London (see Table 1).

The value of  $|\Psi|$  has no effect in BL. On the other hand, the execution time of SOI increases with  $|\Psi|$  as more POIs become relevant (see Table 4) and thus more cells and segments have to be visited. For example in London, the number of cells SOI visits increases from 5% to 13% of the total cells. As a result, SOI outperforms BL by a factor that varies from 1.1 up to 18.



(a) Spatial relevance (S\_Rel).



(b) Textual relevance (T\_Rel).



(c) Spatio-textual relevance and diversity (ST\_Rel+Div).

**Figure 3: Selected photos under different criteria.**

Note that the selected keywords are quite general; they only serve for benchmark purposes in extreme settings. For example, when  $|\Psi|=4$ , we are essentially trying to rank streets that have churches, schools, restaurants or various services within their neighborhood. As a result, around 60% of all street segments are relevant (SOI manages to prune half of them). In practice, the user would pose more selective keywords.

### 5.2.2 Describing Streets of Interest

**Methods.** Next, we evaluate the performance of the ST\_Rel+Div algorithm compared to a baseline method (BL) which, similarly to ST\_Rel+Div, constructs the diversified result set iteratively, but examining all photos in each iteration instead of operating on the grid cells and using the bounds presented in Section 4.2.

**Parameters.** We fix the values of the distance parameters to  $\epsilon = 0.0005^\circ$  and  $\rho = 0.0001^\circ$ , and we vary: (a) the number  $k$  of photos requested (default  $k=20$ ), (b) the weight  $\lambda$  between relevance and diversity (default  $\lambda=0.5$ ), and (c) the weight  $w$  between the spatial and textual components (default  $w=0.5$ ). For each dataset, we randomly selected one of the returned  $k$ -SOIs in the previous experiments. The number of nearby photos for the cases of London, Berlin and Vienna was, respectively, 6572, 788, and 1584.

An interesting discussion concerns the selection of an appropriate value for parameter  $\lambda$ .<sup>3</sup> We can think of the relevance-diversity trade-off as follows. In order to increase the diversity of the result set (the *return*), we have to sacrifice its relevance (the *investment*). Typically, diversity starts to increase quickly in the beginning (when relevance is still high), but its rate slowly decreases, meaning that a greater reduction in relevance is required to achieve

<sup>3</sup>The other parameter in Equation 2,  $w$ , is application dependent.

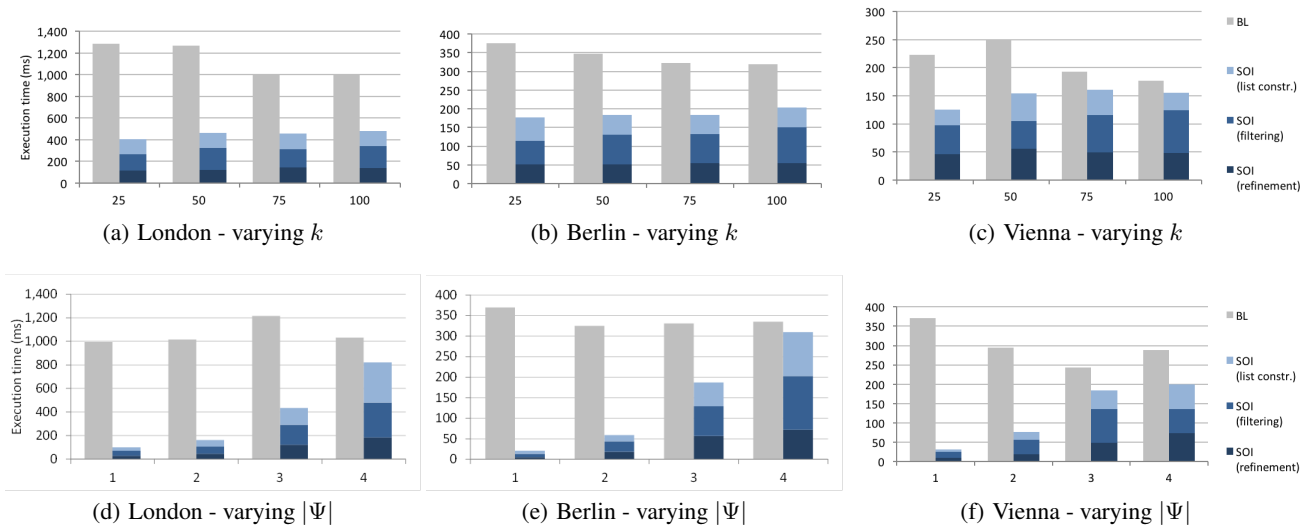


Figure 4: Experimental results for the SOI algorithm.

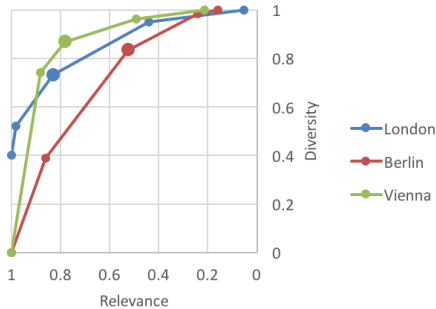


Figure 5: Trade-off between relevance and diversity ( $w = 0.5$ ).

the same increase in diversity. So the goal is to figure out an acceptable investment that is “value for money”.

Figure 5 depicts the normalized relevance (Equation 4) and diversity (Equation 5) scores of the constructed photo summary for the top SOI in the three cities for various values of  $\lambda$ ; note that the relevance axis is reversed. As we go from bottom left to top right, the value  $\lambda$  increases from 0 to 1 in increments of 0.25, and thus relevance decreases while diversity increases. The larger marker indicates the value  $\lambda = 0.5$ . In all cities,  $\lambda$  values around 0.5 achieve the best relevance-diversity trade-off. For example, in Vienna  $\lambda = 0.5$  suggests that by sacrificing 0.22 units of normalized relevance we achieve a diversity of 0.87 normalized units. These findings justify our selection of 0.5 as the default value for  $\lambda$ .

**Metrics.** As previously, we compare the total execution time of the ST\_Rel+Div algorithm and the BL method.

**Results.** The results are shown in Figure 6. The pruning achieved by ST\_Rel+Div via bounds computed for the grid cells drastically reduces the execution time in all experiments. ST\_Rel+Div outperforms BL by a factor that varies from 2 up to 64.

Moreover, it is worth noticing that ST\_Rel+Div has response times of less than a second, in contrast to the BL method that typically requires several seconds to compute the results, thus being unsuitable for online exploration. In fact, the execution time is much higher for London, due to the fact that the selected segment in that case has a much higher number of associated photos, while the inverse holds for Berlin.

For both algorithms, execution time increases with  $k$ , as more iterations are performed; however, ST\_Rel+Div shows much better scalability due to the pruning. These differences in performance also remain consistent when varying the parameters  $\lambda$  and  $w$ .

## 6. CONCLUSIONS

In this paper, we have addressed the problem of finding and exploring *Streets of Interest* based on Points of Interest and photos characterized by geolocation and keywords. The problem addressed is twofold. Given a set of keywords, we first rank streets according to relevant nearby POIs. To that end, we define an interest score for a street, and we present an efficient algorithm that returns the top- $k$  interesting streets. Then, we select for each discovered street a small, diversified set of photos. We formulate this as a diversification problem for spatio-textual objects, and we present an efficient algorithm that performs a greedy search using a spatio-textual grid to speed up the selection of candidates. Our experimental results on real-world data from several Web sources show that the proposed algorithms drastically reduce the computation time, allowing for online discovery and exploration of interesting parts of the road network. In the future, we plan to enhance the diversification criteria with visual features extracted from the photos, as well as to provide route recommendations based on the discovered streets of interest.

## Acknowledgements

This work was partially supported by the EU Projects GEOSTREAM (FP7-SME-2012-315631) and City.Risks (H2020-FCT-2014-653747).

## 7. REFERENCES

- [1] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. In *WSDM*, pages 5–14, 2009.
- [2] R. Ahuja, N. Armenatzoglou, D. Papadias, and G. J. Fakas. Geo-social keyword search. In *SSTD*, pages 431–450, 2015.
- [3] M. Alivand and H. H. Hochmair. Extracting scenic routes from VGI data sources. In *GEOCROWD*, pages 23–30, 2013.
- [4] N. Armenatzoglou, S. Papadopoulos, and D. Papadias. A general framework for geo-social query processing. *PVLDB*, 6(10):913–924, 2013.
- [5] X. Cao, G. Cong, and C. S. Jensen. Retrieving top- $k$  prestige-based relevant spatial web objects. *PVLDB*, 3(1):373–384, 2010.

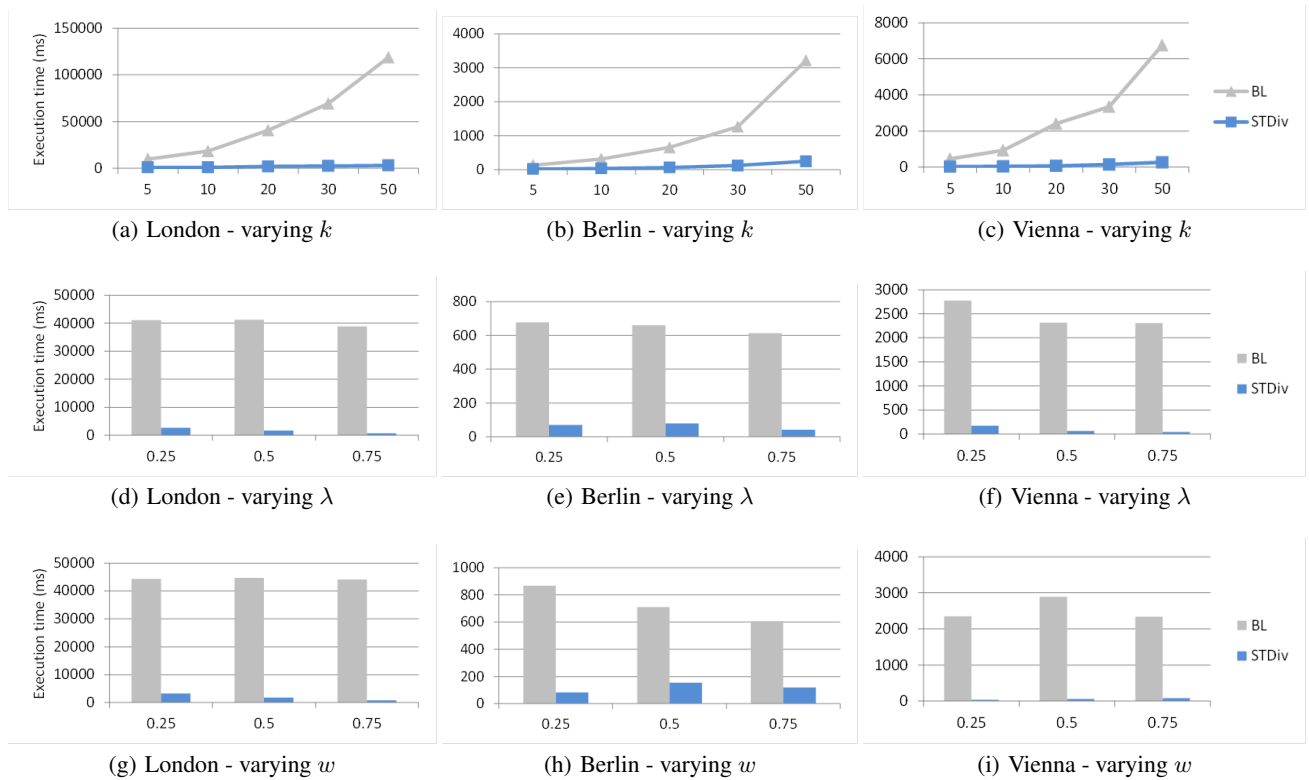


Figure 6: Experimental results for the ST\_Rel+Div algorithm.

- [6] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD*, pages 373–384, 2011.
- [7] X. Cao, G. Cong, C. S. Jensen, and M. L. Yiu. Retrieving regions of interest for user exploration. *PVLDB*, 7(9):733–744, 2014.
- [8] J. G. Carbonell and J. Goldstein. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *SIGIR*, pages 335–336, 1998.
- [9] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.
- [10] D. Choi, C. Chung, and Y. Tao. Maximizing range sum in external memory. *TODS*, 39(3):21:1–21:44, 2014.
- [11] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.
- [12] Z. Dou, S. Hu, K. Chen, R. Song, and J.-R. Wen. Multi-dimensional search result diversification. In *WSDM*, pages 475–484, 2011.
- [13] M. Drosou and E. Pitoura. Search result diversification. *SIGMOD Record*, 39(1):41–47, 2010.
- [14] M. Drosou and E. Pitoura. Disc diversity: result diversification based on dissimilarity and coverage. *PVLDB*, 6(1):13–24, 2012.
- [15] L. Ferrari, A. Rosi, M. Mamei, and F. Zambonelli. Extracting urban patterns from location-based social networks. In *LBSN*, pages 9–16, 2011.
- [16] S. Gollapudi and A. Sharma. An axiomatic approach for result diversification. In *WWW*, pages 381–390, 2009.
- [17] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top- $k$  query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.
- [18] F. Kling and A. Pozdnoukhov. When a city tells a story: urban topic analysis. In *SIGSPATIAL*, pages 482–485, 2012.
- [19] G. Lamprianidis, D. Skoutas, G. Papatheodorou, and D. Pfoser. Extraction, integration and analysis of crowdsourced points of interest from multiple web sources. In *GEOCROWD*, pages 16–23, 2014.
- [20] D. Laptev, A. Tikhonov, P. Serdyukov, and G. Gusev. Parameter-free discovery and recommendation of areas-of-interest. In *SIGSPATIAL*, pages 113–122, 2014.
- [21] J. Liu, G. Yu, and H. Sun. Subject-oriented top- $k$  hot region queries in spatial dataset. In *CIKM*, pages 2409–2412, 2011.
- [22] S. Ravi, D. Rosenkrantz, and G. Tayi. Heuristic and special case algorithms for dispersion problems. *Operations Research*, 42(2):299–310, 1994.
- [23] J. B. Rocha-Junior and K. Nørnvåg. Top- $k$  spatial keyword queries on road networks. In *EDBT*, pages 168–179, 2012.
- [24] Y. Tao, X. Hu, D. Choi, and C. Chung. Approximate MaxRS in spatial databases. *PVLDB*, 6(13):1546–1557, 2013.
- [25] E. Vee, U. Srivastava, J. Shanmugasundaram, P. Bhat, and S. Amer-Yahia. Efficient computation of diverse query results. In *ICDE*, pages 228–236, 2008.
- [26] M. R. Vieira, H. L. Razente, M. C. N. Barioni, M. Hadjieleftheriou, D. Srivastava, C. T. Jr., and V. J. Tsotras. On query result diversification. In *ICDE*, pages 1163–1174, 2011.
- [27] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, pages 688–699, 2009.

# Finding Frequently Visited Indoor POIs Using Symbolic Indoor Tracking Data

Hua Lu    Chenjuan Guo    Bin Yang    Christian S. Jensen

Department of Computer Science, Aalborg University, Denmark

{luhua, cguo, byang, csj}@cs.aau.dk

## ABSTRACT

Indoor tracking data is being amassed due to the deployment of indoor positioning technologies. Analysing such data discloses useful insights that are otherwise hard to obtain. For example, by studying tracking data from an airport, we can identify the shops and restaurants that are most popular among passengers. In this paper, we study two query types for finding frequently visited Points of Interest (POIs) from symbolic indoor tracking data. The snapshot query finds those POIs that were most frequently visited at a given time point, whereas the interval query finds such POIs for a given time interval. A typical example of symbolic tracking is RFID-based tracking, where an object with an RFID tag is detected by an RFID reader when the object is in the reader's detection range. A symbolic indoor tracking system deploys a limited number of proximity detection devices, like RFID readers, at preselected locations, covering only part of the host indoor space. Consequently, symbolic tracking data is inherently uncertain and only enables the discrete capture of the trajectories of indoor moving objects in terms of coarse regions. We provide uncertainty analyses of the data in relation to the two kinds of queries. The outcomes of the analyses enable us to design processing algorithms for both query types. An experimental evaluation with both real and synthetic data suggests that the framework and algorithms enable efficient and scalable query processing.

## 1. INTRODUCTION

Indoor spaces such as shopping malls, office buildings, libraries, metro stations, and airports serve as the settings of significant parts of people's daily lives. The indoor movements of people are increasingly datafied due to advances in indoor positioning [1]. As a result, a new type of data—indoor tracking data—is being accumulated in a variety of formats determined by the particular indoor positioning technologies used.

As in the case for outdoor tracking data [3], analyzing indoor tracking data can reveal how different parts of an

indoor space are used by its inhabitants, e.g., the number of visits to a particular part of space over time can be determined. The findings are potentially useful in practical scenarios. For example, the lease prices of different shop locations in a large shopping mall may be set according to the numbers of people passing by the location. As another example, information on the behavior of past visitors to a museum with multiple exhibitions may be used for making recommendations to new visitors and for planning. These and other example scenarios may benefit from flow counting using indoor tracking data.

Flow counting is non-trivial in indoor spaces, where new, unique technical challenges exist that are different from those in outdoor contexts. These in turn call for novel data management techniques.

First of all, indoor positioning systems differ fundamentally from GPS that is prevalent outdoors but that generally does not work indoors. Having to use wireless technologies such as Wi-Fi, Bluetooth, and RFID that originally are designed for data communication, indoor positioning systems work according to different principles and offer positioning accuracies that are below that of GPS. For example, in an RFID based system, an object with an RFID tag is detected by an RFID reader only when the object is in the reader's detection range. Due to their costs, a limited number of readers are deployed, covering only part of the host indoor space. Consequently, the tracking data is inherently uncertain and only enables the discrete capture of the trajectories of indoor moving objects in terms of coarse regions. Such uncertainty renders flow counting techniques based on GPS data unsuitable for indoor spaces.

Further, indoor spaces are characterized by entities like doors, rooms, and hallways that enable and constrain the movements of indoor objects. Compared to outdoor Euclidean or spatial network space, indoor spaces have more complex topologies. When counting flows in indoor spaces, their complex topologies must be taken into account.

This paper considers flow counting based on symbolic indoor tracking data where object locations are captured as circular regions centered at pre-selected indoor locations. Such circular regions correspond to the detection ranges of proximity detection devices, e.g., RFID readers. We define appropriate ways of counting flows based on the uncertain tracking data. Our definitions capture probabilistically how frequently indoor POIs are visited by tracked visitors. Based on the definitions of indoor flow counting, we define two query types for finding indoor POIs that are visited frequently at a given time point and during a given time range,

respectively. We analyze carefully the data uncertainty with respect to the query types, which enables us to design algorithms for both query types. We use synthetic and real data to evaluate the proposals experimentally. The experimental results show that our proposals are efficient and scalable.

We make the following contributions in this paper.

- We define indoor flow counting methods on symbolic indoor tracking data, and we formulate two types of queries for finding frequently visited indoor POIs.
- We derive query related object uncertainty regions by analysing the relationship between the queries and the tracking data.
- We make use of the uncertainty analysis results to design algorithms for the two query types.
- We perform extensive experiments to evaluate the proposed techniques.

The remainder of the paper is organized as follows. Section 2 presents the format of symbolic indoor tracking data and formulates the research problems. Section 3 derives uncertainty regions for objects. Section 4 details the query processing algorithms. Section 5 reports on the experimental studies. Section 6 reviews the related work, and Section 7 concludes and discusses research directions.

## 2. PROBLEM FORMULATION

We formulate the research problems in this section. Section 2.1 details the symbolic indoor tracking data, and Section 2.2 gives the problem definitions. Table 1 lists notation used throughout the paper.

Symbol	Meaning
$p$	An indoor POI
$P$	A set of indoor POIs
$o$	An indoor moving object
$O$	A set of indoor moving objects
$t$	A time point
$rd$	An indoor tracking record
$\Phi_t(p)$	The flow of $p$ at time $t$
$\Phi_{t_s, t_e}(p)$	The flow of $p$ during interval $[t_s, t_e]$
$V_{max}$	The maximum speed of indoor moving objects

Table 1: Notation

### 2.1 Symbolic Indoor Tracking Data

In symbolic indoor object tracking, raw position readings are reported in the format  $\langle objectID, deviceID, t \rangle$ . Such a 3-tuple means that the object identified by  $objectID$  is seen by the device  $deviceID$  at time  $t$ . As the positioning works at a configured sampling frequency, an object is typically seen in multiple, consecutive raw readings by the same device. Such consecutive raw readings are merged [10] to form a tracking record of the form  $\langle ID, objectID, deviceID, t_s, t_e \rangle$ . Such a tracking record means that the object is continuously seen by the device from time  $t_s$  to time  $t_e$ , i.e.,  $t_s$  is the start time for the continuous detection and  $t_e$  is the end time. An object tracking table (OTT) is used to store such historical tracking records, as exemplified in Table 2, where attribute  $ID$  is a record identifier.

$ID$	$objectID$	$deviceID$	$t_s$	$t_e$
$rd_1$	$o_1$	$dev_4$	$t_1$	$t_2$
$rd_2$	$o_2$	$dev_4$	$t_1$	$t_2$
$rd_3$	$o_1$	$dev_2$	$t_5$	$t_6$
$rd_4$	$o_2$	$dev_1$	$t_7$	$t_8$
$rd_5$	$o_1$	$dev_1$	$t_9$	$t_{10}$
$rd_6$	$o_1$	$dev_{12}$	$t_{15}$	$t_{16}$
$rd_7$	$o_2$	$dev_{13}$	$t_{20}$	$t_{21}$
$rd_8$	$o_1$	$dev_{13}$	$t_{21}$	$t_{22}$
$rd_9$	$o_2$	$dev_{13}$	$t_{29}$	$t_{30}$
...	...	...	...	...

Table 2: Object Tracking Table (OTT)

### 2.2 Problem Definitions

We assume that each indoor POI  $p$  has some fixed extent modeled by a polygon, and for simplicity, we equate a POI  $p$  with its polygon. As an entire indoor space is typically not fully covered by proximity detection devices like RFID readers due to economic constraints, there are considerable time intervals during which an object is not seen at all. Consequently, flow counting is not straightforward in the setting of uncertain indoor tracking data. Since object locations are uncertain, exact counting of objects does not make sense. Instead, we estimate how many objects that appeared in an indoor POI’s range at a particular past time point or during a past time range. For that purpose, we need to determine an indoor moving object’s *uncertainty region* that tells where the object can possibly be located. We differentiate between snapshot and interval uncertainty regions.

Given an object  $o$ , we use  $UR(o, t)$  to denote  $o$ ’s *snapshot uncertainty region* at time point  $t$ . In particular,  $UR(o, t)$  captures an indoor region in which object  $o$  can possibly be at time  $t$ . Next, we use  $UR(o, [t_s, t_e])$  to denote  $o$ ’s *interval uncertainty region* during time interval  $[t_s, t_e]$ . Here,  $UR(o, [t_s, t_e])$  captures the indoor region in which object  $o$  can possibly be during time interval  $[t_s, t_e]$ . We describe how to derive these uncertainty regions in Section 3. Based on the uncertainty regions, we define the following concepts for flow counting.

First, we consider an object  $o$ ’s presence in the range of an indoor POI  $p$ . The presence stipulates how we “count” objects for a POI  $p$ .

**DEFINITION 1 (OBJECT PRESENCE).** *During a given time interval  $[t_s, t_e]$ , an object  $o$ ’s interval presence in a POI  $p$  is*

$$\phi_{t_s, t_e, p}(o) = \frac{\text{area}(UR(o, [t_s, t_e]) \cap p)}{\text{area}(p)}. \quad (1)$$

Similarly,  $\phi_{t, p}(o)$  is defined by replacing  $UR(o, [t_s, t_e])$  with  $UR(o, t)$ . The idea of object presence is to capture the intersection between the object’s uncertainty region and the POI’s range. Intuitively, the larger the intersection, the more likely it is that the object was in the POI. For an arbitrary object  $o$  and an arbitrary POI  $p$ , it is apparent that  $0 \leq \phi_{t, p}(o) \leq 1$  and  $0 \leq \phi_{t_s, t_e, p}(o) \leq 1$ . Therefore,  $o$ ’s object presence can be regarded as the probability that  $o$  is in POI  $p$  at  $t$  or during  $[t_s, t_e]$ .

Next, we define the concept of flow for indoor POIs.

DEFINITION 2 (FLOW). Suppose  $O$  is the set of all objects in an indoor space of interest. Given an indoor POI  $p$  and a time interval  $[t_s, t_e]$ ,  $p$ 's interval flow is defined as

$$\Phi_{t_s, t_e}(p) = \sum_{o \in O} \phi_{t_s, t_e, p}(o). \quad (2)$$

Similarly,  $\Phi_t(p)$  is defined by replacing  $\phi_{t_s, t_e, p}(o)$  with  $\phi_{t, p}(o)$ . The flow definitions perform weighted counting of objects that stay in POI  $p$  at time  $t$  or during time interval  $[t_s, t_e]$ , and the weight assigned to each object is its object presence.

With the above definitions, we formulate two types of queries that return the top- $k$  frequently visited indoor POIs.

PROBLEM 1 (SNAPSHOT TOP- $k$  INDOOR POIS QUERY). Given a set  $P$  of indoor POIs, a time point  $t$ , and an integer  $k$  ( $0 < k \leq |P|$ ), return a  $k$ -subset  $P_T$  of  $P$  such that  $\forall p \in P_T (\forall p' \in P \setminus P_T (\Phi_t(p) \geq \Phi_t(p')))$ .

PROBLEM 2 (INTERVAL TOP- $k$  INDOOR POIS QUERY). Given a set  $P$  of indoor POIs, a time interval  $[t_s, t_e]$ , and an integer  $k$  ( $0 < k \leq |P|$ ), return a  $k$ -subset  $P_T$  of  $P$  such that  $\forall p \in P_T (\forall p' \in P \setminus P_T (\Phi_{t_s, t_e}(p) \geq \Phi_{t_s, t_e}(p')))$ .

These queries return the indoor POIs that are visited by the largest number of visitors at a time point or during a time interval. This functionality has many applications. For example, it can be used to identify the most popular shops in a shopping mall. Shop rental fees can take such information into account. As another example, it can be used to identify possible bottlenecks that slow down movement in an airport.

### 3. DERIVING UNCERTAINTY REGIONS

We elaborate on how to derive uncertainty regions for an given object and specified time parameters. Section 3.1 presents the basic terminology, including uncertainty regions for snapshot queries. Section 3.2 focuses on uncertainty regions for interval queries. Section 3.3 addresses how to finalize the uncertainty regions in a given indoor space.

#### 3.1 Basic Terminology

We first cover basic terminology adopted from previous work [10, 17, 25] that is necessary for understanding the paper's contributions.

##### 3.1.1 Tracking States

Given a time point  $t$ , an object  $o$  may be in either an active state or an inactive state [25]. Specifically, if there is a tracking record  $rd_{cov}$  for  $o$  in  $OTT$  (see Table 2 in Section 2.1) covering time  $t$ , we say  $o$  is in an *active state* at  $t$ . We use  $rd_{pre}$  to refer to  $rd_{cov}$ 's predecessor record in  $OTT$ . If no tracking record for  $o$  exists in  $OTT$  covering time  $t$ , object  $o$  is in an *inactive state* at  $t$ . In this case, we identify two relevant tracking records. Record  $rd_{pre}$  is the tracking record for  $o$  immediately before  $o$  becomes inactive. Record  $rd_{suc}$  is the tracking record for  $o$  immediately after  $o$  leaves the inactive state, i.e.,  $rd_{suc}$  is the first tracking record when  $o$  becomes active again after time  $t$ . Note that  $rd_{pre}.t_e < t < rd_{suc}.t_s$  if object  $o$  is inactive at time  $t$ .

Object  $o_1$ 's tracking records in Table 2 are plotted along the time axis in Figure 1 (originally from [17]). It is in an active state at time  $t_5$ , and it is in an inactive state at time  $t_{19}$ . Relevant particular tracking records are also marked in

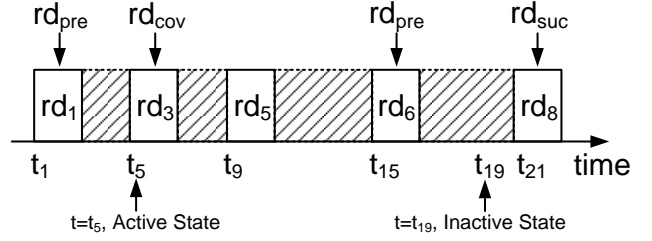


Figure 1: States in Symbolic Object Tracking

the figure. For simplicity, we use  $dev_{pre}$  ( $dev_{cov}$  or  $dev_{suc}$ ) to refer to  $rd_{pre}$ 's ( $rd_{cov}$ 's or  $rd_{suc}$ 's) device.

An object is either in an active or an inactive state at a time point  $t$ , whereas it may change state during a time interval  $[t_s, t_e]$ . In Figure 1, object  $o_1$  changes state five times during  $[t_5, t_{19}]$ .

##### 3.1.2 Snapshot Uncertainty Regions

Two cases exist for the snapshot uncertainty region of an object  $o$  at a time point  $t$  [17]. We suppose that  $V_{max}$  is the maximum speed that object  $o$  can move at in the given indoor space.

**Case 1:** Object  $o$  is in an active state at  $t$ . In this case,  $UR(o, t) = Ring(dev_{pre}, V_{max} \cdot (t - rd_{pre}.t_e)) \cap dev_{cov}.Range$ . I.e.,  $o$ 's uncertainty region is the intersection of  $dev_{cov}$ 's detection range and the ring in which  $o$  can be after leaving  $dev_{pre}$ 's detection range. Specifically, this ring is determined by  $dev_{pre}$ 's range and the maximum distance  $o$  can move from  $rd_{pre}.t_e$  to  $t$ . This case is illustrated in Figure 2(a).

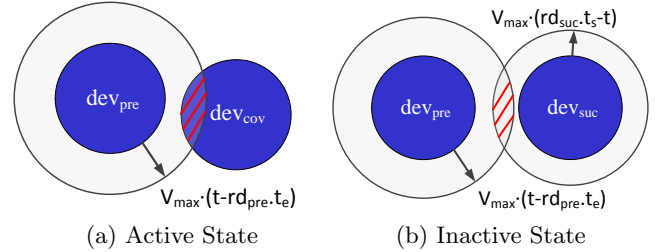


Figure 2: Snapshot Uncertainty Regions

**Case 2:** Object  $o$  is in an inactive state at  $t$ . In this case,  $UR(o, t) = Ring(dev_{pre}, V_{max} \cdot (t - rd_{pre}.t_e)) \cap Ring(dev_{suc}, V_{max} \cdot (rd_{suc}.t_s - t))$ . Here,  $UR(o, t)$  is the intersection of two rings: one involves  $rd_{pre}$  and is the same as that in Case 1; the other involves  $rd_{suc}$  and the maximum distance  $o$  can move from  $t$  to  $rd_{suc}.t_s$ . This case is illustrated in Figure 2(b).

##### 3.1.3 Uncertainty Region Involving Two Consecutive Readings

Without loss of generality, let  $rd_i$  and  $rd_j$  be two consecutive tracking records for object  $o$ . Records  $rd_i$  and  $rd_j$  involve devices  $dev_i$  and  $dev_j$ , respectively. For time intervals  $[rd_i.t_s, rd_i.t_e]$  and  $[rd_j.t_s, rd_j.t_e]$ , object  $o$  is in  $dev_i$ 's and  $dev_j$ 's detection range, respectively. For time interval  $[rd_i.t_e, rd_j.t_s]$ , object  $o$ 's location can be constrained by an

<sup>1</sup> $Ring(dev, \rho)$  denotes the ring whose inner circle is device  $dev$ 's detection circle and whose outer circle extends the inner circle's radius by  $\rho$ .

extended ellipse [10] whose two foci are two points on the boundaries of the two detection ranges (see the two circular regions in dark in Figure 3). Furthermore, the length of the ellipse's major axis is  $2a = V_{max} \cdot (rd_j.t_s - rd_i.t_e)$ . Such an extended ellipse is illustrated in Figure 3. Object  $o$ 's uncertainty region for a time interval is represented by the extended ellipse excluding the two circular regions for the two proximity detection devices. The details of deriving such an extended ellipse can be found in the literature [10, 19]. We use  $\Theta(dev_i, dev_j, rd_i.t_e, rd_j.t_s)$  to refer to the complete region covered by the ellipse, which will be used in our subsequent discussions.

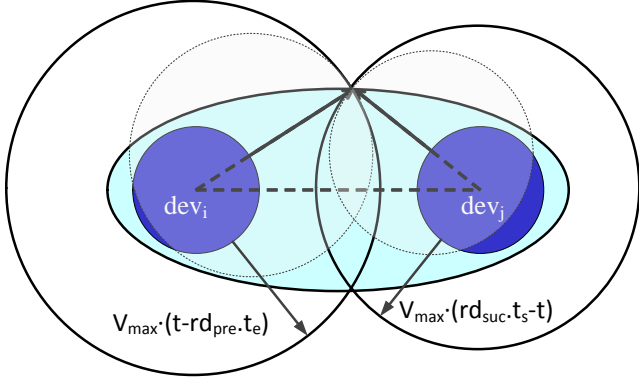


Figure 3: Uncertainty Region Examples

### 3.2 Interval Uncertainty Regions

In order to derive the uncertainty region of object  $o$  in a given time interval  $[t_s, t_e]$ , we need to find all its relevant tracking records in  $OTT$ . Temporally, all those records form a chain of detections of object  $o$ . In particular, we need to find the start record, the end record, and all the records in-between for object  $o$ . We use  $rd_s$  and  $rd_e$  to refer to the start (first in the chain) and end (the last) records, respectively. Although there may be multiple in-between records, we use  $rd_b$  to refer to a concrete record between  $rd_s$  and  $rd_e$  when it is of particular interest in our discussion. Table 3 gives the start and end records for all cases regarding object  $o$ 's state at  $t_s$  and  $t_e$ .

$t_s \backslash t_e$	Active		Inactive	
	$rd_s$	$rd_e$	$rd_s$	$rd_e$
<b>Active</b>	$rd_{cov}(t_s)$	$rd_{cov}(t_e)$	$rd_{cov}(t_s)$	$rd_{suc}(t_e)$
<b>Inactive</b>	$rd_{pre}(t_s)$	$rd_{cov}(t_e)$	$rd_{pre}(t_s)$	$rd_{suc}(t_e)$

Table 3: Start and End Records for  $[t_s, t_e]$

Next, we derive  $UR(o, [t_s, t_e])$ , object  $o$ 's uncertainty region during  $[t_s, t_e]$ , for the four cases given in Table 3.

**Case 1:** Object  $o$  is active at both  $t_s$  and  $t_e$ , i.e.,  $rd_s = rd_{cov}(t_s)$  and  $rd_e = rd_{cov}(t_e)$ . Without loss of generality, we assume that there are two records in-between. An illustration is shown in Figure 4, where  $dev_s$  is  $rd_s.deviceID$  and  $dev_e$  is  $rd_e.deviceID$ . Object  $o$ 's uncertainty region is then the union of the ellipse regions associated with the consecutive tracking records from  $rd_s$  to  $rd_e$ . Formally,  $UR(o, [t_s, t_e]) = \bigcup_{i=1..|R|-1} \Theta(dev_i, dev_{i+1}, rd_i.t_e, rd_{i+1}.t_s)$ ,

where  $rd_i$  is a tracking record from sequence  $R = \langle rd_{cov}(t_s), rd_{b1}, \dots, rd_{cov}(t_e) \rangle$  and  $dev_i = rd_i.deviceID$ .

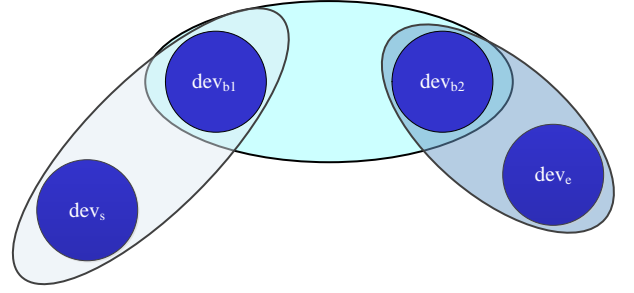


Figure 4: Interval Uncertainty Region for Case 1

**Case 2:** Object  $o$  is inactive at  $t_s$  but active at  $t_e$ , i.e.,  $rd_s = rd_{pre}(t_s)$  and  $rd_e = rd_{cov}(t_e)$ . As for Case 1, an illustration is shown in Figure 5. Here, we need to pay particular attention to the time interval  $[t_s, rd_{b1}.t_s]$  before object  $o$  becomes detected by device  $dev_{b1}$ . For this interval, due to the maximum speed constraint, the object can only be in the intersection of the ellipse region  $\Theta_s = \Theta(dev_s, dev_{b1}, rd_s.t_e, rd_{b1}.t_s)$  and the ring captured as  $Ring_s = Ring(dev_{b1}, V_{max} \cdot (rd_{b1}.t_s - t_s))$ .

As a result, object  $o$ 's uncertainty region for the entire interval  $[t_s, t_e]$  is this intersection unioned with the union of all other ellipse regions associated with all other consecutive tracking records from  $rd_{b1}$  to  $rd_e = rd_{cov}(t_e)$ . Formally,  $UR(o, [t_s, t_e]) = (\Theta_s \cap Ring_s) \cup \bigcup_{i=1..|R|-1} \Theta(dev_i, dev_{i+1}, rd_i.t_e, rd_{i+1}.t_s)$ , where  $rd_i$  is a tracking record from sequence  $R = \langle rd_{b1}, \dots, rd_{cov}(t_e) \rangle$  and  $dev_i = rd_i.deviceID$ .

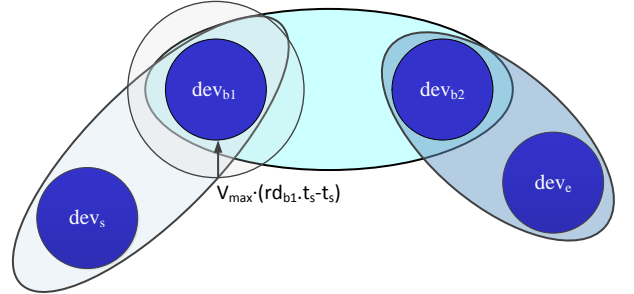


Figure 5: Interval Uncertainty Region for Case 2

**Case 3:** Object  $o$  is active at  $t_s$  but inactive at  $t_e$ , i.e.,  $rd_s = rd_{cov}(t_s)$  and  $rd_e = rd_{suc}(t_e)$ . An illustration is shown in Figure 6. In this case, we need to pay particular attention to the time interval  $[rd_{b2}.t_e, t_e]$  after object  $o$  is last seen by device  $rd_{b2}.deviceID$ . For this interval, due to the maximum speed constraint, the object can only be in the intersection of the ellipse region  $\Theta_e = \Theta(dev_{b2}, dev_e, rd_{b2}.t_e, rd_e.t_s)$  and the ring  $Ring_e = Ring(dev_{b2}, V_{max} \cdot (t_e - rd_{b2}.t_e))$ .

As a result, object  $o$ 's uncertainty region for the entire interval  $[t_s, t_e]$  is the above intersection unioned with the union of all other ellipse regions associated with all other consecutive tracking records from  $rd_s = rd_{cov}(t_s)$  to  $rd_{b2}$ . Formally,  $UR(o, [t_s, t_e]) = (\Theta_e \cap Ring_e) \cup \bigcup_{i=1..|R|-1} \Theta(dev_i, dev_{i+1}, rd_i.t_e, rd_{i+1}.t_s)$ , where  $rd_i$  is a tracking record from sequence  $R = \langle rd_{cov}(t_s), rd_{b1}, \dots, rd_{b2} \rangle$  and  $dev_i = rd_i.deviceID$ .



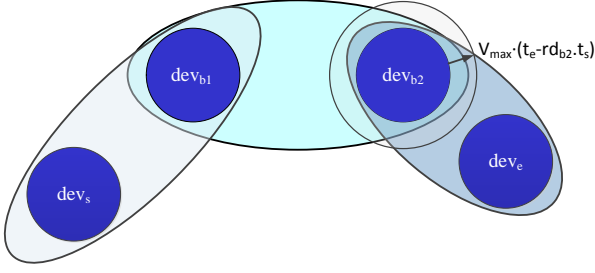


Figure 6: Interval Uncertainty Region for Case 3

**Case 4:** Object  $o$  is inactive at both  $t_s$  and  $t_e$ , i.e.,  $rd_s = rd_{pre}(t_s)$  and  $rd_e = rd_{suc}(t_e)$ . An illustration is shown in Figure 7. This case combines the handling of the beginning and end from Cases 2 and 3, respectively. Therefore, object  $o$ 's uncertainty region in  $[t_s, t_e]$  is  $UR(o, [t_s, t_e]) = (\Theta_s \cap Ring_s) \cup (\Theta_e \cap Ring_e) \cup \bigcup_{i=1..|R|-1} \Theta(dev_i, dev_{i+1}, rd_i, t_e, rd_{i+1}, t_s)$ , where  $rd_i$  is a tracking record from sequence  $R = \langle rd_{b1}, \dots, rd_{b2} \rangle$  and  $dev_i = rd_i.deviceID$ .

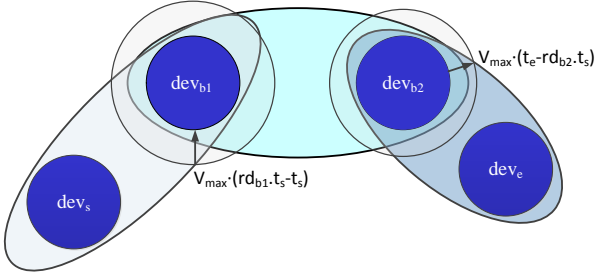


Figure 7: Interval Uncertainty Region for Case 4

### 3.3 Indoor Topology Check

Our coverage so far does not consider the topology of the indoor space. Specifically, we need to check  $UR(o, t)$  against the given indoor space and exclude all parts of the space that are not accessible to object  $o$  at time  $t$ . Also, we need to check  $UR(o, [t_s, t_e])$  against the given indoor space and exclude all parts of the space that are not accessible to object  $o$  from time  $t_s$  and  $t_e$  without exiting  $UR(o, [t_s, t_e])$ . For each type of uncertainty region, the part that remains after the indoor topology check is object  $o$ 's uncertainty region.

Examples are shown in Figure 8, where the shaded parts must be excluded from the object's uncertainty regions. In Figure 8(a), an object  $o$  is inactive at time point  $t$ . Suppose that it was detected by device 1 at time  $t_1 < t$  and then by device 3 after  $t$ . According to the discussion illustrated in Figure 2(b),  $o$ 's snapshot uncertainty region  $UR(o, t)$  is the intersection of the two large circular regions constrained by the maximum speed  $V_{max}$ . However, the shaded part should be excluded as it is too far away for object  $o$  to be able to reach it at time point  $t$ . After leaving device 1's detection range, object  $o$  must go through door 2 to enter the shaded part, which would yield an indoor walking distance that exceeds the maximum Euclidean distance  $o$  can move from  $t_1$  to  $t$ , i.e.,  $V_{max} \cdot (t - t_1)$ . Therefore, this part should be excluded from  $UR(o, t)$ . If the topology check is skipped in this example, the object would be "counted" for room 2

whose flow in turn would be increased incorrectly. Such a miscalculation can result in room 2 entering a top- $k$  query result as a false positive.

It is worth noting that if room 2 had had a door in the region given by the intersection of the two large circular regions (see Figure 8(a)), further checking would be needed to exclude parts of space that are too far away for an object in those parts to be able to move from one device to the other via the door. Based on the maximum speed  $V_{max}$ , we can calculate the earliest time  $t_2$  ( $t_1 < t_2 < t$ ) for an object to reach the assumed door. Then, moving from the door, the possible region for the object to reach before  $t$  is constrained by the distance  $V_{max} \cdot (t - t_2)$ . Any part of space beyond that distance from the assumed door should be excluded from the object's uncertainty region.

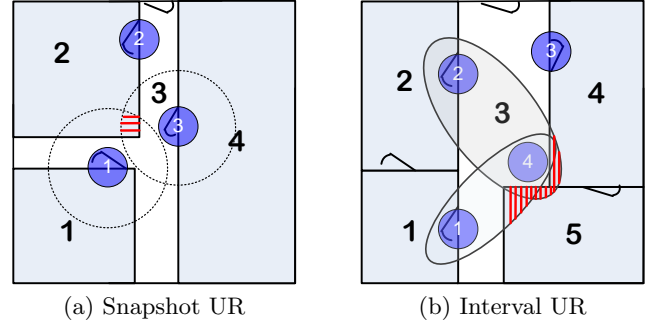


Figure 8: Indoor Topology Check

In Figure 8(b), an object  $o$  is detected by devices 1, 4, and then 2. Due to the maximum speed constraint, the object cannot have entered the shaded regions (in rooms 4 and 5). Therefore, these regions should be excluded from object  $o$ 's interval uncertainty region  $UR(o, [t_s, t_e])$ . Otherwise, the flows of rooms 4 and 5 would be incorrectly increased and they may enter the top- $k$  result as false positives.

In our framework, we do such indoor topology checks to capture the uncertainty regions for indoor moving objects better. Specifically, after an uncertainty region  $UR$  is obtained, we divide it into several disconnected parts according to the indoor topology. For each such part, we check its indoor distance from the involved devices. If the indoor distance exceeds the corresponding maximum Euclidean distance, the part is excluded. For simplicity, we use  $UR(o, t)$  and  $UR(o, [t_s, t_e])$  to refer to the object's uncertainty region also after the topology check in the following.

**Remark** For the sake of conciseness in our presentation, we implicitly assume that the detection ranges of proximity detection devices do not overlap. Overlapping detection ranges can be accommodated by making only slight changes in the corresponding low-level geometric computations; no changes are needed to the overall process of deriving the uncertainty regions for objects. Therefore, the uncertainty analysis presented in this section and the algorithms to be presented in the next section are able to accommodate overlapping detection ranges. We omit the low-level details here in order to keep our discussion concentrated and concise.

## 4. TOP-K INDOOR POI ALGORITHMS

We proceed to design query processing algorithms by making use of the uncertainty regions derived in Section 3. Section 4.1 presents the indexes we use for query processing.

Sections 4.2 and 4.3 detail the algorithms for snapshot and interval queries, respectively.

## 4.1 Indexes

We use two R-tree [4] based indexes for the data to facilitate query processing—one for the tracking data and one for the indoor POIs.

The object tracking table (*OTT*) (refer to Table 2) is indexed by an augmented 1D R-tree named A1R-tree [17] on the temporal attributes as follows. Let  $rd_c$  be a tracking record for object  $o$  in *OTT*. Such a record  $rd_c$  is indexed by an A1R-tree leaf entry of the form  $(t^+, t^-, Ptr_p, Ptr_c)$ . Specifically,  $Ptr_c$  is a pointer to record  $rd_c$ , and  $Ptr_p$  points to record  $rd_p$  that is  $o$ 's previous record in *OTT*, i.e.,  $rd_p$  is  $rd_c$ 's predecessor for the same object. In addition, we set  $t^+ = rd_p.t_e$  and  $t^- = rd_c.t_e$ . We call  $(t^+, t^-)$  (i.e.,  $(rd_p.t_e, rd_c.t_e)$ ) an augmented tracking time interval. A non-leaf entry in an A1R-tree is of the form  $(t^+, t^-, cp)$ , where  $cp$  is a pointer to a child node and  $[t^+, t^-]$  is the minimum bounding interval that contains all time intervals in the child node.

With an A1R-tree, we are able to efficiently obtain all the tracking records relevant to the uncertainty region determination described in Section 2. For a snapshot uncertainty region at query time  $t$ , a point query with  $t$  as the parameter via the A1R-tree will return a leaf entry  $le = (t^+, t^-, Ptr_p, Ptr_c)$  where  $(t^+, t^-)$  covers  $t$ . If  $le.Ptr_p.t_e < t < le.Ptr_c.t_s$ , the object is inactive at time  $t$ , and  $le.Ptr_p$  ( $le.Ptr_c$ ) points to  $rd_{pre}$  ( $rd_{suc}$ ) (see Figures 1 and 2(b)). Otherwise, the object is active at time  $t$  ( $le.Ptr_c.t_s \leq t \leq le.Ptr_c.t_e$ ), and  $le.Ptr_c$  points to the  $rd_{cov}$  (see Figures 1 and 2(a)).

For the uncertainty region in a query time interval  $[t_s, t_e]$ , a range query with  $[t_s, t_e]$  as the parameter via the A1R-tree returns a series of leaf entries such that the first leaf entry's augmented tracking time interval covers  $t_s$  and that of the last leaf entry covers  $t_e$ . Furthermore, the first (last) leaf entry contains the pointer to the particular first (last) tracking record needed for the four cases presented in Section 3.2, depending on the case encountered. All in-between tracking records, if any, are accordingly obtained through those in-between leaf entries.

The set of indoor POIs is indexed by another R-tree, called  $R_P$ . For simplicity, we consider one floor and therefore use a 2D R-tree to index all indoor POIs in our implementation. Nevertheless, our analysis of uncertainty regions as well as the query processing techniques can be extended to multi-floor cases.

## 4.2 Snapshot Query Algorithms

### 4.2.1 Iterative Algorithm

A straightforward approach to compute the snapshot query (Problem 1) is to compute the snapshot flow value for each POI  $p$  in the query and then return the  $k$  POIs with the highest flow values. This iterative algorithm is formalized in Algorithm 1.

The iterative algorithm uses a hash table *flows* to keep flow values for all POIs (lines 1–2). Using a point query on the A1R-tree  $R_O$  on the *OTT* (line 3), the algorithm obtains all the relevant leaf entries whose augmented tracking time interval contains the query time  $t$ , as described in Section 4.1. For each object  $o$  thus obtained from the *OTT*

(lines 4–5), the algorithm derives the uncertainty region  $UR(o, t)$  for either an inactive state (lines 6–9) or an active state (line 11). Then all POIs that intersect  $UR(o, [t_s, t_e])$  are found (line 12). For each such POI  $p$ , its flow value is increased by the current object  $o$ 's presence in  $p$  (lines 13–14). Finally, the top- $k$  POIs are returned (line 15).

---

**Algorithm 1** *iterativeSnapshot*(R-tree  $R_P$  for indoor POIs, A1R-tree  $R_O$  for *OTT*, time point  $t$ , integer  $k$ )

---

```

1: initialize a hash table flows:  $\{POI\} \rightarrow [0, +\infty]$ 
2: for each POI  $p$  do flows[ $p$ ]  $\leftarrow 0$ 
3: LeafEntrySet les  $\leftarrow R_O.PointQuery(t)$ 
4: for each leaf entry  $le \in les$  do
5:    $o \leftarrow le.Ptr_c.objectID$ 
6:    $ring_1 \leftarrow Ring(le.Ptr_p.deviceID, V_{max} \cdot (t - le.Ptr_p.t_e))$ 
7:   if  $le.Ptr_p.t_e < t < le.Ptr_c.t_s$  then  $\triangleright$  The object is in an
      inactive state
8:      $ring_2 \leftarrow Ring(le.Ptr_c.deviceID, V_{max} \cdot (le.Ptr_c.t_s - t))$ 
9:      $UR(o, t) \leftarrow ring_1 \cap ring_2$ 
10:  else  $\triangleright$  The object is in an active state
11:     $UR(o, t) \leftarrow ring_1 \cap le.Ptr_c.deviceID.Range$ 
12:   $ps \leftarrow R_P.IntersectionQuery(UR(o, t))$ 
13:  for each POI  $p \in ps$  do
14:     $flows[p] \leftarrow flows[p] + \frac{Area(UR(o, t) \cap p)}{Area(p)}$ 
15: return the top- $k$  from flows.keys with the highest values

```

---

### 4.2.2 Join Algorithm

The iterative algorithm suffers from two limitations. It iterates over all objects and relevant POIs, which may be inefficient. Also, it needs to compute a considerable number of uncertainty regions. This may not pay off, as some POIs may finally get very low overall flow values while having incurred complex uncertainty region computations. Motivated by these observations, we design a more efficient join based method that is formalized in Algorithm 2.

The join algorithm consists of three phases. The first (lines 1–11) builds an in-memory aggregate R-tree  $R_I$  for all objects whose augmented tracking interval covers query time  $t$ . These objects are again obtained by a point query on the A1R-tree (line 2). If an object  $o$  is inactive at  $t$  (line 5), we obtain the minimum bounding rectangles (MBRs) of the two (*pre* and *suc*) proximity detection devices' detection ranges, and extend each of them by the corresponding maximum possible distance (lines 6–7). The two extended MBRs are then merged to form the object's MBR (line 8). Otherwise, the MBR of device  $dev_{cov}$ 's range is used for the active state (lines 10). After the MBR is determined, the object  $o$  is inserted into tree  $R_I$ , where each node entry is augmented with a field *count* that is the number of all objects in the corresponding sub-tree.

The second phase (lines 12–18) is the initialization for joining the POI R-tree  $R_P$  and the aggregate object R-tree  $R_I$ . Here, the algorithm initializes a priority queue  $Q$  that gives higher priority to  $R_P$  node entries (groups of POIs) that potentially have higher flow values. In particular, each  $R_P$  entry  $e_P$  is associated with a join list of  $R_I$  entries whose MBRs overlap  $e_P$ 's MBR. Note that for any POI  $p$  in  $e_P$ 's sub-tree, those objects that can contribute to  $p$ 's flow value can only come from such  $R_I$  entries. When the two tree roots are joined (line 13–18) initially, the *count* values in the  $R_I$  entries are used to upper bound the flow values as an object's presence in any POI never exceeds 1 (Definition 1).

The third phase carries out the join (lines 19–48). The

---

**Algorithm 2** joinSnapshot(R-tree  $R_P$  for indoor POIs, A1R-tree  $R_O$  for  $OTT$ , time  $t$ , integer  $k$ )

---

```

1: initialize an in-memory aggregate R-tree  $R_I$ 
2: LeafEntrySet  $les \leftarrow R_O.PointQuery(t)$ 
3: for each leaf entry  $le \in les$  do
4:    $o \leftarrow le.Ptr_c.objectID$ 
5:   if  $le.Ptr_p.t_e < t < le.Ptr_c.t_s$  then  $\triangleright$  Inactive state
6:      $mbr_1 \leftarrow \text{extend MBR}(le.Ptr_p.deviceID.Range)$  by
        $V_{max} \cdot (t - le.Ptr_p.t_e)$ 
7:      $mbr_2 \leftarrow \text{extend MBR}(le.Ptr_c.deviceID.Range)$  by
        $V_{max} \cdot (le.Ptr_c.t_s - t)$ 
8:      $mbr \leftarrow \text{MBR}(mbr_1, mbr_2)$ 
9:   else
10:     $mbr \leftarrow \text{MBR}(le.Ptr_c.deviceID.Range)$ 
11:   insert  $(o, mbr)$  into  $R_I$ 
12: initialize a priority queue  $Q$ 
13: for each entry  $e_P$  in  $R_P.root$  do
14:    $ubFlow \leftarrow 0; list \leftarrow \emptyset$ 
15:   for each entry  $e_I$  in  $R_I.root$  do
16:     if  $e_P.mbr$  intersects  $e_I.mbr$  then
17:        $ubFlow \leftarrow ubFlow + e_I.count; list \leftarrow list \cup \{e_I\}$ 
18:    $Q.enqueue((e_P, list, ubFlow))$ 
19:  $result \leftarrow \emptyset$ ; initialize a hash table  $H_U$ 
20: while  $Q$  is not empty do
21:    $(e_P, list) \leftarrow Q.dequeue()$ 
22:   if  $e_P$  is a leaf entry then
23:     if  $list$  is null then
24:       add POI  $e_P.object$  to  $result$ 
25:       if  $result = k$  then return  $result$ 
26:     else
27:       if  $list$  contain leaf entries then
28:          $flow \leftarrow 0$ 
29:         for each entry  $e_I \in list$  do
30:           if  $H_U[e_I.object] = \emptyset$  then
31:              $H_U[e_I.object] \leftarrow UR(e_I.object, t)$ 
32:              $flow \leftarrow flow + \phi_{t, e_P.object}(e_I.object)$ 
33:           if  $flow \neq 0$  then  $Q.enqueue((e_P, null, flow))$ 
34:         else
35:            $expandList(e_P, list)$ 
36:       else
37:         if  $list$  contain leaf entries then
38:           for each sub-entry  $e'_P$  in  $e_P.node$  do
39:              $ubFlow \leftarrow 0; list2 \leftarrow \emptyset$ 
40:             for each entry  $e'_I \in list$  do
41:               if  $e'_P.mbr$  intersects  $e'_I.mbr$  then
42:                  $ubFlow \leftarrow ubFlow + 1$ 
43:                  $list2 \leftarrow list2 \cup \{e'_I\}$ 
44:             if  $list2 \neq \emptyset$  then
45:                $Q.enqueue((e'_P, list2, ubFlow))$ 
46:           else
47:             for each sub-entry  $e'_P$  in  $e_P.node$  do
48:                $expandList(e'_P, list)$ 

```

---

join order is controlled by priority queue  $Q$  (lines 20–21). If the current  $R_P$  entry  $e_P$  is a leaf entry, it is processed as follows. If  $e_P$ 's join list is empty, which means its concrete flow value has been calculated and the value is higher than those yet to be calculated, it is added to the result (line 24), and if the result contains  $k$  POIs, the algorithm terminates (line 25). Otherwise, the join list may contain leaf entries or non-leaf entries from  $R_I$ . In the former case (line 27), the flow value of  $e_P$  is calculated by deriving the uncertainty region and presence for each object in the join list (lines 28–32). If the flow value is non-zero, the POI in  $e_P$  is pushed back to  $Q$  with an empty join list (line 33). To avoid deriving uncertainty regions repeatedly for objects that appear

in multiple join lists, we use a hash table  $H_U$  (lines 19 and 30–31) to store the uncertainty regions for objects. If  $e_P$ 's join list contains non-leaf entries, procedure  $expandList$  (Algorithm 3) is called to join  $e_P$  with sub-entries from the join list. The procedure ensures that  $e_P$  is only associated with those  $R_I$  entries whose MBRs intersect  $e_P$ 's (line 4), and it uses the  $count$  values in the  $R_I$  entries to estimate the upper bound of  $e_P$ 's flow value (line 5).

---

**Algorithm 3** expandList(Entry  $e_P$  in R-tree  $R_P$  for indoor POIs, List  $list$  of entries in R-tree  $R_I$ )

---

```

1:  $ubFlow \leftarrow 0; list2 \leftarrow \emptyset$ 
2: for each entry  $e_I \in list$  do
3:   for each sub-entry  $e'_I$  in  $e_I.node$  do
4:     if  $e_P.mbr$  intersects  $e'_I.mbr$  then
5:        $ubFlow \leftarrow ubFlow + e'_I.count$ 
6:        $list2 \leftarrow list2 \cup \{e'_I\}$ 
7: if  $list2 \neq \emptyset$  then
8:    $Q.enqueue((e_P, list2, ubFlow))$ 

```

---

If the current  $R_P$  entry  $e_P$  is a non-leaf entry, it is processed as follows. If the join list contains leaf entries (line 37), the join algorithm overestimates the flow value for each of  $e_P$ 's sub-entries when joining them with the relevant entries in the join list (lines 38–43). Only sub-entries with a non-empty join list are pushed back into the priority queue (lines 44–45). Otherwise, procedure  $expandList$  is called for each of  $e_P$ 's sub-entries (lines 47–48).

## 4.3 Interval Query Algorithms

### 4.3.1 Iterative Algorithm

Algorithm 4 offers a straightforward way of computing the interval query (Problem 2). Overall, it follows the same iter-

---

**Algorithm 4** iterativeInterval(R-tree  $R_P$  for indoor POIs, A1R-tree  $R_O$  for  $OTT$ , time interval  $[t_s, t_e]$ , integer  $k$ )

---

```

1: initialize a hash table  $flows: \{POI\} \rightarrow [0, +\infty]$ 
2: for each POI  $p$  do  $flows[p] \leftarrow 0$ 
3: LeafEntrySet  $les \leftarrow R_O.RangeQuery([t_s, t_e])$ 
4: initialize a hash table  $H$ 
5: for each leaf entry  $le \in les$  do
6:   append  $le.S$  to  $H[le.objectID]$ 
7: for each key  $objectID \in H.keys$  do
8:   get  $(rd_s, \dots, rd_e)$  from  $H[objectID]$ 
9:   calculate  $UR(objectID, [t_s, t_e])$  from  $(rd_s, \dots, rd_e)$ 
10:   $ps \leftarrow R_P.IntersectionQuery(UR(objectID, [t_s, t_e]))$ 
11:  for each POI  $p \in ps$  do
12:     $flows[p] \leftarrow flows[p] + \frac{Area(UR(o, [t_s, t_e]) \cap p)}{Area(p)}$ 
13: return the top- $k$  from  $flows.keys$  with the highest values

```

---

ative paradigm as does Algorithm 1 for the snapshot query. It uses a range query on the A1R-tree to obtain the relevant leaf entries and objects whose augmented tracking time interval overlap the query time interval  $[t_s, t_e]$  (line 3), as described in Section 4.1. An interval query may involve a sequence of tracking records of an object, and therefore the algorithm uses a hash table to form the sequences for all objects obtained (lines 4–6). The uncertainty region of each object is calculated (lines 8–9) according to the discussion in Section 3.2.

### 4.3.2 Join Based Algorithm

**Basic framework.** Like for snapshot query processing, we have a join based algorithm for interval query processing. As formalized in Algorithm 5, it also contains three phases: aggregate object R-tree  $R_I$  construction (lines 1–9), join initialization (lines 10–16), and join processing (lines 17–46). The differences here lie mainly in the first phase and in the details of deriving the uncertainty regions for objects in a join list in the third phase.

**Algorithm 5** `joinInterval`(R-tree  $R_P$  for indoor POIs, A1R-tree  $R_O$  for  $OTT$ , time interval  $[t_s, t_e]$ , integer  $k$ )

---

```

1: initialize a hash table  $H$ 
2: LeafEntrySet  $les \leftarrow R_O.RangeQuery([t_s, t_e])$ 
3: for each leaf entry  $le \in les$  do
4:   append  $le.S$  to  $H[le.objectID]$ 
5: initialize an in-memory aggregate R-tree  $R_I$ 
6: for each key  $objectID \in H.keys$  do
7:   get  $(rd_s, \dots, rd_e)$  from  $H[objectID]$ 
8:    $mbr \leftarrow MBR(objectID, [t_s, t_e])$ 
9:   insert  $(objectID, mbr)$  into  $R_I$ 
10: initialize a priority queue  $Q$ 
11: for each entry  $e_P$  in  $R_P.root$  do
12:    $ubFlow \leftarrow 0$ ;  $list \leftarrow \emptyset$ 
13:   for each entry  $e_I$  in  $R_I.root$  do
14:     if  $e_P.mbr$  intersects  $e_I.mbr$  then
15:        $ubFlow \leftarrow ubFlow + e_I.count$ ;  $list \leftarrow list \cup \{e_I\}$ 
16:    $Q.enqueue((e_P, list, ubFlow))$ 
17:  $result \leftarrow \emptyset$ ; initialize a hash table  $H_U$ 
18: while  $Q$  is not empty do
19:    $(e_P, list) \leftarrow Q.dequeue()$ 
20:   if  $e_P$  is a leaf entry then
21:     if  $list$  is null then
22:       add POI  $e_P.object$  to  $result$ 
23:       if  $result = k$  then return  $result$ 
24:     else
25:       if  $list$  contain leaf entries then
26:          $flow \leftarrow 0$ 
27:         for each entry  $e_I \in list$  do
28:           if  $H_U[e_I.object] = \emptyset$  then
29:              $H_U[e_I.object] \leftarrow UR(e_I.object, [t_s, t_e])$ 
30:              $flow \leftarrow flow + \phi_{t_s, t_e, e_P.object}(e_I.object)$ 
31:           if  $flow \neq 0$  then  $Q.enqueue((e_P, null, flow))$ 
32:       else
33:          $expandList(e_P, list)$ 
34:   else
35:     if  $list$  contain leaf entries then
36:       for each sub-entry  $e'_P$  in  $e_P.node$  do
37:          $ubFlow \leftarrow 0$ ;  $list2 \leftarrow \emptyset$ 
38:         for each entry  $e'_I \in list$  do
39:           if  $e'_P.mbr$  intersects  $e'_I.mbr$  then
40:              $ubFlow \leftarrow ubFlow + 1$ 
41:              $list2 \leftarrow list2 \cup \{e'_I\}$ 
42:           if  $list2 \neq \emptyset$  then
43:              $Q.enqueue((e'_P, list2, ubFlow))$ 
44:       else
45:         for each sub-entry  $e'_P$  in  $e_P.node$  do
46:            $expandList(e'_P, list)$ 

```

---

Although this basic framework still works for interval query processing, preliminary experimentation suggests that it can be improved significantly. In the following, we identify the performance bottleneck and present improvements to the design of the join based algorithm.

**Improvements.** The uncertainty regions for an interval query are much larger than those for a snapshot query. If a single MBR is used for an interval uncertainty region, the

MBR can cover considerable dead space. This is the case in Algorithm 5 that uses a coarse MBR estimation, especially when an overall MBR is created for all tracking records of an object during the query interval (line 8). To alleviate this problem, we introduce several improvements.

Instead of using a single, large MBR to represent an object's trajectory during  $[t_s, t_e]$ , we use a series of much tighter MBRs, each of which is created based on a pair of consecutive tracking records. Suppose that an object  $o$ 's relevant tracking records during  $[t_s, t_e]$  are  $\langle rd_1, \dots, rd_m \rangle$ . For each pair  $(rd_i, rd_{i+1})$ , we create a small MBR  $mbr_i$  for the extended ellipse (Section 3.1.3) defined by the two tracking records.

After we create all such  $m - 1$  smaller MBR  $mbr_i$ s, we create the overall MBR  $mbr$  for all of them. When we insert  $mbr$  into the aggregate R-tree  $R_I$ , we create additional information at the leaf node level for all such smaller  $mbr_i$ s for  $mbr$ . In particular, we insert a pointer from the entry for  $mbr$  to the list of  $mbr_i$ s, such that we can easily access these when we visit  $mbr$ 's node.

Next, we modify the procedure that expands the join list (Algorithm 3). Instead of simply checking whether two MBRs intersect, we do additional checks when a leaf node entry  $e'_I$  is taken from R-tree  $R_I$  (line 3 in Algorithm 3). In particular, if  $e'_I$  is a leaf node entry and its MBR intersects  $e_P$ 's (line 4), we continue to check  $e_P$ 's MBR against the smaller MBRs covered by  $e'_I$  as described above. We include  $e'_I$  into the join list only if at least one such smaller MBR intersects  $e_P$ 's MBR. These finer-grained checks are expected to eliminate many false positives in the join list that would otherwise result from the large, dead space-dominant MBR of  $e'_I$ . This arrangement reduces the join list size for  $e_P$  and reduces also the subsequent join cost.

In similar fashion, we apply the additional MBR intersection checks to Algorithm 5 when it process leaf entries from  $R_I$  (line 39). This is also expected to reduce the join list and the join cost.

An example of the improvements is shown in Figure 9. Object  $o$ 's overall MBR, whose dead space is indicated by

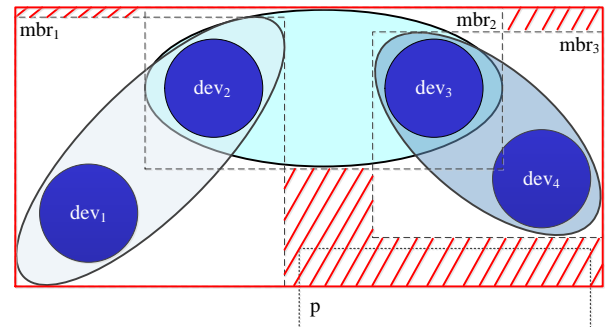


Figure 9: Less Coarse MBR Checks

the shaded parts, overlaps a POI  $p$ , and therefore  $o$  is included in  $p$ 's join list initially. Later,  $o$ 's complex uncertainty region  $UR(o)$  is derived to calculate its presence in  $p$ . It turns out that the complex calculation does not contribute to the query result because  $UR(o)$  does not intersect with  $p$ . Specifically, object  $o$ 's single, large MBR can be replaced by three smaller MBRs. As shown in the figure, each of the three smaller MBRs bounds the ellipse corresponding

to a pair of consecutive tracking records. Using the smaller MBRs, object  $o$  will be excluded from  $p$ 's join list, and the calculation of  $UR(o)$  will be skipped since none of the three smaller MBRs intersects with  $p$ . We only implement the improved framework in the experimental comparisons.

## 5. EXPERIMENTAL STUDIES

This section reports on the experimental studies of our research. Section 5.1 presents the experimental settings. Sections 5.2 and 5.3 cover the experiments on synthetic and real data, respectively.

### 5.1 Experimental Settings

All algorithms are implemented in Java and run on a computer with Windows 7 Enterprise edition, an Intel Core i7-2620M 2.70GHz CPU, and 8.0GB main memory.

**Synthetic data set:** We use a floor plan with about 100 rooms that are all connected by doors to a hallway. We place a total of 143 RFID readers by doors and along the hallways. We generate object movements using the random waypoint model [11]. All objects move with a fixed speed of 1.1 m/s, which is also used as the maximum speed  $V_{max}$ .

We vary multiple parameters when generating the data, as shown in Table 4 where default values are in bold. We vary  $|O|$ , i.e., the number of objects in the  $OTT$  from 10 K to 50 K. We vary the RFID detection range, the radius of the circular region covered by an RFID reader, from 1m to 2.5m. For the complete synthetic data set, the number of the  $OTT$  tracking records falls in the range from 140 K to 2,000 K.

Parameters	Settings
$ O $	10K, 20K, ..., <b>50K</b>
Detection Range (meter)	1, <b>1.5</b> , 2, 2.5
$ P $ (% of all indoor POIs)	2%, <b>4%</b> , 6%, 8%, 10%
$k$	1, 2, 3, 4, <b>5</b> , ..., 10, ... 15
$t_e - t_s$ (minutes)	10, 20, <b>30</b> , ..., 60

Table 4: Parameter Settings in Experiments

**Real-world data set:** We use a real data set collected from Copenhagen International Airport (CPH) where passengers with Bluetooth-enabled mobile phones are tracked by deployed Bluetooth radios. We extract the data for a period of 7 months with the most tracking records. Our  $OTT$  contains approximately 600K records for about 10K passengers. We do not vary the detection range in the experiments on the real data set because we have no access to change the configurations in the real deployment. We use the same  $V_{max}$  as in the synthetic data.

**Query parameters:** We also vary query parameters according to Table 4. Specifically,  $k$  (the number of top ranked indoor POIs to be returned) is varied from 1 to 15. The query POI set is determined as follows. For both synthetic and real data, 375 POIs are determined in the indoor space at distinctive locations and with different areas. Multiple POIs may come from the same large room that is divided into multiple uses. We control the number of query POIs ( $|P|$ ) as a percentage (2% to 10%) of the total number of indoor POIs. Given a percent, the query POI set is determined as a random subset of the total 375 POIs. We start from 2% to make sure there are sufficient query POIs for

returning the top- $k$  results. On the other hand, we do not include more than 10% of all POIs in a query because, intuitively, not all indoor POIs are interesting and frequently visited, so query issuers like building officers may query a subset of all indoor POIs. Furthermore,  $t_e - t_s$  is the query time interval used in the interval top- $k$  indoor POI query. We vary it from 10 to 60 minutes.

## 5.2 Experiments on Synthetic Data

### 5.2.1 Results for Snapshot Queries

We first evaluate the performance of the snapshot query by varying parameters  $k$ ,  $|P|$ , and the detection range. We do not change parameter  $|O|$  for the snapshot query because the number of moving objects retrieved at a given time point is fairly random, and is smaller than and independent of  $|O|$ .

The results of varying  $k$ ,  $|P|$ , and the detection range are presented in Figures 10 and 11(a). As a general observation, the join algorithm outperforms the iterative algorithm. This is because the former is able to prune more objects when their uncertainty regions do not overlap with the regions of POIs.

The effect of varying  $k$  is shown in Figure 10(a). As can be observed, varying  $k$  has only little influence on both algorithms. This indicates that both algorithms are stable with respect to query parameter  $k$ . The relatively high cost for both algorithms at  $k = 1$  is due to the intensive initial computation of the uncertainty regions for many objects, which, however, does not pay off for a simple top-1 query.

The effect of varying  $|P|$  is shown in Figure 10(b). As the number of query POIs increases, the running time increases slightly. This is because more query POIs occupy larger areas, and thus more moving objects are present in POIs. Thus, more object uncertainty regions intersect POIs, which yields longer processing times.

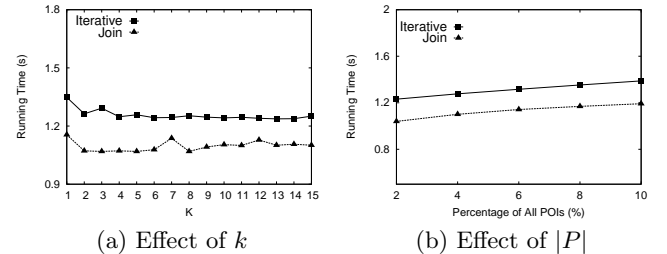


Figure 10: Snapshot Query on Synthetic Data Set

Figure 11(a) presents the effect of varying the detection range of RFID readers. When the detection range increases, the uncertainty region increases as well. Therefore, more computation time is needed to estimate the areas of uncertainty regions. Hence, the running time is the largest when the detection range is set to 2.5m. The slight decrease at 2 meters is attributed to the fluctuation of the  $OTT$  size.

### 5.2.2 Results for Interval Queries

The performance of the interval query algorithms is reported in Figures 11(b) and 12. In this experiment, we vary all five parameters listed in Table 4. Similar to the experiments for snapshot queries, the join algorithm runs faster than the iterative algorithm in almost all settings, which we attribute to its effective pruning strategy.

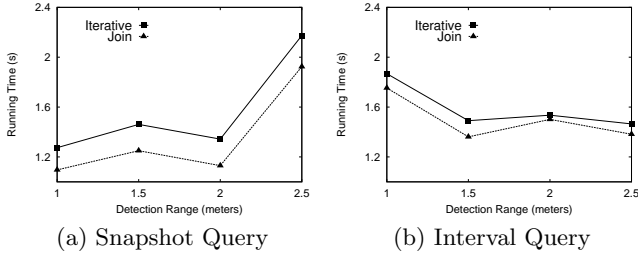


Figure 11: Effect of Detection Range

As shown in Figure 12(a), varying  $k$  does not significantly impact the performance of the algorithms except for  $k = 1$ . Overall, both algorithms are stable with respect to query result size except when  $k = 1$ . The longer running time for  $k = 1$  occurs because the considerable computations on the uncertainty regions do not pay off for the very small top-1 query results. The performance improves when  $k$  increases because neither algorithm works in an incremental manner with respect to  $k$ . The relatively high cost at  $k = 1$  is not observed in the counterpart experiments on the real data set (see Figure 14(a)), which is attributed to the considerably smaller size of the real data.

The effect of varying  $|P|$  is presented in Figure 12(b). When increasing  $|P|$ , the running time of the iterative algorithm increases, while the join algorithm stays stable. Both algorithms have longer running times when compared to their snapshot query counterparts. This is understandable, as uncertainty regions in the snapshot setting are much smaller than those in the interval setting.

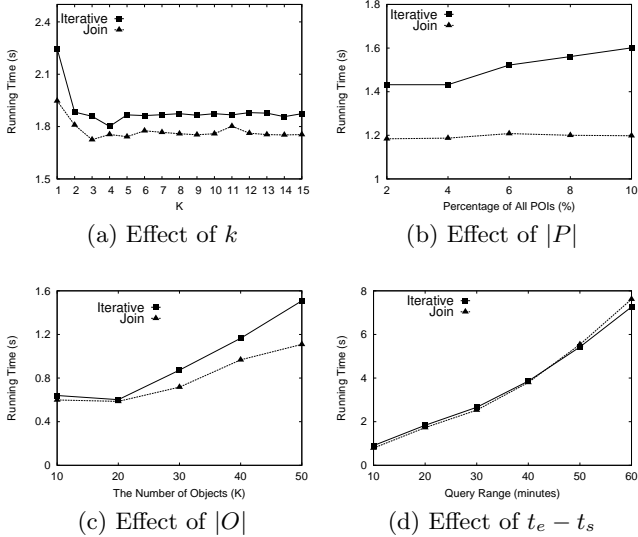


Figure 12: Interval Query on Synthetic Data Set

Figure 12(c) shows that the running time of both algorithms increases as  $|O|$  increases. Nevertheless, the join algorithm remains more efficient. This indicates that the join algorithm is most scalable.

As the query time interval  $t_e - t_s$  increases, the running time of the two algorithms increases accordingly, as presented in Figure 12(d). Longer query intervals tend to yield larger and more complex uncertainty regions that require more time to process.

As presented in Figure 11(b), when the detection range increases, the running time of both algorithms tends to decrease. This trend is opposite to the observation in the experiment for snapshot queries where the detection range is changed. In the interval setting, the algorithms compute on moving objects trajectories in a given time interval and thus involve a series of tracking records and devices. When the detection ranges of the devices increase, an object's uncertainty regions between pairs of consecutive devices shrink. As a result, objects' overall uncertainty regions tend to be smaller and thus take less time to process. At the outlier of 2 meters, the  $OTT$  is small, and the tracking data is sparse, which offsets the benefit of a larger detection range.

### 5.3 Experiments on Real Data

Results for snapshot queries are reported in Figure 13. Clearly the join algorithm outperforms the iterative algorithm. Both algorithms are fairly stable with respect to varying  $k$ , as shown in Figure 13(a). When the number of query POIs is increased, both algorithms' running times increase moderately and almost linearly, as shown in Figure 13(b). These results indicate that our designs are stable and scalable for snapshot queries in real indoor settings.

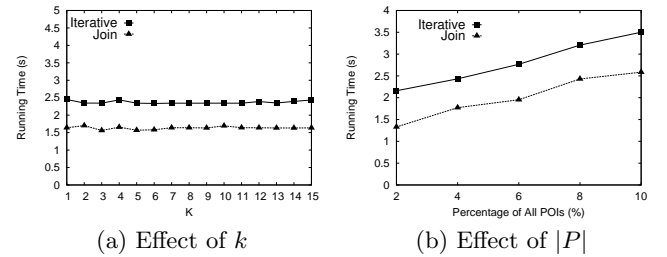


Figure 13: Snapshot Query on CPH Data Set

Interval query performance results are reported in Figure 14. It is clear that the join algorithm outperforms the iterative algorithm for all settings in these experiments.

The effect of varying  $k$  is shown in Figure 14(a). The join algorithm is more efficient and slightly more stable when varying  $k$ . This indicates that our strategy of finer MBRs for interval uncertainty regions pays off.

The effect of varying the number of query POIs is shown in Figure 14(b). The more stable performance of the join algorithm is again attributed to the use of finer MBRs for interval uncertainty regions. Smaller MBRs can help prune objects (and their uncertainty regions) more effectively even when there are more POIs and those POIs collectively occupy larger areas.

The effect of varying the query interval length is shown in Figure 14(c). Longer intervals yield longer object trajectories and larger uncertainty regions. This explains the increase in the running times of both algorithms. Nevertheless, the join algorithm is still more efficient thanks to the use of finer MBRs.

## 6. RELATED WORK

We briefly review related work in this section. Section 6.1 covers the research on indoor-space moving objects, and Section 6.2 covers the density queries in outdoor spaces.

### 6.1 Indoor-Space Moving Objects

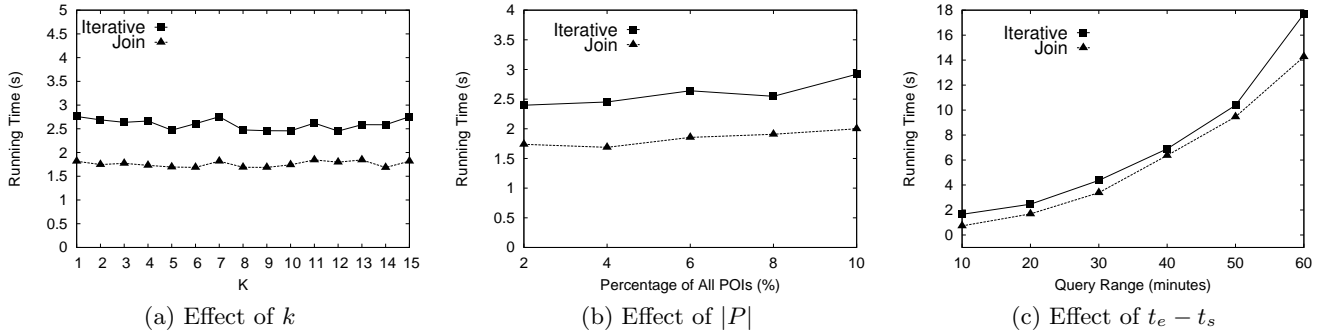


Figure 14: Interval Query on CPH Data Set

Due in part to their different topology, indoor spaces are modeled [10, 13, 16, 21, 22] differently than outdoor spaces. Moreover, indoor moving objects are tracked by means different than outdoor GPS and thus generate different tracking data.

Assuming a generic setting of symbolic indoor tracking, Yang et al. study continuous range monitoring queries [25] and probabilistic  $k$  nearest neighbor queries [26] on indoor moving objects. Uncertain query results are returned as objects' locations are unknown when they are outside any detection range. Yu et al. [28] propose a particle filter based method to infer undetected locations under RFID tracking, which thus improves query result quality. Unlike these works that concern the current locations of indoor moving objects, this paper works on historical indoor tracking data.

Lu et al. [17] define spatio-temporal joins over indoor moving objects whose historical locations are captured in the same format as the symbolic tracking data assumed in this paper. However, while we compute flows for static indoor POIs according to historical tracking data, Lu et al. compute pairs of indoor moving objects that were in the same location according to historical data.

Xie et al. [23,24] study indoor distance-aware spatial queries over online indoor moving objects whose positions are reported as probabilistic samples rather than using RFID detection ranges. Therefore, the proposed techniques are unsuitable for the problems we consider in this paper.

Recently, Ahmed et al. [2] define indoor density queries based on historical RFID indoor tracking data. Their density definition differs from our definition of flow. Moreover, we analyze the inherent uncertainties in the data and design uncertainty-aware solutions, whereas Ahmed et al. do not consider uncertainty.

## 6.2 Density Queries in Outdoor Spaces

Most existing research works on the querying of object flow and density are for outdoor Euclidean spaces or spatial road networks.

Tao et al. [20] use an aggregate RB-tree (aRB-tree) for indexing spatio-temporal objects in a Euclidean space and propose algorithms to count objects in a given spatial window during a given time interval. The proposed solution considers no location uncertainty and the aRB-tree cannot handle the complex indoor topology. Therefore, that solution cannot be used for the problems addressed in this paper.

Employing micro-clustering [29], Li et al. [15] propose techniques that cluster moving objects and dynamically up-

date the clusters as the objects move linearly. The proposed techniques do not apply in our setting, where indoor object movements can not be captured by linear models but are reported based on discrete detection ranges. Yiu and Mamoulis [27] propose density based and hierarchical methods to partition and cluster static objects on a spatial network. Their proposal uses network distances between static positions and therefore does not solve our problems on indoor moving objects.

Hadjieleftheriou et al. [5] study threshold density region queries that find outdoor regions with more objects than a given threshold. Their solution assumes that the objects move according to known linear functions and are indexed by uniform space-time grids. Jensen et al. [9] study snapshot dense region queries in a Euclidean plane where objects move linearly and are indexed by a  $B^x$ -tree [8]. With the same linear motion assumption, Hao et al. [6] study continuous density queries by using a quad-tree to index moving objects. To improve density query results, Ni and Ravishankar [18] redefine the density and use small square neighborhoods to approximate arbitrary outdoor regions. These proposals [5, 6, 9, 18] are unsuitable for our problems, a key reason being that objects in indoor spaces cannot be modeled well by linear functions.

Huang and Lu [7] define online density region queries on moving objects that are observed by sensors at fixed positions in a geographical area. The proposed solution assumes that object locations are captured as certain points in a 2D Euclidean space. This assumption renders the solution unsuitable for our problems where indoor moving object locations are captured as uncertainty regions.

Li et al. [14] devise techniques to return traffic density-based hot routes from historical trajectories in a road network. Cai et al. [12] design a clustering based technique for continuously monitoring dense road segments. Different from these works, we target indoor spaces where objects' historical movements are captured as detection ranges associated with time intervals.

## 7. CONCLUSION AND FUTURE WORK

In this paper, we study how to find the top- $k$  frequently visited indoor Points of Interest (POIs) using symbolic indoor tracking data that captures object movements indoors. We define two types of queries in this regard. A snapshot query finds those indoor POIs that were visited by the most tracked objects (e.g., people) at a given time point, whereas an interval query finds such POIs for a given time interval. As symbolic indoor tracking data can only capture trajec-

tories with a considerable degree of uncertainty, we define appropriate ways to quantify how frequently an indoor POI is visited by probabilistically counting objects' presences in the POI. Subsequently, we conduct a detailed analysis on the uncertainty regions of objects in the settings of the two types of queries. Based on the uncertainty analysis, we design algorithms for both query types. We use both synthetic and real data sets to evaluate the query processing algorithms, and the performance results show that our proposed join-based algorithms are capable of significantly outperforming straightforward baselines and are much more scalable in terms of data set size and query interval length.

Several directions exist for future research. First, it is relevant to consider an object's dwell time when calculating its interval presence in a POI. The interval-related definitions in the paper can be extended for that purpose. In particular, an object's interval uncertainty region can be extended to also reflect the temporal aspect in addition to the spatial aspect. Second, it is of interest to extend the uncertainty analysis to support multiple floors. The new challenge is to track object movement between floors appropriately and to derive the uncertainty regions accordingly. Third, it is of interest to investigate how to solve the problems addressed in the paper using other types of indoor tracking data. To this end, it can be considered whether the proposed techniques can be applied to, or adapted for, other data types. Fourth, it is relevant to develop techniques for finding the currently crowded indoor POIs by using tracking data. Fifth, it is of interest to evaluate the query results against real indoor POIs in order to see how effective the proposed query types are at finding frequently visited indoor POIs. For that purpose, ground truth data on popular indoor POIs is needed.

## Acknowledgments

The work is partly supported by the NILTEK project, funded by European Regional Development Fund.

## 8. REFERENCES

- [1] InLocation Alliance. <http://www.in-location-alliance.com/>, 2016 (accessed January 2016).
- [2] T. Ahmed, T. B. Pedersen, and H. Lu. Finding dense locations in indoor tracking data. In *MDM*, pages 189–194, 2014.
- [3] X. Cao, G. Cong, and C. S. Jensen. Mining significant semantic locations from GPS data. *PVLDB*, 3(1):1009–1020, 2010.
- [4] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.
- [5] M. Hadjieleftheriou, G. Kollios, D. Gunopulos, and V. J. Tsotras. On-line discovery of dense areas in spatio-temporal databases. In *SSTD*, pages 306–324, 2003.
- [6] X. Hao, X. Meng, and J. Xu. Continuous density queries for moving objects. In *MobiDE*, pages 1–7, 2008.
- [7] X. Huang and H. Lu. Snapshot density queries on location sensors. In *MobiDE*, pages 75–78, 2007.
- [8] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B<sup>+</sup>-tree based indexing of moving objects. In *VLDB*, pages 768–779, 2004.
- [9] C. S. Jensen, D. Lin, B. C. Ooi, and R. Zhang. Effective density queries on continuously moving objects. In *ICDE*, page 71, 2006.
- [10] C. S. Jensen, H. Lu, and B. Yang. Graph model based indoor tracking. In *MDM*, pages 122–131, 2009.
- [11] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, pages 153–181. Kluwer Academic Publishers, 1996.
- [12] C. Lai, L. Wang, J. Chen, X. Meng, and K. Zeitouni. Effective density queries for moving objects in road networks. In *WAIM*, pages 200–211, 2007.
- [13] J. Lee. A spatial access-oriented implementation of a 3-D GIS topological data model for urban entities. *GeoInformatica*, 8(3):237–264, 2004.
- [14] X. Li, J. Han, J. Lee, and H. Gonzalez. Traffic density-based discovery of hot routes in road networks. In *SSTD*, pages 441–459, 2007.
- [15] Y. Li, J. Han, and J. Yang. Clustering moving objects. In *SIGKDD*, pages 617–622, 2004.
- [16] H. Lu, X. Cao, and C. S. Jensen. A foundation for efficient indoor distance-aware query processing. In *ICDE*, pages 438–449, 2012.
- [17] H. Lu, B. Yang, and C. S. Jensen. Spatio-temporal joins on symbolic indoor tracking data. In *ICDE*, pages 816–827, 2011.
- [18] J. Ni and C. V. Ravishankar. Pointwise-dense region queries in spatio-temporal databases. In *ICDE*, pages 1066–1075, 2007.
- [19] D. Pfoser and C. S. Jensen. Capturing the uncertainty of moving-object representations. In *SSD*, pages 111–132, 1999.
- [20] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-temporal aggregation using sketches. In *ICDE*, pages 214–225, 2004.
- [21] E. Whiting, J. Battat, and S. Teller. Topology of Urban Environments. *CAAD Futures*, pages 114–128, 2007.
- [22] M. F. Worboys. Modeling indoor space. In *ISA*, pages 1–6, 2011.
- [23] X. Xie, H. Lu, and T. B. Pedersen. Efficient distance-aware query evaluation on indoor moving objects. In *ICDE*, pages 434–445, 2013.
- [24] X. Xie, H. Lu, and T. B. Pedersen. Distance-aware join for indoor moving objects. *IEEE Trans. Knowl. Data Eng.*, 27(2):428–442, 2015.
- [25] B. Yang, H. Lu, and C. S. Jensen. Scalable continuous range monitoring of moving objects in symbolic indoor space. In *CIKM*, pages 671–680, 2009.
- [26] B. Yang, H. Lu, and C. S. Jensen. Probabilistic threshold k nearest neighbor queries over moving objects in symbolic indoor space. In *EDBT*, pages 335–346, 2010.
- [27] M. L. Yiu and N. Mamoulis. Clustering objects on a spatial network. In *SIGMOD*, pages 443–454, 2004.
- [28] J. Yu, W. Ku, M. Sun, and H. Lu. An RFID and particle filter-based indoor spatial query evaluation system. In *EDBT*, pages 263–274, 2013.
- [29] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. In *SIGMOD*, pages 103–114, 1996.



# Designing Access Methods: The RUM Conjecture

Manos Athanassoulis\* Michael S. Kester\* Lukas M. Maas\* Radu Stoica†

Stratos Idreos\* Anastasia Ailamaki‡ Mark Callaghan◊

\*Harvard University †IBM Research, Zurich ‡EPFL, Lausanne ◊Facebook, Inc.

## ABSTRACT

The database research community has been building methods to store, access, and update data for more than four decades. Throughout the evolution of the structures and techniques used to access data, *access methods* adapt to the ever changing hardware and workload requirements. Today, even small changes in the workload or the hardware lead to a redesign of access methods. The need for new designs has been increasing as data generation and workload diversification grow exponentially, and hardware advances introduce increased complexity. New workload requirements are introduced by the emergence of new applications, and data is managed by large systems composed of more and more complex and heterogeneous hardware. As a result, it is increasingly important to develop application-aware and hardware-aware access methods.

The fundamental challenges that every researcher, systems architect, or designer faces when designing a new access method are how to minimize, i) read times (R), ii) update cost (U), and iii) memory (or storage) overhead (M). In this paper, we conjecture that when optimizing the read-update-memory overheads, optimizing in any two areas negatively impacts the third. We present a simple model of the RUM overheads, and we articulate the *RUM Conjecture*. We show how the RUM Conjecture manifests in state-of-the-art access methods, and we envision a trend toward RUM-aware access methods for future data systems.

## 1. INTRODUCTION

**Chasing Access Paths.** Picking the proper physical design (through static autotuning [14], online tuning [13], or adaptively [31]) and access method [27, 49] have been key research challenges of data management systems for several decades. The way we physically organize data on storage devices (disk, flash, memory, caches) defines and restricts the possible ways that we can read and update it. For example, when data is stored in a heap file without an index, we have to perform costly scans to locate any data we are interested in. Conversely, a tree index on top of the heap file, uses additional space in order to substitute the scan with a more lightweight index probe. Over the years, we have seen a plethora of exciting and innovative proposals for data structures and algorithms, each

one tailored to a set of important workload patterns, or for matching critical hardware characteristics. Applications evolve rapidly and continuously, and at the same time, the underlying hardware is diverse and changes quickly as new technologies and architectures are developed [1]. Both trends lead to new challenges when designing data management software.

**The RUM Tradeoff.** A close look at existing proposals on access methods<sup>1</sup> reveals that each is confronted with the same fundamental challenges and design decisions again and again. In particular, there are three quantities and design parameters that researchers always try to minimize: (1) the read overhead (R), (2) the update overhead (U), and (3) the memory (or storage) overhead (M), henceforth called the *RUM overheads*. Deciding which overhead(s) to optimize for and to what extent, remains a prominent part of the process of designing a new access method, especially as hardware and workloads change over time. For example, in the 1970s one of the critical aspects of every database algorithm was to minimize the number of random accesses on disk; fast-forward 40 years and a similar strategy is still used, only now we minimize the number of random accesses to main memory. Today, different hardware runs different applications but the concepts and design choices remain the same. New challenges, however, arise from the exponential growth in the amount of data generated and processed, and the wealth of emerging data-driven applications, both of which stress existing data access methods.

**The RUM Conjecture: Read, Update, Memory – Optimize Two at the Expense of the Third.** An ideal solution is an access method that always provides the lowest read cost, the lowest update cost, and requires no extra memory or storage space over the base data. In practice, data structures are designed to compromise between the three RUM overheads, while the optimal design depends on a multitude of factors like hardware, workload, and user expectations.

We analyze the lower bounds for the three overheads (read - update - memory) given an access method which is perfectly tailored for minimizing each overhead and we show that such an access method will impact the rest of the overheads negatively. We take this observation a step further and propose the RUM Conjecture: *designing access methods that set an upper bound for two of the RUM overheads, leads to a hard lower bound for the third overhead which cannot be further reduced.* For example, in order to minimize the cost of updating data, one would use a design based on differential structures, allowing many queries to consolidate updates and avoid the cost of reorganizing data. Such an approach, however, increases the space overhead and hinders read cost as now queries need to merge any relevant pending updates during processing. Another example is that the read cost can be minimized by

<sup>1</sup>Access methods: *algorithms and data structures for organizing and accessing data* [27].

storing data in multiple different physical layouts [4, 17, 46], each layout being appropriate for minimizing the read cost for a particular workload. Update and space costs, however, increase because now there are multiple data copies. Finally, the space cost can be minimized by storing minimal metadata and, hence, pay the price of increased search time when reading and updating data.

The three RUM overheads form a competing triangle. In modern implementations of data systems, however, one can optimize up to some point for all three. Such optimizations are possible by relying on using inherently defined structure instead of storing detailed information. A prime example is the block-based clustered indexing, which reduces both read and memory overhead by storing only a few pointers to pages (only when an indexed tuple is stored to a different page than the previous one), hence building a very short tree. The reason why such an approach would give us good read performance is the fact that data is clustered on the index attribute. Even in this ideal case, however, we have to perform additional computation in order to calculate the exact location of the tuple we are searching for. In essence, we use computation and knowledge about the data in order to reduce the RUM overheads. Another example, is the use of compression in bitmap indexes; we still use additional computation (compression/decompression) in order to succeed in reducing both read and memory overhead of the indexes. While the RUM overheads can be reduced by computation or engineering, their competing nature manifests in a number of approaches and guides our roadmap for RUM-aware access methods.

**RUM-Aware Access Method Design.** Accepting that a perfect access method does not exist, does not mean the research community should stop striving to improve; quite the opposite. The RUM Conjecture opens the path for exciting research challenges towards the goal of creating RUM-aware and RUM-adaptive access methods.

Future data systems should include versatile tools to interact with the data the way the workload, the application, and the hardware need and not vice versa. In other words, the application, the workload, and the hardware should dictate how we access our data, and not the constraints of our systems. Such versatile data systems will allow the data scientist of the future to dynamically tune the data access methods during normal system operation. Tuning access methods becomes increasingly important if, on top of big data and hardware, we consider the development of specialized systems and tools to cater data, aiming at servicing a narrow set of applications each. As more systems are built, the complexity of finding the right access method increases as well.

**Contributions.** In this paper we present for the first time the RUM Conjecture as a way to understand the inherent tradeoffs of every access method. We further use the conjecture as a guideline for designing access methods of future data systems. In the remainder of this paper we charter the path toward RUM-aware access methods:

- Identify and model the RUM overheads (§2).
- Propose the RUM Conjecture (§3).
- Document the manifestation of the RUM Conjecture in state-of-the-art access methods (§4).
- Use the RUM Conjecture to build access methods for future data systems (§5).

Each of the above points serves as inspiration for further research on data management. Finding a concise yet expressive way to identify and model the fundamental overheads of access methods requires research in data management from a theory perspective. Similarly, proving the RUM Conjecture will expand on this line of research. Documenting the manifestation of the RUM Conjecture entails a new study of access methods with the RUM overheads

in mind when modeling and categorizing access methods. Finally, the first three steps provide the necessary research infrastructure to build powerful access methods.

While we envision that this line of research will enable building powerful access methods, the core concept of the RUM Conjecture is, in fact, that there is no panacea when designing systems. It is not feasible to build the universally best access method, nor to build the best access method for each and every use case. Instead, we envision that the RUM Conjecture will create a trend toward building access methods that can efficiently morph to support changing requirements and different software and hardware environments. The remainder of this paper elaborates one by one the four steps toward RUM-aware access methods for future data systems.

## 2. THE RUM OVERHEADS

**Overheads of Access Methods.** When designing access methods it is very important to understand the implications of different design choices. In order to do so, we discuss the three fundamental overheads that each design decision can affect. Access methods enable us to read or update the main data stored in a data management system (hereafter called *base data*), potentially using auxiliary data such as indexes (hereafter called *auxiliary data*), in order to offer performance improvements. The overheads of an access method quantify the additional data accesses to support any operation, relative to the base data.

**Read Overhead (*RO*).** The *RO* of an access method is given by the data accesses to auxiliary data when retrieving the necessary base data. We refer to *RO* as the read amplification: the ratio between the total amount of data read including auxiliary and base data, divided by the amount of retrieved data. For example, when traversing a  $B^+$ -Tree to access a tuple, the *RO* is given by the ratio between the total data accessed (including the data read to traverse the tree and the base data) and the base data intended to be read.

**Update Overhead (*UO*).** The *UO* is given by the amount of updates applied to the auxiliary data in addition to the updates to the main data. We refer to *UO* as the write amplification: the ratio between the size of the physical updates performed for one logical update, divided by the size of the logical update. In the previous example, the *UO* is calculated by dividing the updated data size (both base and auxiliary data) by the size of the updated base data.

**Memory Overhead (*MO*).** The *MO* is the space overhead induced by storing auxiliary data. We refer to *MO* as the space amplification, defined as the ratio between the space utilized for auxiliary and base data, divided by the space utilized for base data. Following the preceding example, the *MO* is computed by dividing the overall size of the  $B^+$ -Tree by the base data size.

**Minimizing RUM overheads.** Ideally, when building an access method, all three overheads should be minimal, however, depending on the application, the workload, and the available technology they are prioritized. While access time, optimized by minimizing read overhead, often has top priority, the workload or the underlying technology sometimes shift priorities. For example, storage with limited endurance (like flash-based drives) favors minimizing the update overhead, while the slow speed of main memory and the scarce cache capacity justifies reducing the space overhead. The theoretical minimum for each overhead is to have the ratio equal to 1.0, implying that the base data is always read and updated directly and no extra bit of memory is wasted. Achieving these bounds for all three overheads simultaneously, however, is not possible as there is always a price to pay for every optimization.

In the following discussion we reason about the three overheads and their lower bounds using a simple yet representative case for

base data: an array of integers. We organize this dataset consisting of  $N$  ( $N \gg 1$ ) fixed-sized elements in blocks, each one holding a *value*. Every block can be identified by a monotonically increasing ID, *blkID*. The workload is comprised of point queries, updates, inserts, and deletes. We purposefully provide simple examples of data structures in order to back the following hypothesis. The generality of these examples lies in their simplicity.

**Hypothesis.** *An access method that is optimal with respect to one of the read, update, and memory overheads, cannot achieve the optimal value for both remaining overheads.*

**Minimizing Only RO.** In order to minimize *RO* we organize data in an array and we store each *value* in the block with  $blkID = value$ . For example, the relation  $\{1, 17\}$  is stored in an array with 17 blocks. The first block holds value 1, the last block holds value 17, and every block in-between holds a null value. *RO* is now minimal because we always know where to find a specific value (if it exists), and we only read useful data. On the other hand, the array is sparsely populated, with unbounded *MO* since, in the general case, we cannot anticipate what would be the maximum value ever inserted. When we insert or delete a value we only update the corresponding block. When we change a value we need to update two blocks: empty the old block and insert the new value in its new block, effectively, increasing the worst case *UO* to two physical updates for one logical update.

**Prop. 1**  $min(RO) = 1.0 \Rightarrow UO = 2.0$  and  $MO \rightarrow \infty$

**Minimizing Only UO.** In order to minimize *UO*, we append every update, effectively forming an ever increasing log. That way we achieve the minimum *UO*, which is equal to 1.0, at the cost of continuously increasing *RO* and *MO*. Notice that any reorganization of the data to reduce *RO* would result in an increase of *UO*. Hence, for minimum *UO*, both *RO* and *MO* perpetually increase as updates are appended.

**Prop. 2**  $min(UO) = 1.0 \Rightarrow RO \rightarrow \infty$  and  $MO \rightarrow \infty$

**Minimizing Only MO.** When minimizing *MO*, no auxiliary data is stored and the base data is stored as a dense array. During a selection, we need to scan all data to find the values we are interested in, while updates are performed in place. The minimum  $MO=1.0$  is achieved. The *RO*, however, is now dictated by the size of the relation since a full scan is needed in the worst case. The *UO* cost of in-place updates is also optimal because only the base data intended to be updated is ever updated.

**Prop. 3**  $min(MO) = 1.0 \Rightarrow RO = N$  and  $UO = 1.0$

### 3. THE RUM CONJECTURE

Achieving the optimal value for one overhead is not always as important as finding a good balance across all RUM overheads. In fact, in the previous section, we saw that striving for the optimal may create impractical access method designs. The next logical step is to provide a fixed bound only for one of the overheads, however, this raises the question of how to quantify the impact of such an optimization goal for the remaining two overheads. Given our analysis above, we conjecture that it is not possible to optimize across all three dimensions at the same time.

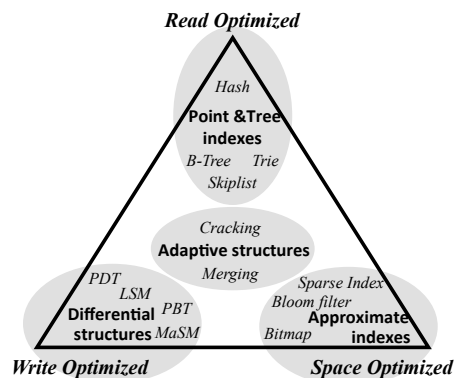
**The RUM Conjecture.** *An access method that can set an upper bound for two out of the read, update, and memory overheads, also sets a lower bound for the third overhead.*

In other words, we can choose which two overheads to prioritize and optimize for, and pay the price by having the third overhead

greater than a hard lower bound. In the following section we describe how the RUM Conjecture manifests in existing access methods by showing that current solutions are typically optimized for one of the three overheads, while in Section 5 we discuss a roadmap for RUM access methods.

### 4. RUM IN PRACTICE

The RUM Conjecture captures in a concise way the tension between different optimization goals that researchers face when building access methods. We present here the competing nature of the RUM tradeoffs, as it manifests in state-of-the-art access methods. Different access method designs can be visually represented based on their RUM balance. The RUM tradeoffs can be seen as a three dimensional design space or, if projected on a two-dimensional plane, as the triangle shown in Figure 1. Each access method is mapped to a point or – if it can be tuned to have varying RUM behavior – to an area. Table 1 shows the time and size complexity of representative data structures, illustrating the conflicting read, update, and memory overheads.



**Figure 1: Popular data structures in the RUM space.**

*Read-optimized access methods* (top corner in Figure 1) are optimized for low read overhead. Examples include indexes with constant time access such as hash-based indexes or logarithmic time structures such as *B-Trees* [22], *Tries* [19], *Prefix B-Trees* [9], and *Skiplists* [45]. Typically, such access methods offer fast read access but increase space overhead and suffer with frequent updates.

*Write-optimized differential structures* (left corner in Figure 1) reduce the write overhead of in-place updates by storing updates in secondary differential data structures. The fundamental idea is to consolidate updates and apply them in bulk to the base data. Examples include the *Log-structured Merge Tree* [44], the *Partitioned B-tree* (PBT) [21], the *Materialized Sort-Merge* (MaSM) algorithm [7, 8], the *Stepped Merge* algorithm [35], and the *Positional Differential Tree* [28]. *LA-Tree* [3] and *FD-Tree* [38] are two prime examples of write optimized trees aiming at exploiting flash while respecting at the same time its limitations, e.g., the asymmetry between read and write performance [6] and the bounded number of physical updates flash can sustain [7]. Typically, such data structures offer good performance under updates but increase read costs and space overhead.

*Space-efficient access methods* (right corner in Figure 1) are designed to reduce the storage overhead. Example categories include compression techniques and lossy index structures such as *Bloom filters* [12], lossy hash-based indexes like *count-min sketches* [16], *bitmaps with lossy encoding* [51], and *approximate tree indexing* [5, 40]. *Sparse indexes*, which are light-weight secondary indexes, like *ZoneMaps* [18], *Small Materialized Aggregates* [42] and *Col-*

Parameter	N	m	B	P	T	MEM
Explanation	dataset size (#tuples)	query result size (#tuples)	block size (#tuples)	partition size (#tuples)	LSM levels ratio	memory (#pages)

Access Method	Bulk Creation Cost	Index Size	Point Query	Range Query (size: $m$ )	Insert/Update/Delete
$B^+$ -Tree	$O(N/B \cdot \log_{MEM/B}(N/B))$	$O(N/B)$	$O(\log_B(N))$	$O(\log_B(N) + m)$	$O(\log_B(N))$
Perfect Hash Index	$O(N)$	$O(N/B)$	$O(1)$	$O(N/B)$	$O(1)$
ZoneMaps	$O(N/B)$	$O(N/P/B)$	$O(N/P/B)$	$O(N/P/B)$	$O(N/P/B)$
Levelled LSM	N/A	$O(\frac{N \cdot T}{T-1})$	$O(\log_T(N/B) \cdot \log_B(N))$	$O(\log_T(N/B) \cdot \log_B(N) + \frac{m \cdot T}{T-1})$	$O(T/B \cdot \log_T(N/B))$
Sorted column	$O(N/B \cdot \log_{MEM/B}(N/B))$	$O(1)$	$O(\log_2(N))$	$O(\log_2(N) + m)$	$O(N/B/2)$
Unsorted column	$O(1)$	$O(1)$	$O(N/B/2)$	$O(N/B)$	$O(1)$

**Table 1: The base data typically exist either as a *sorted column* or as an *unsorted column*. When using an additional index we (i) spend time building it, (ii) allocate space for it, and (iii) pay the cost to maintain it. The I/O cost [2] (time and space complexity) of four representative access methods ( $B^+$ -Trees, Hash Indexes, ZoneMaps, and levelled LSM) illustrates that there is no single winner. ZoneMaps have the smaller size – being a sparse index, but Hash Indexes offer the fastest point queries, while  $B^+$ -Trees offer the fastest range queries. Similarly, the update cost is best for Hash Indexes, while LSM can support efficient range queries having very low update cost as well.**

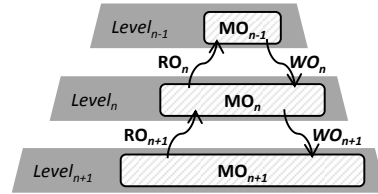
*umn Imprints* [50] fall into the same category. Typically, such data structures and approaches reduce the space overhead significantly but increase the write costs (e.g., when using compression) and sometimes increase the read costs as well (e.g., a sparse index).

*Adaptive access methods* (middle region in Figure 1) are comprised of flexible data structures designed to gradually balance the RUM tradeoffs by using the workload access pattern as a guide. Most existing data structures provide tunable parameters that can be used to balance the RUM tradeoffs offline, however, adaptive access methods balance the tradeoffs online across a larger area of the design space. Notable proposals are *Database Cracking* [31, 32, 33, 48], *Adaptive Merging* [22, 25], and *Adaptive Indexing* [23, 24, 26, 34], which balance the read performance versus the overhead of creating an index. The incoming queries dictate which part of the index should be fully populated and tuned. The index creation overhead is amortized over a period of time, and it gradually reduces the read overhead, while increasing the update overhead, and slowly increasing the memory overhead. Although much more flexible than traditional data structures, existing adaptive data structures cannot cover the whole RUM spectrum as they are designed for a particular type of hardware and application.

The RUM tradeoffs of four representative access methods as well as two data organizations are presented in Table 1 in order to illustrate the need to balance them. The memory overhead is represented in the form of space complexity (index size), and the read and the update overheads in the form of the I/O complexity of the operations. We differentiate between point queries and range queries, to allow for a more detailed classification of access methods. We examine  $B^+$ -Trees<sup>2</sup>, Hash Indexing, ZoneMaps<sup>3</sup>, and levelled LSM [36], assuming that each LSM level is a  $B^+$ -Tree with branch factor  $B$ . Typically, sparse indexing like ZoneMaps gives the lowest access method size, however, it delivers neither the best point query performance, nor the best range query performance. The lowest point query complexity is provided by Hash Indexing, and the lowest range query complexity by  $B^+$ -Trees. In addition, even without any additional secondary index, maintaining a sorted column allows for searching with logarithmic cost, with the downside of having linear update cost. Hence, even without an auxiliary data structure, adding structure to the data affects read and write be-

<sup>2</sup>Bulk loading requires sorting. The best sorting algorithm depends on type of storage. Here we assume external multi-way mergesort.

<sup>3</sup>We consider the best case for ZoneMaps; only a single partition needs to be read or updated.



**Figure 2: RUM overheads in memory hierarchies.**

havior. We envision RUM access methods to take this a step further, and morph between data structures and different data organizations in order to build access methods that have tunable performance and can change behavior both adaptively and on-demand.

**The Memory Hierarchy.** For ease of presentation, the previous discussion assumes that all data objects are stored on the same storage medium. Real systems, however, have a more complex memory/storage hierarchy making it harder to design and tune access methods. Data is stored persistently only at the lower levels of the hierarchy and is replicated, in various forms, across all levels of the hierarchy; each memory level experiences different access patterns, resulting in different read and write overheads, while the space overhead at each level depends on how much data is cached.

Several approaches leverage knowledge about the memory hierarchy to offer better read performance. *Fractal Prefetching  $B^+$ -Trees* [15] use different node sizes for disk-based and in-memory processing in order to have the optimal for both cases. *Cache-sensitive  $B^+$ -Trees* [47] physically cluster sibling nodes together to reduce the number of cache misses, and decrease the node size using offsets rather than pointers. *SB-Trees* [43] operate in an analogous way when the index is disk-based, while *BW-Tree* [37] and *Masstree* [41] presents a number of optimizations related to cache memory, main memory and flash-based secondary storage. *SILT* [39] combines write-optimized logging, read-optimized immutable hashing, and, a sorted store, careful designed around the memory hierarchy to balance the tradeoffs of its various levels.

The RUM tradeoffs, however, still hold for each level individually as shown in Figure 2. The fundamental assumption that data has a minimum access granularity holds for all storage mediums today, including main memory, flash storage, and disks; the only difference is that both access time and access granularity vary. The RUM tradeoffs can also be viewed vertically rather than horizontally. For example, the  $RO_n$  read and the  $WO_n$  update overheads at memory level  $n$  can be reduced by storing more data, updates, or meta-data, at the previous level  $n - 1$ , which results, at least, in a

higher  $MO_{n-1}$ . Overall, we expect this interaction of hardware and software to become increasingly more complex as hierarchies become deeper and as hardware trends shift the relative performance of one level compared to the others, resulting in the need for more fine tuning of access methods.

Viewing the hierarchy from the bottom towards the top, we first have cold storage which today may be shingled disks [29], or traditional rotational disks. They are followed typically by flash storage for buffering and read performance. The next levels are main memory and the different levels of cache memory. In the future an additional layer of non-volatile main memory will be added or it will replace main memory altogether [30]. Different layers of this new storage and memory hierarchy have different requirements. More specifically, shingled disks are similar to flash devices regarding the need to minimize update cost to respect their internal characteristics. On the other hand, when designing access methods for traditional rotational disks – sitting in-between shingled disks and flash in the hierarchy – we need to minimize the read overhead. As we move higher in the hierarchy, read performance and index size is typically more important than update cost.

**Cache-Oblivious Access Methods.** A different way to build access methods is to completely remove the memory hierarchy from the design space, using cache-oblivious algorithms [20]. Cache-oblivious access methods, however, achieve that by having a larger constant factor in read performance [11]. In addition, cache-oblivious access methods have a larger memory overhead because they require more pointers to guarantee that search performance will be orthogonal to the memory hierarchy [10]. Finally, cache-oblivious designs are less tunable. In order to tune a data structure and be able to balance between read performance, update performance, and memory consumption we need to be cache-aware [10]. As a result, in order to build RUM-tunable access methods we have to use a cache-aware design and take into account the exact shape of the memory hierarchy.

## 5. BUILDING RUM ACCESS METHODS

For several decades, the database research community has been redesigning access methods, trying to balance the RUM tradeoffs with every change in hardware, workload patterns, and applications. As a result, every few years new variations of data management techniques are proposed and adaptation to new challenges becomes increasingly harder. For example, most data management software is still unfit to natively exploit solid-state drives, multi-cores, and deep non-uniform memory hierarchies, even though the concepts used to adapt past techniques and structures rely on ideas that were proposed several decades ago (like partitioning, and avoiding random access). In other words, what changes is how one *tunes* the structures and techniques for the new environment.

Both hardware and applications change rapidly and continuously. As a result, we need to frequently adjust data management software to meet the evolving challenges. In the previous sections we laid the groundwork for RUM access methods, by providing an intuitive analysis of the RUM overheads and the RUM Conjecture. Moving further, here we propose the necessary research steps to design versatile access methods that have variable balance between the RUM overheads. In Figure 3 we visualize the ideal RUM access method, which will be able to seamlessly transition between the three extremes: read optimized, write optimized, and space optimized. In practice, it may not be feasible to aim for building a single access method able to cover the whole spectrum. Instead, an alternative approach is to build multiple access methods able to navigate partly in the RUM space, however, covering the whole space in aggregate.

**Studying The RUM Tradeoffs.** The first step towards building

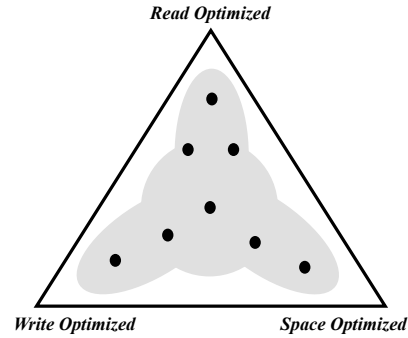


Figure 3: Tunable behavior in the RUM space.

RUM access methods is to extend the discussion in Section 4. A detailed study of the nature of RUM tradeoffs in practice, will lead to a detailed classification of access methods based on their RUM balance. Most existing access methods can be depicted as a static point in the RUM space (Figure 3). The exact position of the point may differ based on some parameters (for example, the fan-out of  $B^+$ -Trees, the number of partitions in PBT, the number of sorted runs in MaSM). Moreover, adaptive indexing techniques like cracking behave in a dynamic manner yet are not tuneable: as they touch more data they add structure to the data and gradually reduce the read overhead at the expense of update overhead.

A concrete outcome of this analysis is the gleaning of all the fundamental building blocks and strategies of access methods. For example, logarithmic access cost (trees, exponentially increasing logs), fixed access cost (tries, hash tables), trading-off computation for auxiliary data size (hashing for hash tables, compression for bitmaps), and lazy updates (log-structure approaches).

**Tunable RUM Balance.** Using the above classification and analysis we can make educated decisions about which access method should be used based on the application requirements and the hardware characteristics, effectively creating a powerful access method wizard. In addition to that, we investigate how to build access methods that have tunable behavior. Such access methods are not single points in the RUM space; instead they can move within an area in the design space.

We envision future data systems with a suite of access methods that can easily adapt to different optimization goals. For example:

- $B^+$ -Trees that have dynamically tuned parameters, including tree height, node size, and split condition, in order to adjust the tree size, the read cost, and the update cost at runtime.
- Approximate (tree) indexing that supports updates with low read performance overhead, by absorbing them in updatable probabilistic data structures (like quotient filters).
- Morphing access methods, combining multiple shapes at once. Adding structure to data gradually with incoming queries, and building supporting index structures when further data reorganization becomes infeasible.
- Update-friendly bitmap indexes, where updates are absorbed using additional, highly compressible, bitvectors which are gradually merged.
- Access methods with iterative logs enhanced by probabilistic data structures that allows for more efficient reads and updates by avoiding accessing unnecessary data at the expense of additional space.

**Dynamic RUM Balance.** We envision access methods that can automatically and dynamically adapt to new workload requirements

or hardware changes, like a sudden increase or decrease of availability of storage or memory. For example, in the case of access methods based on iterative merges, by changing the number of merge trees dynamically, the depth of the merge hierarchy and the frequency of merging, we can build access methods that dynamically adapt to workload and hardware changes.

**Compression and Computation.** Orthogonally to the tension between the three overheads, when accessing data today compression is often used to reduce the amount of data to be moved. This trade-off between computation (compressing/decompressing) and data size does not affect the fundamental nature of the RUM Conjecture. Compression is seldom used only for transferring data through the memory hierarchy. Rather, modern data systems operate mostly on compressed data and decompress as late as possible, usually when presenting the final answer of a query to the user.

## 6. SUMMARY

Changes in hardware, applications and workloads have been the main driving forces in redesigning access methods in order to get the read-update-memory tradeoffs right. In this paper, we show through the RUM Conjecture that creating the ultimate access method is infeasible as certain optimization and design choices are mutually exclusive. Instead, we propose a roadmap towards data structures that can be tuned given hardware and application parameters, leading to the new family of *RUM-aware* access methods.

Although building RUM access methods represents a grand new challenge, we see it as the natural next step inspired by our collective past efforts. Past research in areas such as data structures, tuning tools, adaptive processing and indexing, and hardware-conscious database architectures is the initial pool for concepts and principles to be generalized.

**Acknowledgements.** We thank Margo Seltzer, Niv Dayan, Kenneth Bøgh, the DIAS group at EPFL, and the DASlab group at Harvard for their valuable feedback. This work is partially supported by the Swiss National Science Foundation and by the National Science Foundation under Grant No. IIS-1452595.

## 7. REFERENCES

- [1] D. J. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases*, 5(3):197–280, 2013.
- [2] A. Aggarwal and J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *CACM*, 31(9):1116–1127, 1988.
- [3] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *PVLDB*, 2(1):361–372, 2009.
- [4] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A Hands-free Adaptive Store. In *SIGMOD*, 2014.
- [5] M. Athanassoulis and A. Ailamaki. BF-Tree: Approximate Tree Indexing. *PVLDB*, 7(14):1881–1892, 2014.
- [6] M. Athanassoulis, A. Ailamaki, S. Chen, P. B. Gibbons, and R. Stoica. Flash in a DBMS: Where and How? *IEEE DEBULL*, 33(4):28–34, 2010.
- [7] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. MaSM: Efficient Online Updates in Data Warehouses. In *SIGMOD*, 2011.
- [8] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. Online Updates on Data Warehouses via Judicious Use of Solid-State Storage. *TODS*, 40(1), 2015.
- [9] R. Bayer and K. Unterauer. Prefix B-trees. *TODS*, 2(1):11–26, 1977.
- [10] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuzmaul, and J. Nelson. Cache-Oblivious Streaming B-trees. In *SPAA*, 2007.
- [11] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuzmaul. Concurrent Cache-Oblivious B-Trees. In *SPAA*, 2005.
- [12] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM*, 13(7):422–426, 1970.
- [13] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE*, 2007.
- [14] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, 1997.
- [15] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B+-Trees. In *SIGMOD*, 2002.
- [16] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. of Algorithms*, 55(1):58–75, 2005.
- [17] J. Dittrich and A. Jindal. Towards a One Size Fits All Database Architecture. In *CIDR*, 2011.
- [18] P. Francisco. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. *IBM Redbooks*, 2011.
- [19] E. Fredkin. Trie memory. *CACM*, 3(9):490–499, 1960.
- [20] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *FOCS*, 1999.
- [21] G. Graefe. Sorting And Indexing With Partitioned B-Trees. In *CIDR*, 2003.
- [22] G. Graefe. Modern B-Tree Techniques. *Found. Trends Databases*, 3(4):203–402, 2011.
- [23] G. Graefe, F. Halim, S. Idreos, H. Kuno, and S. Manegold. Concurrency control for adaptive indexing. *PVLDB*, 5(7):656–667, 2012.
- [24] G. Graefe, F. Halim, S. Idreos, H. A. Kuno, S. Manegold, and B. Seeger. Transactional support for adaptive indexing. *VLDBJ*, 23(2):303–328, 2014.
- [25] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, 2010.
- [26] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 5(6):502–513, 2012.
- [27] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a Database System. *Found. Trends Databases*, 1(2):141–259, 2007.
- [28] S. Héman, M. Zukowski, and N. J. Nes. Positional update handling in column stores. In *SIGMOD*, 2010.
- [29] J. Hughes. Revolutions in Storage. In *IEEE Conference on Massive Data Storage*, Long Beach, CA, may 2013.
- [30] IBM. Storage Class Memory: Towards a disruptively low-cost solid-state non-volatile memory. *IBM Almaden Research Center*, 2013.
- [31] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.
- [32] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD*, 2007.
- [33] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.
- [34] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging What’s Cracked, Cracking What’s Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9):586–597, 2011.
- [35] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental Organization for Data Recording and Warehousing. In *VLDB*, 1997.
- [36] B. C. Kuzmaul. A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees. *Tokutek White Paper*, 2014.
- [37] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*, 2013.
- [38] Y. Li, B. He, J. Yang, Q. Luo, K. Yi, and R. J. Yang. Tree Indexing on Solid State Drives. *PVLDB*, 3(1-2):1195–1206, 2010.
- [39] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *SOSP*, 2011.
- [40] W. Litwin and D. B. Lomet. The Bounded Disorder Access Method. In *ICDE*, 1986.
- [41] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.
- [42] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*, 1998.
- [43] P. E. O’Neil. The SB-Tree: An Index-Sequential Structure for High-Performance Sequential Access. *Acta Informatica*, 29(3):241–265, 1992.
- [44] P. E. O’Neil, E. Cheng, D. Gawlick, and E. J. O’Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.
- [45] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *CACM*, 33(6):668–676, 1990.
- [46] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. *VLDBJ*, 12(2):89–101, 2003.
- [47] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *SIGMOD*, 2000.
- [48] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *PVLDB*, 7(2):97–108, 2013.
- [49] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.
- [50] L. Sidirourgos and M. L. Kersten. Column Imprints: A Secondary Index Structure. In *SIGMOD*, 2013.
- [51] K. Wu, et al. FastBit: interactively searching massive data. *Journal of Physics: Conference Series*, 180(1):012053, 2009.

# Self-Curating Databases

Mohammad Sadoghi<sup>#</sup>, Kavitha Srinivas<sup>#</sup>, Oktie Hassanzadeh<sup>#</sup>,  
Yuan-Chi Chang<sup>#</sup>, Mustafa Canim<sup>#</sup>, Achille Fokoue<sup>#</sup>, Yishai Feldman<sup>\*</sup>

<sup>#</sup>IBM T.J. Watson Research Center  
<sup>\*</sup>IBM Research - Haifa

## ABSTRACT

The success of relational databases is due in part to the simplicity of the tabular representation of data, the clear separation of the physical and logical view of data, and the simple representation of the logical view (meta-data) as a flat schema. But we are now witnessing a paradigm shift owing to the explosion of data volume, variety and veracity, and as a result, there is a real need to knit together data that is naturally heterogeneous, but deeply interconnected. To be useful in this world, we argue that today's tabular *data model* must evolve into a *holistic data model* that views meta-data as a new semantically rich source of data and unifies data and meta-data such that the data becomes *descriptive*. Furthermore, given the dynamicity of data, we argue that fundamental changes are needed in how data is consolidated continuously under uncertainty to make the data model naturally more *adaptive*. We further envision that the entire *query model* must evolve into a *context-aware model* in order to automatically discover, explore, and correlate data across many independent sources in real-time within the context of each query. We argue that enriching data with semantics and exploiting the context of the query are the two key prerequisites for building *self-curating databases* in order to achieve a *real-time exploration and fusion of enriched data at web scale*. These needs highlight a series of interesting challenges for database research and alter some of the tenets of Codd's rules for how we think about data.

## 1. INTRODUCTION

We believe that the relational database system will remain the de facto standard for well-structured data. The success of relational database theory is partly due to its simple tabular representation over a predefined relational schema. Tabular data is manipulated through a well-defined declarative relational algebra that is written over the data schema (a logical view). Expressing queries over the logical view has led to decades of query optimization in order to transform queries written over logical views into efficient access methods over the physical layout. As database engines advance, the logical view remains constant, and this has been a key success factor for relational database systems (i.e., data independence).

However, the tabular data represented by the relational schema is limited to a flat schema for describing each table column.<sup>1</sup> One may

<sup>1</sup>Although the database schema has remained a simple flat structure, there have been

argue that the relational schema, in addition to forming a logical view for querying the data, is nothing but a simple blueprint of how to parse the data at the physical level. This blueprint ends at the granularity of columns, which is why often it is referred to as a table schema because it is not at the record level. Homogeneity at the record level is also pre-assumed in the relational theory, in fact, the Boyce-Codd normal forms to some extent already penalize any column heterogeneity [6]. Similarly, NoSQL databases such as key-value stores are still fundamentally tabular, but the "value" column is now heterogeneous with a flexible schema [4].<sup>2</sup>

Although databases were designed for a system of records in order to maintain corporate transactional data, the tabular data model in databases can represent many types of non-transactional data. However, it has certain fundamental limitations. The chief limitation is that the tabular model does not natively capture instance-level relations, which is why a whole class of functional dependency (FD) and referential integrity (RI) constraints had to be developed to express schema-level relationships (e.g., RI) and to avoid record-level inconsistencies (e.g., FDs). In general, integrity constraints are used to ensure that data instances conform to a given schema while only limited knowledge (e.g., relation transitivity) can be expressed using constraints because the primary role of constraints is to restrict the data as opposed to enriching the data [12].

We observe that today's data is no longer limited to systems of records; we now have a variety of data coming from thousands of sources. Data is being generated at an astonishing rate of 2.5 billion gigabytes daily, and further, 80% of data is unstructured and comes in the form of images, video, and audio data to social media (e.g., Twitter, Facebook, Blogosphere) and from embedded sensors and distributed devices [1]. The explosion is partly due to the Internet of Things (IoT) that increasingly connects data sources (including objects and devices) to form a complex network, a network that is expected to exceed one trillion nodes [1].

These emerging data sources are heterogeneous by nature and are independently produced and maintained, yet the data are inherently related. For example, sales patterns correlate with the popularity of the product in social media, and the popularity of the product itself can be measured in terms of how often images or tweets are posted of the product. Even if one considers only the "structured" data after the extraction from the unstructured data, the task of integrating all these disparate data sources leaves islands of data with thousands (if not millions) of tables and schemata that are simply impossible to understand and query by any individual.

Arguably data is a new natural resource in the enterprise world with an unprecedented degree of proliferation and heterogeneity and countless possible ways of aggregating and consuming it to find

attempts to at least model the data conceptually as a hierarchy, e.g., the entity-relation (ER) or object-oriented models.

<sup>2</sup>In fact, several NoSQL initiatives even motivate the need for a *schema-less* paradigm [4] that is in a diametrically opposite position from our self-curating database vision.

actionable insights [1]. However, this inherently interconnected data is trapped in disconnected islands of information, which forces analytics-driven decision making to be carried out in isolation and on stale (and possibly irrelevant) data; thus, making today's *first-ingest-then-process* model insufficient and unnecessary at a time when the cloud is disrupting the entire computing landscape.<sup>3</sup> More importantly, existing database technologies fail to alleviate the data exploration challenges that continue to be a daunting process especially at a time when an army of data scientists are forced to manually and continuously refine their analyses as they sift through these islands of disconnected data sources, a labor-intensive task occupying 50-80% of time spent [11].

We argue that today's database systems need to be fundamentally re-designed to capture data heterogeneity (within local and external data sources) and the semantic relationship among data instances (i.e., data interconnectedness) as first class-citizens. To address these requirements, we propose a *holistic data model* to capture all dimensions of the data, so that we can push the burden of semantic enrichment and integration of the data in a systematic and transparent way into the database engines. We view the data fusion as a *gradual curation* process that transforms the raw data into a new unified entity that has *knowledge-like characteristics*; thereby, we envision the evolution of database systems into *self-curating databases* to meet the continuous enrichment and integration challenges of the information explosion.

In moving from databases to self-curating systems, the schema is no longer a table schema as a separate entity that is limited to necessary information to only parse the data. Instead, the data schema becomes part of the data in order to make the data self-descriptive. Furthermore, it is expected that both the data and meta-data continuously evolve either by ingesting new data sources or through the process of *context-aware query* execution. By context-aware query execution, we emphasize redefining the existing *query model* to enable the discovery of new data linkage and semantic relationships in the specific context of a given query. Thus, while the query must automatically be refined to enable discovery, the data will also become sufficiently enriched in order to enable continuous integration and adjustment of the interconnectedness of instances/types.

In short, our broader vision is a systematic methodology to achieve a *real-time fusion and enrichment of data at web scale* hosted virtually. We argue for a unified view of semantically enriched data by introducing a novel *holistic data model* (i.e., *rethinking the data model*), so queries can be answered by an online consolidation of the most up-to-date data from a variety of sources at query time without the need for offline ingestion and curation. Furthermore, we envision that querying and analytics in general will become explorative in nature to provide deeper and quicker insights by proactively refining and raising new queries based on the context of the query submitted by the user, making the query model *context-aware* (i.e., *rethinking the query model*).

Thus far, we have provided the desired properties of self-curating databases. In the subsequent sections, we elaborate on the specific properties of self-curating databases and highlight the challenges and short- and long-term research opportunities they bring in a systematic fashion. We further broadly classify our proposed statements as either *functional statements* (for adding new capabilities) or *optimization statements* (for improving system performance); these open problems are summarized in Table 1.

## 2. RELATED WORK

Our vision is partly motivated by the recent shift towards semantically enriched information retrieval. We observe a trend among

<sup>3</sup>We anticipate that in the near future all data sources and analytics computation will be hosted virtually on the cloud [1]; thus, there is no need to first ingest data from one computing infrastructure to another before querying the data.

Statement	Description
FS.1	Continuous incremental entity resolution
FS.2	Formalism for assessing interconnectedness richness
FS.3	All-encompassing logical formalism for uncertainty
FS.4	Simplifying logical view of data
FS.5	A Unified language for relational, logical & numerical models
FS.6	Context-aware query refinement semantics
FS.7	Query refinement using query-by-example
FS.8	Incompleteness resolution through crowd
FS.9	Context-aware materialization of ranked & discovered data
FS.10	Parallel world semantics and representational model
FS.11	Concurrency controls for non-deterministic and enriched data model
OS.1	Fine-grained dynamic data clustering
OS.2	Locality-aware multi-hop traversal representation
OS.3	Semantic query optimization
OS.4	Data placement in distributed shared memory

**Table 1: Open problems in self-curating databases**

Web search engines such as Google and Bing in moving away from a pure information retrieval system towards knowledge-based retrieval by not only retrieving a set of documents relevant to the users' queries, but also identifying *entities* and returning *facts* regarding the identified entities [13]. Another prominent initiative is IBM *Watson*, which is an open-domain question answering system for outperforming the best players in Jeopardy [7]. Such knowledge-based retrieval has become possible through the use of rich knowledge bases created by academic and community efforts such as Freebase, DBpedia, and YAGO [13].

What we observe in all these emerging projects [13, 7] is that moving forward, simple information retrieval will be insufficient, and that information will continuously be expanded and semantically enriched as a result of the continuous integration of heterogeneous sources (i.e., the evolution of information to knowledge). Consequently, we argue the need for the evolution of relational databases to handle the challenges in this new enriched data era. We envision that the database systems of the future will no longer be solely responsible for the storage and retrieval of structured data, but they will transform into self-curating databases that are capable of *real-time exploration and fusion of enriched data at web scale*.

We acknowledge that we are not the first to argue for the high-level concepts such as semantic enrichment or continuous integration; in fact, there are several existing efforts in Semantic Web technologies (e.g., [13]) and dataspace and pay-as-you-go integration models (e.g., [8]) that strive for similar high-level objectives. But to achieve these objectives, we argue for a fundamental rethinking of how we view the data and query. We envision the need for a new *holistic data model* to unify the data and meta-data, and to view the meta-data as a new semantically rich source of data. Furthermore, we envision a simpler and more effective way to query and compute answers by automatically refining the query and continuously discovering new data sources within the context of each query, giving rise to a novel *context-aware query model*. Further, what is unique to our vision, in addition to extending past attempts in light of new applications and possibilities (e.g., [7]), is systematically sketching the requisite properties of a self-curating database and providing an extensive list of the concrete research challenges and opportunities needed to make such a vision a reality.

## 3. DATA MODEL: UNIFIED & ENRICHED

In our view, a self-curating database must have a hierarchy of layers to transform raw data incrementally into a *holistic data model* (depicted in Figure 1). First is the *instance layer*, to store the raw data (or data instances) spanning both structured and unstructured. The second layer is the *relation layer*, a horizontal expansion of data to formulate and capture the interconnectedness of data instances within and across data sources (i.e., the fine-grained instance-level linkage). In cases where the raw data layer is unstructured, this layer may additionally capture the results of information extraction. The third layer is the *semantic layer*, a vertical expansion of data to conceptualize data instances and their rela-



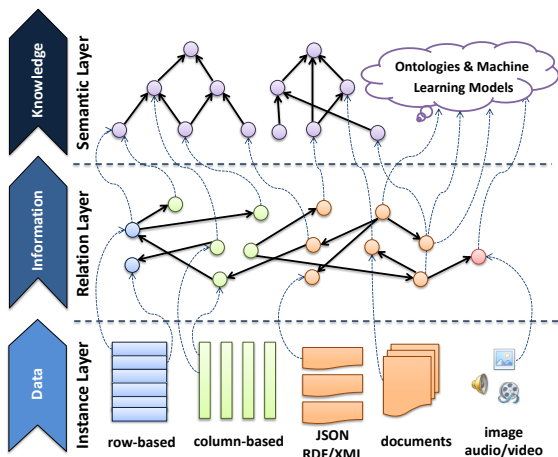


Figure 1: Holistic data model.

tions as semantic types and to formulate the interconnectedness of archetypes and data instantiated types (e.g., ontology). The semantic layer is a way of succinctly capturing conceptual relationships among data instances. This final layer will bring an unprecedented level of expressiveness power and discovery potentials. The last two layers of self-curating databases can be viewed as meta-data, but such a distinction no longer holds in a self-curating database as meta-data is also seen as a rich source of data.

We argue that such a holistic approach is essential to efficiently represent, enrich, manipulate, and query both data and meta-data. Our running example of an enriched data model is extracted from the life science domain, as illustrated in Figure 2. This example is motivated by the overwhelming challenge to unify and enrich data from a variety of heterogeneous sources to develop an assisted diagnosis and personalized treatment and medicine [7].

### 3.1 Instance Layer: Raw Data

The first layer is what today’s relational database systems heavily rely on to represent structured data. But future databases must naively also support semi-structured data such as XML and JSON (already supported by most commercial databases) and unstructured data such as text documents, images, audio, and video. In the example shown in Figure 2, the data comes from different external sources such as DrugBank that offers data about known drugs and diseases, Comparative Toxicogenomics Database that provides information about gene interaction, and Uniprot that provides details about the functions and structure of genes.

One may argue that the proposed instance layer shares similar properties to those already found in the tabular representation of the relational model. However, a deeper question here is whether a tabular representation is an optimal choice for a *holistic data model*. Analytical workloads, for instance, benefit greatly from a columnar decomposition of tabular representation. In contrast, a self-curating database must manage data and meta-data in a unified way, but it is unclear what the optimal representation is for such systems. For example, could the relational model be further decomposed in non-linear and non-tabular form in order to cluster data based on the instance relations and semantic relationships of higher layers?

OPTIMIZATION STATEMENT 1. *Given the abundance of instance relations and semantic relationships, what are the data clustering opportunities to improve retrieval, access locality, and compression? Is it possible to develop dynamic instance-level, fine-grained clustering in the presence of the enriched data model?*<sup>4</sup>

<sup>4</sup>Imagine a representation that could adapt to the locality of access for a workload

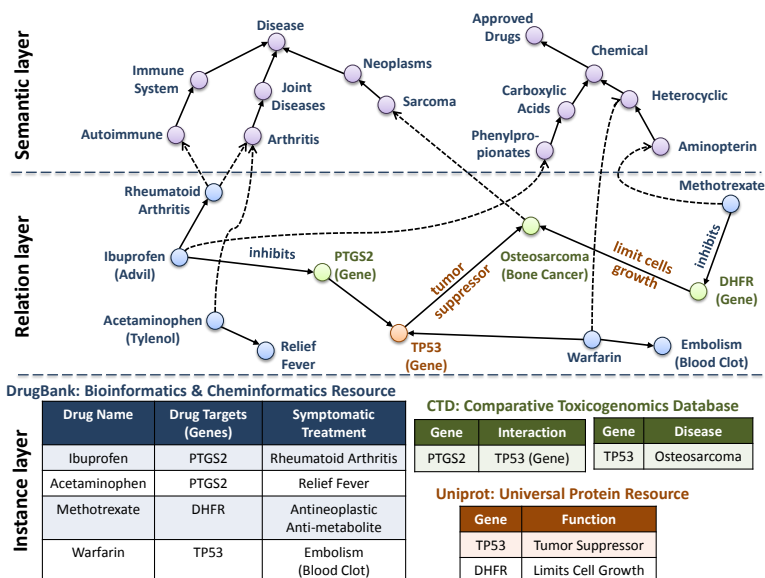


Figure 2: An example of an enriched data model in the life science domain.

### 3.2 Relation Layer: Horizontal Expansion

A relational model has no notion of which columns refer to real world entities (i.e., data instances). But a holistic data model must possess a clear notion of what the entities are, and what relations exist for each instance in order to capture the data interconnectedness. These may be relations to other entities, or the relations of the attributes of the entity to data values. As an example, a database might have a table for *Drug*, and have the columns *Name*, *Targets*, *Symptomatic Treatment*. A rich data model has an identifier for a real world drug *Methotrexate*, and captures its attributes such as *Molecular Structure*, as well as relations to other entities including *Genes* that *Methotrexate* targets (e.g., *DHFR*), and subsequently, *Conditions* that it treats such as *Osteosarcoma (bone cancer)* that are reachable through its target genes, as shown in Figure 2.

The key characteristics of the relation layer are to capture entity interconnectedness and to establish the identity of an entity within and across multiple data sources – a process we term *horizontal data expansion* to transform data into information. An important challenge of the relation layer is to uniquely identify similar entities even when external sources are dynamically changing. There is a long history of entity resolution in the database literature, but the real challenge in this layer is that there is no ability to rely on manual ETL jobs to perform offline schema alignment, and it is not wise to assume that as each source is added to the self-curating database, an all-to-all entity resolution is performed comprehensively across all data sources.

FUNCTIONALITY STATEMENT 1. *A self-curating database must adaptively manage instance relations in light of new information. How does one adapt existing entity resolution techniques so they work across different schemata without requiring prior knowledge about external data sources to enable efficient incremental schema evolution in local data sources?*

FUNCTIONALITY STATEMENT 2. *Furthermore, what is the right formalism to express and capture the interconnectedness in order to assess and measure the richness of each data source based on the connectivity and density? For example, information content and capacity are a common measure for assessing the richness, and graph-theocratic approaches are well suited for studying the connectivity, flow properties, partitioning, and topology, but there is a lack of general formalism to assess the interconnectedness of data.*

based on the interconnectedness of data. The frequently accessed data could be packed together to be used efficiently in the limited, but fast-access memory of modern hardware including CPU cache or GPU and FPGA on-chip memory.

Another challenge is how to efficiently manage relation interconnectedness. One may argue that a graph is the right abstraction model, but it leaves open the question of how to provide fast traversal abilities. Alternatively, one may argue that traditional indexes (e.g., B-Tree) may improve lookup, but at the high-level, indexes only provide one-hop away direct accesses, which are already captured in the explicit interconnectedness of the data. Thus, direct access is no longer beneficial, but rather the open challenge is how to improve the locality of multi-hop traversal.

**OPTIMIZATION STATEMENT 2.** *Given that the instance interconnectedness already encompasses the benefit of one-hop away direct access, what is an optimal representation that provides efficient locality-aware traversal that is tightly coupled with the instance and semantic layers and is update-friendly?*

### 3.3 Semantic Layer: Vertical Expansion

The instance layer together with relations between the instances, as discussed thus far, constitute what is often conceptually referred to as the ABox in the description logic and semantic web literature [3]. That is, instances refer to individual entities in the real world, relations among them are expressed in terms of semantic properties, and each instance is a member of one or more concepts or types. The concepts and semantic properties that are used in the ABox constitute meta-data about the instance data. Concepts themselves may have relationships to each other and semantic properties.

As a somewhat simple example, a *Drug* can be defined as a chemical with an existential quantification over the relation *has Target*. This means that if the actual instance data only stated that *Acetaminophen (Tylenol)* is a *Drug*, a self-curating database could infer that *Acetaminophen* has a target, even if the specific relation has yet to be discovered and expressed as a relationship between *Acetaminophen* and any particular gene. In fact *Acetaminophen* targets *PTGS2* (even though it is not shown in Figure 2).

These richer semantic reasonings are formulated and expressed in taxonomies or web ontology language (OWL), a subset of first-order logic (FOL). Relationships among the concepts and properties are typically referred to in the semantic web as the TBox [3]. To formalize our discussion, we focus on a widely employed OWL-DL language, which is based on the semantics of SHIN. The SHIN semantics is defined as  $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ , where  $\mathcal{I}$  refers to an interpretation,  $\Delta^{\mathcal{I}}$  is a non-empty set (the domain of the interpretation), and  $\cdot^{\mathcal{I}}$  is the interpretation function that maps every atomic concept  $C$  to a set  $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$  (*Approved Drugs* is an example of the concept), every atomic role  $R$  to a binary relation  $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$  (e.g., *has Therapeutic Efficacy* as a role), and every individual  $a$  to  $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ .

An RBox  $\mathcal{R}$  is a finite set of transitivity axioms and role inclusion axioms of the form  $R \sqsubseteq P$  where  $R$  and  $P$  are roles. A Tbox  $\mathcal{T}$  is a set of concept inclusion axioms of the form  $C \sqsubseteq D$ , where  $C$  and  $D$  are concept expressions (e.g., *Neoplasms*  $\sqsubseteq$  *Disease*). An Abox  $\mathcal{A}$  is a set of axioms of the form  $a : C$  ( $a$  is a member of the concept  $C$ ),  $R(a, b)$  (there is an  $R$  relationship between  $a$  and  $b$ ), e.g., *has Target(Acetaminophen, PTGS2)*. Finally, an interpretation  $\mathcal{I}$  is a model of an Abox  $\mathcal{A}$  with respect to a Tbox  $\mathcal{T}$  and a Rbox  $\mathcal{R}$  iff it satisfies all the axioms in  $\mathcal{A}$ ,  $\mathcal{R}$ , and  $\mathcal{T}$ .

A key strength of knowledge representation (KR) formalisms (such as OWL) derived from FOL stems from their capability to represent complex information in a knowledge rich domain, e.g., the biomedical domain. Unfortunately, FOL is incapable of dealing with inconsistency and uncertainty, which naturally arise when information from independent data sources is combined. The KR formalism should capture and aggregate information from both hard and soft sources. Hard sources may have a clear mathematical model of uncertainty, e.g., sensor data. Soft sources, on the other hand, provide vague statements of truth (often fuzzy), such as “a

sudden stomach bleed was attributed to the *recent intake* of *Ibuprofen*”. In contrast, there have been only isolated efforts to extend the KR languages to handle only a particular form of uncertainty, e.g., probabilistic or fuzziness [3]; thus, we raise the following question.

**FUNCTIONALITY STATEMENT 3.** *Is it possible to define a new unifying approach, but perhaps less expressive, to aggregate these isolated forms of uncertainty in a single tractable formalism?*

In general, we view the enrichment of data with semantics as *vertical data expansion* because this layer allows the database to infer new facts about the lower layers. We note that there is an increasing need for the vertical data expansion layer to be more general than the current notion of TBox  $\mathcal{T}$ . Increasingly, conceptual statistical models are being derived from the data to derive new connections between instance data. We therefore propose that the vertical data expansion be enriched by adding statistical models, such as those offered by machine learning, specifically to improve the linkage coverage and accuracy as well, considering that the purpose of this layer is to add semantic inference and reasoning capabilities about the instance types and the relationships among types.

**FUNCTIONALITY STATEMENT 4.** *In the semantic web literature, the assumption is that a user can specify the ontology as a logical view that can be applied over data with respect to a given query. Is it reasonable to have users be aware of the meta-models needed to understand the structure of the data, especially as one allows statistical models? And how does one describe a specific statistical model that should be applied over the data declaratively?*

To further benefit from the enriched data, there is a need for new formalism to combine the expressiveness power of database querying languages (e.g., SQL) with the semantic formalism of description logic (e.g., OWL) to capture the knowledge about the data.

**FUNCTIONALITY STATEMENT 5.** *Is it possible to develop a new semantically enriched query language that combines the expressiveness and declarativeness power of SQL (subset of FOL) and the leading semantic formalisms such as OWL (also subset of FOL) while retaining decades long advancement of query optimization and scalable query execution? Furthermore, is it possible to extend this new combined language with machine learning models that are based on non-declarative statistical, mathematical, or numerical formalism rather than the logical FOL formalism?*

Through semantically enriched data, there is an enormous opportunity to improve query optimization by inferring statistics given that today’s optimizers fail completely in the absence of statistics on the data.

**OPTIMIZATION STATEMENT 3.** *How to extend the predominant rule- and cost-based query optimization to leverage the explicit semantics within our data model, so the optimizers are no longer limited to only statistics on data (e.g., selectivity estimates) to guide the query optimization (often missing or unavailable for external sources)? Is it possible to exploit the available semantics (e.g., exploiting class and subclass relationships) by inferring the selectivity and rewriting the query to a more efficient query (e.g., by inferring that certain predicates can be collapsed together semantically or can be dropped because they are redundant or unsatisfiable)?*

## 4. QUERY MODEL: EMBRACE CONTEXT

There is a compelling case to make queries less complicated through automatic exploration and refinement given the query context while the results must become evidence-based and justified (not limited to just a confidence score). Considering our proposed *holistic data model*, there are new opportunities to formalize and leverage the context of queries throughout the entire query pipeline, giving rise to a new way of thinking about how to query islands of data. We declare a pressing need to *rethink the entire query model* in a self-curating database; in particular, we focus on refining queries and computing answers through the continuous discovery and integration of data made possible by the rich data model.

## 4.1 Continuous Discovery and Refinement

In the database literature, we have the notion of adaptive query processing for collecting more accurate statistics during query execution to proactively optimize the query plan [2]. But conceptually our proposed *context-aware query model* opens up new avenues of research, in which not only more accurate statistics are gathered, but the query is also refined. In addition to refining the query, the data is also being adapted. Specifically, new instance relations or semantics relationships are discovered within the context of a given query (and its refined queries) as part of an online incremental integration, a step towards achieving the continuous integration.

Consider the task of determining an effective dosage of a drug by querying multiple clinical data sources. It is well-known that ethnicity and race have a major role in determining drug responses [9]. Now if these isolated data sources correspond to populations that are biased to genetic, ethnicity, and environmental conditions, then there is a tremendous value in automatically and judiciously navigating through these data sources without forcing the user to be fully aware of the semantics and interpretation of data that would be embedded in the enriched data model.

Suppose the initial query is “What is an effective dosage of Warfarin for preventing a blood clot?” (captured in Figure 2). Now to offer an accurate and justified answer in the presence of many disconnected data sources, there is a crucial need to develop an explorative querying framework that exploits the context of the query. To discover the necessary information and to fill the gap, the following refined queries may be posed automatically: “Is Warfarin sensitive to ethnic background?” (necessary to be aware of any medical facts); “What are the disjoint classes of population with respect to Warfarin?” (necessary for drilling down further); or “Does Warfarin have a narrow therapeutic range?” (necessary to quantify the dosage sensitivity and its range). We argue that such exploration is only possible by enriching data with sufficient semantics in order to interpret the context of queries and raise additional questions.

**FUNCTIONALITY STATEMENT 6.** *A new formalism is needed to express and execute the context-aware query model such that the discovery of new data connections and the refinement of query are feasible. Is it possible to formulate the discovery and refinement process as a random walk problem, where the initial seeds or the probability of each step taken is driven by query predicates and/or query partial results?*

**FUNCTIONALITY STATEMENT 7.** *Alternatively, is it possible to extend the query-by-example formalism [14] for filling missing data to introduce an incremental process so the query answer is partially computed, and the partial answer becomes an example with incompleteness (missing values) for raising/refining additional queries?*

To judge and choose the right formalism for context-aware query answering, we also need to revisit the existing evaluation criteria both in terms of completeness and feasibility.

**FUNCTIONALITY STATEMENT 8.** *To improve the discovery, is it possible to extend the crowdsourcing formalism to identify and assess the necessity to fetch incomplete data given certain qualitative (to improve the accuracy and coverage of answers) or quantitative (to find information faster) cost functions?*

## 4.2 Continuous Online Integration

The importance of online incremental integration for the *context-aware query model* is twofold. First, in a setting consisting of independently managed (but linked) data sources, individual data sources may change over time and one cannot be assured that all updates are propagated in a timely fashion. In fact, one of the main shortcomings of today’s linked data initiative, in which large data sources are linked statically once, is the inability to deal with stale linked data [13]. Second, large scale one-time integration requires

*a priori* knowledge to perform the integration, and this is not always feasible [8]. Moreover, it discards the knowledge of the users of the systems. Every time a user submits a query, the query may contain knowledge about the data, e.g., how two pieces of data are connected. One can think of a submitted query as a small scale but more focused and accurate integration that is at the instance-level and not necessarily at the schema-level.

**FUNCTIONALITY STATEMENT 9.** *There is a need for a new formalism to assess the correctness of query answering within the context of a single query while we discover and consult overlapping or even conflicting sources of information. More importantly, how do we formulate the feedback mechanism to materialize the discovered information guided by the context of query? If the discovered information is conflicting, then how could we automatically assess the richness or validity of discovered entities based on the degree of richness of each source (e.g., information content)?*

Today’s formalisms for computing query answers focus on the inconsistencies, incompleteness, and uncertainty that arise within each data source or a set of integrated sources (i.e., a single consolidated view of data). The traditional probabilistic query answering relies on possible world semantics to assess the likelihood of answers by enumerating all possible worlds [5]. A well-known expressive representational model is a conditional table (c-table), in which each tuple  $t_i$  is associated with a Boolean formula (the condition  $c_i$ ) [10]. The existence of a tuple in a possible world is subject to the satisfaction of its condition [10], c-tables are formally expressed as the valuation function of conditions  $v(c)$ .

Given an instance of data with uncertainty, we have a discrete probability space of  $\mathcal{P} = (W, \mathbf{P})$ , where  $W$  is a set of all the possible worlds given by  $W = \{I_1, \dots, I_n\}$  and  $\mathbf{P}$  is a probability model that assigns probability  $\mathbf{P}(I_i)$  to each possible world  $I_i$  such that  $0 \leq \mathbf{P}(I) \leq 1$  and  $\sum_{i=1}^n \mathbf{P}(I_i) = 1$ . The probability of any tuple  $t$  is the total probability of all worlds in which  $t$  exists and can simply be computed by  $\sum_{i=1, t \in I_i}^n \mathbf{P}(I_i)$ .

Similarly, the incompleteness semantics  $\llbracket \rrbracket$  is defined for an incomplete database  $D$  as a set of complete databases  $\llbracket D \rrbracket$  that are constructed given an interpretation of null values  $\mathcal{I}^{\text{null}}$  under either an open- or closed-world assumption,  $\llbracket \rrbracket_{\text{OWA}}$  or  $\llbracket \rrbracket_{\text{CWA}}$ , respectively [10]. The domain of the database consists of a set of constants (denoted by  $\text{Const}$ ) and a set of nulls (denoted by  $\text{Null}$ ), where the null represents the missing/unknown values. An example of a different interpretation of null values  $\mathcal{I}^{\text{null}}$  is Codd’s three-valued logic.

Subsequently, the problem of query answering is reformulated as finding certain answers for the query  $Q$ . Given an interpretation of nulls  $\mathcal{I}^{\text{null}}$ : the certain answer is defined as  $\text{certain}(Q, D) = \bigcap_i \{Q(D_i) \mid D_i \in \llbracket D \rrbracket\}$ , which amounts to finding an intersection among a set of possible worlds. Notably, an incomplete database can be represented by a c-table [5], an important step towards unifying the representation model for both uncertainty and incompleteness [5, 10]. For instance, to capture both incompleteness and uncertainty, the c-tables semantics can be extended to include the valuation of nulls  $v(t_i)$  and the valuation of conditions  $v(c_i)$  so that a possible complete database instance  $I$  can be computed.

The existing techniques based on possible world semantics focus on deriving possible data instances from a single consolidated representation of data with uncertainty/incompleteness. However, there is no formalism to deal with multiple databases, where each source is complete and certain, but when viewed together without sufficient semantics, then uncertainty, incompleteness, and inconsistencies could arise. Let us revisit querying a set of independent sources, where each source captures clinical trials carried out in a different country and data is demographically biased; thus, naively combining the data from these sources may result in conflicting outcomes, even if data in each source is consistent/certain [9].

Consider a simple Boolean query “Is 5.0 mg an effective dosage

of Warfarin for preventing blood clot?”. If the data was collected in white-dominant population, the effective daily dosage is expected to be around 5.1 mg, while in Asian and black population, daily doses of 3.4 mg and 6.1 mg are recommended, respectively [9]. Now, a naive evaluation may return false as the certain answer to our question (because not all sources report a 5.0 mg dosage rate) while semantically enriched data can infer that these reported dosage rates belong to three disjoint ethnic classes, and to compute the certain answer it is sufficient to have at least one dataset with a daily dose of “close” to 5.0 mg. Now the notion of closeness can further be formulated based on fuzzy logic in light of the fact that “Warfarin has a very narrow therapeutic range” [9]. Therefore, we argue that sufficient semantics are needed to capture the knowledge about the data premises (beyond today’s lineage and provenance information) when integrating multiple data sources, and a new query answering formalism is needed to leverage the added knowledge.

In general, *derived possible worlds* are all constructed from a single integrated and consolidated actual world with incompleteness and/or uncertainty. But data at the web scale consisting of a large set of actual worlds (independent data sources) not just postulated probable worlds. These independent actual worlds, which we refer to as “*parallel worlds*” to distinguish them from the existing *possible worlds* semantics, may have conflicting facts, an alternative view of worlds, or relative facts that are only locally consistent given the premise of the particular world (i.e., semantics of the data). In short, information is relative with respect to the perspective of each independent source, and even in the absence of local inconsistency or uncertainty, the data may become contradictory when combined in the absence of sufficient semantics.

**FUNCTIONALITY STATEMENT 10.** *Firstly, is the exiting c-table formalism sufficiently expressive and concise to model our notion of parallel worlds with our proposed enriched and unified data model? For example, is the c-table representation required to be extended with relation and semantic layers (analogous to our holistic model) to faithfully capture the answers? Now, assuming a representational model, how do we formulate the notion of parallel world semantics for computing justified answers that may not always be globally justified in the presence of overlapping, complementary, and/or opposite relative views of worlds, where “justify” is taken as a fuzzy definition of “certain” to capture, possibly in a relaxed form, correctness and consistency for query semantics?*

In addition to the need for formalism and the semantics of query answering, there are other research challenges related to the execution semantics. As we continuously seek to discover and integrate new data sources and our holistic data model becomes more expressive, a whole set of challenges arise for transaction processing. For example, how do we ensure repeatability and guard against non-deterministic phantoms in transaction processing?

**FUNCTIONALITY STATEMENT 11.** *If the relation and semantic layers can be changed continuously, even when the instance layer does not change, and these layers are further enhanced with non-deterministic predictive inference power, could the classical isolation semantics (e.g., repeatability or snapshot) ever be satisfied? In what ways must concurrency control be extended to account for the non-determinism that is not the result of explicit update queries? Is it possible to introduce relaxed isolation semantics (e.g., eventual consistencies) to account not only for a delay in receiving changes (i.e., pushed and eventually received), but also to account for situations where changes may never be sent explicitly and once received may be non-deterministic (i.e., pulled and eventually received with uncertainty)? These fundamental changes to the concurrency model will inevitably have implication for other components such as logging and recovery protocols.*

A system-level dimension of continuous integration and avoidance of today’s pre-dominant *first-ingest-then-process* arises when

considering the landmark shift of pushing both query execution and hosting of data sources on the cloud [1]. This synergy will introduce a whole new class of workload orchestration and optimization to reduce the cost of online integration and query answering.

**OPTIMIZATION STATEMENT 4.** *How can existing placement strategies be adapted to transition from disk data placement to placing data in distributed main memory at cloud scale? How can the data be judiciously placed in distributed shared memory with close affinity when online integration of data sources is likely in order to eliminate the storage access cost and to reduce the main memory footprint by avoiding data cache duplication?*

## 5. REVISITING DATABASE PRINCIPLES

In conclusion, to characterize our vision of self-curating databases, we revisit Codd’s classical rules for relational systems and elaborate on how these rules must be extended to account for self-curating databases. In the process, we develop a comprehensive list of criteria that may serve as a test for self-curating databases.

- Deviation from *the foundation rule*: A self-curating database cannot assume that all data is managed locally and all data is in a relational model as was prescribed by Codd.
- Deviation from *the information rule*: Information is not limited to only the tabular form. A richer representation is essential to store information about the data. Meta-data and data representations must be unified and their distinction eliminated. Furthermore, every piece of information needs to be represented in the hierarchical multi-layered data model, where each layer semantically enriches the data, unlike Codd’s vision that information is represented in only one way, namely, as a value in a table.
- Extending *the systematic treatment of null values rule*: The data model must allow each data item to be noisy, fuzzy, uncertain, or incomplete so that it can be manipulated systematically, in addition to the need for the nulls to represent missing values as advocated by Codd.
- Extending *the comprehensive data sublanguage rule*: The employed language must also support (1) data discovery and refinement operators and (2) multi-source transactions with limited access and concurrency enforcement on external sources, in addition to the language requirements stated by Codd.
- Deviation from *the view updating rule*: External views may not be updatable or forced to be updated incrementally and lazily, whereas Codd assumes all views must be strictly updatable.
- Deviation from *the integrity independence rule*: Constraints on data and meta-data are not limited to an independent set of rules maintained in the catalog (as required by Codd) because constraints are now modeled at the relation and semantic layers and data instances are physically linked.

## 6. REFERENCES

- [1] The IBM strategy. Annual Report’13. <http://www.ibm.com/annualreport/2013/>, 2013.
- [2] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD’00*.
- [3] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. 2003.
- [4] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.’10*.
- [5] N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: Diamonds in the dirt. *Commun. ACM’09*.
- [6] C. J. Date. *Date on Database: Writings 2000-2006*.
- [7] D. A. Ferrucci, A. Levas, S. Bagchi, D. Gondek, and E. T. Mueller. Watson: Beyond Jeopardy! *Artif. Intell.’13*.
- [8] M. Franklin, A. Halevy, and D. Maier. From databases to dataspace: A new abstraction for information management. *SIGMOD Rec.’05*.
- [9] J. A. Johnson. Ethnic differences in cardiovascular drug response potential contribution of pharmacogenetics. *Circulation’08*.
- [10] L. Libkin. Incomplete data: What went wrong, and how to fix it. In *PODS’14*.
- [11] S. Lohr. For big-data scientists, “janitor work” is key hurdle to insights. *The New York Times*. 2014.
- [12] R. Reiter. Towards a logical reconstruction of relational database theory. In *On Conceptual Modelling (Intervale)*, 1982.
- [13] F. M. Suchanek and G. Weikum. Knowledge bases in the age of big data analytics. *PVLDB’14*.
- [14] M. M. Zloof. Query-by-example: the invocation and definition of tables and forms. In *VLDB’75*.

# Data Wrangling for Big Data: Challenges and Opportunities

Tim Furche  
Dept. of Computer Science  
Oxford University  
Oxford OX1 3QD, UK  
tim.furche@cs.ox.ac.uk

Georg Gottlob  
Dept. of Computer Science  
Oxford University  
Oxford OX1 3QD, UK  
georg.gottlob@cs.ox.ac.uk

Leonid Libkin  
School of Informatics  
University of Edinburgh  
Edinburgh EH8 9AB, UK  
libkin@ed.ac.uk

Giorgio Orsi  
School. of Computer Science  
University of Birmingham  
Birmingham, B15 2TT, UK  
G.Orsi@cs.bham.ac.uk

Norman W. Paton  
School of Computer Science  
University of Manchester  
Manchester M13 9PL, UK  
npaton@manchester.ac.uk

## ABSTRACT

Data wrangling is the process by which the data required by an application is identified, extracted, cleaned and integrated, to yield a data set that is suitable for exploration and analysis. Although there are widely used Extract, Transform and Load (ETL) techniques and platforms, they often require manual work from technical and domain experts at different stages of the process. When confronted with the 4 V's of big data (volume, velocity, variety and veracity), manual intervention may make ETL prohibitively expensive. This paper argues that providing cost-effective, highly-automated approaches to data wrangling involves significant research challenges, requiring fundamental changes to established areas such as data extraction, integration and cleaning, and to the ways in which these areas are brought together. Specifically, the paper discusses the importance of comprehensive support for context awareness within data wrangling, and the need for adaptive, pay-as-you-go solutions that automatically tune the wrangling process to the requirements and resources of the specific application.

## 1. INTRODUCTION

Data wrangling has been recognised as a recurring feature of big data life cycles. Data wrangling has been defined as:

*a process of iterative data exploration and transformation that enables analysis. ([21])*

In some cases, definitions capture the assumption that there is significant manual effort in the process:

*the process of manually converting or mapping data from one "raw" form into another format that allows for more convenient consumption of the data with the help of semi-automated tools. ([35])*

The general requirement to reorganise data for analysis is nothing new, with both database vendors and data integration companies providing Extract, Transform and Load (ETL) products [34]. ETL platforms typically provide components for wrapping data sources, transforming and combining data from different sources, and for loading the resulting data into data warehouses, along with some means of orchestrating the components, such as a workflow language. Such platforms are clearly useful, but in being developed principally for enterprise settings, they tend to limit their scope to supporting the specification of wrangling workflows by expert developers.

Does big data make a difference to what is needed for ETL? Although there are many different flavors of big data applications, the 4 V's of big data<sup>1</sup> refer to some recurring characteristics: *Volume* represents scale either in terms of the size or number of data sources; *Velocity* represents either data arrival rates or the rate at which sources or their contents may change; *Variety* captures the diversity of sources of data, including sensors, databases, files and the deep web; and *Veracity* represents the uncertainty that is inevitable in such a complex environment. When all 4 V's are present, the use of ETL processes involving manual intervention at some stage may lead to the sacrifice of one or more of the V's to comply with resource and budget constraints. Currently,

*data scientists spend from 50 percent to 80 percent of their time collecting and preparing unruly digital data. ([24])*

and only a fraction of an expert's time may be dedicated to value-added exploration and analysis.

In addition to the technical case for research in data wrangling, there is also a significant business case; for example, vendor revenue from big data hardware, software and services was valued at \$13B in 2013, with an annual growth rate of 60%. However, just as significant is the nature of the associated activities. The UK Government's Information Economy Strategy states:

*the overwhelming majority of information economy businesses – 95% of the 120,000 enterprises in the sector – employ fewer than 10 people. ([14])*

As such, many of the organisations that stand to benefit from big data will not be able to devote substantial resources to value-added

<sup>1</sup><http://www.ibmbigdatahub.com/infographic/four-vs-big-data>.

data analyses unless massive automation of wrangling processes is achieved, e.g., by limiting manual intervention to high-level feedback and to the specification of exceptions.

**Example 1** (e-Commerce Price Intelligence). When running an e-Commerce site, it is necessary to understand pricing trends among competitors. This may involve getting to grips with: Volume – thousands of sites; Velocity – sites, site descriptions and contents that are continually changing; Variety – in format, content, targeted community, etc; and Veracity – unavailability, inconsistent descriptions, unavailable offers, etc. Manual data wrangling is likely to be expensive, partial, unreliable and poorly targeted.

As a result, there is a need for research into how to make data wrangling more cost effective. The contribution of this vision paper is to characterise research challenges emerging from data wrangling for the 4Vs (Section 2), to identify what existing work seems to be relevant and where it needs to be further developed (Section 3), and to provide a vision for a new research direction that is a prerequisite for widespread cost-effective exploitation of big data (Section 4).

## 2. DATA WRANGLING – RESEARCH CHALLENGES

As discussed in the introduction, there is a need for cost-effective data wrangling; the 4 V's of big data are likely to lead to the manual production of a comprehensive data wrangling process being prohibitively expensive for many users. In practice this means that data wrangling for big data involves: (i) *making compromises* – as the perfect solution is not likely to be achievable, it is necessary to understand and capture the priorities of the users and to use these to target resources in a cost-effective manner; (ii) *extending boundaries* – as relevant data may be spread across many organisations and of many types; (iii) *making use of all the available information* – applications differ not only in the nature of the relevant data sources, but also in existing resources that could inform the wrangling process, and full use needs to be made of existing evidence; and (iv) *adopting an incremental, pay-as-you-go approach* – users need to be able to contribute effort to the wrangling process in whatever form they choose and at whatever moment they choose.

The remainder of this section expands on these features, pointing out the challenges that they present to researchers.

### 2.1 Making Compromises

Faced with an application exhibiting the 4 V's of big data, data scientists may feel overwhelmed by the scale and difficulty of the wrangling task. It will often be impossible to produce a comprehensive solution, so one challenge is to make well informed compromises.

The *user context* of an application specifies functional and non-functional requirements of the users, and the trade-offs between them.

**Example 2** (e-Commerce User Contexts). In price intelligence, following on from Example 1, there may be different user contexts. For example, routine *price comparison* may be able to work with a subset of high quality sources, and thus the user may prefer features such as *accuracy* and *timeliness* to *completeness*. In contrast, where sales of a popular item have been falling, the associated *issue investigation* may require a more complete picture for the product in question, at the risk of presenting the user with more incorrect or out-of-date data.

Thus a single application may have different *user contexts*, and any approach to data wrangling that hard-wires a process for se-

lecting and integrating data risks the production of data sets that are not always fit for purpose. Making well informed compromises involves: (i) capturing and making explicit the requirements and priorities of users; and (ii) enabling these requirements to permeate the wrangling process. There has been significant work on decision-support, for example in relation to multi-criteria decision making [37], that provides both languages for capturing requirements and algorithms for exploring the space of possible solutions in ways that take the requirements into account. For example, in the widely used Analytic Hierarchy Process [31], users compare criteria (such as timeliness or completeness) in terms of their relative importance, which can be taken into account when making decisions (such as which mappings to use in data integration).

Although data management researchers have investigated techniques that apply specific user criteria to inform decisions (e.g. for selecting sources based on their anticipated financial value [16]) and have sometimes traded off alternative objectives (e.g. precision and recall for mapping selection and refinement [5]), such results have tended to address specific steps within wrangling in isolation, often leading to bespoke solutions. Together with high automation, adaptivity and multi-criteria optimisation are of paramount importance for cost-effective wrangling processes.

### 2.2 Extending the Boundaries

ETL processes traditionally operate on data lying within the boundaries of an organisation or across a network of partners. As soon as companies started to leverage big data and data science, it became clear that data outside the boundaries of the organisation represent both new business opportunities as well as a means to optimize existing business processes.

Data wrangling solutions recently started to offer connectors to external data sources but, for now, mostly limited to open government data and established social networks (e.g., Twitter) via formalised APIs. This makes wrangling processes dependent on the availability of APIs from third parties, thus limiting the availability of data and the scope of the wrangling processes.

Recent advances in web data extraction [19, 30] have shown that fully-automated, large scale collection of long-tail, business-related data, e.g., products, jobs or locations, is possible. The challenge for data wrangling processes is now to make proper use of this wealth of “wild” data by coordinating extraction, integration and cleaning processes.

**Example 3** (Business Locations). Many social networks offer the ability for users to check-in to places, e.g., restaurants, offices, cinemas, via their mobile apps. This gives to social networks the ability to maintain a database of businesses, their locations, and profiles of users interacting with them that is immensely valuable for advertising purposes. On the other hand, this way of acquiring data is prone to data quality problems, e.g., wrong geo-locations, misspelled or fantasy places. A popular way to address these problems is to acquire a curated database of geo-located business locations. This is usually expensive and does not always guarantee that the data is really clean, as its quality depends on the quality of the (usually unknown) data acquisition and curation process. Another way is to define a wrangling process that collects this information right on the website of the business of interest, e.g., by wrapping the target data source directly. The extraction process can in this case be “informed” by existing integrated data, e.g., the business url and a database of already known addresses, to identify previously unknown locations and correct erroneous ones.

### 2.3 Using All the Available Information

Cost-effective data wrangling will need to make extensive use of

automation for the different steps in the wrangling process. Automated processes must take advantage of all available information both when generating proposals and for comparing alternative proposals in the light of the user context.

The *data context* of an application consists of the sources that may provide data for wrangling, and other information that may inform the wrangling process.

**Example 4** (e-Commerce Data Context). In price intelligence, following on from Example 1, the data context includes the catalogs of the many online retailers that sell overlapping sets of products to overlapping markets. However, there are additional data resources that can inform the process. For example, the e-Commerce company has a product catalog that can be considered as master data by the wrangling process; the company is interested in price comparison only for the products it sells. In addition, for this domain there are standard formats, for example in `schema.org`, for describing products and offers, and there are ontologies that describe products, such as The Product Types Ontology<sup>2</sup>.

Thus applications have different *data contexts*, which include not only the data that the application seeks to use, but also local and third party sources that provide additional information about the domain or the data therein. To be cost-effective, automated techniques must be able to bring together all the available information. For example, a product types ontology could be used to inform the selection of sources based on their relevance, as an input to the matching of sources that supplements syntactic matching, and as a guide to the fusion of property values from records that have been obtained from different sources. To do this, automated processes must make well founded decisions, integrating evidence of different types. In data management, there are results of relevance to data wrangling that assimilate evidence to reach decisions (e.g. [36]), but work to date tends to be focused on small numbers of types of evidence, and individual data management tasks. Cost effective data wrangling requires more pervasive approaches.

## 2.4 Adopting a Pay-as-you-go Approach

As discussed in Section 1, potential users of big data will not always have access to substantial budgets or teams of skilled data scientists to support manual data wrangling. As such, rather than depending upon a continuous labor-intensive wrangling effort, to enable resources to be deployed on data wrangling in a targeted and flexible way, we propose an incremental, pay-as-you-go approach, in which the “payment” can take different forms.

Providing a pay-as-you-go approach, with flexible kinds of payment, means automating all steps in the wrangling process, and allowing feedback in whatever form the user chooses. This requires a flexible architecture in which feedback is combined with other sources of evidence (see Section 2.3) to enable the best possible decisions to be made. Feedback of one type should be able to inform many different steps in the wrangling process – for example, the identification of several correct (or incorrect) results may inform both source selection and mapping generation. Although there has been significant work on incremental, pay-as-you-go approaches to data management, building on the dataspace vision [18], typically this has used one or a few types of feedback to inform a single activity. As such, there is significant work to be done to provide a more integrated approach in which feedback can inform all steps of the wrangling process.

**Example 5** (e-Commerce Pay-as-you-go). In Example 1, automated approaches to data wrangling can be used to select sources of

<sup>2</sup><http://www.productontology.org/>

product data, and to fuse the values from such sources to provide reports on the pricing of different products. These reports are studied by the data scientists of the e-Commerce company who are reviewing the pricing of competitors, who can annotate the data values in the report, for example, to identify which are correct or incorrect, along with their relevance to decision-making. Such feedback can trigger the data wrangling system to revise the way in which such reports are produced, for example by prioritising results from different data sources. The provision of domain-expert feedback from the data scientists is a form of payment, as staff effort is required to provide it. However, it should also be possible to use crowdsourcing, with direct financial payment of crowd workers, for example to identify duplicates, and thereby to refine the automatically generated rules that determine when two records represent the same real-world object [20]. It is of paramount importance that these feedback-induced “reactions” do not trigger a re-processing of all datasets involved in the computation but rather limit the processing to the strictly necessary data.

## 3. DATA WRANGLING – RELATED WORK

As discussed in Section 2, cost-effective data wrangling is expected to involve best-effort approaches, in which multiple sources of evidence are combined by automated techniques, the results of which can be refined following a pay-as-you-go approach. Space precludes a comprehensive review of potentially relevant results, so in this section we focus on three areas with overlapping requirements and approaches, pointing out existing results on which data wrangling can build, but also areas in which these results need to be extended.

### 3.1 Knowledge Base Construction

In *knowledge base construction* (KBC) the objective is to automatically create structured representations of data, typically using the web as a source of facts for inclusion in the knowledge base. Prominent examples include YAGO [33], Elementary [28] and Google’s Knowledge Vault [15], all of which combine candidate facts from web data sources to create or extend descriptions of entities. Such proposals are relevant to data wrangling, in providing large scale, automatically generated representations of structured data extracted from diverse sources, taking account of the associated uncertainties.

These techniques have produced impressive results but they tend to have a single, implicit *user context*, with a focus on consolidating slowly-changing, common sense knowledge that leans heavily on the assumption that correct facts occur frequently (instance-based redundancy). For data wrangling, the need to support diverse user contexts and highly transient information (e.g., pricing) means that user requirements need to be made explicit and to inform decision-making throughout automated processes. In addition, the focus on fully automated KBC at web-scale, without systematic support for incremental improvement in a pay-as-you-go manner, tends to require expert input, for example through the writing of rules (e.g., [28]). As such, KBC proposals share requirements with data wrangling, but have different emphases.

### 3.2 Pay-as-you-go Data Management

Pay-as-you-go data management, as represented by the dataspace vision [18], involves the combination of an automated *bootstrapping* phase, followed by incremental *improvement*. There have been numerous results on different aspects of pay-as-you-go data management, across several activities of relevance to data wran-

gling, such as data extraction (e.g., [12]), matching [26], mapping [5] and entity resolution [20]. We note that in these proposals a single type of feedback is used to support a single data management task. The opportunities presented by crowdsourcing have provided a recent boost to this area, in which, typically, paid micro-tasks are submitted to public crowds as a source of feedback for pay-as-you-go activities. This has included work that refines different steps within an activity (e.g. both blocking and fine-grained matching within entity resolution [20]), and the investigation of systematic approaches for relating uncertain feedback to other sources of evidence (e.g., [13]). However, the state-of-the-art is that techniques have been developed in which individual types of feedback are used to influence specific data management tasks, and there seems to be significant scope for feedback to be integrated into all activities that compose a data wrangling pipeline, with reuse of feedback to inform multiple activities [6]. Highly automated wrangling processes require formalised feedback (e.g., in terms of rules or facts to be added/removed from the process) so that they can be used by suitable reasoning processes to automatically adapt the wrangling workflows.

Data Tamer [32] provides a substantially automated pipeline involving schema integration and entity resolution, where components obtain feedback to refine the results of automated analyses. Although Data Tamer moves a significant way from classical, largely manually specified ETL techniques, user feedback is obtained for and applied to specific steps (and not shared across components), and there is no user context to inform where compromises should be made and efforts focused.

### 3.3 Context Awareness

There has been significant prior work on context in computing systems [3], with a particular emphasis on mobile devices and users, in which the objective is to provide data [9] or services [25] that meet the evolving, situational needs of users. In information management, the emphasis has been on identifying the portion of the available information that is relevant in specific ambient conditions [8]. For data wrangling, classical notions of context such as location and time will sometimes be relevant, but we anticipate that for data wrangling: (i) there may be many additional features that characterise the *user and data contexts*, for individual users, groups of users and tasks; and (ii) that the information about context will need to inform a wide range of data management tasks in addition to the selection of the most relevant results.

## 4. DATA WRANGLING – VISION

In the light of the scene-setting from the previous sections, Figure 1 outlines potential components and relationships in a data wrangling architecture. To the left of the figure, several (potentially many) *Data Sources* provide the data that is required for the application. A *Data Extraction* component provides wrappers for the potentially heterogeneous sources (files, databases, documents, web pages), providing syntactically consistent representations that can then be brought together by the *Data Integration* component, to yield *Wrangled Data* that is then available for exploration and analysis.

However, in our vision, these extraction and integration components both *use all the available data* and *adopt a pay-as-you-go approach*. In Figure 1, this is represented by a collection of *Working Data*, which contains not only results and metadata from the *Data Extraction* and *Data Integration* components, but also:

1. all relevant *Auxiliary Data*, which would include the *user context*, and whatever additional information can represent the *data context*, such as *reference data*, *master data* and *do-*

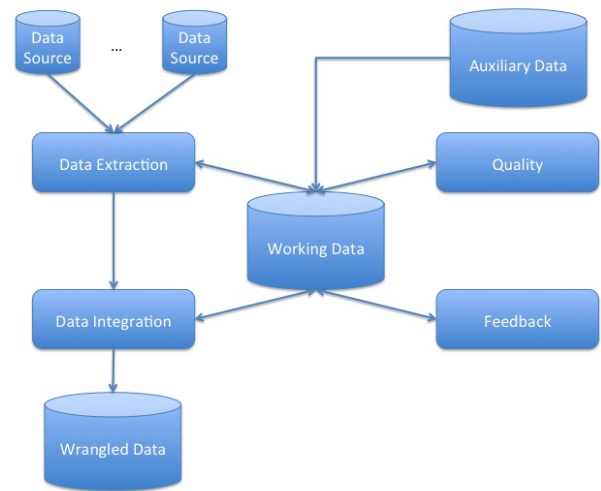


Figure 1: Abstract Wrangling Architecture.

*main ontologies*;

2. the results of all *Quality* analyses that have been carried out, which may apply to individual data sources, the results of different extractions and components of relevance to integration such as matches or mappings; and
3. the feedback that has been obtained from users or crowds, on any aspect of the wrangling process, including the extractions (e.g. users could indicate if a wrapper has extracted what they would have expected), or the results of integrations (e.g. crowds could identify duplicates).

To support this, data wrangling needs substantial advances in data extraction, integration and cleaning, as well as the co-design of the components in Figure 1 to support a much closer interaction in a context-aware, pay-as-you-go setting.

### 4.1 Research Challenges for Components

This section makes the case that meeting the vision will require changes of substance to existing data management functionalities, such as *Data Extraction* and *Data Integration*.

To respond fully to the proposed architecture, *Data Extraction* must make effective use of all the available data. Consider web data extraction, in which wrappers are generated that enable deep web resources to be treated as structured data sets (e.g., [12, 19]). The lack of context and incrementality in data extraction has long been identified as a weakness [11], and research is required to make extraction components responsive to quality analyses, insights from integration and user feedback. As an example, existing knowledge bases and intermediate products of data cleaning and integration processes can be used to improve the quality of wrapper induction (e.g. [29]).

Along the same lines, *Data Integration* must make effective use of all the available data in ways that take account of the *user context*. As data integration acts on a variety of constructs (sources, matches, mappings, instances), each of which may be associated with its own uncertainties, automated functionalities such as those for identifying matches and generating mappings need to be revised to support multi-criteria decision making in the context of uncertainty. For example, the selection of which mappings to use must take into account information from the user context, such as the number of results required, the budget for accessing sources, and quality requirements. To support the complete data wrangling



process involves generalising from a range of point solutions into an approach in which all components can take account of a range of different sources of evolving evidence.

## 4.2 Research Challenges for Architectures

This section makes the case that meeting the vision will require changes of substance to existing data management architectures, and in particular a paradigm-shift for ETL.

Traditional ETL operates on manually-specified data manipulation workflows that extract data from structured data sources, integrating, cleaning, and eventually storing them in aggregated form into data warehouses. In Figure 1 there is no explicit control flow specified, but we note that the requirements of automation, refined on a pay-as-you-go basis taking into account the user context, is at odds with a hard-wired, user-specified data manipulation workflow. In the abstract architecture, the pay-as-you-go approach is achieved by storing intermediate results of the ETL process for on-demand recombination, depending on the user context and the potentially continually evolving data context. As such, the *user context* must provide a declarative specification of the user's requirements and priorities, both functional (data) and non-functional (such as quality and cost trade-offs), so that the components in Figure 1 can be automatically and flexibly composed. Such an approach requires an autonomic approach to data wrangling, in which self-configuration is more central to the architecture than in self-managing databases [10].

The resulting architecture must not only be autonomic, it must also take account of the inherent uncertainty associated with much of the *Working Data* in Figure 1. Uncertainty comes from: (i) *Data Sources* in the form of unreliable and inconsistent data; (ii) the wrangling components, for example in the form of tentative extraction rules or mappings; (iii) the auxiliary data, for example in the form of ontologies that do not quite represent the user's conceptualisation of the domain; and (iv) the feedback which may be unreliable or out of line with the user's requirements or preferences. With this complex environment, it is important that uncertainty is represented explicitly and reasoned with systematically, so that well informed decisions can build on a sound understanding of the available evidence.

This raises an additional research question, on how best to represent and reason in a principled and scalable way with the *working data* and associated *workflows*; there is a need for a uniform representation for the results of the different components in Figure 1, which are as diverse as domain ontologies, matches, data extraction and transformation rules, schema mappings, user feedback and provenance information, along with their associated quality annotations and uncertainties.

In addition, the ways in which different types of user engage with the wrangling process is also worthy of further research. In Wrangler [22], now commercialised by Trifacta, data scientists clean and transform data sets using an interactive interface in which, among other things, the system can suggest generic transformations from user edits. In this approach, users provide feedback on the changes to the selected data they would like to have made, and select from proposed transformations. Additional research could investigate where such interactions could be used to inform upstream aspects of the wrangling process, such as source selection or mapping generation, and to understand how other kinds of feedback, or the results of other analyses, could inform what is offered to the user in tools such as Wrangler.

## 4.3 Research Challenges in Scalability

In this paper we have proposed responding to the *Volume* as-

pect of big data principally in the form of the number of sources that may be available, where we propose that automation and incrementality are key approaches. In this section we discuss some additional challenges in data wrangling that result from scale.

The most direct impact of scale in big data results from the sheer volume of data that may be present in the sources. ETL vendors have responded to this challenge by compiling ETL workflows into big data platforms, such as map/reduce. In the architecture of Figure 1, it will be necessary for extraction, integration and data querying tasks to be able to be executed using such platforms. However, there are also fundamental problems to be addressed. For example, many quality analyses are intractable (e.g. [7]), and evaluating even standard queries of the sort used in mappings may require substantial changes to classical assumptions when faced with huge data sets. Among these challenges are understanding the requirement for query scalability [2] that can be provided in terms of access and indexing information [17], and developing static techniques for query approximation (i.e., without looking at the data) as was initiated in [4] for conjunctive queries. For the architecture of Figure 1 there is the additional requirement to reason with uncertainty over potentially numerous sources of evidence; this is a serious issue since even in the classical settings data uncertainty often leads to intractability of the most basic data processing tasks [1, 23]. We also observe that knowledge base construction has itself given rise to novel reasoning techniques [27], and additional research may be required to inform decision-making for data wrangling at scale.

## 5. CONCLUSIONS

Data wrangling is a problem and an opportunity:

- A problem because the 4 V's of big data may all be present together, undermining manual approaches to ETL.
- An opportunity because if we can make data wrangling much more cost effective, all sorts of hitherto impractical tasks come into reach.

This vision paper aims to raise the profile of data wrangling as a research area within the data management community, where there is a lot of work on relevant functionalities, but where these have not been refined or integrated as is required to support data wrangling. The paper has identified research challenges that emerge from data wrangling, around the need to make compromises that reflect the user's requirements, the ability to make use of all the available evidence, and the development of pay-as-you-go techniques that enable diverse forms of payment at convenient times. We have also presented an abstract architecture for data wrangling, and outlined how that architecture departs from traditional approaches to ETL, through increased use of automation, which flexibly accounts for diverse user and data contexts. It has been suggested that this architecture will require changes of substance to established data management components, as well as the way in which they work together. For example, the proposed architecture will require support for representing and reasoning with the diverse and uncertain working data that is of relevance to the data wrangling process. Thus we encourage the data management research community to direct its attention at novel approaches to data wrangling, as a prerequisite for the cost-effective exploitation of big data.

## Acknowledgments

This research is supported by the VADA Programme Grant from the UK Engineering and Physical Sciences Research Council, whose support we are pleased to acknowledge. We are also grateful to our colleagues in VADA for their contributions to discussions on data wrangling: Peter Buneman, Wenfei Fan, Alvaro Fernandes, John

## 6. REFERENCES

- [1] S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. *TCS*, 78(1):158–187, 1991.
- [2] M. Armbrust, E. Liang, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. Generalized scale independence through incremental precomputation. In *SIGMOD*, pages 625–636, 2013.
- [3] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *IJAHUC*, 2(4):263–277, 2007.
- [4] P. Barceló, L. Libkin, and M. Romero. Efficient approximations of conjunctive queries. *SIAM J. Comput.*, 43(3):1085–1130, 2014.
- [5] K. Belhajjame, N. W. Paton, S. M. Embury, A. A. A. Fernandes, and C. Hedeler. Incrementally improving dataspace based on user feedback. *Inf. Syst.*, 38(5):656–687, 2013.
- [6] K. Belhajjame, N. W. Paton, A. A. A. Fernandes, C. Hedeler, and S. M. Embury. User feedback as a first class citizen in information integration systems. In *CIDR*, pages 175–183, 2011.
- [7] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, pages 143–154, 2005.
- [8] C. Bolchini, C. Curino, E. Quintarelli, F. A. Schreiber, and L. Tanca. A data-oriented survey of context models. *SIGMOD Rec.*, 36(4):19–26, 2007.
- [9] C. Bolchini, C. A. Curino, G. Orsi, E. Quintarelli, R. Rossato, F. A. Schreiber, and L. Tanca. And what can context do for data? *CACM*, 52(11):136–140, 2009.
- [10] S. Chaudhuri and V. R. Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, pages 3–14, 2007.
- [11] S. Chuang, K. C. Chang, and C. X. Zhai. Collaborative wrapping: A turbo framework for web data extraction. In *ICDE*, 2007.
- [12] V. Crescenzi, P. Merialdo, and D. Qiu. A framework for learning web wrappers from the crowd. In *WWW*, pages 261–272, 2013.
- [13] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. Large-scale linked data integration using probabilistic reasoning and crowdsourcing. *VLDBJ*, 22(5):665–687, 2013.
- [14] Department for Business, Innovation & Skills. Information economy strategy. <http://bit.ly/1W4TPGU>, 2013.
- [15] X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmman, S. Sun, , and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *KDD*, pages 601–610, 2014.
- [16] X. L. Dong, B. Saha, and D. Srivastava. Less is more: Selecting sources wisely for integration. *PVLDB*, 6(2):37–48, 2012.
- [17] W. Fan, F. Geerts, and L. Libkin. On scale independence for querying big data. In *PODS*, pages 51–62, 2014.
- [18] M. J. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspace: a new abstraction for information management. *SIGMOD Record*, 34(4):27–33, 2005.
- [19] T. Furche, G. Gottlob, G. Grasso, X. Guo, G. Orsi, C. Schallhart, and C. Wang. DIADEM: Thousands of websites to a single database. *PVLDB*, 7(14):1845–1856, 2014.
- [20] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. W. Shavlik, and X. Zhu. Corleone: hands-off crowdsourcing for entity matching. In *SIGMOD*, pages 601–612, 2014.
- [21] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck, and P. Buono. Research directions in data wrangling: Visualizations and transformations for usable and credible data. *Information Visualization*, 10(4):271–288, 2011.
- [22] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *CHI*, pages 3363–3372, 2011.
- [23] L. Libkin. Incomplete data: what went wrong, and how to fix it. In *PODS*, pages 1–13, 2014.
- [24] S. Lohr. For big-data scientists, ‘janitor work’ is key hurdle to insights. <http://nyti.ms/1Aqif2X>, 2014.
- [25] Z. Maamar, D. Benslimane, and N. C. Narendra. What can context do for web services? *CACM*, 49(12):98–103, 2006.
- [26] R. McCann, W. Shen, and A. Doan. Matching schemas in online communities: A web 2.0 approach. In *ICDE*, pages 110–119, 2008.
- [27] F. Niu, C. Ré, A. Doan, and J. W. Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an RDBMS. *PVLDB*, 4(6):373–384, 2011.
- [28] F. Niu, C. Zhang, C. Ré, and J. W. Shavlik. Elementary: Large-scale knowledge-base construction via machine learning and statistical inference. *IJSWIS*, 8(3):42–73, 2012.
- [29] S. Ortona, G. Orsi, M. Buoncristiano, and T. Furche. Wadar: Joint wrapper and data repair. *PVLDB*, 8(12):1996–2007, 2015.
- [30] D. Qiu, L. Barbosa, X. L. Dong, Y. Shen, and D. Srivastava. DEXTER: large-scale discovery and extraction of product specifications on the web. *PVLDB*, 8(13):2194–2205, 2015.
- [31] T. L. Saaty. The modern science of multicriteria decision making and its practical applications: The AHP/ANP approach. *Operations Research*, 61(5):1101–1118, 2013.
- [32] 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, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.
- [33] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.
- [34] P. Vassiliadis. A survey of extract-transform-load technology. *IJDWM*, 5(3):1–27, 2011.
- [35] Wikipedia: Various Authors. Data wrangling. <http://bit.ly/1KslZb7>, 2007.
- [36] X. Yin, J. Han, and P. S. Yu. Truth discovery with multiple conflicting information providers on the web. *IEEE Trans. Knowl. Data Eng.*, 20(6):796–808, 2008.
- [37] C. Zopounidis and P. M. Pardalos. *Handbook of Multicriteria Analysis*. Springer, 2010.

# Road to Freedom in Big Data Analytics

Divy Agrawal\* Sanjay Chawla Ahmed Elmagarmid Zoi Kaoudi Mourad Ouzzani  
 Paolo Papotti Jorge-Arnulfo Quiané-Ruiz Nan Tang Mohammed J. Zaki\*

*Data Analytics Center, Qatar Computing Research Institute, HBKU*

## ABSTRACT

The world is fast moving towards a data-driven society where data is the most valuable asset. Organizations need to perform very diverse analytic tasks using various data processing platforms. In doing so, they face many challenges; chiefly, platform dependence, poor interoperability, and poor performance when using multiple platforms. We present RHEEM, our vision for big data analytics over diverse data processing platforms. RHEEM provides a three-layer *data processing* and *storage* abstraction to achieve both platform independence and interoperability across multiple platforms. In this paper, we discuss our vision as well as present multiple research challenges that we need to address to achieve it. As a case in point, we present a data cleaning application built using some of the ideas of RHEEM. We show how it achieves *platform independence* and the performance benefits of following such an approach.

## 1. WHY TIED TO ONE SINGLE SYSTEM?

Data analytic tasks may range from very simple to extremely complex pipelines, such as data extraction, transformation, and loading (ETL), online analytical processing (OLAP), graph processing, and machine learning (ML). Following the dictum “one size does not fit all” [23], academia and industry have embarked on an endless race to develop data processing platforms for supporting these different tasks, *e.g.*, DBMSs and MapReduce-like systems. Semantic completeness, high performance, and scalability are key objectives of such platforms. While there have been major achievements in these objectives, users still face two main roadblocks.

The **first roadblock** is that applications are tied to a single processing platform, making the migration of an application to new and more efficient platforms a difficult and costly task. Furthermore, complex analytic tasks usually require the combined use of different processing platforms. As a result, the common practice is to develop several specialized analytic applications on top of different platforms. This requires users to manually combine the results to draw a conclusion. In addition, users may need to re-implement existing applications on top of faster processing platforms when

these become available. For example, Spark SQL [3] and MLlib [2] are the Spark counterparts of Hive [24] and Mahout [1].

The **second roadblock** is that datasets are often produced by different sources and hence they natively reside on different storage platforms. As a result, users often perform tedious, time-intensive, and costly data migration and integration tasks for further analysis.

Let us illustrate these roadblocks with an Oil & Gas industry example [13]. A single oil company can produce more than 1.5TB of diverse data per day [6]. Such data may be structured or unstructured and come from heterogeneous sources, such as sensors, GPS devices, and other measuring instruments. For instance, during the exploration phase, data has to be acquired, integrated, and analyzed in order to predict if a reservoir would be profitable. Thousands of downhole sensors in exploratory wells produce real-time seismic data for monitoring resources and environmental conditions. Users integrate these data with the physical properties of the rocks to visualize volume and surface renderings. From these visualizations, geologists and geophysicists formulate hypotheses and verify them with ML methods, such as regression and classification. Training of the models is performed with historical drilling and production data, but oftentimes users have to go over unstructured data, such as notes exchanged by emails or text from drilling reports filed in a cabinet. Thus, an application supporting such a complex analytic pipeline has to access several sources for historical data (relational, but also text and semi-structured), remove the noise from the streaming data coming from the sensors, and run both traditional (such as SQL) and statistical analytics (such as ML algorithms) over different processing platforms.

Similar examples can be drawn from many other domains such as healthcare: *e.g.*, IBM reported that North York hospital needs to process 50 diverse datasets, which are on a dozen different internal systems [15]. These emerging applications clearly show the need for complex analytics coupled with a diversity of processing platforms, which raises two major research challenges.

**Data Processing Challenge.** Users are faced with various choices on where to *process* their data, each choice with possibly orders of magnitude differences in terms of performance. However, users have to be intimate with the intricacies of the processing platform to achieve high efficiency and scalability. Moreover, once a decision is taken, users may end up being tied up to a particular platform. As a result, migrating the data analytics stack to a more efficient processing platform often becomes a nightmare. Thus, there is a need to build a system that offers *data processing platform independence*. Furthermore, complex analytic applications require executing tasks over different processing platforms to achieve high performance. For example, one may aggregate large datasets with traditional queries on top of a relational database such as PostgreSQL, but ML tasks might be much faster if executed on Spark [28]. How-

\*Work done while at QCRI.

ever, this requires a considerable amount of manual work in selecting the best processing platforms, optimizing tasks for the chosen platforms, and coordinating task execution. Thus, this also calls for *multi-platform task execution*.

**Data Storage Challenge.** Data processing platforms are typically tightly coupled with a specific *storage* solution. Moving data from a certain storage (*e.g.*, a relational DB) to a more suitable processing platform for the actual task (*e.g.*, Spark on HDFS) requires shuffling data between different systems. Such shuffling may end up dominating the execution time. Moreover, different departments in the same organization may go for different storage engines due to legacy as well as performance reasons. Dealing with such heterogeneity calls for *data storage independence*.

To tackle these two challenges, we envision a system, called RHEEM<sup>1</sup>, that provides both platform independence and interoperability (Section 2). In the following, we first discuss our vision for the data processing abstraction (Section 3), which is fully based on user-defined functions (UDFs) to provide adaptability as well as extensibility. This processing abstraction allows both users to focus only on the logic of their data analytic tasks and applications to be independent from the data processing platforms. We then discuss how to divide a complex analytic task into smaller subtasks to exploit the availability of different processing platforms (Section 4). As a result, RHEEM can run simultaneously a single data analytic task over multiple processing platforms to boost performance. Next, we present our first attempt to build an instance application based on some of the ideas of RHEEM and the resulting benefits (Section 5). We then show how we push down the processing abstraction idea to the storage layer (Section 6). This storage abstraction allows both users to focus on their storage needs and the processing platforms to be independent from the storage engines.

Some initial efforts are also going into the direction of providing data processing platform independence [11, 12, 21] (Section 7). However, our vision goes beyond the data processing. We not only envision a data processing abstraction but also a data storage abstraction, allowing us to consider data movement costs during task optimization. We give a research agenda highlighting the challenges that need to be tackled to build RHEEM in Section 8.

## 2. OUR VISION

We envision a system that frees applications and users from being tied to a single data processing platform (platform independence) and provides interoperability across different platforms (multi-platform task execution). We discuss these two aspects in the following. We discuss data storage independence in Section 6.

**Processing Platform Independence.** Whenever a new platform that achieves better performance than existing ones becomes available, one is enticed to move to the new platform. However, such move does not usually come without pain. There is a clear need for a system that frees us from the burden and cost of re-implementing applications from one platform to another. Mahout [1] and MLlib [2] clearly illustrate this need, as all ML algorithms initially implemented in Hadoop had to be re-implemented in Spark. In addition, there are cases where, for the same task but with a different input, one platform is better than another. Thus, the system we envision should not only provide platform independence, but also should be able to select the best available platform to execute a given task in order to deliver better performance.

**Multi-Platform Task Execution.** We are witnessing the emergence of complex data analytic pipelines in many different do-

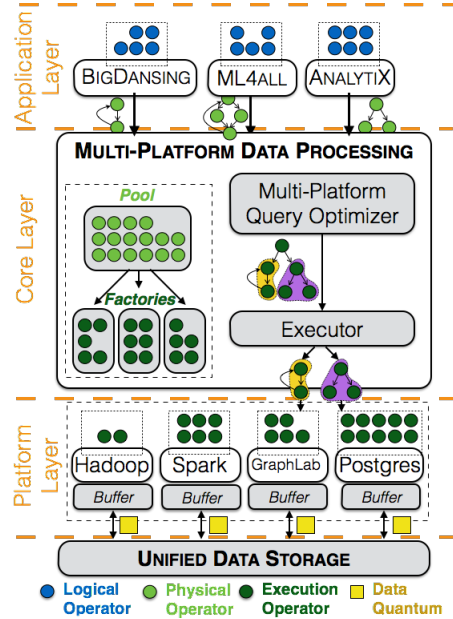


Figure 1: RHEEM data processing abstraction.

main [4, 6, 13, 15]. These pipelines require first combining multiple processing platforms to perform each task of the process and then integrating the results. For instance, many companies are already adopting a lambda architecture, which combines both batch and stream processing. Our vision goes beyond batch or stream processing to any kind of data analytics paradigm. We envision a system that eases the integration among different processing platforms by automatically dividing a task into subtasks and determining the underlying platform for each subtask.

**RHEEM.** The foundation of our vision is a three-layer data processing abstraction that sits between user applications and data processing platforms (*e.g.*, Hadoop or Spark). Figure 1 depicts these three layers: the *application* layer models all application-specific logic; the *core* layer provides the intermediate representation between applications and processing platforms; and the *platform* layer embraces the underlying processing platforms. In contrast to DBMSs, RHEEM decouples physical and execution levels. This separation allows applications to express physical plans in terms of algorithmic needs only, without being tied to a particular processing platform. The communication among these levels is enabled by operators defined as UDFs. These three layers allow RHEEM to provide applications with platform independence. Providing platform independence is the first step towards realizing multi-platform task execution, which is crucial to achieve the best performance at all times. For example, Figure 2 shows the benefits of running the SVM algorithm on different datasets from LIBSVM<sup>2</sup> with only one hundred iterations, as a Spark job and as a plain Java program. We observe that, for small datasets, executing SVM as a plain Java program is up to one order of magnitude faster than executing it on Spark. Indeed, this performance gap gets bigger with the number of iterations. Using Spark pays off for big datasets only. Such results show a great potential for platform independence and ultimately multi-platform execution. RHEEM will be able to receive a complex analytic task, seamlessly divide it into subtasks, schedule each task on the best processing platform, monitor task execution, and aggregate results for users or applications. Achieving our vision requires tackling several challenges that we will discuss throughout

<sup>1</sup>Rheem is a native gazelle species in Qatar.

<sup>2</sup><http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/>

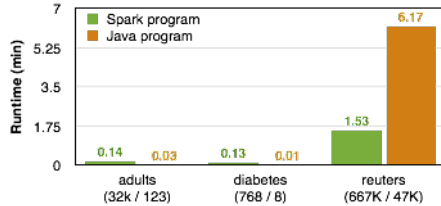


Figure 2: SVM on Spark and Java.

the paper and then summarize in Section 8.

To show the benefits of our RHEEM vision, we have fully developed one data cleaning application based on it [19]. While this initial instance provides only platform independence, its performances are encouraging and already demonstrate the advantages of our vision (see Section 5).

Similar to the data processing abstraction, we envision a three-level data storage abstraction to uphold data processing tasks. The data storage abstraction is also composed of an *application*, a *core*, and an *execution* level. The RHEEM data storage abstraction functions symmetrically as the data processing abstraction. We shall further discuss this storage abstraction in Section 6.

### 3. DATA PROCESSING ABSTRACTION

In this section, we detail the abstraction layers of RHEEM and show how users can interact with the system at each layer.

#### 3.1 Abstraction Layers

RHEEM provides a set of operators at each layer, namely, logical operators, physical operators, and execution operators. The input of the application layer is the logical operators provided by users (or generated by a declarative query parser) and the output is a physical plan. The physical plan is then passed to the core layer where multi-platform optimizations take place to produce an execution plan. In contrast to a DBMS, RHEEM decouples the physical level from the execution one. This separation allows applications to express a physical plan in terms of algorithmic needs only, without being tied to a particular processing platform.

**Application Layer.** A logical operator is an abstract UDF that acts as an *application-specific* unit of data processing. One can see a logical operator as a template where users provide the logic of their tasks. Such abstraction enables both ease-of-use, by hiding implementation details from users, and high performance, by allowing several optimizations, *e.g.*, seamless distributed execution.

A logical operator works on *data quanta*, which are the smallest units of data elements from the input datasets. For instance, a data quantum represents a tuple in the input dataset or a row in a matrix. This fine-grained data model allows RHEEM to apply operators in a highly parallel fashion and thus achieve better performance.

*Example 1:* Consider a developer who wants to offer end users logical operators to implement various ML algorithms. The developer can define three basic operators: (i) Initialize, for initializing algorithm-specific parameters, *e.g.*, initializing cluster centroids, (ii) Process, for the computations required by the ML algorithm, *e.g.*, finding the nearest centroid of a point, (iii) Loop, for specifying the stopping condition. Users implement algorithms such as SVM, K-means, and linear/logistic regression with them. □

The application optimizer translates logical operators into physical operators that will form the *physical plan* at the core layer.

**Core Layer.** This layer is the heart of RHEEM. It exposes a pool of *physical* operators, each representing an algorithmic decision for executing an analytic task. A physical operator is a *platform-independent* implementation of a logical operator. These operators

are available to the developer to deploy a new application on top of RHEEM. Developers can still define new operators as needed.

*Example 2:* In the above ML example, the application optimizer maps Initialize to a Map physical operator and Process to a GroupBy physical operator. RHEEM provides two different implementations for GroupBy: the SortGroupBy (sort-based) and HashGroupBy (hash-based) operators from which the optimizer of the core level will have to choose. □

Once an application has produced a physical plan for a given input task, RHEEM divides this physical plan into *task atoms*, *i.e.*, sub-tasks, which are the units of execution. A task atom (a part of the execution plan) is a sub-task to be executed on a single data processing platform. It will then translate the task atoms into an *execution plan* by optimizing each task atom according to a target platform. Finally, it schedules each task atom to be executed on its corresponding platform. Therefore, in contrast to DBMSs, RHEEM produces execution plans that can run on multiple platforms.

**Platform Layer.** At this layer, *execution* operators (in an execution plan) define how a task is executed on the underlying processing platform. In other words, an execution operator is the *platform-dependent* implementation of a physical operator. RHEEM relies on existing data processing platforms to actually run input tasks.

*Example 3:* Again in the above ML example, the MapPartitions and ReduceByKey execution operators for Spark are one way to perform Initialize and Process. □

In contrast to a logical operator, an execution operator works on multiple data quanta rather than a single one, which enables the processing of multiple data quanta with a single function call.

**Flexible operator mappings.** Defining mappings between execution and physical operators is the developers' responsibility whenever a new platform is plugged to the core. Our goal is to rely on a mapping structure to model the correspondences between operators together with context information. Such context is needed for the effective and efficient execution of each operator. For instance, the Process logical operator maps to two different physical operators (SortGroupBy and HashGroupBy). In this case, a developer could use the context to provide hints to the optimizer for choosing the right physical operator at run time. Developers will provide only a declarative specification of such mappings; the system will use them to translate physical operators to execution operators.

#### 3.2 User Interaction

We distinguish between two types of users: *end-users*, who interact with the applications, and *developers*, who interact with the system at all the three layers. We discuss below how developers define operators (UDFs) at every layer of the abstraction.

**Application layer.** At this layer, developers model a data processing application by specifying a set of abstract logical operators. End-users implement these operators to express their analytic tasks. RHEEM provides an abstract LogicalOperator that defines the method applyOp. Logical operators of any application extend LogicalOperator and provide an implementation of applyOp. RHEEM invokes this method at runtime to apply a logical operator. In addition to logical operators, an application developer could also expose a declarative language for users to define their tasks (*e.g.*, queries). The application is then responsible for translating a declarative query into a logical plan. Then, the application optimizer translates the logical plan into a physical plan.

**Core layer.** RHEEM provides a pool of physical operators for applications to produce physical plans. To enable extensibility, the

system also provides an abstract `PhysicalOperator`, with the abstract method `applyOp`, for developers to define their own physical operators. Developers define a new physical operator to fill two different needs. First, developers define a *wrapper* operator to execute the logical operator together with some physical details, such as algorithmic decisions and schema details. The wrapper operator follows the signature of the logical operator. Second, since the output of a specific operator might not fit as input of a subsequent operator, developers define *enhancer* operators to fill possible gaps between wrapper operators. For example, an application for  $K$ -means clustering might only expose the `GetCentroid` (for getting the closest centroid of a data point) and `SetCentroids` (for computing the new centroids) logical operators. `GetCentroid` outputs a data point and its closest centroid, while `SetCentroids` requires a centroid and all its closest data points. Here, the developer provides a `GroupBy` enhancer operator between `GetCentroid` and `SetCentroid`.

**Platform layer.** To model a data processing platform, developers extend the abstract `ExecutionOperator` and implement its `applyOp` method. There are two main scenarios. If a new physical operator has been defined, *e.g.*, because the developer is adding a new application, then it must be supported with a corresponding execution operator in the actual execution platform. In a different scenario, the developer is adding a new platform to the execution layer. In this case, every physical operator must be supported with a corresponding execution operator in the new execution platform. RHEEM uses these execution operators to produce an execution plan and pushes “down” execution details to the underlying platform, such as data distribution, parallel execution, and data storage. At the end, the target processing platform simply performs an execution plan in its own data and processing model.

## 4. MULTI-LAYER OPTIMIZATIONS

In contrast to traditional data management systems, which are tied to a specific data processing platform, RHEEM’s goal is not only platform independence but also multi-platform execution. To efficiently deal with both aspects, we envision optimizations at each layer: (i) at the application layer, we validate an input task, translate it into a logical plan, and then produce an optimized physical plan, (ii) at the core layer, we translate a physical plan into an execution plan by dividing the query into *task atoms*, and (iii) at the platform layer, we further refine a task atom based on the actual platform.

### 4.1 Application-Layer Optimizations

In our envisioned system, users will be able to express their tasks either procedurally (via logical operators) or declaratively (using a query language). Given an input task, the application optimizer builds a logical plan and performs some pre-defined optimizations, such as operator push-down. Once the logical plan is built, the application optimizer produces an optimized physical plan by translating each logical operator into a wrapper physical operator. Recall that a wrapper receives a logical operator as input. Additionally, the application optimizer might use enhancer physical operators to boost performance; for example, it may plug-in a `GroupBy` physical operator followed by a `CrossProduct` operator to perform a cross product inside a group only. This avoids a costly cross product over the entire input dataset. As an example of this kind of optimization we refer to [19]. Then, the application sends the optimized physical plan to the core layer.

### 4.2 Core-Layer Optimizations

The optimizations at the core layer are the responsibility of the multi-platform task optimizer (see Figure 1). RHEEM receives

a physical plan from an application and passes it to the multi-platform task optimizer to generate a plan for execution on the underlying processing platforms. We envision the multi-platform task optimizer to deal with five aspects. First, the optimizer should consider operators as first-class citizens. That is, its optimization process should be fully based on UDFs optimization techniques. We will base our solutions on different existing optimization techniques, such as Manimal [16], PACTs [25], and SOFA [22], but we also need to devise new optimization techniques to support different processing platforms. Second, the optimizer should consider rules and cost models for its optimizations as plugins and not hard-coded as in traditional database optimizers. In other words, these two aspects should be decoupled from the optimizer in order to allow for extensibility when new processing platforms are added by developers. Third, it has to consider inter-platform cost models to effectively take into account the cost of moving data and computation across underlying processing platforms. The main difficulty here comes from the fact that underlying frameworks are typically highly heterogeneous in terms of both data representations and processing paradigms. Fourth, it should divide a physical plan into task atoms according to the supported underlying data processing platforms. Recall that it is the underlying processing platform that ensures the execution of task atoms. The main challenge in this aspect is to find a way to divide a task into atoms seamlessly from users. Last, but not least, it should also apply traditional physical optimizations, whenever possible. Examples are shared scans and optimized data access paths, such as index access. Achieving this is difficult as such optimizations should be general in order to be efficient on any processing platform.

Once an execution plan is built, the multi-platform task optimizer passes it to the `Executor` (see Figure 1) for: (i) scheduling the resulting execution plan on the selected data processing frameworks, (ii) monitoring the progress of plan execution, (iii) coping with failures, and (iv) aggregating and returning results to users.

### 4.3 Platform-Layer optimizations

Once at a target processing platform, we envision a third optimization phase that uses plugged-in platform-specific optimization tools. For instance, if the selected platform for a task atom is Hadoop, we could further optimize an execution plan by using Starfish [14]. Notice that the data processing platform itself can also perform some additional optimizations, *e.g.*, if an execution plan is given as input to Spark in the form of a Shark query [26].

## 5. APPLICATIONS

Clearly, a large number of applications benefit from our vision. As a proof of concept, we present here a data cleaning application we developed using part of RHEEM’s vision, mainly platform independence. We are currently developing two other applications: a machine learning application and a graph processing application.

### 5.1 Data Cleaning in RHEEM

**Demand.** Ensuring high quality data is challenging because of the variety of data dirtiness, such as typos, duplicates, and missing values. However, detecting errors is a combinatorial problem that quickly becomes expensive with the size of the data, thus limiting the applicability of cleaning systems.

**Our solution.** We built BIGDANSING, a Big Data Cleansing on top of RHEEM [19]. The two distinct features of BIGDANSING are its *ease-of-use* and *high scalability*; both natural consequences of the RHEEM abstraction vision. BIGDANSING models data quality rules with five operators, namely `Scope`, `Block`, `Iterate`, `Detect`, and `GenFix`. These operators allow users to capture the semantics

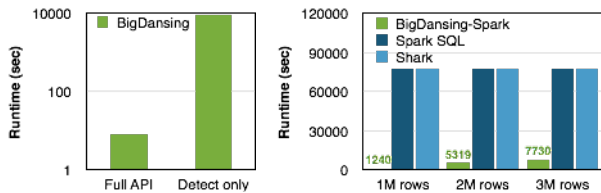


Figure 3: RHEEM execution times for violations detection.

of error detection and possible repairs generation at the application layer (see [19] for details).

**Ease-of-use.** The developer of the application has to come up with the physical plan of the cleaning process (preferably via an automatic optimization process). The detection part of BIGDANSING is composed of a sequence of five physical operators, which are fed with the logics coming from the corresponding logical operators. Similarly, the logical operators require only a few lines of code [19], allowing for ease-of-use.

**High scalability.** Figure 3 shows a comparison of the performance for a single Detect UDF versus our operators for the same task. The left-side subfigure clearly shows the benefits of the abstraction with operators that enables finer granularity for the distributed execution. The right-side subfigure shows a comparison of BIGDANSING against state-of-the-art approaches on Spark. We observe that RHEEM enables orders of magnitude better performance than baselines, which we had to stop after 22 hours. Here, as an example of extensibility, we extended the set of physical RHEEM operators with a new join operator (called IEJoin [20]) to boost performance.

## 5.2 When to Use RHEEM?

It should be clear at this point how the proposed three layers enable better performance and more freedom in developing applications with respect to existing solutions. However, there is a trade-off. A developer who decides to use RHEEM instead of one of the alternative systems may need to implement new operators as required by the target application. This is because our pool of default physical operators is not as exhaustive as the operators provided by the specific underlying platforms. For example, we had to implement the IEJoin operator in RHEEM to boost the performance of our data cleaning application. While this may require extra effort from a developer, we believe that this a reasonable price to pay for platform independence when performance is crucial.

## 6. DATA STORAGE ABSTRACTION

So far we have focused our attention on the data processing side of our vision. Symmetrically to the data processing, we envision a data storage abstraction to provide interoperability among different data storage platforms.

Figure 4 illustrates the RHEEM data storage abstraction. Each layer of abstraction has a set of operators (*i.e.*, UDFs): logical operators (*l-store*) at the application layer, physical operators (*p-store*) at the core layer, and execution operators (*x-store*) at the platform layer. At the application layer different storage applications, *e.g.*, Dropbox, or data processing platforms, *e.g.*, Hadoop or PostgreSQL, output a physical storage plan in a homogeneous format defined by RHEEM. Then, at the core layer, RHEEM takes the physical storage plan as input and produces an optimized execution storage plan. Finally, at the platform layer, a data storage platform stores or accesses a dataset according to the execution storage plan. Note that an execution storage plan is composed of *storage atoms*, *i.e.*, the counterpart of task atoms, which are processed by a different data storage platform. It is worth noting that while a data quantum is the data unit itself (*e.g.*, a tuple), a storage atom is the

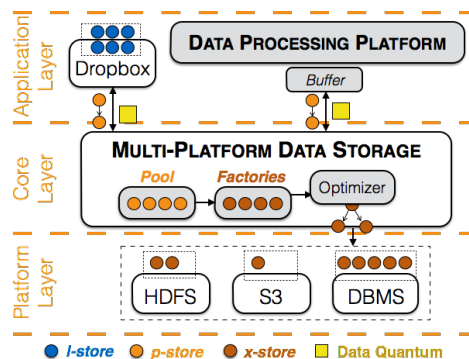


Figure 4: RHEEM data storage abstraction.

minimum unit of data quanta transformation (*e.g.*, projection).

The benefit of such a data storage abstraction is twofold. First, it provides interoperability across storage platforms to simplify users specification about how to store or transform their datasets from one platform to another. Second, it offers opportunities to fully optimize a data flow in order to further improve the performance of data processing tasks. Still, two main challenges make this problem hard: (i) how to unify the abstraction for data storage and access over multiple platforms, and (ii) how to seamlessly decide where and how to store data. Our current efforts into this direction include Cartilage [18], which is a unified data storage representation. In summary, Cartilage introduces the notion of data transformation plans, analogous to logical query plans, that specify a sequence of data transformations that should be applied to raw data as it is uploaded into a storage system. This allows for intermediate storage optimizations based on users and applications needs. For example, WWHow! [17] is a first effort for a unified data storage optimizer.

*Embracing hot data.* Accessing data through a unified data storage might degrade the performance of processing platforms because the data might not be in the required format. Thus, we envision processing platforms or storage applications with specialized buffers for embracing frequently accessed data in their native format.

## 7. RELATED WORK

The closest work to us is Musketeer [12], which provides an intermediate representation between applications and data processing platforms. While Musketeer has the merit of proposing an optimizer for the supported applications and platforms, it considers neither the costs of data movement across processing platforms nor the fact that multiple platforms may be able to perform the same job. Furthermore, it lacks the extensibility that we advocate with our proposal. In fact, only Musketeer developers can integrate new processing platforms or applications. This is in fact similar to integrating a new storage system on an existing processing platform, such as Spark or Hadoop MapReduce. In contrast, in RHEEM, users can achieve these tasks with mappings and new physical operators.

DBMS<sup>+</sup> [21] is another work that aims at embracing several processing and storage platforms for declarative processing. However, DBMS<sup>+</sup> is not adaptive and extensible as it is limited by the expressiveness of its declarative language. Furthermore, it is unclear how it abstracts underlying platforms seamlessly. BigDAWG [11] has recently been proposed as a federated system that enables users to run their queries over multiple vertically-integrated systems such as column stores, NewSQL engines, and array stores. As a result, users can leverage the advantages of each of them. However, users explicitly specify the underlying platforms (called islands) on which their queries must run on. This implies that users need to know how to divide their queries into subqueries and which underlying platform is best suited for each of them.

Other groups have been working on a general platform for big data analytics [5, 7–9, 27, 29]. For example, AsterixDB [5] offers an open data model, native data storage and indexing, declarative querying over multiple datasets, and a rule-based optimizer. SimSQL [10] compiles SQL queries into Java code that can run on top of Hadoop. Moreover, users can use UDFs to materialize views with simulated data, which enables a range of applications requiring stochastic analytics. PACTs [7] extends the MapReduce programming model with second-order functions on top of Nephele, a processing platform that RHEEM can also use as underlying platform. However, none of the above systems provides the multi-platform data processing and storage we propose with RHEEM.

## 8. ROAD TO FREEDOM

*“I have walked that long road to freedom. I have tried not to falter; I have made missteps along the way. But I have discovered the secret that after climbing a great hill, one only finds that there are many more hills to climb... and I dare not linger, for my long walk is not yet ended.”*

– Nelson Mandela –

Oftentimes, users are confronted with the hard decision to choose the right processing platform given the requirements of their analytic application. In addition, their data, born out of various processes, ends up in different storage platforms. To make things worse, the same application may have tasks requiring different platforms to be performed efficiently. Thus, there is a real urgency to free both users and data from (i) being tied to a specific platform, either for processing or storage, and (ii) going through the pain of moving from one platform to another, depending on the requirements of their applications and the characteristics of their data. While the *road to freedom* is full of challenges, RHEEM data processing and storage abstractions hold promise to achieve this freedom. As a case in point, a data cleaning application [19] is our first success towards this goal. IEJoin [20] also showcases the extensibility of RHEEM. While we have laid down the basic ideas on how to build RHEEM, many challenges remain to be addressed.

**(1) Extensibility.** *How to adapt to extensions and improvements in a data processing platform without requiring the developers to go into the source code? What is the right language to provide hints to the optimizer?* We envision an optimization process based on a flexible data model, such as RDF. Developers will specify mappings between operators as well as encode rule- and cost-based models in RDF triples. The optimizer will use this RDF representation as a first-class citizen in its optimization process.

**(2) Multi-platform optimization.** *How to divide a task into atoms, assign the best platform to each atom, and combine results?* We envision a solution based on data processing profiles and inter-platform cost models. A data processing profile denotes the type of data processing a platform can support, e.g., batch-processing profile for Hadoop. An inter-platform cost model will capture different multi-platform aspects, such as the cost of transferring and transforming data from one processing platform to another.

**(3) Unified storage abstraction.** *How to provide a unified abstraction for data storage and access for multiple storage platforms? How to decide where and how to store data?* We envision a three-layer abstraction as discussed in Section 6. This abstraction will enable storage platforms with specialized data buffers to embrace frequently accessed data in their native format.

In summary, the above challenges can be categorized into five main research themes: (i) processing and storage abstractions, (ii) platform-independent task specification, (iii) multi-platform optimization, (iv) multi-platform execution, and (v) data storage and data movement optimizations.

## 9. REFERENCES

- [1] Apache Mahout. <http://mahout.apache.org/>.
- [2] Spark MLlib. <http://spark.apache.org/mllib/>.
- [3] Spark SQL. <http://spark.apache.org/sql/>.
- [4] Powering Big Data at Pinterest. Interview with Krishna Gade. <http://goo.gl/UMGSvy>, April 2015.
- [5] S. Alsubaiee et al. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14), 2014.
- [6] A. Baaziz and L. Quoniam. How to use big data technologies to optimize operations in upstream petroleum industry. In *21<sup>st</sup> World Petroleum Congress*, 2014.
- [7] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *SoCC*, 2010.
- [8] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: a flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.
- [9] F. Bugiotti, D. Bursztyn, A. Deutsch, I. Ileana, and I. Manolescu. Invisible Glue: Scalable Self-Tuning Multi-Stores. In *CIDR*, 2015.
- [10] Z. Cai, Z. Vagena, L. L. Perez, S. Arumugam, P. J. Haas, and C. M. Jermaine. Simulation of database-valued markov chains using simsql. In *SIGMOD*, 2013.
- [11] A. Elmore et al. A Demonstration of the BigDAWG Polystore System. In *VLDB 2015 (demo)*, 2015.
- [12] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand. Musketeer: All for One, One for All in Data Processing Systems. In *EuroSys*, 2015.
- [13] A. Hems, A. Soofi, and E. Perez. How innovative oil and gas companies are using big data to outmaneuver the competition. Microsoft White Paper, <http://goo.gl/2Bn0xq>, 2014.
- [14] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11), 2011.
- [15] IBM. Data-driven healthcare organizations use big data analytics for big gains. White paper, <http://goo.gl/AFIHpk>.
- [16] E. Jahani, M. J. Cafarella, and C. Ré. Automatic Optimization for MapReduce Programs. *PVLDB*, 4(6):385–396, 2011.
- [17] A. Jindal, J. Quiané-Ruiz, and J. Dittrich. WWHow! Freeing Data Storage from Cages. In *CIDR*, 2013.
- [18] A. Jindal, J. Quiané-Ruiz, and S. Madden. CARTILAGE: adding flexibility to the hadoop skeleton. In *SIGMOD*, 2013.
- [19] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzanni, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and S. Yin. BigDancing: A System for Big Data Cleansing. In *SIGMOD*, 2015.
- [20] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and P. Kalnis. Lightning Fast and Space Efficient Inequality Joins. *PVLDB*, 8(13), 2015.
- [21] H. Lim, Y. Han, and S. Babu. How to Fit when No One Size Fits. In *CIDR*, 2013.
- [22] A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann. SOFA: An extensible logical optimizer for UDF-heavy data flows. *Inf. Syst.*, 52:96–125, 2015.
- [23] M. Stonebraker and U. Çetintemel. “One Size Fits All”: An Idea Whose Time Has Come and Gone (Abstract). In *ICDE*, 2005.
- [24] 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), 2009.
- [25] K. Tzoumas, J.-C. Freytag, V. Markl, F. Hueske, M. Peters, M. Ringwald, and A. Krettek. Peeking into the Optimization of Data Flow Programs with MapReduce-style UDFs. In *ICDE*, 2013.
- [26] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *SIGMOD*, 2013.
- [27] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *OSDI*, 2008.
- [28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud’10*, 2010.
- [29] J. Zhou et al. SCOPE: Parallel Databases Meet MapReduce. *VLDB*, 21(5), 2012.



# Data Management for Next Generation Genomic Computing

Stefano Ceri, Abdulrahman Kaitoua, Marco Masseroli, Pietro Pinoli, Francesco Venco

DEIB, Politecnico di Milano  
Piazza L. da Vinci 32, 20133 Milano  
first.last@polimi.it

## ABSTRACT

Next-generation sequencing (NGS) has dramatically reduced the cost and time of reading the DNA. Huge investments are targeted to sequencing the DNA of large populations, and repositories of well-curated sequence data are being collected. Answers to fundamental biomedical problems are hidden in these data, e.g. how cancer arises, how driving mutations occur, how much cancer is dependent on environment. So far, the bio-informatics research community has been mostly challenged by *primary analysis* (production of sequences in the form of short DNA segments, or "reads") and *secondary analysis* (alignment of reads to a reference genome and search for specific features on the reads); yet, the most important emerging problem is the so-called *tertiary analysis*, concerned with multi-sample processing of heterogeneous information. Tertiary analysis is responsible of *sense making*, e.g., discovering how heterogeneous regions interact with each other.

This new scenario creates an opportunity for rethinking genomic computing through the lens of fundamental data management. We propose an essential data model, using few general abstractions that guarantee interoperability between existing data formats, and a new-generation query language inspired by classic relational algebra and extended with orthogonal, domain-specific abstractions for genomics. They open doors to the seamless integration of descriptive statistics and high-level data analysis (e.g., DNA region clustering and extraction of regulatory networks). In this vision, computational efficiency is achieved by using parallel computing on both clusters and public clouds; the technology is applicable to federated repositories, and can be exploited for providing integrated access to curated data, made available by large consortia, through user-friendly search services. Our most far-fetching vision is to move towards an *Internet of Genomes* exploiting data indexing and crawling.

## Categories and Subject Descriptors

H.2.1 [Logical design]: Data models; H.2.3 [Languages]: Query languages; H.2.8 [Database applications]: Scientific databases.

## Keywords

Genomic data management

## 1. INTRODUCTION

Modern genomics promises to answer fundamental questions for biological and clinical research, e.g., how protein-DNA interactions and DNA three-dimensional conformation affect gene activity, how cancer develops, how driving mutations occur, how much complex diseases such as cancer are dependent on personal

© 2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

genomic traits or environmental factors. Unprecedented efforts in genomics are made possible by **Next Generation Sequencing (NGS)**, a family of technologies that is progressively reducing the cost and time of reading the DNA. Huge amounts of sequencing data are continuously collected by a growing number of research laboratories, often organized through world-wide consortia (such as ENCODE [1], TCGA [2], the 1000 Genomes Project [3], and Epigenomic Roadmap [4]); personalized medicine based on genomic information is becoming a reality.

Several organizations are considering genomics at a global level. Global Alliance for genomics and Health<sup>1</sup> is a large consortium of over 200 research institutions with the goal of supporting voluntary and secure sharing of genomic and clinical data; their work on data interoperability is producing a data conversion technology<sup>2</sup> recently provided as an API to store, process, explore, and share DNA sequence reads, alignments, and variant calls, using Google's cloud infrastructure<sup>3</sup>. Parallel frameworks are used to support genomic computing, including Vertica<sup>4</sup> (used by Broad Institute and NY Genome Center) and SciDB<sup>5</sup> (used by NCBI for storing the data of the 1000 Genomes project [3]). A survey of current challenges in computational analysis of genomic big data can be found in [5]. According to many biologists, answers to crucial genomic questions are hidden within genomic data already available in these repositories, but such research questions go simply unanswered (or even unasked) due to the lack of suitable tools for genomic data management and processing.

So far, the bio-informatics research community has been mostly challenged by **primary analysis** (production of sequences in the form of short DNA segments, or "reads") and **secondary analysis** (alignment of reads to a reference genome and search for specific features on the reads, such as variants/mutations and peaks of expression); but the most important emerging problem is the so-called **tertiary analysis**, concerned with multi-sample processing, annotation and filtering of variants, and genome browser-driven exploratory analysis [6]. While secondary analysis targets *raw data* in output from NGS processors by using specialized methods, tertiary analysis targets processed data in output from secondary analysis and is responsible of *sense making*, e.g., discovering how heterogeneous regions interact with each other (see Figure 1).

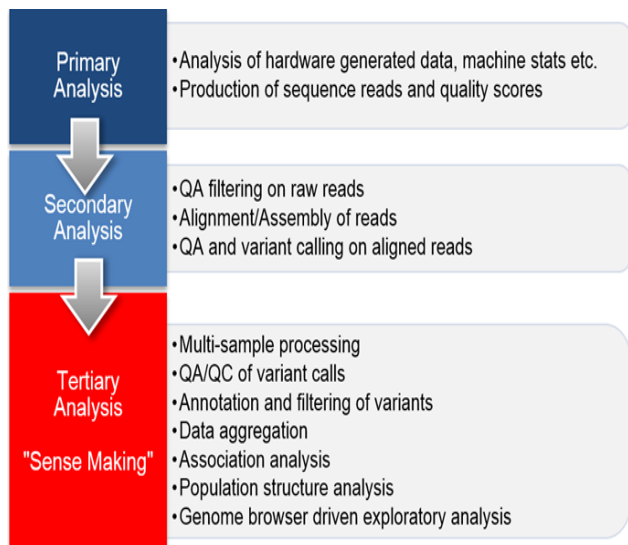
<sup>1</sup> <http://genomicsandhealth.org/>

<sup>2</sup> <http://ga4gh.org/#/api>

<sup>3</sup> <https://cloud.google.com/genomics/>

<sup>4</sup> <https://www.vertica.com/>

<sup>5</sup> <http://www.paradigm4.com/>



**Figure 1. Phases of genomic data analysis, source:**  
<http://blog.goldenhelix.com/grudy/a-hitchhiker%E2%80%99s-guide-to-next-generation-sequencing-part-2/>

Tertiary processing consists of integrating DNA features; these can be specific DNA variations (e.g., a variant or mutation in a DNA position), or signals and peaks of expression (e.g., regions with higher DNA read density). Processing can also give structural properties of the DNA, e.g., break points (where the DNA is damaged) or junctions (where DNA creates loops, and then locations which are distant on the 1D string become close in the 3D space).

While gigantic investments are targeted to sequencing the DNA of larger and larger populations, comparably much smaller investments are directed towards a computational science for mastering tertiary analysis. Bio-informatics resources are dispersed in provisioning a huge number of tools for ad-hoc processing of genomic data, targeted to specific tasks and adapted to technology-driven formats, with little emphasis on powerful abstractions, format-independent representations, and out-of-the-box thinking and scaling. Programming data manipulation operations directly in Python or R is customary.

Another source of difficulty comes from “metadata”, which describe DNA region-invariant properties of the biological sample processed by NGS, i.e., the sample cell line, tissue, preparation (antibody used), experimental conditions, and in case of human samples the race, gender, and other phenotype-related traits. This information should be stored in principled data schemes of a “LIMS” (laboratory information management system) and be compliant with standards, but biologists are very liberal in omitting most of it, even in well-cured repositories.

## 2. OUR CONTRIBUTION

Bio-informatics suffers its interdisciplinary nature and is considered by biologists and clinicians as a commodity that should immediately respond to their pressing needs, while it stays too far from foundational science to attract the interest of many core computer scientists. We understood that it is “mission impossible” for basic computer science to have an impact on primary and secondary analysis: algorithms are biologically driven and very specialized and efficient. Hence, we decided not

to interfere with current biologists’ practices, but rather to empower them with radically new data processing capabilities.

We propose a paradigm shift based on introducing a very simple data model which mediates all existing data formats, and a high-level, declarative query language which supports data extraction as well as the most standard data-driven computations required by tertiary data analysis. The **Genomic Data Model (GDM)** is based on just two entities: genomic region and metadata. Regions (upper part of Figure 2) have a normalized schema (i.e., a table of typed attributes) where the first five attributes are fixed and the next attributes are variable and reflect the “calling process” that produced them. The fixed attributes include the sample identifier and the region coordinates (the chromosome whom the region belongs to, its left and right ends, and the strand - i.e., the “+” or “-” of the two DNA strands on which the region is read, and “\*” if the region is not stranded). The model can be adapted to the rare cases of regions across chromosomes. Metadata (lower part of Figure 2) are even simpler. They are arbitrary, semi-structured attribute-value pairs, extended into triples to include the sample identifier. We consider this model a paradigm shift, because a single model describes, though simple concepts, all types of processed data (peaks, signals, mutations, DNA sequences, loops, break points).

ID	(CHR, LEFT, RIGHT, STRAND)	P_VALUE
1	(1, 3245, 4535, +)	0.000024
1	(1, 6340, 7400, -)	0.000053
1	(1, 7540, 8563, -)	0.000013
1	(2, 1440, 2506, -)	0.000034
1	(2, 3540, 4541, +)	0.00006
2	(1, 4020, 5073, *)	0.000017
2	(1, 7020, 8061, *)	0.000035
2	(2, 1020, 3064, *)	0.000016
2	(2, 4020, 4101, *)	0.000022

ID	ATTRIBUTE	VALUE
1	cell	CLL
1	karyotype	cancer
1	tissue	blood
1	sex	M
2	cell	H9ES
2	sex	F
2	tissue	embryonic

**Figure 2. GDM schema and instances for NGS ChIP-Seq data.**

The data model is completed by a constraint: data samples can be included into a named dataset when their genomic regions have the same schema. Thus, the above figure shows the PEAKS dataset for “ChIP-Seq” data with two samples (1 and 2) whose regions fall within two chromosomes (1 and 2) and whose variable part of the schema consists of the attribute P\_VALUE (each peak’s statistical significance). Note that the sample ID provides a many-to-many connection between regions and metadata of the same sample; e.g., sample 1 has 5 regions and 4 metadata attributes, sample 2 has 4 regions and 3 metadata attributes; regions of the first sample are stranded (positively or negatively oriented along the DNA), while regions of the second sample are not stranded. Metadata tell us that sample 1 has karyotype “cancer” and sample 2 was taken from a “female”. This example is simple, but we can associate a schema with arbitrarily complex processed data, where typed and named attributes serve the purpose of any numerical or statistical operation across compatible values. An important operation is the **schema merging**, which allows merging datasets with different schemas (the operation builds a new schema such that fixed attributes are

in common and variable attributes are concatenated; in this way, we provide interoperability across heterogeneous processed data.

We also defined a query language, called **GenoMetric Query Language (GMQL)** - the name derives from its ability of computing distance-related queries along the genome, seen as a sequence of positions. GMQL is a **closed algebra over datasets**: results are expressed as new datasets derived from their operands. Thus, GMQL operations compute both regions and metadata, connected by IDs; they perform schema merging when needed. GMQL operations include classic algebraic transformations (SELECT, PROJECT, UNION, DIFFERENCE, JOIN, SORT, AGGREGATE) and domain-specific transformations (e.g., COVER deals with replicas of a same experiment; MAP refers genomic signals of experiments to user selected reference regions; GENOMETRIC JOIN selects region pairs based upon distance properties). The language brings to genomic computing the classic algebraic abstractions, rooted in Ted Codd's seminal work, and adds suitable domain-specific abstractions. Tracing provenance both of initial samples and of their processing through operations is a unique aspect of our approach; knowing why resulting regions were produced is quite relevant. In [7], we show GMQL at work in many heterogeneous biological contexts.

We give an intuition of GMQL through a simple example, consisting of three operations. We start from two datasets called ANNOTATIONS and ENCODE, the former includes samples with the reference regions from the UCSC database<sup>6</sup>, the latter includes thousands of samples from ENCODE (in BED format); both are available at our server, with both regions and metadata. Two selections are used to produce two intermediate datasets: PROMS extracts from ANNOTATIONS a single sample with all the promoter regions of known genes; PEAKS extracts the samples of type 'ChIPSeq' from ENCODE. Then, a map operation applies to the intermediate datasets PROMS and PEAKS and produces the RESULT dataset. The MAP operation, as well as all GMQL operations, implicitly iterates over all the samples of its operand datasets; it counts, for each input peak sample, all the peaks of expression over each region of PROMS, representing gene promoters. Thus, RESULT contains one output sample for each PEAK input sample, each with all the regions of PROMS; for each of such regions, it has the counter of peaks of the sample which fall within such region. This simple example shows the power of the language: with tree algebraic operations, we select reference regions and experiments and then compute aggregate properties of each experiment over each reference region, with implicit iteration over all the experiment samples.

```
PROMS = SELECT(annType == 'promoter') ANNOTATIONS;
PEAKS = SELECT(dataType == 'ChIPSeq') ENCODE;
RESULT = MAP(peak_count AS COUNT) PROMS PEAKS;
```

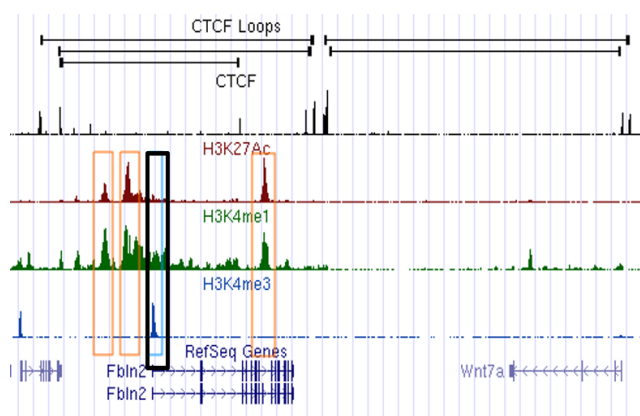
This query above was executed over 2,423 ENCODE samples including a total of 83,899,526 peaks, which were mapped to 131,780 promoters, producing as result 29 GB of data.

### 3. DATA-DRIVEN GENOMIC PROBLEMS

An open problem that we are nowadays studying concerns the search for a correlation of cancer-inducing mutations and DNA string breaks with abnormal gene activity during cell replication

[8], as one of the possible basic mechanisms of cancer. The assumption under consideration is that the abnormal production of DNA string breaks correlates with the presence of mutations (simply explained: mutations occur where the genome is most fragile, fragility is revealed by DNA break points); this in turn may be caused by gene dis-regulation during the process of cell replication (certain genes omit to perform a regulatory function that should prevent mutations during replication, or should fix them afterwards). In this problem, we are therefore confronted with correlating the cell replication with gene regulations; we do it in experimental conditions (exposure of cells to oncogenes), and we study how the induction of the oncogene changes both replication time and expression of other genes. The study requires genome-wide comparison of heterogeneous datasets (breakpoints, mutations, gene replication times and gene expressions under different experimental conditions), challenging both GDM and GMQL, and then calling for specific data analysis; specifically, GMQL can extract differentially dis-regulated genes, intersect them with regions where string breaks occur, and then count the mutations in various conditions.

Another open problem is concerned with the tri-dimensional layout of DNA, which is induced by the chromatin structure revealed by peaks of the CTCF transcription factor, and **understanding how CTCF loops influence gene regulation** [9]; a loop is simply a binding of the DNA, so that two DNA regions which are far away from a 1D perspective become very close from a 3D perspective. In Figure 3, within yellow (thin) rectangles we see three signals which identify three non-coding regions of the genome, called enhancers, and within a black (thick) rectangle we see signals which identify the promoter of the gene Fbln2. They are enclosed within regions which represent short CTCF loops, and the assumption to be tested is whether there is a direct relationship between active enhancers and active genes (where activity is revealed by experiments) when enhancers and promoters are enclosed within CTCF loops (as this spatial condition may favor the enhancer-to-gene relationship); *determining the relationships of genes with enhancers is a fundamental aspect of epigenetics*. Such question corresponds to searching a pattern within the whole genome; GMQL can be used to extract candidate gene-enhancer pairs by suitable intersections of the signals in Figure 3 - i.e., CTCF regions, the regions of the three methylation experiments (H3K27AC, H3K4me1, H3K4me3), and gene promoter regions (from RefSeq).



**Figure 3. Interaction between CTCF loops and gene regulation by enhancers.**

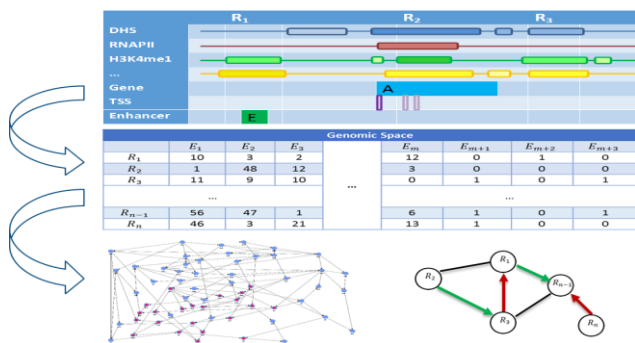
<sup>6</sup> [http://genome.ucsc.edu/cgi-bin/hgTables?hgid=445319346\\_kHaTO493uLRZhuqCvaKTAf7HL3](http://genome.ucsc.edu/cgi-bin/hgTables?hgid=445319346_kHaTO493uLRZhuqCvaKTAf7HL3)

## 4. VISION

GDM and GMQL open new scenarios in approaching tertiary analysis of genomic data. We next discuss them.

### 4.1 Data Analysis

Data analysis methods which are most useful for genomic computing can be bridged to the high-level language, with a bottom-up, problem driven approach. In particular, query results can be expressed in the form of interaction networks between genomic regions. In biology, many genes are involved in complex regulatory processes; for us, genes are just DNA regions of a specific sample (they are “known annotations of the genome”) and thus we can MAP (using the domain-specific operation shown in the example of Section 2) arbitrary experiments to genes. MAP is the first transformation in Figure 4, which computes aggregates over those regions of regions of experiments that intersect with genes (represented by regions  $R_1, R_2, R_3$ ). In general, every map operation produces what we call a **genome space**, i.e., a tabular space of regions vs. experiments (in the middle of Figure 4), which is the starting point for data analysis (including advanced data mining and computational intelligence). Such table can be also interpreted as an adjacency matrix representing a network, where regions are nodes and arcs have a weight obtained by further aggregating properties across experiments; thus, the second transformation in Figure 4 yields to a **gene network**, producing as well the strength of gene-to-gene interactions. The interpretation of genome spaces in the form of networks is particularly important in genomics, as regulatory gene activities typically depend on multiple interacting genes.



**Figure 4. Interpretation of GMQL “map” query as a genome space, and further transformation of the genome space into a gene network.**

Several data mining and computational intelligence approaches, including advanced *latent semantic analysis* and *topic modelling*, can be applied to evaluate relationships among genomic data, and between them and biological or clinical features of experimental samples expressed in their metadata, i.e., for *genotype-phenotype* correlation analysis.

### 4.2 Distributed Processing and Cloud Computing

With the growth of NGS experiments (whose cost is expected to drop to about 100 Euro in less than a decade, <https://www.genome.gov/sequencingcosts/>), we will see a deluge of NGS data. Although processed data are “smaller” than raw data (0.3 TB per full genome sample), we are still talking of samples with tens of thousands or even millions of regions. Genomic

repositories store thousands of full genome samples (i.e., 4,660 samples in [1], 5,400 samples in [2], and 2,500 samples in [3]). Our simple query in Section 3 produced 83 million regions, and simple queries over genes may produce genome spaces of 10K genes and 100M relationships between them, whose analysis requires using large-scale network management packages. Moreover, NGS is increasingly used for massive testing on restricted, pathology-specific mutation panels, so as to accelerate the use for diagnostics and for clinics. We are clearly facing one of the **most important “big data” problems for mankind**.

We are currently working towards a new GMQL release, that will be available in 2016, and will support two parallel implementations, respectively using Flink<sup>7</sup> and Spark<sup>8</sup>, two emerging data frameworks. In our architecture, the two implementations differ only in the encoding of about twenty GMQL language components, while the compiler, logical optimizer, and APIs/UIs are independent from the adoption of either framework. In a recent paper [10] we present an early comparison of Flink and Spark at work on three genomic queries inspired by GMQL. Several tools were developed within the Hadoop framework for primary and secondary analysis, including BioPig [11], SeqPig [12] and SparkSeq [13]. Our preliminary work shows open source frameworks are effective computing systems also for tertiary data analysis; we foresee a growth of systems for genomic based upon parallel computing frameworks.

So far, our focus on tertiary data analysis is shared just by Paradigm4, a startup company founded by the Turing award Mike Stonebraker, whose products include genomic add-ons to SciDB, a vector-based data management system for scientific applications. They provide access to data from TCGA and 1000 Genomes Project, and they advocate the use of specialized databases for scientific computing rather than cloud computing – indeed, we find in [6] several arguments against the use of Spark. We expect that the alternative between open frameworks and specialized systems will shape the evolution of genomic data management in the forthcoming years.

### 4.3 Integrated Access to Repositories

Very large-scale sequencing projects are emerging; as of today, the most relevant ones include:

- The **Encyclopedia of DNA elements (ENCODE)** [1], the most general and relevant world-wide repository for basic biology research. It provides public access to more than 4,000 experimental datasets, including the just released data from its Phase 3, which comprise hundreds of epigenetic experiments of processed data in human and mouse;
- The **Cancer Genome Atlas (TCGA)** [2], a full-scale effort to explore the entire spectrum of genomic changes involved in human cancer;
- The **1000 Genomes Project** [3], aiming at establishing an extensive catalogue of human genomic variations from 26 different populations around the globe;
- The **Epigenomic Roadmap Project** [4], a repository of “normal” (not involved in diseases) human epigenomic data from NGS processing of stem cells and primary ex vivo tissues.

<sup>7</sup> <https://flink.apache.org/>

<sup>8</sup> <https://spark.apache.org/>

Data collected in these projects are open and public; all the Consortia release both raw and processed data, but biologists in nearly all cases trust the processing, which is of high-quality and well controlled and explained. All consortia provide portals for data access; some systems already provide integrated access to some of them (e.g., [14]; see also <http://www.paradigm4.com/>). The use of a high-level model and language, such as GDM and GMQL, is the ideal setting for provisioning next generation services over data collected and integrated from these and other repositories, improving over the current state-of-the-art in four directions:

- All the processed datasets available in the above data sources will be provided of compatible metadata;
- It will be possible to choose among a set of custom queries, representing the typical/most needed requests;
- It will be possible to provide user input samples to the services, whose privacy will be protected;
- Deferred result retrieval will be possible, through limited amount of staging at the sites hosting the services.

A simple protocol will facilitate input and output file transmissions and it will also be possible to visualize results on genome browsers or to selectively retrieve regions or metadata. Users will be enabled to write personalized queries, whose privacy will be protected. The main challenges in this vision include two new research objectives: the mediation of ontological knowledge and the statistical description of custom queries.

- Ontological reasoning will be required in order to establish the appropriate **conceptual relationships between the metadata** which are present at the various sources. The best option is to use the global ontology provided by the Unified Medical Language System (UMLS) [15], which collects and integrates well-established biomedical ontologies. Our initial solution, presented in [16], consists in semantically annotating the metadata of each repository's datasets by means of UMLS, and completing the information by performing the semantic closure [17] of such annotations. Then, a suitable UI would allow users to search for relevant experiments through keyword-based or free text queries.
- **Custom queries** will need to be augmented with suitable mechanisms for reasoning about data; such services could imitate the *Great* service developed by Gill Bejerano's group at Stanford [18], which includes powerful statistics to indicate the significance of query results.

#### 4.4 Federated Query Processing and Protocols

The availability of a core data model as a data interoperability solution and of a high-level data processing language is a strong prerequisite for defining data exchange protocols. We expect that each data repository will be the owner of the data that are locally produced, and that nodes of cooperating organizations will be connected to form a federated database. In such systems, queries move from a requesting node to a remote node, are locally executed, and results are communicated back to the requesting node; this paradigm allows for distributing the processing to data, transferring only query results which are usually small in size.

Supporting a high-level query interface to a server is already making one big step forward, which is similar to the gigantic step made by SQL in the context of client-server architectures (which

dates a couple of decades). Indeed, once a system supports an API for submitting GMQL queries, these have the following properties: they are short texts and produce short answers. This comes from the nature of problems: the more they are biologically inspired, the more they produce results which are both short and ranked, and these will eventually be transmitted along any GMQL API; in contrast, most of today's implementations requires first a full data transmission and then to evaluate server-side imperative programs. This scenario opens up to the design of simple interaction protocols, typically for:

- **Requesting information about remote datasets**, facilitated by the availability of metadata (for locating data of interest) and of their region schemas (for formalizing queries).
- **Transmitting a query in high-level format and obtain data about its compilation**, not only limited to correctness, but including also estimates of the data sizes of results.
- **Launching query execution and then controlling the transmission of results**, so as to be in control of staging resources and of communication load.

#### 4.5 Search Methods and Internet of Genomes

After having provided access to integrated sources of sequence data, we come to the question of how such knowledge can be searched. The problem can be approached progressively, starting first with opening search services over the integrated repositories. There are two intertwined problems:

- **Metadata search.** Search methods should locate relevant samples within very large bodies, using classical measures of precision and recall; keyword-based search or free text querying should be supported.
- **Feature-based region search.** Best-matching regions with user-specified features should be provided. For some regions (e.g., known genes) it is possible to define a priori the typical features, store them as attributes, and then use indexing; but in general features should be computed. We envision general search mechanisms where the user selects interesting regions, then provides information about the features of interest, then those features are computed, and finally regions are ordered based on their computed features and presented to the user. So, search and feature evaluation have to intertwine in a clever way.

The most ambitious and challenging vision is **building a search system upon an Internet of genomes**. The prerequisite to this vision is of course not in today's reach, and requires all research centers to agree on a deployment technology playing the role of HTML and HTTP for the Web. However, biologists are forced to publish the data which go together with their experiments: it is already in their practice to provide a link to a download site where experimental data should be available for downloading by reviewers. In such context, it is possible to envision the definition of a simple protocol for data publishing, prescribing how to publish a link to genomic data in their native format with suitable metadata; the protocol should offer the possibility of making such link public, i.e., visible within a host open to the visits of crawlers. With such infrastructure, a third party hosting a search service could periodically launch the crawlers, and these would download the metadata and links from the host; the search service could also download datasets from the hosts by using those links, with an agreed, non-intrusive protocol. The search service would then have all the required information for indexing all the

metadata and for storing some of the samples within a large repository, possibly pre-computing some features of their regions. Such search system could accept search queries and produce result snippets, with an indication of the presence of each dataset in the repository. In any case, users of the search system would be able to locate genomic data available at another host (a research or clinical center) and could download them asynchronously.

## 5. CONCLUSIONS

The progress in DNA and RNA sequencing technology has been so far coupled with huge computational efforts in primary and secondary genomic data management, consisting of producing “raw” data, aligning them to the reference genomes, and calling for specific features such as expression peaks and mutations. However, a new pressing need is emerging: making sense of data produced by these methods, in the so-called tertiary analysis. This need requires a substantial change of the dominating approach to bio-informatics. While primary and secondary analyses produce data formats which are typically intricate and incompatible, tertiary analysis must worry about their interoperability and ease of use. Tertiary analysis calls for raising the level of abstractions of models, languages and tools for genomics, going towards a broader vision where biologists and clinicians can observe the huge and complex body of genomic knowledge at a much higher level, using simple interfaces similar to search queries which have become widely available in the Internet.

In this paper, we have shown that a change of paradigm is possible, by means of a new data model and query language; we have then shown the biological applications that have become feasible thanks to this approach, and examined the relevant advantages that this approach may bring in the contexts of data analysis, distributed processing, integrated repository access, federated data management, and search of genomic data over the Internet. The corresponding scenario traces a five-to-ten year research trajectory.

## 6. ACKNOWLEDGEMENTS

We acknowledge the essential contributions of Pier Giuseppe Pelicci, Ivan Dellino, Lucilla Luzi, Laura Riva, Mattia Pelizzola (IEO), of Heiko Muller (IIT) and of Michele Bertoni, Gianpaolo Cugola, Vahid Yalili, Matteo Matteucci, Fernando Paluzzi (PoliMI) in developing our approach to genomics. This research is supported by the PRIN Project GenData 2020 and by a grant from Amazon Web Services.

## 7. REFERENCES

- [1] ENCODE Project Consortium. 2012. An integrated encyclopedia of DNA elements in the human genome. *Nature* 489(7414), 57-74.
- [2] Weinstein, J. N., et al. 2013. The Cancer Genome Atlas pan-cancer analysis project. *Nat. Genet.* 45(10), 1113-1120.
- [3] 1000 Genomes Project Consortium. 2010. A map of human genome variation from population-scale sequencing, *Nature* 467, 1061-1073.
- [4] Romanoski, C. E., et al. 2015. Epigenomics: Roadmap for regulation. *Nature* 518, 314-316.
- [5] Qin, Y., et al. 2015. The current status and challenges in computational analysis of genomic big data. *Big Data Research* 2(1), 12-18.
- [6] Accelerating bioinformatics research with new software for big data knowledge. White paper retrieved from: <http://www.paradigm4.com/>, April 2015.
- [7] Masseroli, M., Pinoli, P., Venco, F., Kaitoua, A., Jalili, V., Paluzzi, F., Muller, H., Ceri, S. 2015. GenoMetric Query Language: A novel approach to large-scale genomic data management. *Bioinformatics* 12(4), 837-843.
- [8] Dellino, G. I., et al. 2013. Genome-wide mapping of human DNA-replication origins: Levels of transcription at ORC1 sites regulate origin selection and replication timing. *Genome Res.* 23(1), 1-11.
- [9] Handoko, L., et al. 2011. CTCF-mediated functional chromatin interactome in pluripotent cells. *Nat. Genet.* 43(7), 630-638.
- [10] Bertoni, M., Ceri, S., Kaitoua, A., Pinoli, P. 2015. Evaluating cloud frameworks on genomic applications. *IEEE Conference on Big Data Management*, Santa Clara, CA.
- [11] Nordberg, H., et al. 2013. BioPig: a Hadoop-based analytic toolkit for large-scale sequence data. *Bioinformatics* 29(23), 3014-3019.
- [12] Schumacher, A., et al. 2014. SeqPig: simple and scalable scripting for large sequencing data sets in Hadoop. *Bioinformatics* 30(1), 119-120.
- [13] Weiwiorka, M. S., et al. 2014. SparkSeq: Fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision. *Bioinformatics* 30(18), 2652-2653.
- [14] Nielsen, C. B., et al. 2012. Spark: A navigational paradigm for genomic data exploration. *Genome Res.* 22(11), 2262-2269.
- [15] Bodenreider, O. 2004. The Unified Medical Language System (UMLS): Integrating biomedical terminology. *Nucleic Acids Res.* 32(Database issue), D267-D270.
- [16] Fernandez, J. D., Lenzerini, M., Ceri, S., Masseroli, M., Venco, F. 2016. Ontology-based search of genomic metadata. *IEEE/ACM Trans. Comput. Biol. Bioinform.* (in press).
- [17] Kontchakov, R., Lutz, C., Toman, D., Wolter, F., Zakharyashev, M. 2010. The combined approach to query answering in *DL-Lite*. *Int. Conf. Knowledge Representation and Reasoning*, 247-257.
- [18] McLean, C. Y., et al. 2010. GREAT improves functional interpretation of cis-regulatory regions. *Nat. Biotechnol.* 28(5), 495-501.

# Exploring Text Classification for Messy Data: An Industry Use Case for Domain-Specific Analytics

## Industrial Paper

Laura B. Kassner  
Daimler AG  
71059 Sindelfingen  
laura\_bernadette.kassner@daimler.com

Bernhard Mitschang  
Institut für Parallele und Verteilte Systeme  
Universitätsstraße 38  
70569 Stuttgart  
bernhard.mitschang@ipvs.uni-stuttgart.de

### ABSTRACT

Industrial enterprise data present classification problems which are different from those problems typically discussed in the scientific community – with larger amounts of classes and with domain-specific, often unstructured data. We address one such problem through an analytics environment which makes use of domain-specific knowledge. Companies are beginning to use analytics on large amounts of text data which they have access to, but in day-to-day business, manual effort is still the dominant method for processing unstructured data. In the face of ever larger amounts of data, faster innovation cycles and higher product customization, human experts need to be supported in their work through data analytics. In cooperation with a large automotive manufacturer, we have developed a use case in the area of quality management for supporting human labor through text analytics: When processing damaged car parts for quality improvement and warranty handling, quality experts have to read text reports and assign error codes to damaged parts. We design and implement a system to recommend likely error codes based on the automatic recognition of error mentions in textual quality reports. In our prototypical implementation, we test several methods for filtering out accurate recommendations for error codes and develop further directions for applying this method to a competitive business intelligence use case.

### Categories and Subject Descriptors

H.3.1 [Content Analysis and Indexing]: Linguistic Processing; H.3.3 [Information Search and Retrieval]: Information Filtering; H.4.2 [Information Systems Applications]: Types of Systems—*Decision Support*; J.1 [Administrative Data Processing]: Manufacturing

©2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

### General Terms

Text-Based Classification, Domain-Specific Semantic Resources

### Keywords

recommendation system, automotive, text analytics, domain-specific language, automatic classification

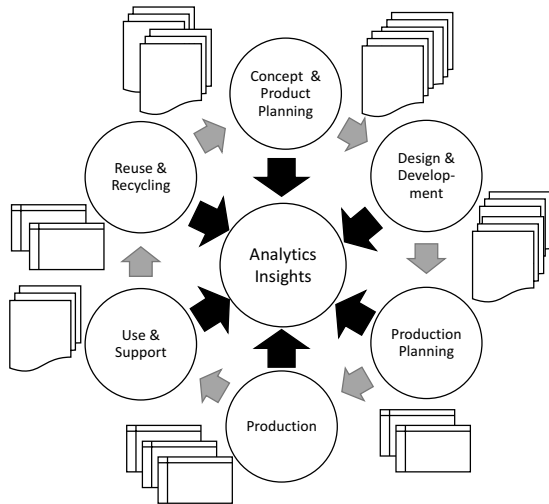
## 1. INTRODUCTION

Analytics and automatized processing of unstructured data to support business processes and decisions have become a topic of interest for the research community and the enterprise world in recent years only [15]. Companies are realizing that valuable knowledge can be gained especially from the large amounts of unstructured text data which they are storing internally and able to access publicly on the social web. In past decades, this knowledge was only accessible to the human mind. The means for storing and processing large amounts of data and scalable text analytics or cognitive computing tools [5] are both relatively new developments.

In cooperation with a large automotive original equipment manufacturer (OEM), we have developed an overarching use case in the area of quality management, with a concrete example for supporting human labor through text analytics. This use case presents a particularly challenging classification problem with several hundred possible classes and mainly unstructured text data as input. We develop a modular environment for classifying these data with the help of natural language processing. In this paper, our main points are (1) a first proof of concept for a “messy” industrial data source and (2) the investigation of the usefulness of a legacy domain-specific resource in the context of a new analytics task. To this end, we evaluate various adaptations of an established classification algorithm in order to customize it to the given situation.

### 1.1 Motivation

Industrial enterprises are generating and collecting large amounts of unstructured text data. These data are often highly domain-specific. Most current approaches to automatically analyzing these unstructured data with traditional



**Figure 1: Data Analytics around the Product Life Cycle**

analytics for structured data are either very specific and case-based or too generic [11]. What is needed is a flexible framework for analytics with re-usable and modular resources and analytics toolboxes.

Enterprises are beginning to transition to more widespread and streamlined use of text analytics. But in day-to-day business, a lot of manual effort is still used to process unstructured data, for example in quality management or customer relationship management. This effort is important because these data contain value-adding knowledge, and human expertise cannot and should not be fully replaced in these tasks. But in the face of ever larger amounts of data, faster innovation cycles and higher product customization especially in the manufacturing industry, human experts need to be better supported through data analytics. Beyond the support of current manual analytics tasks, data analytics can also provide important novel business insights (cf. Figure 1).

Domain-specific resources, for example taxonomies of domain-specific languages, are often developed for a single case-based analytics scenario and rarely re-used for others. This leads to inefficiencies and loss of valuable knowledge. Related research on this topic [11, 12] has proposed requirements and an architectural paradigm for flexible, re-usable and value-adding analytics software. Within this framework we develop a proof of concept for a toolbox using legacy code and semantic resources, unstructured data from several sources, and modular, tailored text analytics.

## 1.2 Contribution and Outline

When processing damaged car parts for quality improvement and warranty handling, quality experts have to read large numbers of text reports from various sources and assign error codes to damaged parts from a large pool of options. We investigate methods to handle these unstructured text data analytically and to support this labor through automatic recommendation of likely error codes, which is a specialized application of automatic classification.

This use case is different from typical classification problems in several respects: The amount of potential classes is

larger (several hundred), and the data to be classified are short text reports which we term “messy data”: Text which consists of non-standard, domain-specific language, riddled with spelling errors, idiosyncratic and non-idiomatic expressions and OEM-internal abbreviations.

We design and implement the Quality Engineering Support Tool QUEST with an included Quality Analytics Toolkit (QATK), a system to recommend likely error codes based on the automatic recognition of error mentions in textual quality reports. It also includes the functionality for comparing error distributions across different data sources, which we have implemented for a public data source, the database of automotive malfunctioning complaints maintained by the Office of Defects (ODI) of the National Highway Traffic Safety Administration (NHTSA) [13].

We test a domain-specific and a domain-ignorant version of a custom classification algorithm derived from k-Nearest-Neighbors (kNN), and evaluate both against a baseline which ignores the text content as well as with respect to their industrial feasibility.

We also investigate in more detail the influence of the report source and its place in the error classification process – early or late and contributed by mechanics or part suppliers.

In sum, we address three different industry-focused goals: (1) to make classification work easier for the workers who do it by sorting error codes in a meaningful way, (2) to do this as early as possible in the life cycle of a damaged car part, and (3) to make data comparable to other data sources through text analytics.

The remainder of this paper is structured as follows: In the following chapter 2, we present the research background. In chapter 3, we describe in detail the industrial context of our application (3.1) and discuss the challenges presented by the data (3.2). The methods and implementation are explained in chapter 4. We then present and evaluate two experiments and an extension of the original use case in chapter 5. We look at the feasibility of text-based error classification (5.2) and the role of the report source for classification accuracy (5.3) as well as the potential use of automated error code assignment to compare the performance of a product with competitors (5.4). Chapter 6 concludes our paper with a summary of the results and outlook on future research.

## 2. BACKGROUND

In prior research, we have motivated the need for Product Life Cycle Analytics integrating structured and unstructured data within a holistic framework [11]. In this chapter, we present two background foci which are relevant for the present research topic, namely text analytics in the automotive industry (2.1) and the general field of automatic text categorization (2.2), which deals with applying classification algorithms to text data.

### 2.1 Text Analytics in the Automotive Industry

In the automotive domain, there have been a number of efforts to use unstructured text data for business analytics. Most recently, several interconnected research projects [3, 16, 8] have developed software for extracting information about frequent problems from internal error reports and customer sentiment related to problems from social media. This research has also led to the creation of a valuable semantic resource, a taxonomy of parts and errors [17, 18], which we



use as a central component in our analytics framework (cf. 4.5.3) as well as in the domain-specific classification algorithm.

## 2.2 Automatic Text Categorization

Automatic text categorization has been a widely researched field since the late 1990s / early 2000s [19]. Typically, the task is to label texts as belonging to one of a small number of classes, e.g. one of five different topics for news texts or one of three known potential authors for literary works. Our task differs from this in that we have a very large number of classes.

Features for classification are usually derived by their information content across a large number of texts. Using pre-defined features such as author, user mentions and signal words in tweets [20] has also been shown to achieve high accuracies. We investigate the suitability of features drawn from a domain-specific knowledge resource.

An important part of this investigation is the mapping of words in the text to concepts from a semantic resource. We agree with the argument of [10] that words and stems do not represent the semantic content of a text very well. They try to map words to concepts using WordNet [4]. This is important for disambiguation, but also for highlighting and exposing shared concepts as latent features across texts with no shared word material. Because our semantic resource is rich in synonyms, we can map text to concepts via surface entity recognition.

[7] point out a weakness of the kNN algorithm which we also encounter – it is instance-based and thus potentially memory-intensive. They develop a modified kNN which creates generalized instances and representatives of instances based on local neighborhoods. We modify kNN to use representatives of instances based on abstractions of texts to contained concepts in our domain-specific variant, and also store these instances in a relational database with on-the-fly access to further address memory concerns.

## 3. PROBLEM DESCRIPTION

In this chapter, we present the industrial context of our use case (3.1) and discuss the challenges of the data we are working with (3.2).

### 3.1 Industrial Context

An automotive OEM is evaluating car parts which were removed during repairs in customer-owned vehicles for the warranty process and in order to gain insight into quality issues. The evaluation is a complex and multi-step process which involves unstructured text data from many sources and large amounts of human work:

The removed and potentially damaged car part is first evaluated in a short textual report by the mechanic who removed it. It is then shipped to the OEM, where an optional initial report can be written. Next, the car part is sent on to the supplier who manufactured it. The supplier evaluates the part's damage, writes a textual report and assigns a damage responsibility code (indicating whom they hold responsible for the problem). Eventually, a quality expert at the OEM assigns the car part a final error code and writes a short final report. This process of data accumulation is depicted in Fig. 2.

In order to assign the correct error code, the quality expert needs to read all reports written about the part at hand and

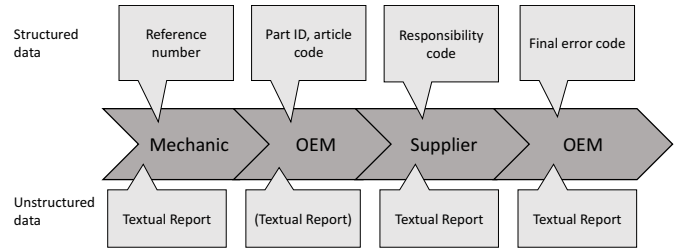


Figure 2: Process of data accumulation

then pick the error code from a list of potentially over 100 error codes available for this type of car part.

Due to the largely textual nature of the data and the large number of potential error codes, a substantial amount of the quality experts' time is taken up by assigning error codes to known errors. We want to support the quality workers by offering them automatically derived error code recommendations to speed up the decision process. If the set of error codes for a given part is smaller and sorted, the final error code assignment will take less time. The quality experts can then spend more time investigating novel or more complex errors, thus improving the overall quality of the evaluation.

### 3.2 Challenges of the Data

We developed our prototype around a randomized and anonymized subset of the original data input from the evaluation documentation tool. We extracted a fraction of the data concerning three larger component classes which have already been assigned error codes, such that a portion of the data can be used for evaluation. Other component classes are subject to future research to further validate the approach. Any information about individual persons (quality workers assigned to tasks, supplier contacts, etc.) was removed, as well as select mentions of supplier names and the OEM name, and the fields listing vehicle identification numbers and information about vehicle make and model. In total, we are working with data for 7500 individual car parts. These data, including the text reports, are stored across several tables in a relational database.

We define all data pertaining to an individual component as a data bundle. The data bundles for each component are structured in the following way:

- A component is identified by a unique reference number and also assigned an article code and a part ID. These have vastly different levels of granularity: In our data set, there are 831 distinct article codes and 31 distinct part IDs.
- A further challenge is the extremely high number of distinct error codes: There are 1271 distinct error codes in our data set of 7500 data bundles. 718 of these error codes only appear a single time, so we remove them for our experiments since nothing can be learned from them for the classification task at hand (for information extraction tasks we would of course consider them). This leaves us with 553 potential classes and 6782 data bundles with an error code that appears more than once. The largest number of distinct error codes for one part id in our data set is 146, and 25 of the 31 part IDs have instances of over 10 error codes.

Ref.No	Art.Code	Part ID	Error Code
Part Description	Error Description		
Mechanic Report			
Cleint says taht radio turns on and off by itself. Electiral smell, crackling sound.			
Supplier Report			
Unit non-functional. Lüfter funktioniert nicht. Kontakt defekt, durchgeschmort.			
Initial OEM Report			
Did test A470, no clear results, sending on to supplier. Removed some dirt.			

**Figure 3: Structure of the data bundles before final classification, with missing structured data fields highlighted in red**

- Each component is further associated with three or four textual reports: (1) the mechanic report, (2) the initial OEM report (optional), (3) the supplier report and (4) the final OEM report.

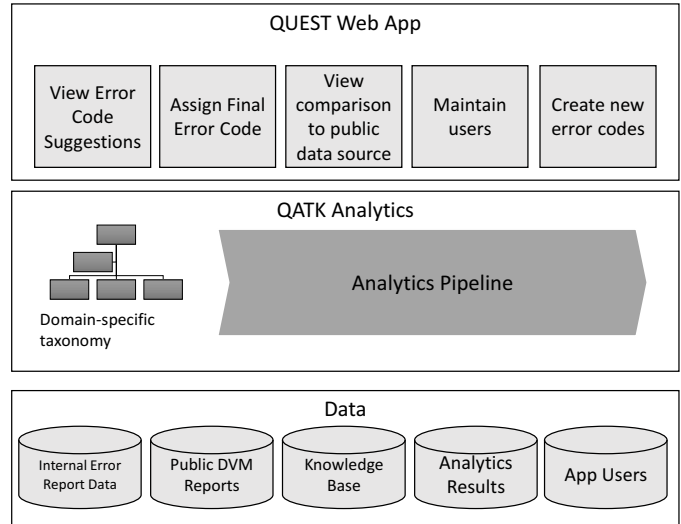
Other textual resources are the standardized descriptions of part id and error code in German and English. Fig. 3 shows a schematic overview of one data bundle and contains a fictional but representative text example.

These text resources can be used to derive textual indicators of error codes during the training phase of classification. In the testing phase, we use only the mechanic report, the optional initial report, the supplier report and the part id description. This reflects the circumstance that the final OEM report and the error code description are unavailable as sources for textual indicators in data which have not yet been assigned an error code. In 5.3, we investigate the performance of classification when using only the mechanic report or only the supplier report as input to the classifier.

The reports in our data sample are mostly a mix of German and English, but the entire data set contains several other languages. Robustly recognizing meaning in multilingual input is therefore a requirement for our system. In our prototype, we achieve this by using a multilingual semantic resource for the domain-specific approach and primarily relying on natural language processing steps which are language-independent. The domain-ignorant approach does not address multilinguality. Future investigations will also deal with how to incorporate language-specific tools.

## 4. METHODS AND IMPLEMENTATION

In this chapter, we present the general architecture of our analytics toolkit (4.1), develop a basic algorithm derived from kNN (4.2), discuss our adaptation of this algorithm in the domain-specific and the domain-ignorant variant (4.3), and describe the processing pipeline (4.4) as well as the prototypical implementation (4.5).



**Figure 4: General architecture of the recommendation system**

### 4.1 Architecture

The architecture of our general analytics framework comprises data sources and two main components (cf. Fig. 4), the Quality Analytics Toolkit (QATK) and the Quality Engineering Support Tool (QUEST) web application.

The QATK provides a highly modular analytics pipeline to process the text data, build a knowledge base representing structure extracted from the unstructured data, and assign scored and sorted potential error codes to new data bundles using the classification approach outlined in 4.3. This pipeline can be seen in detail in Fig. 8 and is further discussed in 4.4.

The functionality of the QATK can easily be adapted to classify data from a different source according to the same classification schema of part IDs and error codes as the internal source. This allows for comparisons of error distributions across different data sources. In 5.4, we show how such comparisons can be implemented for a public data source, the complaints database of the NHTSA ODI available via [safercar.gov](http://safercar.gov) [13].

### 4.2 Basic Classification

The reports and the error key and part ID labels contain unstructured text descriptions of the problems encoded by the error keys. If we can abstract from these descriptions to structured features, we can use them as input for a classification algorithm.

In our approach and in the prototypical implementation, we employ a classification algorithm derived from the k-Nearest-Neighbor algorithm. The standard kNN algorithm (Fig. 6) determines class membership of a data point by majority vote from the classes of the k nearest neighbors of this data point, where nearness is equivalent to similarity with respect to the chosen classification features. This majority vote can also be weighted by the individual nearness of neighbors, which is determined by a similarity measure of choice between the data points.

kNN is a „lazy” machine learning algorithm – it does not build a statistical model, it just holds instances of already

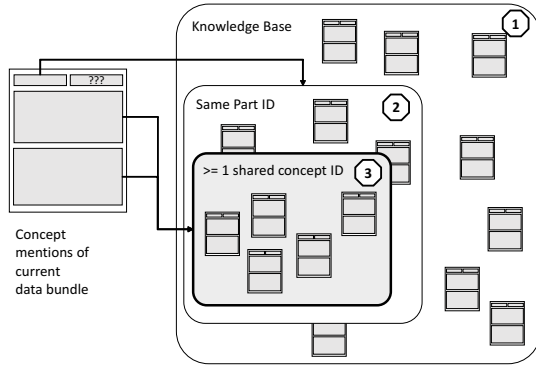


Figure 5: Selecting candidate nodes

classified data in memory (or on disk, as is the case in our implementation) for comparison with the data instances to be classified.

We have decided to focus on variants of kNN in our proof of concept for several reasons: We are dealing with an extreme multi-class classification problem, and in contrast to many other algorithms, kNN handles multi-class classification in a straightforward manner. It is also instance-based and therefore allows for predictions about class membership even with a small data set and a large number of classes, which makes it especially suitable for our example data. Further, since our focus is on establishing whether this particular multi-class classification problem can be solved semi-automatically and on investigating the role of domain-specific vs. domain-ignorant features, we decided in favor of an algorithm which allows for very straightforward control over the features, is easy to implement and easy to understand, and can easily be used with different similarity or distance measures, such that we are not fixed on representing our data in one particular way. Other algorithms are to be investigated in future research after we have established that the challenges of the data can be met at all.

Thus, we can derive a bare-bones classification algorithm with maximum parametrizability:

**Given** set of objects  $S$  with assigned classes, object  $o$  without assigned class

**for**  $o_i$  **in**  $S$ : calculate  $\text{similarity}(o, o_i)$

**sort**  $S$  by descending similarity

**assign** class to  $o$  based on sorting of  $S$

The similarity measure, the choice of features on which to base the similarity measure and the method for deriving the class assignment from the similarity ranking can be adjusted to the needs of the use case.

### 4.3 Adaptation and Parametrization

Starting from the basic algorithm described in 4.2, we make the following modifications: It is unlikely that a clear majority of the nearest neighbors of a data bundle will all share the same error key because of the sparsity introduced by the large number of classes and the small size of the data set. Therefore, instead of majority vote to determine one definitive class, we output a list of all potential error keys ranked by the distance of the knowledge base instances to the data bundle, then cut off the list at  $k$  for initial presentation to the expert (see the schematic depiction in Fig. 7). A similar type of ranking categorization is already mentioned

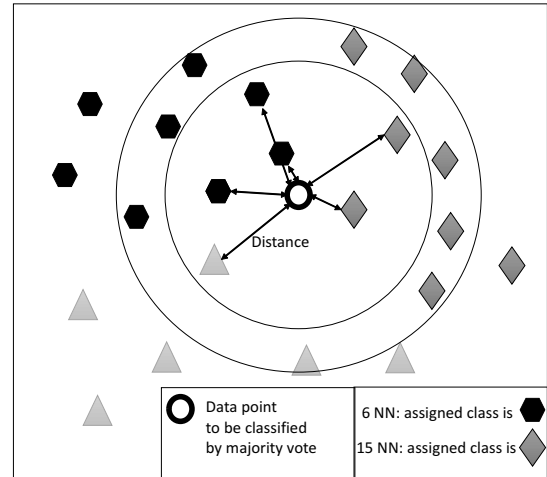


Figure 6: Standard unweighted instance-based kNN classification for  $k = 6$  and  $k = 15$  with class assignment by majority vote

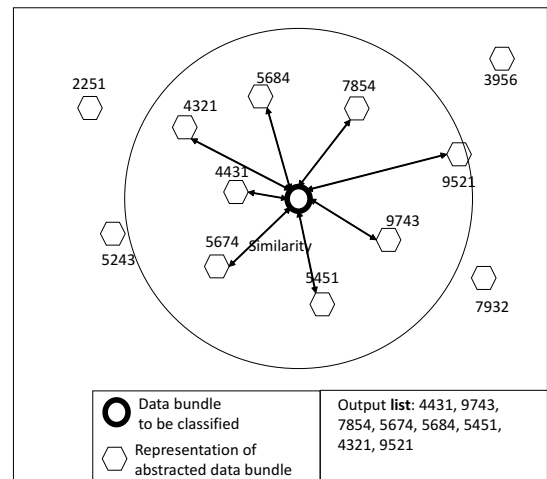


Figure 7: Adapted ranked-list kNN classification for abstracted data bundle representations

in [19]. Thus, we also address a weakness of standard kNN which becomes evident in Fig. 6 – the sensitivity to local data structures. For  $k = 6$ , the class assigned by majority vote is different from that for  $k = 15$ . Since our goal is to support the human expert, not fully automatic classification, we can avoid this inconsistency. Items lower on the list can be accessed by the quality expert in the user interface (cf. Fig. 4).

We determine the closeness of the data bundles by comparing them with respect to features derived from the text. For the domain-ignorant approach, we use all words in the text (*bag-of-words*), for the domain-specific approach, we choose mentions of parts and errors as features (*bag-of-concepts*). On average, a text has about 70 words, resulting in as many bag-of-word features. With the domain-specific approach, we detect on average 26 part/error mentions per text. We use the concept mentions as attributes without distinguishing between types of concepts (part or error). Annotating the text with the help of a domain-specific, synonym-

rich part-and-error taxonomy from prior research with the OEM [16] allows us to collapse mentions of the same part in different wordings into identical features. For example, „mud guard“, „splashboard“ and „fender“ all belong to the same concept within the taxonomy and are all represented by the same concept ID. The taxonomy has about 1.800 / 1.900 distinct concepts in German and English, respectively. Thus, we can represent each unique combination of part ID, error key and concept mentions as a node in a knowledge base, which is derived in a first training step.

This also allows us to abstract from data instances to configuration instances, reducing the size of the knowledge base and allowing for faster data comparisons when calculating similarity measures. We thus address one of the weaknesses of the standard kNN approach in a way which is similar to the kNN Model algorithm in [7]. For the bag-of-words approach we accordingly store combinations of part ID, error key and individual words (excluding punctuation) in a knowledge base of the same structure.

We retrieve error code suggestions for data bundles from the knowledge structure by computing the similarity of potential nearest neighbors. In order to do this efficiently, we filter the knowledge nodes to first retrieve a neighbor *candidate set* fitting to the data bundle under consideration (Fig. 5). From the entire set of knowledge nodes (1), we first select the subset of nodes with the same part ID as the data bundle to be classified (2). From this subset, we select those knowledge nodes which share at least one concept mention with the data bundle under investigation (3) – or one word for the bag-of-words approach. This selection is made via the indexes of the knowledge structure. If the part ID is not found in the knowledge structure, we select all nodes into our neighbor candidate set.

Next, we compute a pairwise similarity score for each candidate node with reference to the current data bundle. We retrieve the error codes of the 25 best-scored candidate nodes. For each of these error codes, we assign an error code with associated score to the data bundle under investigation. These scored error codes are stored in a relational database and presented to the quality worker via the web app interface for final error code assignment.

To evaluate performance of the classification algorithm, we have experimented with two established similarity measures – the Jaccard similarity, computed as the number of shared attributes divided by the total number of attributes, and the overlap similarity, computed as the number of shared attributes divided by the size of the smaller attribute set.

**Jaccard Similarity Coefficient:** The similarity of two items (knowledge nodes) with feature sets A and B is represented by:

$$\frac{|A \cap B|}{|A \cup B|}$$

**Overlap Similarity Coefficient:** The similarity of two items (knowledge nodes) with feature sets A and B is represented by:

$$\frac{|A \cap B|}{\min(|A|, |B|)}$$

## 4.4 Processing Pipeline

The classification step is embedded in a pipeline which includes linguistic preprocessing and the creation of a knowl-

edge base. This processing pipeline is detailed in the following.

Figure 8 shows the entire analytics pipeline for the domain-specific approach. It can conceptually be split in two parts: one to extract structure from unstructured data, which includes data preparation, linguistic preprocessing and annotation with domain-specific knowledge, and one for processing the extracted structured data, which includes the building of a knowledge base and the classification step.

We assume the classical phase-oriented data mining process which differentiates a training phase from the subsequent test phase and the application phase.

In the *training phase*, we extract domain-specific classification features from within the unstructured data portion, following several steps:

1. Creating Data Bundles: read data from the database and combine related reports into one document.
2. Unstructured Data Analytics
  - (a) Text Preprocessing: Tokenization and Language Recognition
  - (b) Concept Annotation: mark up domain-specific concepts in the text (words describing parts and errors)
3. Structured Data Analytics
  - (a) Knowledge Base Extraction: for each data bundle, extract into a "knowledge node" (cf. Fig. 9)...
    - the error code
    - the part number
    - the occurring concepts (numeric IDs)
  - (b) Knowledge Base Persistence: store knowledge nodes in a relational database

After the knowledge base has been created, we exploit this knowledge for the classification step (cf. 4.3) in the *test and application phases*. Steps 1 - 2 of the process are identical to the training phase:

1. Creating Data Bundles: read data from the database and combine related reports into one text document.
2. Unstructured Data Analytics
  - (a) Text Preprocessing: Tokenization and Language Recognition
  - (b) Concept Annotation: mark up domain-specific concepts in the text (words describing parts and errors)
3. Structured Data Analytics
  - (a) Candidate Set Generation: select knowledge nodes which share a minimum of 1 feature with the data bundle to be classified (cf. Fig. 5)
  - (b) Classification of the data bundle (cf. 4.3)
  - (c) Result Persistence: store scored error code suggestions in a relational database

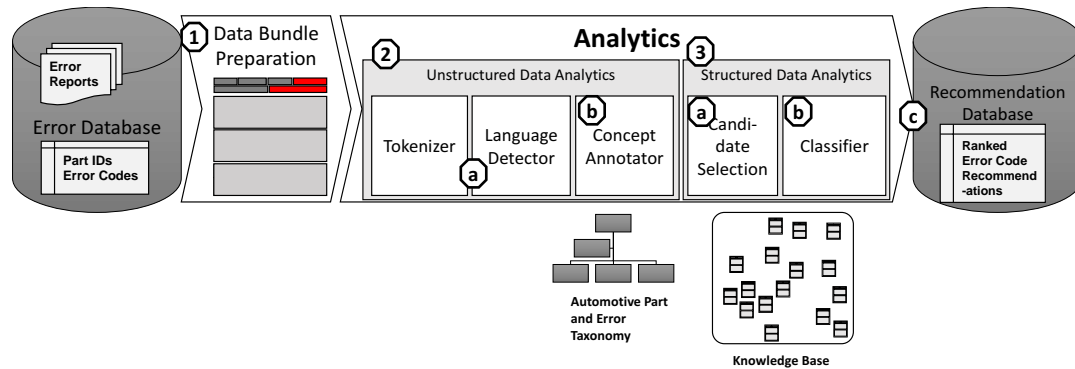


Figure 8: Detailed view of the domain-specific classification pipeline

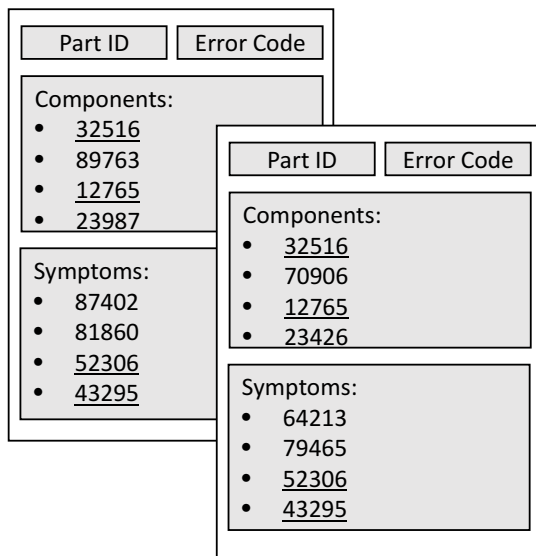


Figure 9: Two Knowledge Nodes with shared concepts underlined

The domain-ignorant approach proceeds accordingly but eliminates the concept annotation step and instead extracts all words of the document as features to be stored in the knowledge nodes.

It is obvious that this approach to a processing pipeline reflects a high degree of flexibility and extensibility for both preprocessing and classification steps. In our current setting we use the kNN-derived algorithm described in 4.3 for the classification step. This step is realized as an extension point where different classification algorithms can be plugged in easily.

## 4.5 Prototypical Implementation

In the following, we give a short overview of the technologies used in the prototypical implementation of the QATK framework and QUEST app.

### 4.5.1 Data Storage

For data storage, we use relational databases. We store raw data from the industrial source as well as from the NHTSA ODI source and the knowledge bases and classi-

fication results.

### 4.5.2 Text Analytics and Classification

In our implementation of the Quality Analytics Toolkit, we build on the Java version of the open-source Apache standard UIMA (Unstructured Information Management Architecture) [6], which enables us to easily build modular linguistic processing pipelines. These pipelines are composed of Analysis Engines containing annotators with single text analytics functionalities. Annotations on the text are recorded as typed feature structures with a start and end index relative to the document text in the Common Analysis Structure (CAS) which is handed over from one Analysis Engine to the next, such that annotators can build on findings from previous steps of analysis. In our case, one CAS contains one data bundle, including all available reports and text descriptions plus the part ID and error code.

We chose this framework for several reasons: It is open-source, extremely modular and well supported by the research community. High-quality implementations exist for a large number of standard natural language processing components, e.g. in the DKPro repository [21]. Functionality for quick and flexible pipeline building and testing is provided by the uimaFIT library [14].

The core of the QATK toolkit is a UIMA pipeline corresponding to the processing steps explained in 4.4 and depicted in Fig. 8. It reads the report and identifier data bundles from a database, segments the text into words using a simple custom whitespace-/punctuation-tokenizer, identifies occurrences of car part and problem synonyms in the text, builds a knowledge base from these identified concepts and uses the knowledge base to assign error code suggestions to previously unseen data bundles.

### 4.5.3 Domain-specificity: The Automotive Part and Error Taxonomy

A domain-specific resource used in our analytics module is a taxonomy of car parts and error symptoms, developed in prior research and originally used for an information extraction task on social media data [18, 9]. The taxonomy is stored in a custom XML format and has a shallow structure which is nevertheless well suited to the differentiations we want to make: It distinguishes components, symptoms, location and solutions. The error codes we want to recommend correspond to symptoms and also depend on components, which is why we choose to annotate the texts with occur-

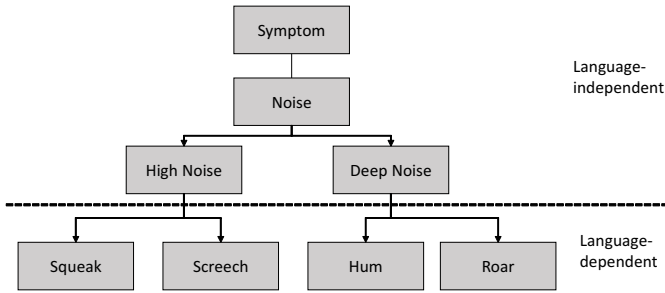


Figure 10: Automotive Taxonomy (graphic adapted from [18])

rences of components and symptoms from the taxonomy as features for our classification task. The taxonomy is multilingual – its upper category levels are language-independent with multilingual labels, its leaf categories are language-specific and contain synonyms of terms for the same concept (cf. Fig. 10).

QATK builds upon some closed-source legacy libraries for maintaining and using the taxonomy resource: An editor GUI for adding, changing and removing taxonomy concepts and concept features, as well as compiled Java archives which contain the classes needed for modifying and using the taxonomy. Among them are UIMA components for annotating occurrences of taxonomy concept words in text documents. These libraries do not entirely meet the requirements of the present use case. Therefore, some efforts had to be made in order to be able to use the taxonomy for text annotation.

We made a number of changes to the representation of the taxonomy and to the taxonomy annotator component which improve performance (cf. [12]): Annotation becomes faster, less memory-intensive, achieves higher coverage and is more accurate for multiwords. We represent the taxonomy as a trie data structure, a tree structure which allows for fast search and retrieval. Like the original approach, we expand the concepts of the taxonomy with synonyms of concept label substrings as found in the taxonomy itself.

Our optimized implementation has a left-bounded greedy longest-match approach for mapping text sequences to taxonomy concepts, eliminating concept matches which are completely enclosed by other concept matches. By applying multilingual annotation and correctly capturing multiwords, we achieve an overall higher recall of concepts than the annotator from the legacy code. For instance, the original taxonomy annotator does not recognize any taxonomy concepts in 2530 out of the 7500 data bundles, but the new annotator finds concepts in all of these.

#### 4.5.4 User Interface

The QUEST web application partly reconstructs the user interface and functionality of the original quality engineering software which is used by the automotive OEM to record, maintain, and classify the data. In QUEST as in the original software, users can view the data and assign error codes. The core difference is that in the QUEST error code assignment interface, the user is first presented with a selection of the 10 most likely error codes in descending order of likelihood. If the user decides that the correct error code is not among these 10 codes, they can access the list of all error

codes available for the part ID of the current data bundle, as is the default in the original software.

Also, users with extended rights can define new error codes right in the QUEST interface. Furthermore, all users can view the comparison of error code distributions between the OEM data set and the public US complaints database (cf. 5.4). The QUEST web app is compatible with most browsers and implements responsive design to be viewable on mobile devices. It is written in Java, uses PrimeFaces graphical components [1] and is deployed on a WSO2 web server [2].

## 5. EXPERIMENTS

In this section we present and discuss the results of two experiments into the feasibility of text-based classification with our data set. In 5.1 we establish the conditions for our experiments, in 5.2 we compare the performances of the domain-specific and the domain-ignorant modified classification algorithms with respect to a baseline, and in 5.3 we test the performance of classification on different report types (mechanic and supplier report). In 5.4, we describe the setup of an extended use case in the web app which is currently under development and evaluation.

### 5.1 Experiment Setup

We test the classification algorithm (cf. chapter 4.3) with different similarity measures and with different data abstraction models.

As a performance measure, we report accuracy defined as the percentage of test data which include the correct error code in the error code list at  $k \leq 1, 5, 10, 15, 20$  and  $25$ , respectively.

#### Accuracy@k:

For  $D_k$  the set of data bundles where the correct error code is found within the first  $k$  suggestions, corresponding to the  $k$  nearest neighbors, and  $T$  the test set,

$$A@k = \frac{|D_k|}{|T|}$$

We run all experiments with stratified 5-fold cross-validation on the 6782 data bundles whose error code appears more than once in the data. This means that for each error code, we use 4/5 of the data bundles with this error code as input to the knowledge base and assign error codes to the remaining 1/5 using the knowledge base built from the rest of the data. We do this five times with distinct splits of the data and average the accuracies obtained in each iteration. The test data sets consist of 1250 data bundles on average.

We use two similarity measures, the Jaccard coefficient and the overlap measure (cf. 4.3). We work on whitespace- and punctuation-tokenized text without further preprocessing or normalization.

We compare the results of the classification to two baselines obtained without or with very little consideration of the text:

1. the *code frequency* baseline, where all error codes which are available in the database for the part ID of the data bundle under consideration are sorted by their frequency in this database, and the first  $k$  returned
2. the unsorted *candidate set* baseline (cf. 4.3), containing all nodes in the knowledge base which share the

part ID and at least one concept / word with the data bundle under consideration

The baselines themselves merit a look at their performance: The *candidate set* baseline depends on the applied variant of the algorithm, but all candidate set baselines have similar accuracy profiles. Accuracies are rather low throughout, with  $<1\%$  accuracy for  $k = 1$  and approximately linear development towards around  $83\%$  accuracy for  $k = 25$  (cf Fig. 11). This baseline could obviously not be used for automated recommendations of error codes.

The *code frequency* baseline performs better than the candidate set baseline, with an  $\text{accuracy}@1$  of  $35\%$ ,  $\text{accuracy}@5$  of  $76\%$  and  $\text{accuracy}@10$  of  $88\%$ . At  $k = 25$ , it even has perfect accuracy of  $100\%$ . Since we know that there are potentially over hundred error codes for one part ID, we assume that this is an artifact of our randomly selected data set. In any case, sorting available codes by their frequency can be a first step towards supporting quality workers in finding the correct error code more quickly.

## 5.2 Experiment 1 – Text-Based Error Code Prediction

In this experiment we establish whether error codes can be predicted at all on the basis of the text reports alone. We compare two variants of the classifier, a domain-ignorant bag-of-words model on the tokenized text, and a domain-specific bag-of-concept model created with the help of domain-specific text annotations from the automotive part and error taxonomy.

### 5.2.1 Results

The results of experiment 1 can be seen in Figure 11. We find that both the bag-of-words and the bag-of-concepts classifier outperform the baselines for  $k < 25$  when using Jaccard similarity. The four variants we tested – bag-of-words and bag-of-concepts with each similarity measure respectively – differ considerably in their performance.

In general, overlap similarity performs worse than Jaccard, and the bag-of-concepts classifier does not perform significantly better than the code frequency baseline (in fact, slightly worse for  $k = 1$ ) when combined with the overlap measure.

Combined with the Jaccard measure, the bag-of-concepts approach out-performs both baselines with  $\text{accuracy}@1$  of  $56\%$ ,  $\text{accuracy}@5$  of  $85\%$ , and  $\text{accuracy}@10$  of  $92\%$ . For  $k$  of 15, 20 and 25, all approaches as well as the baseline deliver accuracies between 90 and  $100\%$ .

For smaller  $k$  (1 and 5), the accuracy of the bag-of-words classifier is markedly better than that of the bag-of-concepts classifier regardless of similarity measure used, with  $\text{accuracy}@1$  of  $76\%$  (overlap) and  $81\%$  (Jaccard),  $\text{accuracy}@5$  of  $93\%$  (overlap) and  $94\%$  (Jaccard), respectively.

### 5.2.2 Discussion

Three out of the four text-based classification algorithm variants provide accuracies which out-perform the code frequency baseline, and all four could be used for recommending error codes on the basis of text reports. The bag-of-words model is currently providing better accuracies than the bag-of-concept model, especially for small  $k$ . This means that its ranking of the potential error codes more closely resembles the actual probabilities of error codes based on the content of the problem reports.

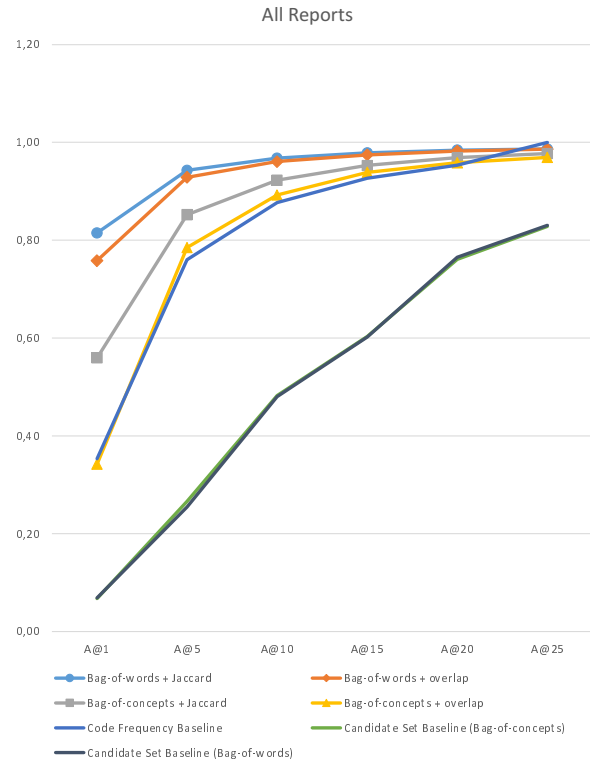


Figure 11: Results of experiment 1, with  $k$  on the x-axis and  $\text{accuracy}@k$  on the y-axis.

This tells us that the concepts which are currently being recognized using the automotive parts and error taxonomy do not represent ultimately accurate features for classification and are not the best option for recommending error codes to the quality worker. This is not altogether surprising, since the taxonomy was originally developed for a different task (information extraction, cf. [17]) and has not yet been adapted to the current data source. Adapting the taxonomy thus suggests itself as a next step.

However, the bag-of-words variant is not a feasible industrial solution because of time and memory needs due to the larger number of features per data bundle and the larger number of pairwise similarity computations to be made. On our small data set which includes only 3 out of hundreds of components, running the bag-of-words classifier takes about 11 minutes for one iteration of the five-fold cross-validation, classifying ca. 1250 data bundles, which means computation time per data bundle is at ca. 0.5 seconds. In contrast, running the bag-of-concepts classifier takes about three minutes for one iteration, which means computation time per data bundle is at ca. 0.14 seconds. When applying our processing pipeline to the entire data set with a larger number of data bundle to data bundle comparisons, it is important to keep the number of pairwise feature comparisons low. Removing German and English stopwords (articles and personal pronouns) as an additional step during the bag-of-words approach has no impact on the accuracy of classification, but shortens the runtime to ca. 7 minutes for one iteration and ca. 0.3 seconds per data bundle. This is still slower than the bag-of-concepts approach.

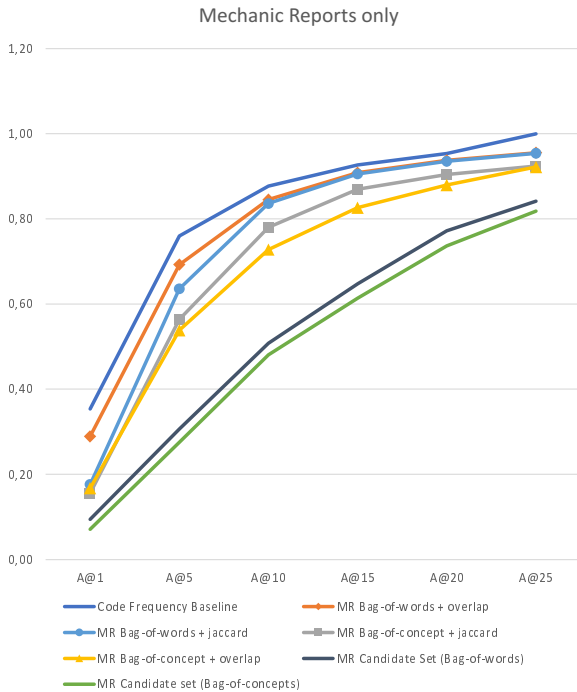


Figure 12: Classification results for features derived from the mechanic reports only, with  $k$  on the x-axis and accuracy@ $k$  on the y-axis.

In contrast to the bag-of-concepts approach, the bag-of-words approach is also not suited for further domain-specific analysis steps. Improving the coverage of the taxonomy used for the bag-of-concepts approach is therefore a worthwhile avenue to pursue. An overview of the issue of taxonomy extension and an argument for re-using semantic resources across tasks is given in [12]. Investigations into methods to automate the extension of a domain-specific semantic resource are on-going.

There are thus two conclusions to be drawn from experiment 1: (1) the text reports can indeed be used to support quality workers by automatic recommendation of error codes, (2) to create a feasible industrial solution, an improved domain-specific resource is needed.

### 5.3 Experiment 2 – Point of Entry for Error Code Prediction

In the second experiment, we test how early it is possible to make a prediction about the error code of a report bundle. Recall that data about the nature of the problem from different sources – mechanic, OEM and supplier – accumulate over time during the quality evaluation process (cf. Fig. 2). The earliest report which reaches the OEM is the mechanic report, whereas the supplier report is added later. It would be beneficial if the error code could already be predicted on the basis of the mechanic report only.

Retaining the knowledge base models learned on all reports, we have therefore attempted classification with all variants of our adapted classification algorithm on test data bundles which included only one type of report, namely, the mechanic report or the supplier report.

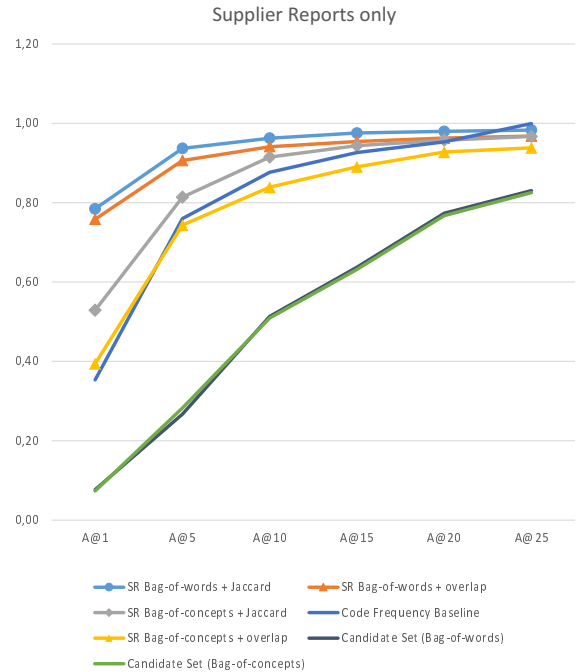


Figure 13: Classification results for features derived from the supplier reports only, with  $k$  on the x-axis and accuracy@ $k$  on the y-axis.

#### 5.3.1 Results

The results of experiment 2 can be seen in Figures 12 and 13. We find that the classifier performs very badly on test data which only include the mechanic report (cf. Fig. 12): All four variants of the algorithm have lower accuracies across the board than those provided by the code frequency baseline, with accuracy@1 between 16 and 29 % vs. the baseline’s 35 %. Still, the bag-of-word models perform slightly better than the bag-of-concept models.

On test data which only include the supplier report, we observe accuracies which are nearly as good as those for the test data including mechanic report and supplier report (and in some cases an early OEM report): 78 % accuracy@1 for the bag-of-words model with Jaccard similarity, accuracies of > 90 % starting at  $k = 5$  for the bag-of-words model and at  $k = 10$  for the bag-of-concepts model, and a very close resemblance of accuracies between the bag-of-concepts with overlap similarity and the code frequency baseline (cf. Fig. 13).

#### 5.3.2 Discussion

It is evident that the mechanic reports alone do not contain good features for predicting error codes with the adapted classifier, either as bag-of-words or as bag-of-concepts representations. In contrast, the supplier reports are a more reliable source of features. This is in accordance with observations about the information content and the data quality of the respective data sources: Mechanic reports tend to be poor in detail, focused on superficial problem description and often error-riddled, such that even human experts cannot draw conclusions about the detailed nature of the problem, whereas supplier reports tend to contain more



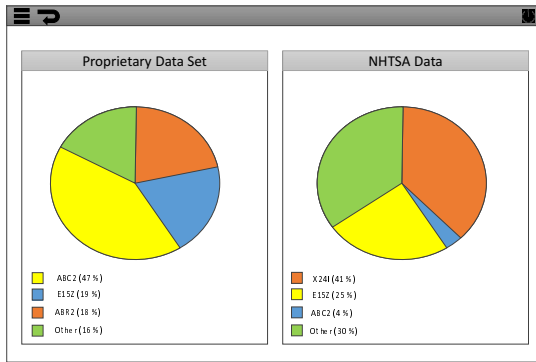


Figure 14: Data Comparison Screen in the QUEST web app, showing a side-by-side comparison of the top 3 error codes in two different databases.

detail and include descriptions of potential causes.

While we have to conclude that (1) an earlier entry point into automatic error code suggestion is not feasible, we also find that (2) even a comparatively simple text-based classification approach accurately reflects the amount of information which human experts can draw from the text sources.

#### 5.4 Extending the Use Case – Error Distribution in Different Data Sources

As mentioned earlier, it is easy to exploit additional data sources using the knowledge bases created with the internal OEM error report data. This allows for an interesting extension of our use case. If we assign error codes from the schema we use to classify our own quality data to texts from a different data source, for instance one which covers complaints from a different market and includes reports about other manufacturers' vehicles, we can gain insights about where we stand in terms of product quality in contrast to the competitors. This is crucial business intelligence for staying competitive, e.g. by identifying brand-specific weaknesses or issues with shared suppliers.

Obviously, there will be substantial inaccuracies in the fully automatic classification of the public data source. In particular, the bag-of-words approach suffers in accuracy as soon as test and training data are different text types or in different languages, whereas the bag-of-concepts approach is in principle independent of the document language or other text features. However, an approximate impression of the distribution of similar errors can still be gained from the data. The usability and desirability of this functionality have been confirmed in conversation with the OEM.

In the QATK/QUEST implementation, we provide a mockup of this use case extension: We use our knowledge base to classify problem reports from the US-American complaints database maintained by the Office of Defects (ODI/NHTSA) [13]. In the web app, we have implemented the function for viewing side-by-side pie charts showing the distribution of the  $n$  most frequent error codes in both data sources (cf. Fig 14).

## 6. CONCLUSIONS AND FUTURE DIRECTIONS

In this paper, we have presented a specific multi-class classification problem from a real-life industry context, namely,

the filtering and recommendation of error codes for bundles of textual reports concerning damaged car parts. This problem is challenging because of the nature of the data, which are mostly unstructured text using domain-specific vocabulary, and because of the high number of classes.

We have tested alternative variants, domain-specific and domain-ignorant, of a custom classification algorithm derived from k-Nearest-Neighbors (kNN), and evaluated both against a baseline which ignores the text content as well as with respect to their industrial feasibility. We have shown the QATK/QUEST toolkit as a viable approach and presented a prototypical implementation.

In order to identify domain-specific terms in the text, we have used a legacy semantic resource originally designed for a different task. We have tested the validity of these domain-specific features with a custom classification algorithm adapted from k-Nearest-Neighbors. We have shown that the use of domain-specific knowledge leads to good accuracies of recommended classes, with up to 92% of correct classes recovered within the first 10 ranked recommendations. We have compared this domain-specific approach to a domain-ignorant bag-of-words approach and found that the domain-ignorant approach currently performs better, although it is not a viable solution in the industrial context due to performance and scalability properties.

Therefore, we plan the following extensions of our work:

- introducing more linguistic preprocessing – here we will profit from the modularity of the UIMA framework and of the QATK/QUEST toolkit
- enhancing the domain-specific taxonomy
- evaluating the extended use case and discovering more use case extensions
- evaluating the web UI in a field study with quality experts

Work on improving the coverage and maintainability of the domain-specific taxonomy is already in progress.

## 7. ACKNOWLEDGMENTS

Many thanks to our colleagues in the quality management and quality engineering departments for providing inspiration for the use case and ongoing discussions, as well as to our colleague C. Kiefer for important feedback and to the students of project „Mobile User-Driven Infrastructure for Factory IT Integration” for crucial implementation work. We also thank the Graduate School advanced Manufacturing Engineering (GSaME) for supporting the broader research context of this paper.

## 8. REFERENCES

- [1] PrimeFaces JSF.
- [2] WSO2 Server.
- [3] M. Bank. *AIM-A Social Media Monitoring System for Quality Engineering*. PhD thesis, 2013.
- [4] C. Fellbaum. *WordNet*. Blackwell Publishing Ltd, 1999.
- [5] D. Ferrucci, E. Brown, J. Chu-Carroll, and J. Fan. Building Watson: An overview of the DeepQA project. *AI magazine*, pages 59–79, 2010.

- [6] D. Ferrucci and A. Lally. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348, 2004.
- [7] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer. Using kNN model for automatic text categorization. *Soft Computing*, 10(5):423–430, 2006.
- [8] C. Hänig. *Unsupervised Natural Language Processing for Knowledge Extraction from Domain-specific Textual Resources*. PhD thesis, Universität Leipzig, 2012.
- [9] C. Hänig and M. Schierle. Relationsextraktion aus Fachsprache - ein automatischer Ansatz für die industrielle Qualitätsanalyse. *eDITion*, 1(1):28 – 32, 2010.
- [10] G. Ifrim, M. Theobald, and G. Weikum. Learning word-to-concept mappings for automatic text classification. *Learning in Web Search Workshop, . . .*, 2005.
- [11] L. Kassner, C. Gröger, B. Mitschang, and E. Westkämper. Product Life Cycle Analytics - Next Generation Data Analytics on Structured and Unstructured Data. In *CIRP ICME 2014*, volume 00. Elsevier Procedia, 2014.
- [12] L. Kassner and C. Kiefer. Taxonomy Transfer: Adapting a Knowledge Representing Resource to new Domains and Tasks. In *Proceedings of the 16th European Conference on Knowledge Management*, 2015.
- [13] NHTSA. NHTSA Data, 2014.
- [14] P. V. Ogren and S. J. Bethard. Building test suites for UIMA components. *SETQA-NLP '09 Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, (June):1–4, 2009.
- [15] P. Russom. BI Search and Text Analytics. *TDWI Best Practices Report*, 2007.
- [16] M. Schierle. *Language Engineering for Information Extraction*. PhD thesis, Universität Leipzig, 2011.
- [17] M. Schierle and D. Trabold. Extraction of Failure Graphs from Structured and Unstructured Data. *2008 Seventh International Conference on Machine Learning and Applications*, pages 324–330, 2008.
- [18] M. Schierle and D. Trabold. Multilingual knowledge based concept recognition in textual data. In *Proceedings of the 32nd Annual Conference of the GfKI*, pages 1–10, 2008.
- [19] F. Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34(1):1–47, Mar. 2002.
- [20] B. Sriram, D. Fuhry, E. Demir, H. Ferhatosmanoglu, and M. Demirbas. Short text classification in twitter to improve information filtering. *Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval - SIGIR '10*, page 841, 2010.
- [21] TUDarmstadt. DKPro Toolkit, 2011.

# Discovering Correlations in Annotated Databases

Xuebin He                      Stephen Donohue                      Mohamed Y. Eltabakh  
 Worcester Polytechnic Institute, Computer Science Department, MA, USA  
 {xhe2, donohues, meltabakh}@cs.wpi.edu

## ABSTRACT

Most emerging applications, especially in science domains, maintain databases that are rich in metadata and annotation information, e.g., auxiliary exchanged comments, related articles and images, provenance information, corrections and versioning information, and even scientists' thoughts and observations. To manage these *annotated databases*, numerous techniques have been proposed to extend the DBMSs and efficiently integrate the annotations into the data processing cycle, e.g., storage, indexing, extended query languages and semantics, and query optimization. In this paper, we address a new facet of annotation management, which is the discovery and exploitation of the hidden correlations that may exist in annotated databases. Such correlations can be either between the data and the annotations (data-to-annotation), or between the annotations themselves (annotation-to-annotation). We make the case that the discovery of these annotation-related correlations can be exploited in various ways to enhance the quality of the annotated database, e.g., discovering missing attachments, and recommending annotations to newly inserted data. We leverage the state-of-art in association rule mining in innovative ways to discover the annotation-related correlations. We propose several extensions to the state-of-art in association rule mining to address new challenges and cases specific to annotated databases, i.e., incremental addition of annotations, and hierarchy-based annotations. The proposed algorithms are evaluated using real-world applications from the biological domain, and an end-to-end system including an Excel-based GUI is developed for seamless manipulation of the annotations and their correlations.

## 1. INTRODUCTION

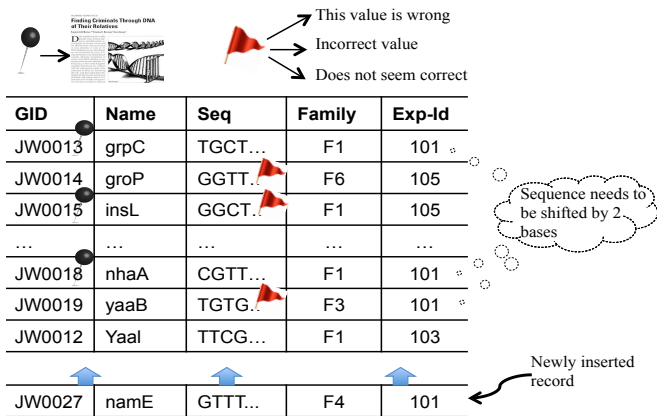
Most modern applications annotate and curate their data with various types of metadata information—usually called *annotations*, e.g., provenance information, versioning timestamps, execution statistics, related comments or articles, corrections and conflict-related information, and auxiliary exchanged knowledge from different users. Interestingly, the number and size of these annotations is growing very fast, e.g., the number of annotations is around 30x, 120x, and 250x larger than the number of data records in Data-

Bank biological database [3], Hydrologic Earth database [4, 47], and AKN ornithological database [5], respectively. Existing techniques in annotation management, e.g., [9, 15, 17, 21, 24], have made it feasible to systematically capture such metadata annotations and efficiently integrate them into the data processing cycle. This includes propagating the related annotations along with queries' answers [9, 15, 17, 24, 46], querying the data based on their attached annotations [21, 24], and supporting semantic annotations such as provenance tracking [11, 14, 20, 43], and belief annotations [23]. Such integration is very beneficial to higher-level applications as it complements the base data with the auxiliary and semantic-rich source of annotations.

In this paper, we address a new facet of annotation management that did not receive much attention before and has not been addressed by existing techniques. This facet concerns the discovery and exploitation of the hidden correlations that may exist in annotated databases. Given the growing scale of annotated databases—both the base data and the annotation sets—important correlations may exist either between the data values and the annotations, i.e., *data-to-annotations correlations*, or among the annotations themselves, i.e., *annotations-to-annotations correlations*. By systematically discovering such correlations, applications can leverage them in various ways as motivated by the following scenarios.

**Motivation Scenario 1—Discovery of Missing Attachments:** Assume the example biological database illustrated in Figure 1. Typically, many biologists may annotate subsets of the data over time—each scientist focuses only on few genes of interest at a time. For example, some of the data records in Figure 1 are annotated with a “Black Flag” annotation. This annotation may represent a scientific article or a comment that is attached to these tuples. By analyzing the data, we observe that most genes having value F1 in the Family column have an attached “Black Flag” annotation. Such correlation suggests that gene JW0012 is probably missing this annotation, e.g., none of the biologists was working on that gene and thus the article did not get attached to it. However, by discovering the aforementioned correlation, the system can proactively learn and recommend this missing attachment to domain experts for verification. Correlations may also exist among the annotations themselves, e.g., between the “Black Flag” and the “Red Flag” annotations. Without discovering such correlations the database may become “under annotated” due to these missing attachments.

**Motivation Scenario 2—Annotation Maintenance under Evolving Data:** Data is always evolving and new records are always added to the database. Hence, a key question is: “For the newly added data records, do any of the existing annotations apply to them?”. Learning the correlations between the data and the annotations can certainly help in answering such question. For ex-



**Figure 1: Examples of Annotation-Related Correlations.**

ample, the cloud-shaped comment in Figure 1 is attached to all data records having value 101 in the Exp-Id column. Based on this correlation, the system can automatically predict—at insertion time—that this annotation also applies to the newly inserted JW0027 tuple. Otherwise, such attachment can be easily missed and important information is lost. Clearly, delegating such task to end-users without providing system-level support—which is the state of existing annotation management engines—is not a practical assumption.

**Motivation Scenario 3—Annotation-Driven Exploration:** The discovered correlations may reveal information about the underlying data that trigger further investigation or exploration by domain experts. For example, as highlighted in Figure 1, the “Red Flag” annotation semantically means invalid or incorrect data. Since these annotations can be added by different biologists and at different times, none of them may observe a pattern in the data. In contrast, by discovering (and reporting) that the “Red Flag” annotation has strong correlation with experiment id 105, the domain experts may re-visit the experimental setup of this wet-lab experiment and may revise and re-validate all data generated from it.

These scenarios demonstrate the potential gain from capturing the annotation-related correlations. Unfortunately, relying on domain experts or DB admins to manually define or capture these correlation patterns is evidently an infeasible approach. This is because the correlations may not be known in advance, hard to capture or express, dynamically changing over time, or even not 100% conformed. Moreover, the manual exploration process is error-prone, will not scale to the size of modern annotated databases, and it is a very time- and resource-consuming process. For example, the UniProt biological database has over 150 people working as full-time to maintain and annotate the database [6, 12]. Certainly, such scale of investments may not be viable to many other domains and scientific groups, e.g., it is reported in a recent science survey [49] that 80.3% of the participant research groups do not have sufficient fund for proper data curation. For these reasons, we argue in this paper that the analysis and discovery of the annotation-related associations and correlations should be an integral functionality of the annotation management engine. As a result, the correlations can be timely discovered and maintained up-to-date, and also systematic actions can be taken based on them as highlighted by the motivation scenarios.

In this paper, we investigate applying the well-known techniques of association rule mining, e.g., [8, 31, 52], to the domain of annotated databases. This is a new and promising domain for association

rule mining due to the following reasons:

- Many emerging applications—especially scientific applications—maintain and rely on large-scale annotated databases [3, 5, 6]. It is reported in [1] that the *ebird* ornithological database receives more than 1.6 million annotations per month from scientists and bird watchers worldwide. These applications will benefit from the proposed techniques.

- Many annotated databases go under the very expensive and time-consuming process of manual curation, e.g., [2, 6]. The goal from this process is to ensure that correct annotations and curation information are attached to the data, and to enrich the annotations whenever possible. Nevertheless as illustrated in the motivation scenarios, the discovery of the annotation-related correlations can help in enhancing the quality of the annotated database in an automated way. And hence, reducing the effort needed in the manual curation process and freeing the domain scientists for their main task, which is scientific experimentation.

- Interestingly, annotated databases stretch the traditional techniques of association rule mining, and present new challenges as discussed in Section 3. For example, the state-of-art techniques in association rule mining fall short in efficiently handling several new cases specific to annotated databases, i.e., they cannot perform incremental maintenance of the discovered rules and they have to re-process the entire database. These cases include:

- (1) *Generalization of Annotations:* Annotations can be free-text comments, which may differ in their values but have the same semantics. And hence, discovering the correlations based on the values of the raw annotations may miss important patterns. For example, referring to Figure 1, the correlation pattern involving the *Black Flag* annotation, i.e., “Family:F1  $\implies$  Black Flag” can be detected based on the raw annotation value. This is because all instances of the *Black Flag* annotation refer to the same scientific article. In contrast, for the *Red Flag* annotation, the actual annotations inserted by scientists have different values, and thus no correlation pattern can be detected based on the raw values. However, by *generalizing* the annotations to a common concept—the *Red Flag* annotation in our case—we can detect the correlation pattern between them and the experiment Id 105. Therefore, building a generalization hierarchy on top of the annotations is an important step.

- (2) *Integration with the Annotation Manager:* We propose to build a coherent integration between the association rule mining module and the Annotation Manager component in contrast to the *offline* mining techniques. As a result, the Annotation Manager can take informed actions based on the discovered rules, e.g., discover potential missing attachments and report them for verification (Motivation Scenario 1), and annotate newly inserted data tuples with existing annotations (Motivation Scenario 2). Moreover, since the Annotation Manager cannot guarantee with 100% confidence that the predicted attachments are correct, we propose developing a verification module that enables domain experts to verify the predicted attachments.

- (3) *Incremental Maintenance under Annotation Addition:* In annotated databases, the discovered correlations and association rules need to be incrementally updated under two scenarios, i.e., the addition of new data tuples, and the addition of new annotations. The former case can be handled by existing techniques that address the incremental update of association rules, e.g., [16]. These techniques assume that the new delta batch changes the size of the database, i.e., the number of data tuples increases. In contrast, in

the latter case, the new annotation batches will not change the number of data tuples, instead they change the content of the tuples—assuming the new annotations are part of the tuples. Therefore, the existing incremental techniques need to be extended to handle the latter case.

In this work, we develop an end-to-end solution that addresses the above challenges in the context of a real-world application and annotation repository, which is a data warehouse for the *Caenorhabditis elegans* (*C. elegans*) Worm from biological sciences. To facilitate scientists’ usage of the developed system, we designed an Excel-based GUI—A tool that most scientists are familiar with—through which all of the proposed functionalities can be performed.

The rest of the paper is organized as follows. In Section 2, we present the needed background, preliminaries, and our case study. In Sections 3, and 4, we present the techniques for the discovery and maintenance of the annotation-related correlations, and their exploitation, respectively. Section 5 overviews the related work while Section 6 contains the experimental evaluation. Finally, the conclusion remarks are included in Section 7.

## 2. PRELIMINARIES

In this section, we give a brief overview on annotation management and association rule mining techniques, and then present our case study.

### 2.1 Background

**Annotation Management in Relational DBs:** Annotation management techniques in relational databases enable end-users to attach auxiliary information to the data stored in the relational tables [9, 15, 17, 21, 24, 46]. Annotations can be attached to individual table cells, rows, columns, or arbitrary sets and combinations of them. Some systems provide a GUI through which the annotations can be added [17, 25], while other systems extend the SQL language with new commands and clauses to enable annotation addition [17, 21, 25]. For example, the work in [21] introduces a new `Add Annotation` command to SQL as follows:

```
Add Annotation
[Value <text-value> | Link <path-to-file>]
On <sql-statement>;
```

This command will first trigger the execution of the specified `<sql-statement>` to identify the data tuples and the attributes to which the annotation will be attached. The annotation can be provided as a text value (using the `Value` clause), or as a link to a file (using the `Link` clause). In all of these systems, the organization of annotations, i.e., storage scheme and indexing, is fully transparent to end-users.

At query time, when a standard SQL query is submitted, the underlying database engine will not only compute the data tuples involved in the answer set, but will also compute the related annotations that should be reported along with the answer set. This is not straightforward since the data may go through complex transformations, e.g., projection, join, grouping and aggregations. Therefore, the semantics of the query operators have been extended to manipulate the data as well as their attached annotations in a systematic way. For example, one possible semantic is to union the annotations when performing grouping, joining, or duplicate elimination over a group of tuples. According to this semantic, the SQL query in Figure 2 produces the illustrated output—assuming duplicate annotations on the same tuple are eliminated.

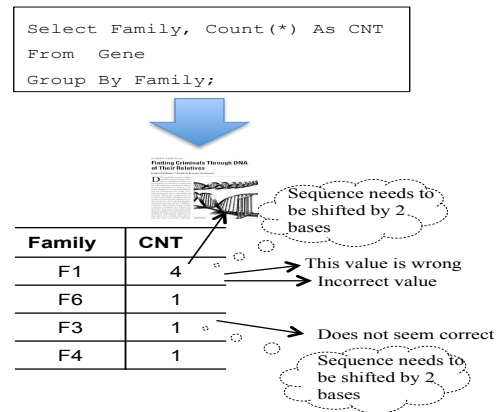


Figure 2: Automatic Propagation of Annotations at Query Time.

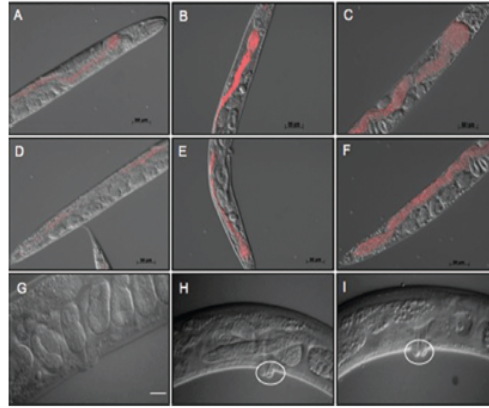
Since in large-scale annotated databases, the number of reported annotations on a single output tuple can be large, the work in [32, 50] proposed summarizing the annotations using data mining techniques, e.g., clustering, classification, and text summarization, and reporting the summarize instead of the raw annotations. The proposed work of discovering the correlations in annotated databases is complementary to the existing techniques and can be integrated with any of the existing systems.

**Association Rule Mining:** Association rule mining is a well-known problem in data mining that concerns the discovery of correlation patterns within large datasets [8, 44, 45, 52]. An association rule in the form of “ $X \implies Y$ , support =  $\alpha$ , confidence =  $\beta$ ” means that the presence of the L.H.S itemset  $X$  implies the presence of the R.H.S itemset  $Y$  (in the same transaction or tuple) with support equals to  $\alpha$  and confidence equals to  $\beta$ . Typically,  $X \cap Y = \phi$ , and the support is computed as the fraction of transactions (or tuples) containing  $X \cup Y$  relative to the database size, while the confidence is computed as  $\text{support}(X \cup Y) / \text{support}(X)$ . Therefore, given a minimum support  $\text{min\_supp}$  and minimum confidence  $\text{min\_conf}$ , the association rule mining technique discovers all rules having support and confidence above the specified  $\text{min\_supp}$  and  $\text{min\_conf}$ , respectively.

It has been observed in many real-world applications, that the number of generated rules can be very large and many of them may not be interesting. Therefore, additional measures have been proposed in [31, 44], which include the *lift* and *conviction* measures. The former is computed as  $\frac{\text{support}(X \cup Y)}{\text{support}(X) * \text{support}(Y)}$ , and the latter is computed as  $\frac{1 - \text{support}(Y)}{1 - \text{confidence}(X \implies Y)}$ . The higher these measures, the more interesting the rule. Another related extension to the standard association rule mining problem is the mining of multi-level rules [44, 45]. In this extension, the technique is given a domain generalization hierarchy over one or more attributes, and we need to discover the association rules that may span different levels of the hierarchy. For example, in market analysis, the items “*pants*”, “*shirts*”, and “*t-shirts*” can be generalized to “*clothes*”. Because of this generalization, some rules may hold at the higher level(s) of the hierarchy which may not be true for the lower more-detailed levels. Association rule mining has numerous applications in various domains including market analysis, biology, healthcare, environmental sciences, and beyond [31]. The proposed work extends these applications to the emerging domain of annotated databases.



(a) *Caenorhabditis elegans*. Lives in garden soil and feeds on bacteria, e.g., *E. coli*.



(b) Time course of intestinal distention in *C. elegans* worms exposed to *S. cerevisiae*. Worms were exposed to RFP-marked, wildtype yeast from hatching and photographed on day 3 (A and D), day 4 (B and E), and day 5 (C and F). The experiment was done three times, and 60 to 75 worms were observed over a 3-day period. (A to C) Anterior region of the worm; (D to F) posterior region of the worm. Accumulation of yeast began in the pharynx region (compare panels A and D) and proceeded to the posterior. (G to I) *S. cerevisiae* also induces vulval swelling in the worms. Worms exposed to *S. cerevisiae* (H to I) show abnormal vulval swelling (white circles) compared to the control sample grown on *E. coli* (G). Bars: A to G, 50  $\mu$ m; H and I, 20  $\mu$ m.

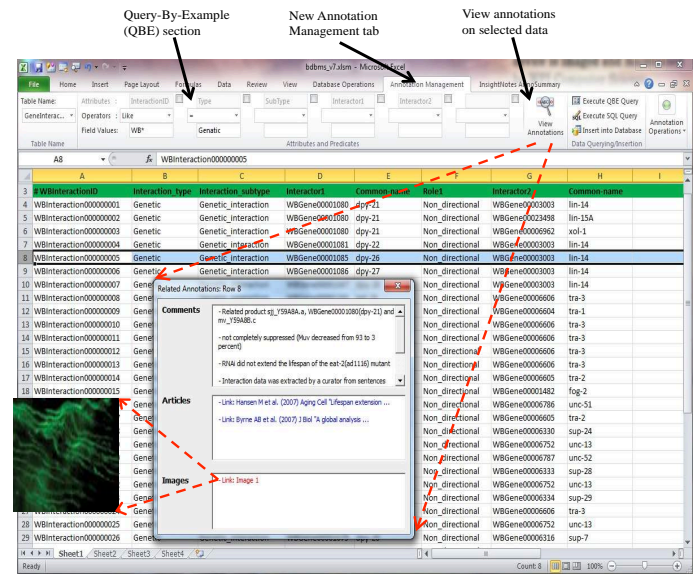
**Figure 3: Case Study: Building Data-Annotation Repository for *Caenorhabditis elegans* (*C. elegans*) Worm.**

## 2.2 Case Study: Annotated Repository for *C. elegans* Worm

Although the proposed work is applicable to annotated databases in general, we consider one case study as an example. This case study focuses on building a database repository for the *Caenorhabditis elegans* (*C. elegans*) worm, which integrates data from multiple databases and labs studying the genetics and fungal pathogenesis of the organism (Refer to Figure 3). Some of these sources maintain relational databases, while other use excel sheets to store their data. Each of the sources maintains various types of curation information and annotations related to the data records, e.g., images, publications, observations, corrections, and experimental setups. In these data sources the curation information is not modeled as *annotations*. Instead, they are modeled as regular data with relationships to the data tuples, e.g., in the relational databases, the images and observations are stored in separate tables linked to the primary data tables through PK-FK constraints.

The disadvantage of this modeling scheme, i.e., modeling the annotations as data, is that applications lose the benefits of annotation management. That is, annotation management tasks are entirely delegated to end-users and higher-level applications starting from the storage and indexing of annotations and ending by explicitly encoding the propagation semantics within each of the users' queries. Both tasks have been shown to be very complex and sophisticated. For example, the storage and indexing mechanisms need to deal with the combinatorial relationship between annotations and data, e.g., annotations can be attached to single table cells (attributes), rows, columns, arbitrary sets and combinations of them, or even attached to sub-attributes [21, 24]. Moreover, manually encoding the annotations' propagation within each query is not only error-prone, and lacks optimizations, but also renders even simple queries very complex [9, 13, 26, 46]. That is why annotation management engines have been proposed to efficiently and transparently manage such complexities across applications.

To address the above limitations, we opt for leveraging our previous systems and work in annotation management [21, 50] to model the metadata information as annotations. These systems are built on top of PostgreSQL DBMS, and thus the repository will acquire the benefits of both a DBMS and an annotation management engine. We developed an Excel-based user interface to enable seamless visualization of the data as well as the annotations (See Figure 4). For the purpose of this project, the annotations are categorized into three basic types, which are: (1) Regular comments or observations, which covers any free-text values, (2) Articles and



**Figure 4: Excel-Based GUI for Annotation Management.**

documents, and (3) Images. In the Excel-based GUI, scientists can query their data either by writing direct SQL queries, or through a QBE interface as illustrated in Figure 4. Then, they can select specific rows or table cells of interest, and click the *View Annotation* button, which reports the annotations related to the selected datasets in a new window. This window has three sections for the three annotation types mentioned above as shown in the figure. For the articles and images, they are uploaded to a web-server and can be opened by clicking the corresponding link.

## 3. DISCOVERY OF CORRELATIONS

Given an annotated database, the primary challenge is to discover and incrementally maintain the hidden annotation-related correlations within the database, which is the focus of this section. The basic unit in an annotated database is an "*annotated relation*". In the following, we formally define an annotated relation and the target correlations.

**Definition 3.1 (Annotated Relation).** An annotated relation  $\mathcal{R}$  is defined as  $\mathcal{R} = \{r = \langle x_1, x_2, \dots, x_n, a_1, a_2, a_3, \dots \rangle\}$ , where

each data tuple  $r \in \mathcal{R}$  consists of  $n$  data values  $x_1, x_2, \dots, x_n$ , and a variable number of attached annotations  $a_1, a_2, \dots, a_k$ .

**Definition 3.2** (Data-to-Annotation Correlations). *Given an annotated relation  $\mathcal{R}$ , a minimum support  $\alpha$ , and a minimum confidence  $\beta$ , the data-to-annotation correlations over  $\mathcal{R}$  is the problem of discovering all association rules in the form of:  $x_1:v_1, x_2:v_2, \dots, x_k:v_k \implies a$ , where the L.H.S is a set of column names ( $x_i$ ) and corresponding data value ( $v_i$ ), the R.H.S is a single annotation, the rule's support  $\geq \alpha$ , and the rule's confidence  $\geq \beta$ .*

**Definition 3.3** (Annotation-to-Annotation Correlations). *Given an annotated relation  $\mathcal{R}$ , a minimum support  $\alpha$ , and a minimum confidence  $\beta$ , the annotation-to-annotation correlations over  $\mathcal{R}$  is the problem of discovering all association rules in the form of:  $a_1 a_2 \dots a_k \implies a$ , where the L.H.S is a set of annotations, the R.H.S is a single annotation, the rule's support  $\geq \alpha$ , and the rule's confidence  $\geq \beta$ .*

According to Definitions 3.2 and 3.3, the rules to be discovered must involve an annotation in the R.H.S of the rule. In addition, these rules focus on the raw annotations without generalization. This is applicable especially for annotations of type *image* or *publication*, where a single image or publication can be attached to many data tuples. Discovering these rules is straightforward using any of the state-of-art techniques, e.g., the A-priori [8], or FP-Tree [30] algorithms. The only modification that we introduced to these algorithms is the early elimination of candidate patterns that do not include at least one annotation value.

### 3.1 Incremental Maintenance of Correlations

An annotated database may evolve and change in three different ways, which are: (1) Adding new un-annotated data tuples (refer to it as  $\Delta_{unannotated}$ ), (2) Adding new annotated data tuples (refer to it as  $\Delta_{annotated}$ ), and (3) Adding new annotations to existing data tuples (refer to it as  $\delta$ ). Each of these changes may affect the discovered association rules as summarized in Figure 5. The maintenance of association rules under incremental data updates has been studied in existing work [16, 41]. However, these existing techniques can handle only the first two cases mentioned above, i.e., the  $\Delta_{unannotated}$  and  $\Delta_{annotated}$  cases, but not the third case, i.e., the  $\delta$  case. The reason is that the assumption in these techniques is that the number of data tuples change (get increased), which is true for the first two cases. In contrast, in the third case, the number of the data tuples is fixed, but their content changes due to the addition of new annotations.

Figure 5 summarizes the three cases mentioned above and their effect on the association rules. The  $\Delta_{unannotated}$  case does not add any new rules since no new annotations are added. For the existing rules, both the support and confidence of the data-to-annotation rules may get decreased and need to be re-computed (Column 2 in Figure 5). Nevertheless, for the annotation-to-annotation rules, only the support may decrease and need to be re-computed, but the confidence remains unchanged (Column 3 in Figure 5). The  $\Delta_{annotated}$  case may introduce new association rules since the new data tuples are annotated. Moreover, all of the existing rules may get affected positively or negatively. For these two cases, the existing techniques in [29, 44] can be directly applied to efficiently and incrementally update the association rules.

The third case—which is not handled by existing techniques—concerns the addition of new annotations to existing data tuples. For this case, all existing data-to-annotation rules are guaranteed to remain valid because the support and confidence of these rules

$\Delta_{unannotated}$	New un-annotated data tuples	= Remain Fixed $\updownarrow$ May increase or decrease $\downarrow$ May only decrease $\uparrow$ May only increase		
$\Delta_{annotated}$	New annotated data tuples			
$\delta$	New annotations on existing data tuples			
	Effect	Existing Rules		
	Change	New Rules	$x_1, x_2, \dots, x_k \implies a$	$a_1, a_2, \dots, a_k \implies a$
	$\Delta_{unannotated}$	✗	S $\downarrow$ & C $\downarrow$	S $\downarrow$ & C =
	$\Delta_{annotated}$	✓	S $\downarrow\uparrow$ & C $\downarrow\uparrow$	S $\downarrow\uparrow$ & C $\downarrow\uparrow$
	$\delta$	✓	S $\uparrow$ & C $\uparrow$	In L.H.S: S $\uparrow$ & C $\downarrow\uparrow$ In R.H.S: S $\uparrow$ & C $\uparrow$

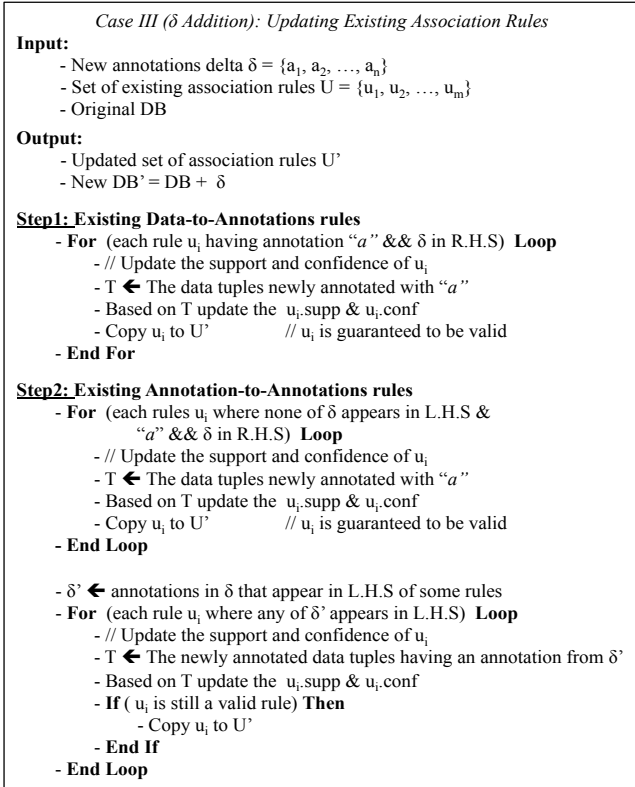
Figure 5: Effect of Evolving Data on Support (S) and Confidence (C).

cannot be decreased. The same intuition applies to the annotation-to-annotation rules if the new annotation appears in the R.H.S of the rule, i.e., the support and confidence may only increase. However, if the new annotation appears in the L.H.S of a rule, then the confidence of this rule needs to be re-computed because it may decrease and becomes below the *min\_conf* threshold. Finally, this third case may introduce new association rules that need to be discovered as indicated in Figure 5. In the following, we present a pseudocode on how these changes take place incrementally.

The algorithm depicted in Figure 6 presents the main steps of updating the existing rules. In Step 1, the data-to-annotation rules are updated. Basically, the denominator in the support and confidence of these rules does not change, and thus only the numerator values need to be re-computed. This update can be performed by checking only the newly annotated data tuples and counting the number of new occurrences of the rule's pattern ( $L.H.S \cup R.H.S$ ). This count will be added to the old numerator to compute the new values. Since all of these rules are guaranteed to be in the output set  $U'$ , they are directly copied to  $U'$  after updating their support and confidence values.

In Step 2, the annotation-to-annotation rules are updated. The first `For...End For` loop handles the case where the new annotations do not appear in the L.H.S of an existing rule, but appear on the R.H.S. This case is very similar to Step 1, where all the rules will get their support and confidence updated (only the numerator values), and then copied to the output set  $U'$ . The second `For...End For` loop handles the case where the new annotations appear on the L.H.S of the rules. In this case both the numerator and the denominator values of the confidence may change and hence, it may increase or decrease. Fortunately, updating these values can be also performed by only checking only the newly annotated data tuples and counting the number of new occurrences that will be added to either of the numerator or denominator values. Depending on the new confidence, if the rule is still valid, then it will be copied to the output set  $U'$ . It is worth highlighting that in updating the existing association rules (Steps 1 & 2 in Figure 6), we only need to process the newly annotated data tuples without touching the rest of the database.

The addition of the new annotations (the  $\delta$  batch) may also create new association rules. The algorithm depicted in Figure 7 outlines the procedure of incrementally discovering the new rules. In Step 1, the new data-to-annotation rules in the form of  $x_1 x_2 \dots x_k \implies a$  will be discovered, where  $a \in \delta$ . First,  $a$  must be a frequent an-

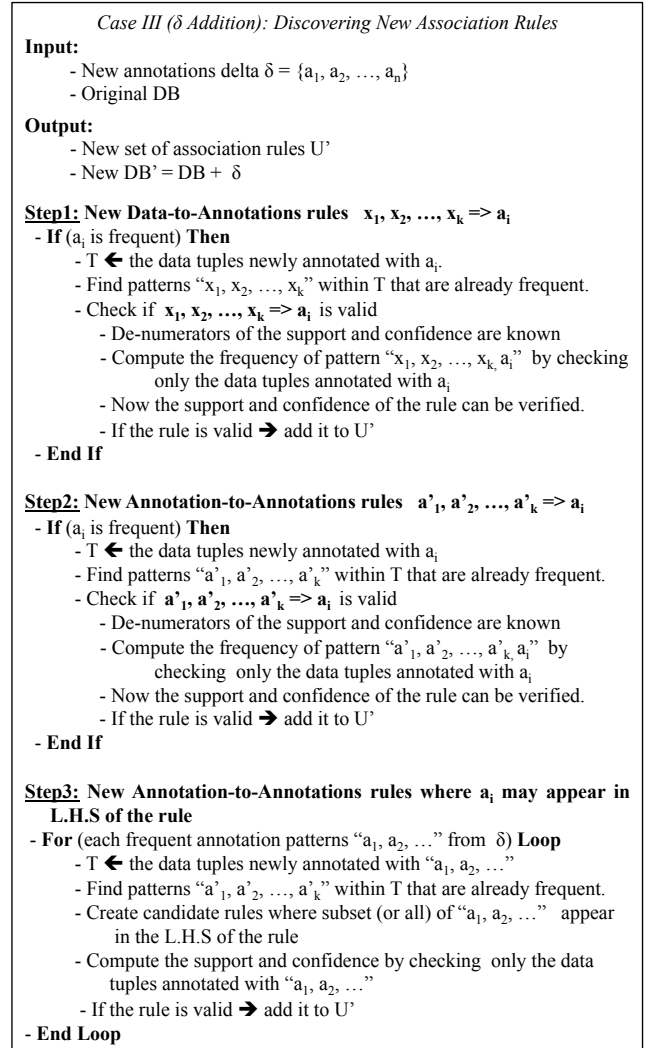


**Figure 6: Case III ( $\delta$  Addition): Updating Existing Rules.**

notation by itself. To perform this check efficiently, the system maintains a table containing the frequency of each annotation, and it is updated whenever a new annotation is added. If  $a$  is frequent, then from the newly annotated tuples, denoted as  $T$ , we extract the data-value patterns that are already frequent, say  $x_1 x_2 \dots x_k$ . Notice that since  $x_1 x_2 \dots x_k$  is already frequent, then the denominator for the support and confidence of rule  $x_1 x_2 \dots x_k \implies a$  is already known. What is left is to compute the frequency of pattern  $x_1 x_2 \dots x_k, a$ , which can be performed by checking only the data tuples in the database annotated with  $a$ . As illustrated in Figure 7, a similar procedure will be taken in Step 2, i.e., discovering the new annotation-to-annotation rules where the new annotations  $a \in \delta$  contribute only to the R.H.S of the rule.

Discovering the new annotation-to-annotation rules where the new annotations  $a \in \delta$  contribute to the L.H.S is slightly different (Step 3). This is because the denominator of the new rules is no longer known and it has to be computed. The procedure works by considering each new annotation  $a \in \delta$ , and verifying first that it is frequent (if not, then the process stops). And then, for each data tuple  $t$  that is receiving  $a$  as a new annotation, we extract the already-frequent annotation patterns, say  $p = a'_1, a'_2, \dots, a'_k$ , i.e.,  $p$  is already frequent and attached to  $t$ . By augmenting  $a$  to  $p$ , we generate several candidate new rules in the form of  $a'_1, a, \dots, a'_k \implies a'_i$ . Notice that  $a$  can be a new annotation over tuple  $t$ , but it is an already-existing annotation over many other tuples in the database. Therefore, to compute the support and confidence of these rules, we need to check all data tuples in the database having annotation  $a$ . This is enough to compute the support and the confidence of the rule and to verify whether or not it is a valid rule.

It is clear that the algorithm of maintaining the existing rules (Figure 6) is less expensive than that of discovering new rules (Fig-



**Figure 7: Case III ( $\delta$  Addition): Discovering New Rules.**

ure 7). This is because the former requires access to only the newly annotated data tuples, whereas the latter requires access to all data tuples that have annotation  $a \in \delta$  (even if the tuples are not newly annotated with  $a$ ). To efficiently support the latter case, the system indexes the annotations such that given a query annotation, we can efficiently find all data tuples having this annotation. In all cases, there is no need for full database processing or re-discovering the rules from scratch.

### 3.2 Generalization-Based Correlations

Generalizing the raw annotation values to higher concepts may lead to discovering important association rules that cannot be discovered from the raw values. The *Red-Flag* annotation in Figure 1 is a good example of this case. The reason this case is important in annotated databases is that the annotations can be added by many curators, and they may not follow specific ontology. And thus, multiple annotations may carry the same semantics but differ in their raw values. There are extensions to association rule mining techniques that can discover the rules under the presence of a generalization hierarchy [29, 44, 45]. However, these techniques assume that the hierarchy, and the assignments between the raw values to



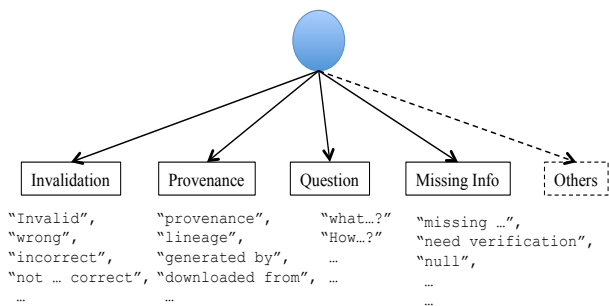


Figure 8: Example of Annotation-Generalization Hierarchy.

the hierarchy elements are given as input, which is usually not the case in annotated databases.

In this work, we assume the more practical case where the domain experts know the generalization hierarchy, i.e., the generic types of annotations of interest, but the raw annotations are not yet labeled. An example of this hierarchy is illustrated in Figure 8, where several types can be of interest, e.g., “Invalidation” (comments that highlight errors or invalid values), “Provenance” (comments capturing the source of data or how it is generated), “Question” (comments including questions), and “Missing Info” (comments highlighting missing values or more investigation). The hierarchy will always include a separate class, called “Others” to which any un-generalized annotation will belong.

In general, any text classification technique can be used. As a proof of concept, we use the technique proposed in [18]. Since we assume no training set or classifier model is given, the model is first created as follows:

- 1- Select  $\alpha$  real annotations at random and manually label them to build the 1<sup>st</sup> classifier model.
- 2- Select  $\beta\%$  of the real annotations at random and classify them using the trained classifier.
- 3- Manually verify the results, and re-label the wrong classification to refine the model.
- 4- Repeat Steps 2 & 3 until achieving an acceptable accuracy.

In Step 1, in addition to manually labeling an initial set of  $\alpha$  annotations, we also create synthetic annotations to each class capturing the keywords in that class. For example, as highlighted in Figure 8, keywords like “wrong”, “incorrect”, “invalid” are embedded in synthetic annotations under the 1<sup>st</sup> class label, while keywords like “source”, “generated from” are added under the 2<sup>nd</sup> class label. The creation process iterates between labeling and verifying a small subset of annotations ( $\beta\%$ ) until the classifier reaches an acceptable accuracy (Step 4).

After building the classifier model, it is applied over all annotations in the dataset, and data tuples get annotated with the class labels corresponding to their raw annotations—Except for “Others” label for which its annotations are not generalized. A data tuple can have a given label at most once even if there are multiple raw annotations mapping to the same label. This is the same model used in association rule mining techniques that handle a generalization hierarchy [29, 44, 45]. For example, the top data tuple in Figure 9 has two attached annotations classified into the “Invalidation” label, i.e., the Red Flag, and thus after attaching the classification decision, it generates the tuple at the bottom of the figure.

After building the extended annotated database, existing tech-

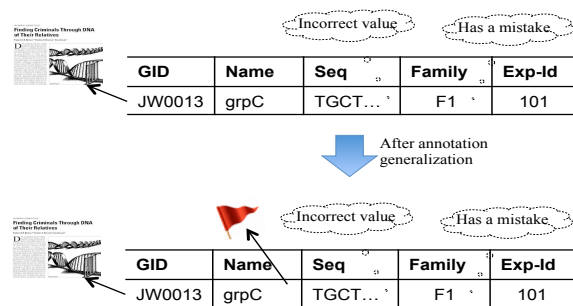


Figure 9: Applying Annotation-Generalization over Example Tuple.

niques can mine and extract the data-to-annotation association rules in the form of:  $x_1 x_2 \dots x_k \implies b$ , where  $b$  is either a raw annotation or an annotation’s class label. The same applies for annotation-to-annotation rules, which are in the form of:  $b_1 b_2 \dots b_k \implies b$ , where none of the L.H.S or R.H.S values (either raw or class label values) have an ancestor-descendant relationship in the generalization hierarchy.

#### 4. EXPLOITATION OF CORRELATIONS

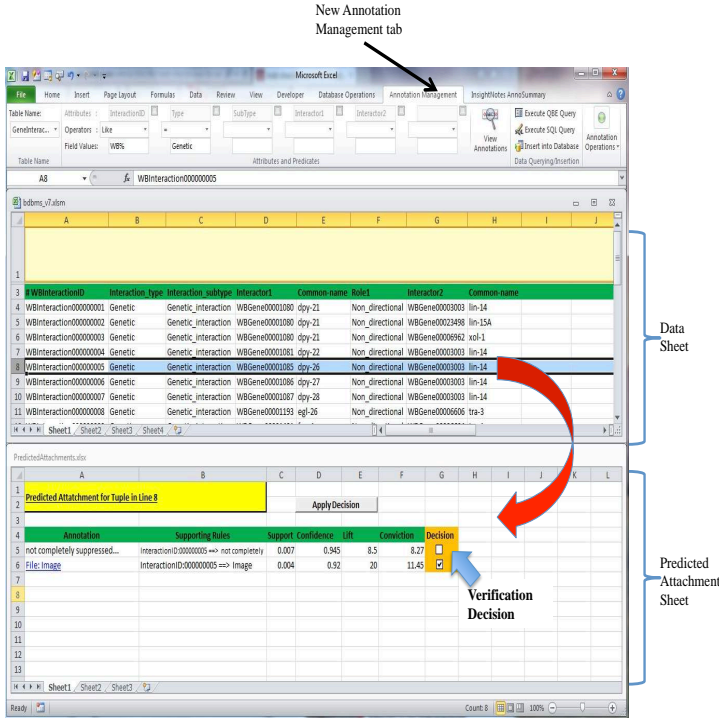
As discussed in Section 1, one of our goals is to exploit the discovered correlations to enhance the quality of the annotated database. We consider two exploitation scenarios: (1) The prediction of related annotations to newly inserted tuples (Motivation Scenario 2), and (2) The discovery of missing attachments when new association rules are found (Motivation Scenario 1).

**Insertion of New Data Tuples:** For this case, an automatic database trigger (at *After Insert at Row Level*) is created for each database table. The trigger forwards a newly inserted tuple  $t$  to the Annotation Manager, which checks  $t$  against the available association rules. If the L.H.S pattern of a rule is present in  $t$  without the R.H.S annotation, then the system creates a recommendation that the R.H.S annotation is potentially applicable to  $t$ . For example, referring to the motivation example in Figure 1, the following rule can be derived from the data:

Exp-Id:101  $\implies$  “Sequence need to be shifted by 2 bases.”

The newly added tuple JW0027 contains the L.H.S of the rule, and hence the system predicts and creates a recommendation that the R.H.S annotation may be related to the new tuple. The end-user will be notified that the system has created annotation predictions for the new tuple, and the prediction(s) will be stored in a system table along with its supporting rule (the rule generated the recommendation) for later verification and approval as explained in sequel.

To enable efficient searching for the association rules matching the newly inserted tuple, the L.H.S of the rules are indexed. Since the L.H.S may contain several pairs in the form of `columnName:value`, we first itemize and store the L.H.S in a normalized form, i.e., one record for each pair, and then index them using a B-Tree index. In this case, given a new data tuple, the *After Insert* database trigger will create lookup keys from the new tuple, i.e., each column name and its value will form one lookup key, and then search the normalized table to find a “superset” of the candidate rules. The tuple will be checked against this candidate set to find the actual matches.



**Figure 10: Exploitation of Correlations and Annotation-Related Recommendations.**

**Updating the Association Rules:** As presented in Section 3.1, the annotation-related association rules may change periodically when new batches of data or annotations are applied. The change may take three forms: (1) Valid rules remain valid, (2) Valid rules become invalid, and (3) New rules are discovered. In the 1<sup>st</sup> case, nothing will change in the system. In the 2<sup>nd</sup> case, the pending recommendations awaiting verification whose supporting rules become invalid will be eliminated. This is because the system will not recommend attachments without rules supporting available as evidences.<sup>1</sup> In the 3<sup>rd</sup> case, the discovery of new rules will trigger the system to search for data tuples matching the L.H.S of the new association rules, but missing the R.H.S annotation in the rule. If a matching is found, then a new pending recommendation will be added to the system table. Given a newly discovered rule having a L.H.S in the form of: “ $C_1 : v_1, C_2 : v_2, \dots, C_m : v_m$ ”, where  $C_i$  is a column name and  $v_i$  is the column’s value, the searching for the matching data tuples is performed as follows. A query on the same table to which the new rule is related is formed, and it consists of a set of conjunctive predicates in the form of  $C_i = v_i \forall 1 \leq i \leq m$ . The returned tuples will be then checked if they are missing the R.H.S of the rule, i.e., the annotation value, and if so, then a new recommendation is generated.

It is worth highlighting that the number of discovered rules by considering solely the *support* and *confidence* thresholds is in most cases very high [31, 44]. This will result in many uninteresting predictions and recommendation. To overcome this problem, we integrate additional properties such as *lift* and *conviction* measures that measure how interesting the rule is (See Section 2). That is,

<sup>1</sup>Recommended attachments that have been approved by the end-users will remain in the system even if their supporting rules become invalid at a later point in time. This is because an approved attachment is viewed as a correct and permanent one.

among all the rules that satisfy the support and confidence requirements, we use in the exploitation process only a subset of these rules that also satisfy the lift and conviction requirements.

**Manipulation Interface:** To seamlessly enable the verification process, we extend the Excel-based GUI presented in Section 2.2 such that domain experts can visualize the recommendations from the tool and decide whether or not each prediction will be accepted (See Figure 10). The tool enables reporting and visualizing the pending predictions either by providing a database table name, or by specifying a select statement to limit the scope of interest. The reported predictions can be then sorted according to various criteria, e.g., the confidence of the association rule suggested the prediction. As Figure 10 illustrates, the data tuples from a given table (or a select statement) are reported on the top-level excel sheet, and then when a tuple is selected, its related predictions and recommendations are dynamically reported on the bottom-level sheet. For each prediction, the supporting association rule is displayed along with its properties, e.g., the support, confidence, lift, and conviction. Curators can then use the checkbox highlighted in the figure to decide on whether or not to accept the recommendation.

## 5. RELATED WORK

Annotation management is widely applicable to a broad range of applications, yet it gained a significant importance within the context of scientific applications [27, 35, 40]. Therefore, to help scientists in their scientific exterminations and to boost the discovery process, several generic annotation management frameworks have been proposed for annotating and curating scientific data in relational DBMSs [9, 17, 24, 25, 26, 46]. Several of these systems, e.g., [9, 17, 24, 46], focus on extending the relational algebra and query semantics for propagating the annotations along with the queries’ answers at query time. The techniques presented in [46] address the annotation propagation for containment queries, while the techniques [13] address the propagation in the presence of logical database views. They address, for example, the minimum amount of data that need to annotated in the base table(s) in order for the annotation to appear (propagate) to the logical view. The work in [21] proposed compact storage mechanisms for storing multi-granular annotations at the raw-, cell-, column-, and table-levels, as well as defining behaviors for annotations under the different database operations. Moreover, the techniques proposed in [21, 36] enable registering annotations in the database system and automatically applying them to newly inserted data tuples if they satisfy pre-defined predicates.

Other systems have addressed special types of annotations, e.g., [15, 23]. For example, the work in [15] have addressed the ability to annotate the annotations, and hence they proposed a hierarchical approach that treats annotations as data. On the other hand, the BeliefDB system in [23] introduced a special type of annotations, i.e., *the belief annotation*, that captures the different users’ beliefs either about the data or others’ beliefs. In both systems, the query engine is extended to efficiently propagate/query these annotations. It have been also recognized in [28, 34] that annotations may have semantics and based on these semantics the propagation in the query pipeline may differ, e.g., instead of getting the union of annotations under the join operation, getting the intersection makes more sense under some annotation semantics. In our previous work [50], we addressed the challenge of managing number of annotations that can be orders of magnitude larger than the number of base data tuples. In this case, reporting the raw annotations will be overwhelming and useless. Instead, we proposed summarizing the annotations into concise forms, and then proposed an extended query engine to efficiently propagate these summaries.

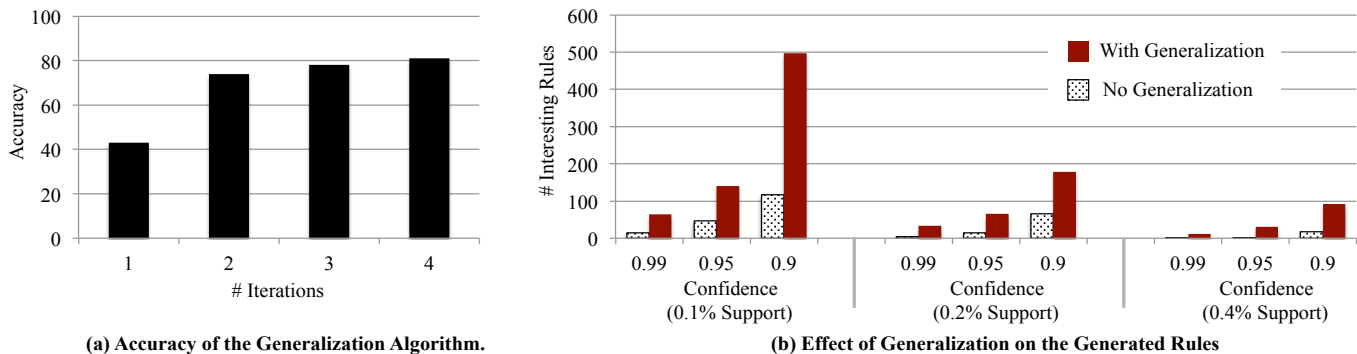


Figure 11: Annotation Generalization and Rule Generation.

Although the above systems provide efficient query processing for annotations, none of them have addressed the facet of mining the rich repositories of annotations to discover interesting patterns, e.g., discovering the annotation-related correlations proposed in this paper. Therefore, the proposed work is complementary to existing systems and creates an automated mechanism for enhancing the quality of annotated databases, which is currently handled through a manual curation process [2, 6]. In this process, domain experts manually curate the data, ensure correct and high-quality annotations are attached to the data, and potentially add or remove further attachments between the annotations and the data. Certainly, this curation process is very time consuming, error-prone and does not scale well, and more importantly consumes valuable cycles from domain experts and scientists. With the proposed work, a significant effort in discovering missing attachments and relationships between the data and the annotations can be automated.

Annotations have been supported in contexts other than relational databases, e.g., annotating web documents [10, 33, 37, 51], and pdf files [38, 39, 48]. These techniques focus mostly on annotating different pieces within a document (or across documents) with useful information, e.g., extracting the key objects or providing links to other related objects. In the domains of e-commerce, social networks, and entertainment systems [22, 42], the annotations are usually referred to as *tags*. These systems deploy advanced mining and summarization techniques for extracting the best insight possible from the annotations to enhance users' experience. They use such extracted knowledge to take actions, e.g., providing recommendations and targeted advertisements [7, 19, 42]. However, none of these systems focus on mining and discovering correlations within the annotation repositories.

## 6. EXPERIMENTS

**Setup and Dataset:** The experiments are performed using our annotation management engines [21, 50], which are based on the open-source PostgreSQL DBMS. The experiments are conducted using an AMD Opteron Quadputer compute server with two 16-core AMD CPUs, 128GB memory, and 2 TBs SATA hard drive. Our objectives are: (1) To quantify the effectiveness of the annotation generalization technique, (2) The performance of discovering the annotation-based association rules over static data, and (3) The efficiency of the proposed incremental techniques to update the rules under new batches of annotations. The experimental dataset represent the *C. elegans* repository, which we are building on site. The repository consists of 27 database tables, where the main table of our focus is the *Gene* table that contains approximately 17,500 genes integrated from multiple sources. The table has nine

columns, e.g., *Gene Id*, *Locus*, *Strain*, *Stage*, *Location*, and *Length*. The table has a total number of 43,000 publications attached to the genes in addition to other 8,120 free-text comments. These publications and comments represent the annotations attached to the *Gene* table. In the dataset, each record has between 0 annotations (the minimum) and 16 annotations (the maximum).

In Figure 11, we study the effect of annotation generalization on the generation of interesting annotation-related association rules. Figure 11(a) illustrates the accuracy of the generalization algorithm presented in Section 3.2. The x-axis indicates the number of iterations from 1 to 4, while the y-axis represents the obtained accuracy from the manual verification step (Step 3 of the algorithm). As the figure shows, there is a big jump in accuracy from Iteration 1 to Iteration 2, and then as more iterations are performed the accuracy slightly increases. This is mostly because our generalization model is simple and consists of one level only. However, as the generalization model becomes more complex, we expect to more iterations will be needed to reach the desired accuracy. In the subsequent experiments, we will use the results obtained after performing 4 iterations.

In Figure 11(b), we present the number of interesting association rules discovered in the entire dataset under varying confidence and support degrees (the x-axis). Since the expected rules are not necessarily globally frequent, we use very low support, e.g., 0.1% to 0.4%, while setting the confidence to a very high threshold as indicated in the figure. As expected, as the support or confidence thresholds increase, less number of interesting rules are discovered. The number of discovered rules is relatively manageable and not very large because we also enforce minimum lift and conviction thresholds to narrow down the reported rules to the strongest ones only. Both thresholds are set to value 5. In the remaining experiments, we will consider the dataset under the annotation generalization case, i.e., the annotations have been generalized to enable the discovery of more rules.

In Figure 12, we study the execution time of discovering the annotation-related rules in the entire dataset. We vary the confidence and support thresholds as depicted in the figure. In this experiment that dataset is static, i.e., there are no new batches of data or annotations. As the figure illustrates the execution time ranges from 8 secs to 22 secs depending on the used thresholds. We have also studied the execution time of the mining algorithm without annotation generalization and the observed differences are not significant, e.g., the technique without generalization are faster by 1% to 7% compared to the other case.

In Figure 13, we study the execution time under the addition of new annotation batches. We focus on the 3<sup>rd</sup> case presented in

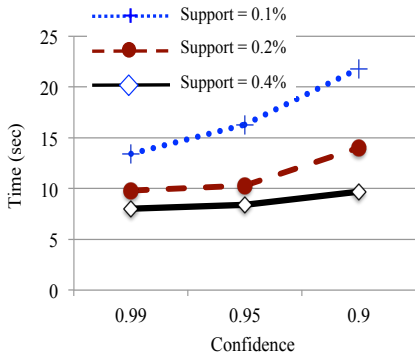


Figure 12: Evaluation of Execution Time.

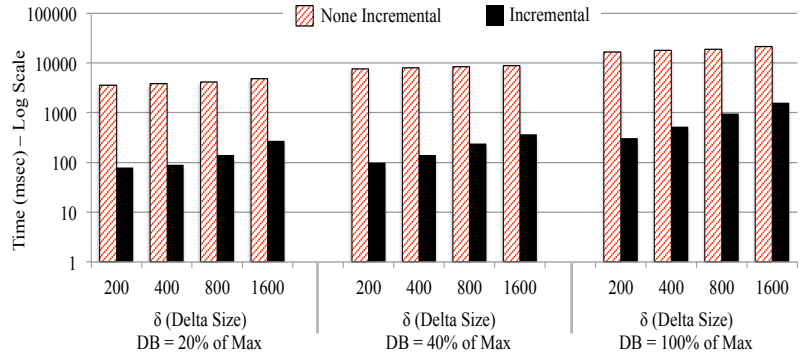


Figure 13: Execution Time of Incremental vs. Non-Incremental Algorithms (Case III).

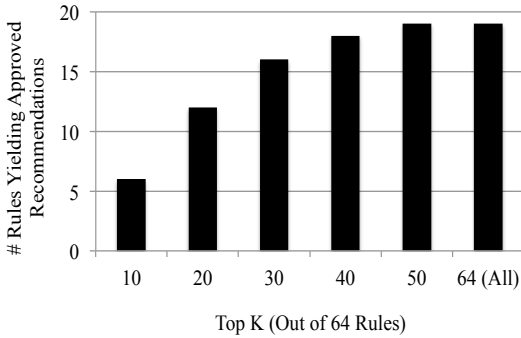


Figure 14: Number of Rules Yielding Approved Recommendations.

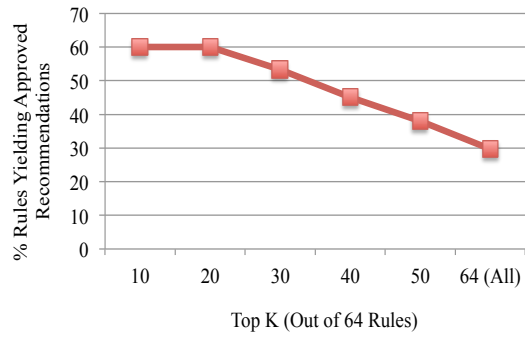


Figure 15: Percentage of Rules Yielding Approved Recommendations.

Section 3.1 since this case is not supported by existing association rule mining techniques. In the experiment, we set the confidence and support thresholds to values 95%, and 0.2%, respectively. In the x-axis of the figure, we vary the number of newly added annotations ( $\delta$ ) between 200 and 1,600. This is performed by isolating *delta* annotations (publications) from the original dataset, discovering the association rules on the modified dataset, and then adding the isolated annotations back as the new delta batch. We perform the experiment on three different dataset sizes, i.e., small (20% of the entire dataset), medium (40% of the entire dataset), and large (the entire dataset). We compared between a naive non-incremental A-priori technique [8] versus the incremental technique proposed in Section 3.1. As the results show, the incremental algorithm outperforms the non-incremental one by up to two orders of magnitude while producing the same exact results.

In Figures 14 and 15, we investigate the virtue of the exploitation process within which the system proactively uses the discovered association rules and provides recommendations to enrich the annotated database. In the experiment, we set the confidence and support thresholds to 95% and 0.2%, respectively (resulting in 64 discovered rules as illustrated in Figure 11(b)). We then, in Figures 14 and 15 select the top  $K$  strongest rules for generating recommendations. The  $K$  varies over the values between 10 to 64 as indicated on the x-axis. Each rule, may generate many recommendations, e.g., attaching a specific annotations to some data tuples, and each recommendation is supported by some evidences (Refer to Figure 10). The results presented in Figure 14 and 15 illustrate that yield to at least one recommendation being approved by the database admin. This means that these rules were valuable as they

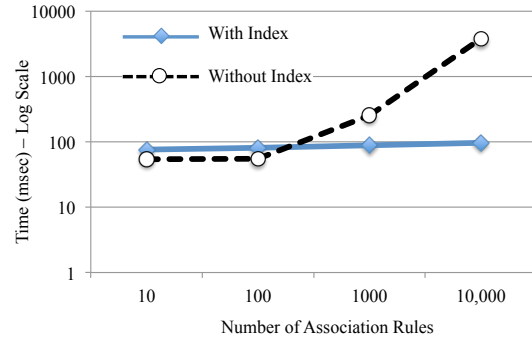


Figure 16: Searching for Matching Rules under New Tuples Insertions.

helped discovering missing attachments. The results show that a fair number of rules have yielded to approved recommendations. Moreover, the strongest rules, e.g., Top 10 or 20, usually yield to more realistic recommendations compared to the weaker rules, e.g., Top 50, or 60. The results in Figures 14 and 15 are interesting as they show that we may not even use all the discovered interesting rules for recommendation purposes. We can depend on only the top ranked ones to achieve high acceptance rate while reducing the domain experts' efforts in the verification process.

In Figure 16, we study the performance of searching for matching association rules under the insertion of new data tuples. We compare the two cases where the search does not use an index, i.e., scanning all existing rules, versus the use of index, i.e., the rules

are normalized and indexed using the B-Tree index (Refer to Section 4). In the experiment, we varied the number of rules from 10 to 10,000 (the x axis), and measured the search time (the y axis). The experiment is repeated 10 times, and the average values are presented in the figure. As expected, the “*With Index*” case scales much better and remains stable as the number of association rules gets larger. Whereas, the “*Without Index*” case has slightly better performance when the number of rules is very small. This is because the key lookups over the index—which are multiple lookups per tuple—add unnecessary overhead when the number of rules is very small. In this case, one scan over all rules becomes faster.

## 7. CONCLUSION

In this paper, we investigated a new facet of annotation management, which is the discovery and exploitation of the hidden annotation-related correlations. The addressed problem is driven by the emerging real-world applications that create and maintain large-scale repositories of annotated databases. The proposed work opens a new application domain to which the well-known association rule mining can be applied. We show cased several scenarios specific to annotated databases that cannot be efficiently handled by the state-of-art in association rule mining. We then proposed algorithms for efficient and incremental maintenance of the discovered association rules under these scenarios. We proposed two important applications for leveraging the discovered annotation-related correlations and enhancing the quality of the underlying database, which are the discovery of missing attachments, and the recommendation of applicable annotations to newly inserted data. To enable seamless use by scientists, we integrated the proposed algorithms within the annotation management engine and developed an end-to-end system including an Excel-based GUI through which all of the proposed functionalities can be performed.

## 8. REFERENCES

- [1] eBird Trail Tracker Puts Millions of Eyes on the Sky. [https://www.fws.gov/refuges/RefugeUpdate/MayJune\\_2011/ebirdtrailtracker.html](https://www.fws.gov/refuges/RefugeUpdate/MayJune_2011/ebirdtrailtracker.html).
- [2] EcoliHouse: A Publicly Queryable Warehouse of E. coli K12 Databases. <http://www.porteco.org/>.
- [3] Gene Ontology Consortium. <http://geneontology.org>.
- [4] Hydrologic Information System CUAHSI-HIS. (<http://his.cuahsi.org>).
- [5] The Avian Knowledge Network (AKN). <http://www.avianknowledge.net/>.
- [6] The Universal Protein Resource Databases (UniProt). <http://www.ebi.ac.uk/uniprot/>.
- [7] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE TKDE*, 17(6):734–749, 2005.
- [8] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB*, pages 487–499, 1994.
- [9] D. Bhagwat, L. Chiticariu, and W. Tan. An annotation management system for relational databases. In *VLDB*, pages 900–911, 2004.
- [10] A. J. B. Brush, D. Barger, A. Gupta, and J. Cadiz. Robust Annotation Positioning in Digital Documents. In *Proceedings of the 2001 ACM Conference on Human Factors in Computing Systems (CHI)*, pages 285–292. ACM Press, 2001.
- [11] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD*, pages 539–550, 2006.
- [12] P. Buneman, J. Cheney, W.-C. Tan, and S. Vansummeren. Curated databases. In *Proceedings of the 27th ACM symposium on Principles of database systems (PODS)*, pages 1–12, 2008.
- [13] P. Buneman and et. al. On propagation of deletions and annotations through views. In *PODS*, pages 150–158, 2002.
- [14] P. Buneman, S. Khanna, and W. Tan. Why and where: A characterization of data provenance. *Lec. Notes in Comp. Sci.*, 1973:316–333, 2001.
- [15] P. Buneman, E. V. Kostylev, and S. Vansummeren. Annotations are relative. In *Proceedings of the 16th International Conference on Database Theory, ICDT '13*, pages 177–188, 2013.
- [16] D. Cheung, J. Han, V. Ng, and C. Wong. Maintenance of discovered association rules in large databases: an incremental updating technique. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 106–114, Feb 1996.
- [17] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. DBNotes: a post-it system for relational databases based on provenance. In *SIGMOD*, pages 942–944, 2005.
- [18] P. R. Christopher D. Manning and H. Schütze. Book Chapter: Text classification and Naive Bayes, in Introduction to Information Retrieval. In *Cambridge University Press*, pages 253–287, 2008.
- [19] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web (WWW)*, pages 271–280, 2007.
- [20] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, pages 1345–1350, 2008.
- [21] M. Eltabakh, W. Aref, A. Elmagarmid, and M. Ouzzani. Supporting annotations on relations. In *EDBT*, pages 379–390, 2009.
- [22] A. Gattani and et. al. Entity extraction, linking, classification, and tagging for social media: a wikipedia-based approach. *Proc. VLDB Endow.*, 6(11):1126–1137, 2013.
- [23] W. Gatterbauer, M. Balazinska, N. Khoussainova, and D. Suciu. Believe it or not: adding belief annotations to databases. *Proc. VLDB Endow.*, 2(1):1–12, 2009.
- [24] F. Geerts and et. al. Mondrian: Annotating and querying databases through colors and blocks. In *ICDE*, pages 82–93, 2006.
- [25] F. Geerts, A. Kementsietsidis, and D. Milano. iMONDRIAN: a visual tool to annotate and query scientific databases. In *Proceedings of the 10th international conference on Advances in Database Technology (EDBT)*, pages 1168–1171, 2006.
- [26] F. Geerts and J. Van Den Bussche. Relational completeness of query languages for annotated databases. In *Proceedings of the 11th international conference on Database Programming Languages (DBPL)*, pages 127–137, 2007.
- [27] J. Gray, D. T. Liu, M. Nieto-Santesteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4):34–41, 2005.

- [28] T. J. Green. Containment of conjunctive queries on annotated relations. In *ICDT*, pages 296–309, 2009.
- [29] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *In Proc. 1995 Int. Conf. Very Large Data Bases*, pages 420–431, 1995.
- [30] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD, pages 1–12, 2000.
- [31] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Algorithms for association rule mining &mdash; a general survey and comparison. *SIGKDD Explor. Newsl.*, 2(1):58–64, June 2000.
- [32] K. Ibrahim, D. Xiao, and M. Y. Eltabakh. Elevating Annotation Summaries To First-Class Citizens In InsightNotes. In *EDBT Conference*, 2015.
- [33] J. Kahan and M.-R. Koivunen. Annotea: an open RDF infrastructure for shared Web annotations. In *Proceedings of the 10th international conference on World Wide Web (WWW)*, pages 623–632, 2001.
- [34] E. V. Kostylev and P. Buneman. Combining dependent annotations for relational algebra. In *Proceedings of the 15th International Conference on Database Theory (ICDT)*, pages 196–207, 2012.
- [35] L. S. P. Lee, Woei-Jyh; Raschid. Mining Meaningful Associations from Annotations in Life Science Data Resources. *Proceedings of the Conference on Data Integration for the Life Sciences*, 1(1), 2008.
- [36] Q. Li, A. Labrinidis, and P. K. Chrysanthis. ViP: A User-Centric View-Based Annotation Framework for Scientific Data. In *Proceedings of the 20th international conference on Scientific and Statistical Database Management (SSDBM)*, pages 295–312, 2008.
- [37] M. Markovic, P. Edwards, D. Corsar, and J. Z. Pan. DEMO: Managing the Provenance of Crowdsourced Disruption Reports. In *IPAW*, pages 209–213, 2012.
- [38] C. MARSHALL. Annotation: from paper books to the digital library. *ACM Digital Libraries* (1997).
- [39] C. MARSHALL. The future of Annotation in a Digital (paper) world. The 35th Annual GSLIS Clinic: Successes and Failures of Digital Libraries University of Illinois at Urbana-Champaign (1998).
- [40] L. Martšnez-Cruz, A. Rubio, M. Martšnez-Chantar, A. Labarga, L. Barrio, and A. Podhorski. GARBAN: genomic analysis and rapid biological annotation of cDNA microarray and proteomic data. *Bioinformatics*, 19(16):2158–2163, 2003.
- [41] B. Nath, D. K. Bhattacharyya, and A. Ghosh. Incremental association rule mining: a survey. *Wiley Interdisciplinary, Data Mining and Knowledge Discovery*, 3(3):157–169, 2013.
- [42] A. Rae, B. Sigurbjörnsson, and R. van Zwol. Improving tag recommendation using social networks. In *Adaptivity, Personalization and Fusion of Heterogeneous Information*, RIAO, pages 92–99, 2010.
- [43] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.
- [44] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the 21th International Conference on Very Large Data Bases, VLDB*, pages 407–419, 1995.
- [45] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 1996.
- [46] W.-C. Tan. Containment of relational queries with annotation propagation. In *DBPL*, 2003.
- [47] D. Tarboton, J. Horsburgh, and D. Maidment. CUAHSI Community Observations Data Model (ODM), Version 1.1, Design Specifications. In *Design Document*, 2008.
- [48] P. TUCKER and D. JONES. Document annotation - to write, type or speak. In *International Journal of Man-Machine Studies*, pages 885–900, 1993.
- [49] M. Twombly. Science online survey: Support for data curation. *Science Journal*, 331, 2011.
- [50] D. Xiao and M. Y. Eltabakh. InsightNotes: Summary-Based Annotation Management in Relational Databases. In *SIGMOD Conference*, pages 661–672, 2014.
- [51] H. Yang, D. T. Michaelides, C. Charlton, W. J. Browne, and L. Moreau. DEEP: A Provenance-Aware Executable Document System. In *IPAW*, pages 24–38, 2012.
- [52] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000.

# Query Performance Problem Determination with Knowledge Base in Semantic Web System OptImatch

Guilherme Damasio\*  
University of Ontario Institute of Technology  
Faculty of Science, Computer Science  
guilherme.fetterdamasio@uoit.ca

Jaroslav Szlichta\*  
University of Ontario Institute of Technology  
Faculty of Science, Computer Science  
jaroslav.szlichta@uoit.ca

Piotr Mierzejewski  
IBM Canada Ltd  
piotrm@ca.ibm.com

Calisto Zuzarte  
IBM Canada Ltd  
calisto@ca.ibm.com

## ABSTRACT

Database query performance problem determination is often performed by analyzing query execution plans (QEPs) in addition to other performance data. As the query workloads that organizations run have become larger and more complex, analyzing QEPs manually even by experts has become a very time consuming and cumbersome task. Most performance diagnostic tools help with identifying problematic queries and most query tuning tools address a limited number of known problems and recommendations. We present the OptImatch system that offers a way to (a) look for varied user defined problem patterns in QEPs and (b) automatically get recommendations from an expert provided and user customizable knowledge base. Existing approaches do not provide the ability to perform workload analysis with flexible user defined patterns, as they lack the ability to impose a proper structure on QEPs. We introduce a novel semantic web system that allows a relatively naive user to search for arbitrary patterns and to get solution recommendations stored in a knowledge base. Our methodology includes transforming a QEP into an RDF graph and transforming a GUI based user-defined pattern into a SPARQL query through handlers. The SPARQL query is matched against the abstracted RDF graph, and any matched portion of the abstracted RDF graph is relayed back to the user. With the knowledge base, the OptImatch system automatically scans and matches interesting stored patterns in a statistical way as appropriate and returns the corresponding recommendations. Although the knowledge base patterns and solution recommendations are not in the context of the user supplied QEPs, the context is adapted automatically through the handler tagging interface. We test the performance and scalability of our framework to demonstrate its efficiency using a real query workload. We also perform a user study to quantify the benefits of the approach in terms of precision and time compared to manually searching for patterns.

## Categories and Subject Descriptors

H.2.4 [Database Management]: Systems – Query Processing

\* Damasio is a Student Fellow in IBM Centre for Advanced Studies (CAS) and Szlichta is a Faculty Fellow in IBM CAS in Toronto.

© 2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

## General Terms

Performance, Design and Experimentation

## Keywords

Query Performance, Problem Determination, Semantic Web, Knowledge Bases and Business Intelligence.

## 1. INTRODUCTION

### 1.1 Background and Motivation

Much of the world's high-valuable data remain in relational databases (e.g., operational databases and data warehouses [12]). Access to this data is gained through relational query languages such as Structured Query Language (SQL). Complex analytic queries on large data warehouse system are not only done as weekend or end of period canned batch reports. Ad hoc complex queries are increasingly run as part of business operations. As such it is critical to pay attention to performance of these queries.

Database systems themselves are certainly increasingly becoming more sophisticated and able to automatically tune the environments they operate in. General query performance problem determination tools [4], [6], [23], [24] also offer an automated way to database administrators to analyze performance issues that neither requires mastery of an optimizer, nor deep knowledge about the query execution plans (QEPs). However, due to the complexity while the general approach has merit, there is a lack of customization and many refinements are needed, so that the problem determination and tuning process can be truly effective and consumable by the general end-user. Given the specific circumstances and limitations of existing tools, performance analysis today is often best done by manually analyzing optimizer QEPs that provide detail of how queries are executed. Manually analyzing these QEPs can be very demanding and often requires deep expertise particularly with complex queries that are often seen in data warehouse environments. Very often the end users and database administrators resign themselves to opening problem reports to the database vendors so that experts who are well versed in both SQL and analyzing optimizer QEPs can provide recommendations. This can be a time consuming exercise and does not scale well.

Existing tools such as IBM® Optim Query Tuner® and IBM Optim Workload Tuner® provide tuning recommendations for specific known problems. While very effective, they do not, however, provide the ability to perform query performance problem determination with flexible user-defined patterns (examples listed below). This is mainly because these tools are agnostic to the

```

19860.9
NLJOIN
( 2)
16246.59
4909.624
/-----+-----\
19.12          4043
FETCH          TBSCAN
( 3)          ( 5)
26.0884       15771
2.624         4907
/---+-----\
19.12      1228      812130
IXSCAN    SALES_FACT  CUST_DIM
( 4)      Q2          Q1
11708.7
5250
|
9.18948e+07
IDX1
Q2

```

**Figure 1 Query with NLJOIN**

complex structure of QEPs. There does not exist a general purpose automated system that would allow for interactive analysis and diagnosis of performance problems by searching for arbitrary patterns within a large number of QEPs. A user not so experienced with QEPs may want to answer simple questions. For example, after searching and determining the cost of a table scan on a particular table, the user may want to know how many queries in the workload do an index scan access on the table and get a sense of the implications of dropping the index by comparing the index access cost to that of the table scan. Even with more experienced database administrators, often there are clues from monitoring data that provide hints of certain characteristics of QEPs that are not easily found by using typical search tools like *grep*. For instance, given a large number of queries, say 1000 queries, and the corresponding workload QEPs:

- Find all the queries in the workload that might have a spilling hash join below an aggregation and the cost is more than a constant  $N$ .
- Find all the subqueries that have a cost that is more than 50% of the total cost of the query and provide details of the subquery operators (name, cost, and input operators).
- Find all the queries that have an outer join involving the same table somewhere in the plan below both sides of a hash join.
- Perform cost based clustering and correlate results of applying expert patterns to each cluster.

We consider making it easier and faster to automatically answer questions like the above in our work. We provide a flexible system OptImatch that performs analysis over large and complex query workloads, in order to help diagnosis optimizer problems and retrieve solutions that were previously provided by experts. The optImatch system drastically lowers the skill level required for optimizer access plan problem determination through advanced automated pattern matching and retrieving of solution recommendations of previously discovered performance problems for single queries and large query workloads. OptImatch is very well received and is proving to be very valuable in the IBM support

of business clients and database optimizer development organization.

At the enterprise level, major commercial state-of-the-art relational database systems such as IBM DB2®, Oracle®, and Microsoft® SQL Server® are deployed in environments where finding all available optimizations and performance tuning strategies becomes necessary to maintain the usability of the database. Traditional optimization methods often fail to apply when logical subtleties in queries and database schemas circumvent them. The examples of this include cases, where the recommended performance enhancement is to index a table in a particular way, prescribe an integrity constraint such as functional dependency [16] or order dependency [17], create a materialized view [7] or to rewrite manually the proposed SQL query, where orthogonal approach with machine optimization [5], [21], [25] failed to rewrite the query to get the same answer but with a better performance.

The problem pattern comprises a list of operators having particular properties that are of interest to a user, as exemplified in some of the aforementioned problems. By incorporating our query performance problem determination system many optimization problems could be automatically identified and resolved. Figure 1 depicts an example of a text graph version of a snippet of a QEP from IBM DB2. The snippet shows a nested loop join (NLJOIN) of the SALES\_FACT table accessed using an index scan (IXSCAN) with other columns fetched (FETCH) from the table and then joined to the CUST\_DIM table. The numbers immediately above the operator or table name show the estimated number of rows flowing out (cardinality). The numbers in parenthesis show the operator number. Operators are also referred to as Plan Operators (pop) or LOW LEVEL Plan Operators (LOLEPOP) in this paper. Each operator has an estimated Input/Output (I/O) cost, the bottom number below the operator number, and a cumulative cost for itself and all operator below it, the number immediately below the operator number. In the depicted example, a user could be concerned with NLJOIN that has an inner stream of type table scan (TBSCAN). Such query is costly as the NLJOIN operator scans the entire inner table CUST\_DIM for each of the rows from the outer SALES\_FACT table. An example of a solution recommendation might be to provide a recommendation to create an index of the target table of the TBSCAN, in this case CUST\_DIM.

In recent years, more and more customer queries are generated automatically by query managers (such as IBM Cognos®) with business users providing only specific parameters through graphical interfaces [9], [10]. Specific parameters are then automatically translated by query managers into executable SQL queries. Based on analyzing IBM customer workloads there is essentially no limit to the length of the query generated automatically by query managers. It is quite usual to find queries with over one thousand lines of SQL code (hundreds of operators). Such queries are very complex and time consuming to analyze with nesting and stitching of several subqueries into a larger query being a common characteristic. Another common feature is repetitiveness, where similar (or even identical) expressions appear in several different parts of the same query, for instance, in the queries referring to the same view or nested query block multiple times [15], [22]. If there is need to improve the performance of such complex queries, when optimizer failed, it could be time consuming to do this manually. It could take hours or even days to analyze a large query workload. Our goal is automate this process as much as possible, and therefore save significant amount of time spent by users on query performance problem determination. The OptImatch system makes this process easier. While optimizers are constantly improving, OptImatch allows experts through their



experience to create the interesting problem patterns and recommendations to overcome issues.

We decided to use the RDF format as it allows one to easily retrieve information with the SPARQL query language. SPARQL has the capability for querying optional and required graph patterns. Another powerful feature exploited in SPARQL is that of property paths. A property path is a possible route through a graph between two graph nodes. SPARQL property paths provide a succinct way to write parts of graph patterns and to also extend matching patterns to arbitrary length paths. With property paths, we can handle recursive queries, for instance, search for a descendant operator that does not necessarily have an immediate relationship (connection) with its parent. We can also search for patterns that appear multiple times in the same QEP. Last but not least, SPARQL allows graph traversal and pattern matching in a very efficient way [3], enabling analysis of a large number of complex QEPs in a short period of time.

While the focus of this work is on query performance problem determination, our methodology can be applied to other general software problem determination [26], assuming that there exists automatically or dynamically generated diagnostic information that needs to be further analyzed by an expert. Broadly, the contemplated diagnostic data may be human-readable and intended for review by human users of the system to which the diagnostic data relates. Examples of possible diagnostic data include log data relating to network usage, security, or compiling software, as well as software debug data or sensor data relating to some physical external system. In these scenarios, the problem pattern may correspond to any sequence of data points or interrelationships of data points that are of diagnostic interest.

## 1.2 Contributions

The main contributions of this paper appear in Section 2 and Section 3 as follows.

1. We developed a semantic web tool to transform a QEP into an abstracted artefact structure (RDF graph). We propose in our framework to model features of the QEP into a set of entities containing properties with relationships established between them. (Section 2.1)
2. We provide a web-based graphical interface for the user to describe a problem pattern (pattern builder). The tool transforms this pattern into a SPARQL query through handlers. Handlers provide the functionality of automatically generating variable names used as part of the SPARQL query. The SPARQL query is executed against the abstracted RDF structure and any matched portions of RDF structure are relayed back to the user. We present a suite of real-world IBM customer problem patterns that illustrate the issues related to query performance, which are then used in Section 3 for experimental performance evaluation. (Section 2.2).
3. We added a knowledge base capability within the tool that could be populated with some expert provided patterns and solution recommendations as well as allow users to add their own patterns and recommendations. The system automatically matches problem patterns in knowledge base to the QEPs and if there are any search results ranks them using statistical correlation analysis. OptImatch distinguishes between a pattern builder and a tagging handler interface to achieve generality and extensibility. In a nutshell, the pattern builder allows the users to specify what is wrong with the query execution plan (static semantics), and the handler tagging interface defines how to report and fix it (dynamic semantics) through automatically adopting the context. Since

the knowledge base patterns and solution recommendations are not in the context of the user supplied QEPs, we have defined the language that users can use to add dynamic context to the recommendations. (Section 2.3)

4. An experimental evaluation showing the performance and effectiveness of our techniques was carried out using real IBM customer datasets. We experimented with different problem patterns, and show that our framework runs efficiently over large and complex query workloads. Our performance evaluation reveals that the time needed to compute a search over a specified problem pattern against a QEP increases linearly with the size of the workload, the number of operators in the QEP and the number of pattern/recommendations in the knowledge base. Finally, we show through a user study that our system is able to save a significant amount of time to analyze QEPs. Moreover, we quantify in the user study the benefits of our approach in terms of precision over manual pattern searching. (Section 3)

In Section 4, we discuss related work. We conclude and consider future work in Section 5.

To the best of our knowledge, we are the first to provide a system for query performance problem determination by applying QEP feature transformation through RDF and SPARQL. This work we feel opens exciting venues for future work to develop a powerful new family of problem determination techniques over existing optimizer performance analysis tools and other diagnostic data exploiting graph databases.

## 2. SYSTEM

### 2.1 Transforming Diagnostic Data

Even though optimizer diagnostic data may differ in some ways between various database management systems, their major characteristics remain the same. Query performance diagnostic information is usually in the form of QEPs formatted in readable text form. An example of the portion of the QEP generated by the IBM DB2 database engine is presented in Figure 1. A QEP includes diagnostic information about base objects (e.g., tables, views and indexes), operators (e.g., join, sort and group-by) as well as costs and characteristics associated with each operator.

Some properties of operators are included in a QEP in the tree diagram as in Figure 1 (e.g., cardinality total cost, Input/Output cost, cumulative cost), wherein other properties appear as separate textual blocks identified by operator number (e.g., cumulative CPU cost, cumulative first row cost and estimated bufferpool buffers). Furthermore, some properties are common between different types of operators (e.g., cardinality, total cost and CPU cost), while others are specific to certain operators. For instance, NLJOIN has a property fetch max, and TBSCAN has a property max pages, but not vice versa. A QEP also contains some other detailed diagnostic data, including information about the DBMS instance and environment settings. All of the techniques described in this paper have been implemented given IBM DB2 QEPs. Hence, much of the discussion through the rest of the paper is framed in the terminology and characteristics of IBM DB2. However, the techniques that are described have general applicability, and can be used with any other DBMS product or other diagnostic data that lends itself to property graph representation.

A QEP can be viewed as a directed graph that indicates the flow of operations processing data within the plan. QEPs resemble a tree structure, where each node (operator) possesses numerous properties and is considered as one of the inputs to a derived

```

<http://explainPlan/PlanPop/2> <http://explainPlan/PlanPred/hasPopType> "NLJOIN" .
<http://explainPlan/PlanPop/2> <http://explainPlan/PlanPred/hasOuterInputStream> <http://explainPlan/PlanPop/3> .
<http://explainPlan/PlanPop/2> <http://explainPlan/PlanPred/hasInnerInputStream> <http://explainPlan/PlanPop/5> .
<http://explainPlan/PlanPop/3> <http://explainPlan/PlanPred/hasPopType> "FETCH" .
<http://explainPlan/PlanPop/3> <http://explainPlan/PlanPred/hasEstimatedCardinality> "19.12" .
<http://explainPlan/PlanPop/5> <http://explainPlan/PlanPred/hasPopType> "TBSCAN" .
<http://explainPlan/PlanPop/5> <http://explainPlan/PlanPred/hasEstimatedCardinality> "4043.0" .
<http://explainPlan/PlanPop/5> <http://explainPlan/PlanPred/hasTotalCost> "15771.0" .
<http://explainPlan/PlanPop/5> <http://explainPlan/PlanPred/hasIOCost> "49007.0" .
<http://explainPlan/PlanPop/5> <http://explainPlan/PlanPred/hasJoinInputLeg> "INNER" .
<http://explainPlan/PlanPop/5> <http://explainPlan/PlanPred/hasOutputStream> <http://explainPlan/PlanPop/2> .
<http://explainPlan/PlanPop/5> <http://explainPlan/PlanPred/hasInputStreamPop> <http://explainPlan/PlanBaseObject/CUST_DIM> .
<http://explainPlan/PlanBaseObject/CUST_DIM> <http://explainPlan/PlanPred/hasEstimateCardinality> "812130.0"

```

Figure 2 Generated RDF in textual representation

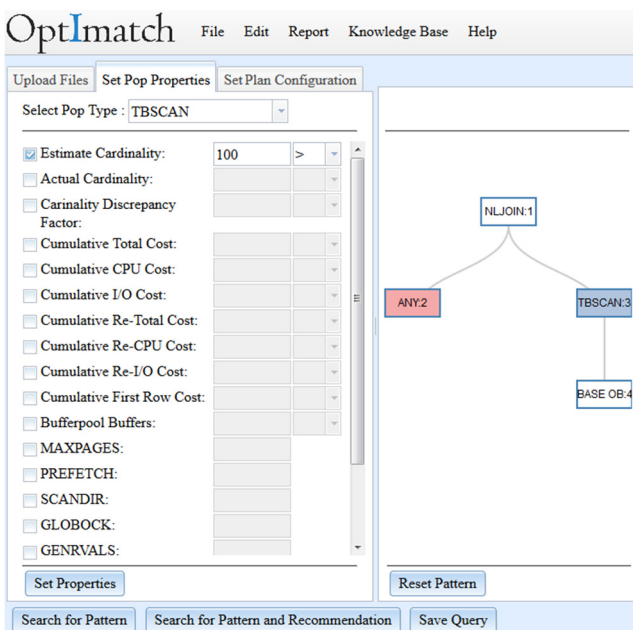


Figure 3 Web-based Graphical Interface (Pattern Builder)

ancestor node. LOLEPOPs in a QEP are connected to their parents as inputs streams. These inputs can be identified with three different types 1) outer input (left input of the parent operator) 2) inner input (right input of the parent operator) and 3) general input stream (generic input used for any kind of operator).

The LOLEPOPs may be understood, at the level of abstraction of the DBMS user, as indivisible operations that are directly executed by the DBMS, with each LOLEPOP carrying a stated cost. The stated cost for each LOLEPOP represents an estimate of server resources, generated by the DBMS system based on a proposed SQL query by taking into account the particular properties of the database. The overall QEP is machine-generated by the DBMS Optimizer [14]. It is machine-optimized to gravitate towards the lowest total cost LOLEPOPs attainable by the DBMS's optimizer. The plan structure is highly dynamic and can change based on configuration, statistics of the data associated with referenced base objects and other factors even if query characteristics remain similar. However, plan changes are difficult to spot manually as they tend to spawn thousands of lines of informative details for more complex queries in the workload.

RDF is a labeled directed graph built out of triples, each containing subject (resource), predicate (property or relationship) and object (resource or value). RDF does not enforce specific schema, hence, two resources in addition to sharing properties and relationships, can also be described by their own unique predicates. This property of RDF is beneficial to describe and preserve various types of complex diagnostic information about QEPs. Even though RDF inherently does not possess a particular structure, such structure can be enforced by specifying predicates (for example, defining predicates, such as *hasInputStreamPop* or *hasOutputStreamPop*, and *hasInnerInputStreamPop* or *hasOuterInputStreamPop* and using them to establish relationships between resources (LOLEPOPs)). This allows one to recreate the tree structure and characteristics used in QEPs.

#### Algorithm 1 Transforming QEPs

**Input:** query execution plan files  $QEPFs[ ]$

**Output:** execution plans represented as RDF Graphs,  $RDFGs[ ]$

- 1: **forall**  $qepf$  in  $QEPFs[ ]$
- 2:      $i := 0$
- 3:      $rdfg :=$  convert  $qepf$  into RDF graph model by traversing through base objects, operators and relationship (input streams) with Jena RDF API
- 4:      $RDFGs[i] := rdfg$
- 5:      $i := i + 1$
- 6: **end forall**
- 7: **return**  $RDFGs[ ]$

We propose in our framework to model features of the QEP into a set of entities containing properties with relationships established between them. In these terms, a QEP can be modelled into LOLEPOPs (entities), type, cardinality and costs (properties) and input/output streams (relationships). This model, represented in our framework by means of Apache Jena RDF API, is applied to QEPs provided by the user and persisted in a transformation engine (Algorithm 1). Jena is a Java API which can be used to create and manipulate RDF graphs. Jena has object classes to represent graphs, resources, properties and literals. The result is a transformation of the QEP into an RDF graph, where each LOLEPOP represents an RDF Resource, each property and relationship represents an RDF Predicate and each property value is represented by an RDF Object. During the transformation from the QEP file to the RDF graph additional derived properties can be defined by analyzing resource properties. For instance, the *hasTotalCostIncrease* predicate allows us to calculate and store the total cost of the LOLEPOP by subtracting the cost of the input LOLEPOPs from currently LOLEPOP being analyzed. The

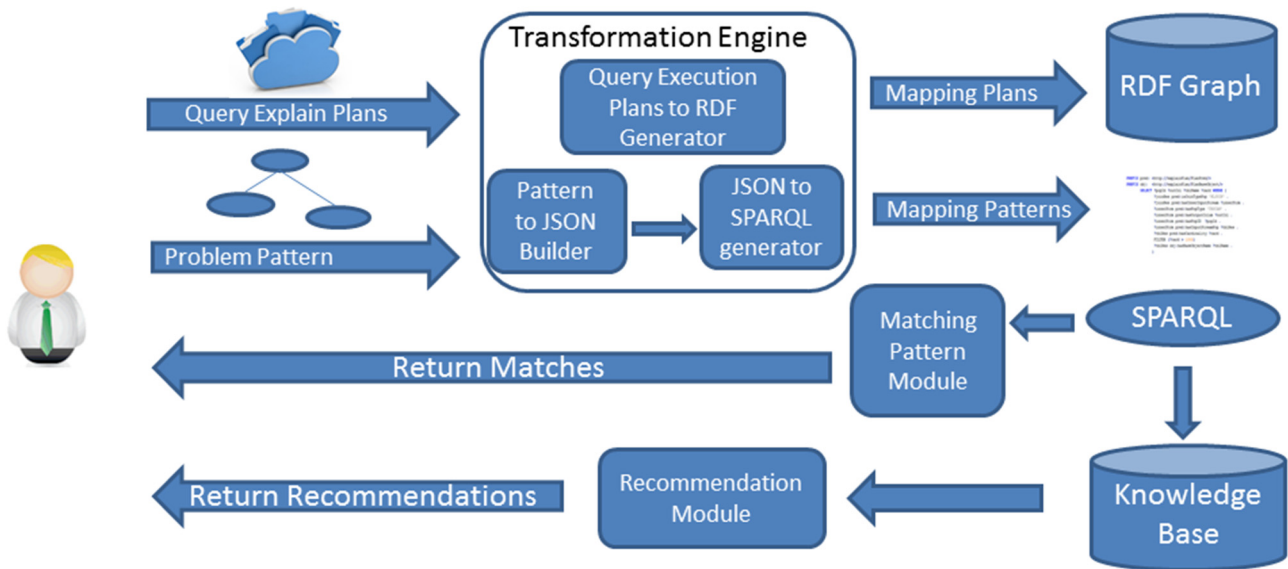


Figure 4 System Architecture

Generated RDF graph can then be preserved in memory ready for analysis by the transformation engine. The OptImatch System architecture is illustrated in Figure 4.

An example of an auto-generated RDF graph in textual representation is presented in Figure 2. Figure 2 depicts an RDF representation of the LOLEPOPs shown in Figure 1. Each RDF statement is in the form of a triplet including a resource, a predicate and an object. In the code presented, statements referring to the resource “http://explainPlan/PlanPop/5” represents LOLEPOP #5. Various predicates are shown, each encoding a piece of information from the QEP. For example, there are predicates that specify LOLEPOP #5’s total cost (15771) and estimated cardinality (4043). The RDF representation of all other LOLEPOPs is generated accordingly.

## 2.2 Searching for Problem Patterns

SPARQL is the RDF Query Language. The SPARQL standard is maintained by the W3C. Our system can accomplish the step of searching for user-defined problem patterns in QEPs by transforming patterns into SPARQL queries directed to the abstracted RDF derived from the QEPs. SPARQL performs graph traversal and pattern matching efficiently. This allows one to analyze complex patterns over large query workloads in a short period of time.

We decided to use RDF and SPARQL as SPARQL contains the capability for querying optional and required patterns of the graph with arbitrary length paths, and moreover, SPARQL property paths provide a succinct way to match patterns in the RDF graph. This includes recursive queries, such as looking for descendants operators that do not necessarily have an immediate connection with their parent (see Pattern B in Section 2.3), and searching for patterns that appear multiple times in the same query execution plan. We decided to use RDF and SPARQL as SPARQL contains the capability for querying optional and required patterns of the graph with arbitrary length paths, and moreover, SPARQL property paths provide a succinct way to match patterns in the RDF graph. While one could consider using any property graph representation framework, RDF was used also for convenience since DB2

supports RDF file format and SPARQL querying across all editions from DB2 10.1, when the RDF specific layer, DB2 RDF Store, was added. The DB2 RDF Store is optimized for graph pattern matching.

### Algorithm 2 TransformingProblemPattern

**Input:** problem pattern *probPat*

**Output:** problem pattern *probPat* transformed to SPARQL query

- 1: *probPatJSON[]* := translate problem pattern *probPat* into JSON Object (an array)
- 2: *sparql* := initialize prefixes
- 3: **forall** *probPat* in *probPatJSON[]*
- 4: *sparql* += transform an element *probPat* from JSON object *probPatJSON* with handlers into the SPARQL query
- 5: **return** *sparql*

Query performance problems can usually be described as problem patterns in the QEP. A problem pattern is a set of optimizer plan features and characteristics specified in a particular order and containing properties with predefined values. Figure 3 displays a web-based graphical user interface (pattern builder) used in our system wherein a user can express the problem pattern by selecting various properties of LOLEPOPs and plan properties that a user might be interested in within the QEP. In the depicted example of problem pattern (**Pattern A**), the user is concerned with a LOLEPOP that: (i) is of type NLJOIN; (ii) has an outer input stream of type any (ANY) with cardinality greater than one (meaning that the outer is likely to be more than one row and consequently the inner will be accessed multiple times); (iii) has an inner input stream of type TBSCAN; and (iv) the inner input stream has large cardinality (greater than 100). The depicted graphical user interface generates an example structure of a LOLEPOP that matches the selected properties. In this case, the described LOLEPOP is a nested loop join operator (NLJOIN) with some operator (ANY) on the outer input stream and a table scan (TBSCAN) on the inner input stream. Such a pattern is costly as deduced by satisfying the cardinality conditions. The NLJOIN operator scans the entire table (TBSCAN) for each of the rows from the outer operator ANY. It would likely be of value for a subject

matter expert to spend time and attention to try to optimize queries matching this problem pattern in the QEP. (System recommendations are described in details in Section 2.3.)

When specifying a problem pattern using the graphical user interface (GUI) for generality and flexibility sake, the user can choose between two types of relationships: immediate and descendant. Descendants are operators that are successors but not necessarily immediately below the current LOLEPOP. In that case, the path between the parent and the descendant child is the portion of the graph that in the general case can contain any arbitrary number of operators. For instance, in Figure 1, LOLEPOP #4 is an immediate child of LOLEPOP #3 and LOLEPOP #4 is a descendant child of LOLEPOP #2.

Once the desired problem pattern is defined by the user by describing LOLEPOPs, their characteristics and relationships, it is then automatically translated (Algorithm 2) into a JavaScript Object Notation (JSON). This object is constructed to contain a transformation of the properties specified in the pattern builder to the RDF resources and the predicates defined in the model used in the QEP. In Figure 5, we present an example JSON Object that contains properties specified in the pattern builder (Figure 3). The generated JSON Object is an array of objects describing each resource operator and its relationships. For instance, the portion of JSON Object describing LOLEPOP with ID 1 has specified type NLJOIN, an estimated cardinality value of more than 100 and relationship with two immediate children operators, LOLEPOP with ID 2 and LOLEPOP with ID 3.

```
{
  "pops": [
    {
      "ID": 1, "type": "NLJOIN", "popProperties": [
        {
          "id": "hasOuterInputStream", "value": 2, "sign": "Immediate Child",
          "id": "hasInnerInputStream", "value": 3, "sign": "Immediate Child"
        }
      ]
    },
    {
      "ID": 2, "type": "ANY", "popProperties": [
        {
          "id": "hasOutputStream", "value": 1
        }
      ]
    },
    {
      "ID": 3, "type": "TBSCAN", "popProperties": [
        {
          "id": "hasEstimateCardinality", "value": "100", "sign": ">"
        },
        {
          "id": "hasInputStream", "value": 4, "sign": "Immediate Child",
          "id": "hasOutputStream", "value": 1
        }
      ]
    },
    {
      "ID": 4, "type": "BASE OB", "popProperties": [
        {
          "id": "hasOutputStream", "value": 3
        }
      ],
      "planDetails": []
    }
  ]
}
```

**Figure 5 JSON Object**

The transformation engine uses JSON Objects to auto-generate an executable SPARQL query. An example of the autogenerated SPARQL query is presented in Figure 6. The URIs broadly match the RDF graph generated based on the QEP in Figure 1, and various SPARQL query operators and operands match the elements of the problem pattern indicated by the user.

An autogenerated SPARQL query is composed of two main parts, the SELECT clause that defines variables that appear in the query results, and the WHERE clause that defines resource properties that should be matched against the specified RDF graph. The variables that appear in query results are specified by prefixing variable name with “?” symbol, i.e., “?variable\_name” (and can be referenced multiple times in the WHERE clause). The same convention is used to define variables to establish relationship between resources and the ones used to filter retrieved resources.

Our framework allows us to autogenerate SPARQL queries with a wide range of characteristics, including nesting, filtering, multiple resource mapping, and specifying property paths as well as blank nodes. Blank nodes in RDF indicate the existence of unnamed or previously undefined resources. We introduce the concept of *handlers* to facilitate this. Handlers provide the functionality of automatically generated variable names used for the retrieval of query results, filtering of retrieved values, and establishing relationships between resources and blank nodes.

Handler generation is performed in a modular manner, by building the SPARQL query one layer (one operator) at the time over portions of JSON Object. In order to generate the SPARQL query, we define four types of handler variables: *result handlers*, *internal handlers*, *relationship handlers* and *blank node handlers*. Result handlers are created based on identifiers (sequential identifiers assigned to each LOLEPOP as shown in the graphical user interface in Figure 3), i.e., *?pop1* and *?pop2* etc. For instance, in our SPARQL query, the result handler *?pop1* is a resource returned to the user, and is also used in the WHERE clause to identify this resource as NLJOIN by adding the predicate *hasPopType*.

```
PREFIX popURI: <http://explainPlan/PlanPop/>
SELECT (?pop1 AS ?TOP) (?pop2 AS ?ANY2)
      (?pop4 AS ?BASE4)
WHERE {
  ?pop1 predURI:hasPopType "NLJOIN" .
  ?pop1 predURI:hasOuterInputStream
    ?BNodeOfpop2_to_pop1 .
  ?BNodeOfpop2_to_pop1 predURI:hasOuterInputStream
    ?pop2 .
  ?pop2 predURI:hasOutputStream ?BNodeOfpop2_to_pop1 .
  ?BNodeOfpop2_to_pop1 predURI:hasOutputStream ?pop1 .
  ?pop1 predURI:hasInnerInputStream
    ?BNodeOfpop3_to_pop1 .
  ?BNodeOfpop3_to_pop1 predURI:hasInnerInputStream
    ?pop3 .
  ?pop3 predURI:hasOutputStream ?BNodeOfpop3_to_pop1 .
  ?BNodeOfpop3_to_pop1 predURI:hasOutputStream ?pop1 .
  ?pop3 predURI:hasPopType "TBSCAN" .
  ?pop3 predURI:hasEstimateCardinality ?internalHandler1 .
  FILTER ( ?internalHandler1 > 100) .
  ?pop3 predURI:hasInputStream ?BNodeOfpop4_to_pop3 .
  ?BNodeOfpop4_to_pop3 predURI:hasInputStream ?pop4 .
  ?pop4 predURI:hasOutputStream ?BNodeOfpop4_to_pop3 .
  ?BNodeOfpop4_to_pop3 predURI:hasOutputStream ?pop3 .
  ?pop4 predURI:isABaseObj ?internalHandler2 .
} ORDER BY ?pop1
```

**Figure 6 Autogenerated SPARQL Executable Query**

Internal handlers are used to filter results. Identifiers of internal handlers are not tied to a specific resource. Their identifiers are automatically incremented on the server. For instance, the handler *?internalHandler1* is generated to provide the filtering of cardinality property by first associating it with *?pop1* (*?pop1 predURI:hasEstimatedCardinality ?internalHandler1*) and then

utilizing it in the FILTER clause (FILTER (?internalHandler1 > 100)).

Relationship handlers establish connection between resources based on information about the hierarchy of operators retrieved from the JSON Object (e.g., {"id": "hasOuterInputStream", "value": 2, "sign": "Immediate Child"}). The relationship handlers are used in conjunction with blank node handlers to resolve ambiguity problems. Ambiguity problems are encountered when the same LOLEPOP is absorbed in the different parts of the QEP. Such a LOLEPOP, for example, a common sub expression with a temporary table (TEMP) that has multiple consumers, has the same cardinality in all the consumers which may produce different results. This might be the case, for example, when a common sub expression TEMP is consumed by both a NLJOIN and a HSJOIN in the different parts of the QEP applying different predicates. In such a case, the output columns of NLJOIN and HSJOIN might differ even though the input common sub expression TEMP into each of them is the same. In the above example, ?pop1 resource has the predicate *hasOuterInputStream* connecting it to ?pop2 via the blank node ?BnodeOfPop2\_to\_pop1 (?pop1 predURI: hasOuterInputStream ?BnodeOfpop2\_to\_pop1). This design ensures the uniqueness of each resource instance in the received QEP.

The autogenerated SPARQL query through handlers is matched against the abstracted RDF structure containing information about the QEP. It maps any matched portions of the abstracted RDF structure back to the corresponding diagnostic data (Algorithm 3). Figure 1 represents an example of the DBMS QEP that contains problem pattern specified in Figure 3.

---

#### Algorithm 3 FindingMatches

---

**Input:** problem pattern *probPat*,  
query execution plan files *QEPFs*[ ]

**Output:** matches found in query execution plans

- 1: *RDFGs*[ ] := TransformingQEPs(*QEPFs*[ ])
- 2: *sparql* := TransformingProblemPattern(*probPat*)
- 3: **forall** *rdfg* in *RDFGs*[ ]
- 4:   *matchProbPat*[ ] := match abstracted problem pattern *sparql* against query execution plan *rdfg*
- 5:   **if** (*matchProbPat* != empty)
- 6:     *matchProbPatDet*[ ] := detransformation by relating any matched portions of RDF structure *matchProbPat* back to corresponding query plan
- 7:     *MATCHES*[ ].append(*matchProbPatDet*[ ])
- 8:   **endif**
- 9: **end forall**
- 10: **return** *MATCHES*[ ]

---

Matching problem patterns against diagnostic data allows for dynamic analysis of ad-hoc patterns. However, beyond single pattern matching, the tool usage can vary from problem identification and analysis to solution recommendations as described in the following section.

### 2.3 Finding Solutions in Knowledge Base

The OptImatch system has the ability to access the knowledge base to provide solutions to the known problems (Algorithm 4 and Algorithm 5). The knowledge base is populated with predetermined problem patterns and associated query plan recommendations by subject matter experts (e.g. IBM employees or expert database administrators). The OptImatch system promotes and supports collaboration among developers, experts and database administrators to create library of patterns and recommendations.

Once defined, the problem pattern is preserved in the knowledge base in two forms: an executable SPARQL query that is applied to the QEP provided by the user and as an RDF structure describing this pattern. Although the knowledge base problem patterns and solution recommendations are not in the context of the user supplied QEPs, the context for problem patterns is adapted automatically through the handlers tagging with the defined language.

Once a problem pattern to be stored in the knowledge base is described by an expert, it is translated into the SPARQL query that includes result handlers (Section 2.2). The result handlers can have *aliases* associated with them. Looking at the example SPARQL snippet we can see that the result handler ?pop1 has been assigned an alias ?TOP and ?pop4 an alias ?BASE4. These aliases are used to tag the recommendation to the specified result handlers. Tagging allows for identifying a specific result handler or a set of result handlers to be returned. This allows OptImatch to list table names, column names and predicates etc., in the context of the QEP provided by the user even though these are not available when the recommendations were created.

---

#### Algorithm 4 SavingRecommendationsKB

---

**Input:** problem patten *probPat*  
suggested recommendations *recomms*[ ]  
current knowledge base *KB*[ ]

**Output:** updated knowledge base *KB*[ ]

- 1: *sparql* := TransformingProblemPattern(*probPat*)
- 2: save abstracted problem *sparql*, problem pattern represented as RDF and corresponding recommendations *recomms*[ ] in knowledge base *KB*[ ] with handlers tagging interface
- 3: **return** *KB*[ ]

---

Our language allows for surrounding static parts of recommendations with dynamic components generated through aliases by preceding each alias of the handler with an "@" sign. This approach is also used to limit the number of resource handlers returned to the user since in complex queries there can be large number or result handlers generated, however, only some of them might be significant to the recommendation.

---

#### Algorithm 5 FindingRecommendationsKB

---

**Input:** query execution plan files *QEPFs*[ ]  
knowledge base *KB*[ ]

**Output:** solution recommendations for queries that match *QEPFs*[ ]

- 1: **forall** *qepf* in *QEPFs*[ ]
- 2:   *queryReccomendation*[ ] := match specified qepf against knowledge base *KB*[ ] using statistical analysis and provide recommendations to diagnostic data through tags of handlers
- 3:   **if** (*queryReccomendation* != empty)
- 4:     *queryRecommendations*[ ].append(*queryReccomendation*)
- 5:   **else**
- 6:     *queryRecommendations*[ ].append("There is currently no recommendation in knowledge base")
- 7:   **endif**
- 8: **end forall**
- 9: **return** *queryRecommendations*[ ]

---

A user may include multiple result handlers and apply the same rules to each of them by using array brackets e.g., [@TOP, @ANY2]. For common patterns (appearing multiple times in the same QEP) a user may limit the number of occurrences of the

```

0.157686
NLJOIN
( 5)
644901
751020
/-----+-----\
8.78417e+06      1.79511e-08
>HSJOIN          TBSCAN
( 6)            ( 13)
633711          2267.08
750436          583.334
/---+---\
78417e+06  5.99144e+06  0.174681
^HSJOIN    TBSCAN      TEMP
( 7)      ( 12)      ( 14)
561520    68023.4     2267.07
664808    85628       583.334
          |            |
          5.99144e+06  0.174681
TELEPHONE_DETAIL >NLJOIN
Q1              ( 15)
                2267.07
                583.334

```

**Figure 7 Query with Left Outer Join**

pattern that is returned in recommendation results. In the following example, [ @TOP, @ANY2]:1, only the first occurrence of @TOP and @ANY2 is returned and the specifics of the LOLEPOP types and names are obtained from the context of each occurrence.

Furthermore, a user can make use of various *helper functions* constructed to allow for interactions with base tables, indexes and materialized query tables (MQTs). These functions provide means to list column predicates and table names specific to each occurrence of the pattern in the context of the user provided query execution plan. For instance, a following expression @TOP.listColumns("PREDICATE") lists columns from an alias handler in the predicate indicated by the keyword PREDICATE.

An expert can also use ?TOP alias tagging handler to indicate that when such pattern is encountered all input columns (using keyword INPUT) coming from ?BASE4 object into the NLJOIN should be listed and are valid candidates for the index creation. This can be accomplished by tagging recommendation with following expression:

```

"Create index on table @BASE4 on columns
@TOP.listColumns("INPUT")",

```

and adding it to the knowledge base with the corresponding pattern.

Our system can look through all the QEPs supplied and iterate through both the user-defined problem patterns and the library of expert provided patterns with corresponding recommendations. If there is a match between the problem pattern in the knowledge base and the QEP, one or more query plan recommendations are returned with the appropriate context.

Our system returns ranked recommendations by using statistical correlation analysis. QEPs typically have operators, estimated or actual cost, frequency or priority metrics associated with them (as described in more details in Section 2.1). These characteristics are critical to the database system in terms of performance. Based on these characteristics a prioritized list of recommendations is provided by the system. The ranked recommendations are provided

```

1.311e-08
IXSCAN
( 38)
16.9825
3
|
2.55276e+08
IDX9
TRAN_BASE
Q21

```

**Figure 8 Estimation of the execution cost**

with a confidence score. For instance, in the example described in Section 2.2 with NLJOIN, the query plan problem determination program could output the recommendation (by automatically generating context) to create an index of the CUST\_DIM table that is the source for the TBSCAN, as this could be the recommendation stored in the knowledge base created by the experts. An example of the syntax for creating index is illustrated in the previous paragraph. An alternate recommendation may be to collect column group statistics in order to get better cardinality estimates so that the optimizer may choose a hash join instead of a nested loop join. Ranking between these two recommendations can be aided with statistical correlation analysis comparing the QEP context of cardinality and cost estimates with that in the expert provided patterns.

OptImatch can provide advanced guidance with a variety of recommendations for example, changing database configuration, improving statistics quality, recommending materialized views, suggesting alternate query and schema design changes, and recommending integrity constraints that promote performance. We illustrate some examples of these below.

As an example of a problem related to query rewrite, we describe the pattern that represents the problem of poor join order. This pattern (**Pattern B**) is given by the following properties: (i) LOLEPOP of type JOIN (which means any type of JOIN method, e.g. NLJOIN, hash join (HSJOIN) and merge scan join (MSJOIN)); (ii) has a descendant (i.e., not necessarily immediately below) outer input stream of type JOIN; (iii) has a descendant inner input stream of type JOIN; (iv) the descendant outer input stream join is a Left Outer Join; (v) descendant inner input stream join is a Left Outer Join. The recommendation for this pattern is to rewrite the query from the following structure (T1 LOJ T2) ... JOIN ... (T3 LOJ T4) to ((T1 LOJ T2).... JOIN ....T3) LOJ T4 as the rewritten query is more efficient. This optimization is now automatically done in DB2 but was found to be a limitation in early versions of DB2. This illustrates the usefulness of the tool in database optimizer development as well as supporting clients that use previous version of the DB2 system. We found QEPs matching this problem pattern in the real customer workload used in experiments, since the customer uses previous version of DB2. Figure 7 represents an example of the DBMS QEP that contains specified problem pattern. (Left outer join operators are prefixed in a QEP with ">" symbol, e.g. >HSJOIN and >NLJOIN.) This pattern is an example of the recursive problem pattern, since descendant outer and inner input stream of type LOJ do not have to be necessary immediate child of JOIN. (For instance, see LOLEPOP #5 and LOLEPOP #15) in Figure 7.

An alternate recommendation for this pattern, in case T1 = T3, is to materialize the column(s) from table T4 into table T1 and change the order of the operators from (T1 LOJ T2)... JOIN... (T1 LOJ

T4) to ((T1 LOJ T2)... JOIN ....T1), eventually allowing to eliminate T4 as well as one instance of T1, because it had a unique key join to itself. This optimization is not automatically done in current version of DB2 optimizer.

The next pattern (**Pattern C**) represents the problem related to estimation of the execution cost by optimizer. This pattern is given by the following properties: (i) LOLEPOP of type index Scan (IXSCAN) or table scan (TBSCAN) (ii) has cardinality smaller than 0.001; (iii) has a generic input stream of type Base Object (BASE OB); (iv) the generic input stream has cardinality bigger than 100000. The recommendation in this case is to create column group statistics (CGS) on equality local predicate columns and CGS on equality join predicate columns of the Base Object. Figure 8 represents an example of the DBMS query explain plan that contains specified problem pattern. With column group statistics, the optimizer can determine a better QEP and improve query performance. This is a common tuning recommendation when there is statistical correlation between column values associated with multiple equality local predicates or equality join predicates.

A user can also encounter a problem related to SORT spilling (**Pattern D**) given by the following properties: (i) LOLEPOP of type SORT; (ii) has an input stream immediately below with an I/O cost less than the I/O cost of the SORT. The recommendation may be to change the database memory configuration to increase sort memory if the number of QEPs containing this pattern is large enough to benefit the performance of many queries in the workload.

With expert or user provided patterns and recommendations in the knowledge base, OptImatch can iterate over all of the predetermined problem patterns in the knowledge base. Specifically, each problem pattern may, in such cases, be understood as being received from the knowledge base. If matched, OptImatch recalls and returns the recommendations corresponding to the particular problem pattern. Such a technique enables query plan checks to be routinized – a user can, with no particular knowledge or training, run a general test of all predetermined problem patterns against a given query workload.

### 3. EXPERIMENTAL STUDY

In this section, we present an experimental evaluation of our techniques. Our evaluation focuses on three objectives.

- a) An evaluation of the effectiveness of our approach using real IBM customer query workload (1000 QEP files). (Section 3.2 and Section 3.3)
- b) Scalability and performance over different problem characteristics: sizes of the query workload (Section 3.2.1), number of LOLEPOPs (Section 3.2.2) and number of recommendations in the knowledge base (Section 3.2.3).
- c) A comparative study with manual search for patterns by experts, quantifying the benefits of our approach in terms of time and precision. (Section 3.3)

#### 3.1 Setup

Our experiments were run on an Intel® Core™ i5-4330M machine with 2.80GHz processor and 8GB of memory. For all the conducted experiments, we used three patterns created by IBM experts. Each pattern has associated recommendations for the user (stored in the knowledge base), providing a diagnosis of the artefact. The patterns used throughout the experimental study and their respective recommendations are as follows (detailed description of each pattern can be found in Section 2).

- a) **Pattern #1 – Pattern A** (Section 2.2) that represents a problem with recommendation related to indexing.

- b) **Pattern #2 – Pattern B** (Section 2.3) of a problem with a recommendation related to rewriting the query.
- c) **Pattern #3 – Pattern C** (Section 2.3) that represents problem with a recommendation related to statistics for better cardinality (and consequently cost) estimation.

We measure the performance by computing system time for running OptImatch over the IBM customer query workload. We demonstrate the benefits of our framework by quantifying the running time against the size of the query workload (number of QEPs), number of LOLEPOPs (complexity of individual QEPs) and number of recommendations in the knowledge base.

### 3.2 Performance and Scalability

#### 3.2.1 Size of Query Workload

In the first experiment, we measure the performance of our tool by dividing the IBM customer query workload into ten buckets, each containing a different number of execution files. The first bucket contains 100 QEP files, and for each following bucket another 100 unique QEP files is added up to 1000. In other words, the distribution of the QEP files over buckets is as follows: [100, 200, ..., 1000].

The purpose of this experiment is to verify how efficient our tool is to search for portions of QEP files that match the prescribed pattern against different sizes of the query workload. The test was repeated six times (for each pattern), by dividing the QEP files into buckets randomly. (The average time is reported.)

Figure 9 reveals that the time needed to compute the search increases *linearly* with the number of QEP files. The linear dependence allows our problem determination tool to scale well to large query workloads. (There is a possibility to even further reduce the time by optimizing the communication between the client and server in our system.) Furthermore, the time to perform the search even for a large query workload with 1000 QEPs (with hundreds of operators each) that involves complex SPARQL request is less than 70 seconds. Therefore, we can conclude that our tool allows for efficient search for patterns over complex diagnostic data.

Note that Pattern #2 takes more time to be searched for than the others (around two times more). This is because Pattern #2 is more complex, as it contains descendant nodes. Therefore, recursion is used to analyze all LOLEPOPs inside the query explain plans which are fairly complex involving on average 100+ operators in the experimental workload.

#### 3.2.2 Number of LOLEPOPs

In the second experiment, we measured the performance of our system over QEPs with varying number of LOLEPOPs. We divided the IBM customer query workload into eleven buckets. The first bucket contains QEPs with the number of LOLEPOPs from 0 to 50, the second one from 50 to 100, and so on, until the last bucket that contains from 500 to 550 LOLEPOPs. (The maximum number of LOLEPOPs encountered in the workload was 550.) However, buckets 7-10 with the number of LOLEPOPs from 250 to 500 turned out to be empty, because the tested query workload contains only query explain plans with number of LOLEPOPs below 250 or above 500. Hence, as a consequence we report numbers for six buckets, 1-5 and 11. In other words, the distribution of the buckets is [0-50], [50-100], [100-150], [150-200], [200-250], and [500-550]. The number of pops is tied to the size of the explain file, the larger number of pops, the larger the size of the file.

The objective to run this experiment is to verify how efficient searching for patterns is as a function of number of pops. The experiment was repeated 6 times for each pattern, and the average

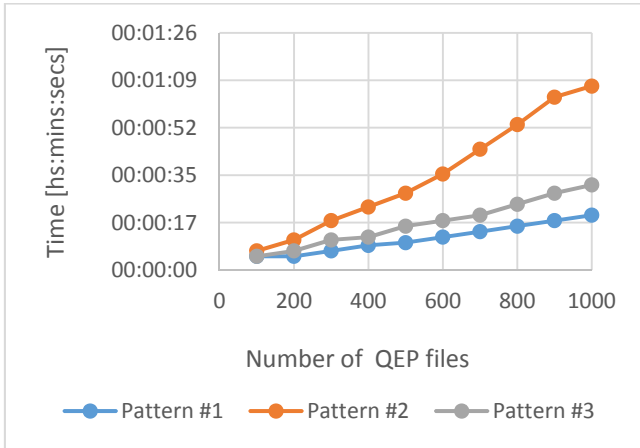


Figure 9 Search time versus number of QEP files

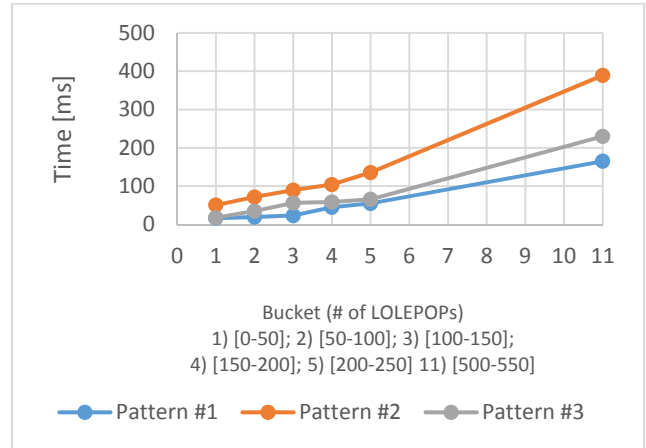


Figure 10 Search time versus number of LOLEPOPs

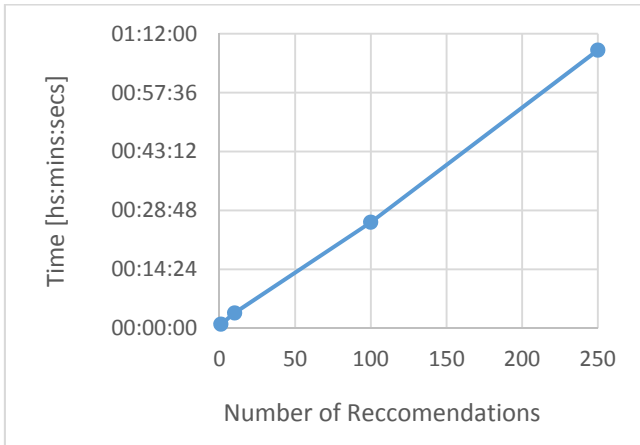


Figure 11 Matching recommendations in knowledge base

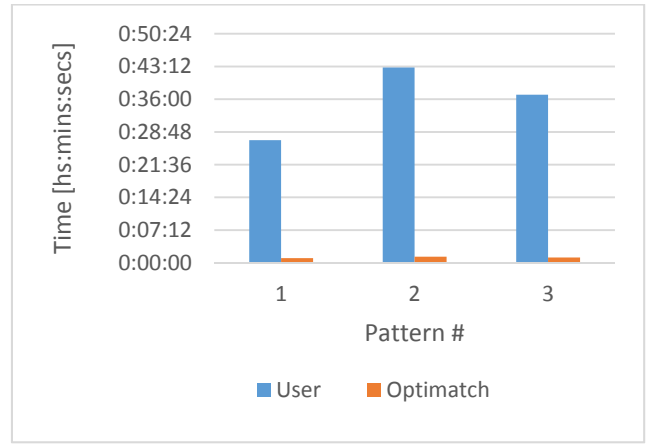


Figure 12 Comparative user study

time is reported. For each bucket, we report the average time in milliseconds, to analyze a single explain plan.

The results of this experiment are presented in Figure 10. As expected, the time spent to analyze QEPs increases as the number of LOLEPOPs increases. However, the time spent to analyze the QEPs increases in a linear fashion. Therefore, our system scales well for complex queries with a large number of LOLEPOPs. Moreover, even large and complex queries (with around 500 LOLEPOPs) can be processed efficiently by our tool (less than 400 milliseconds).

### 3.2.3 Number of Recommendations in Knowledge Base

In the next experiment, we quantify the performance of our system against the number of recommendations in the knowledge base. We measure the running time to analyze 1000 QEP files against 1, 10, 100 and 250 recommendations in the knowledge base, respectively.

We perform this experiment to simulate the important use case for our system described in Section 2.3 to routinize query plan checks with expert provided predetermined problem patterns and corresponding recommendations against a given complex query workload. Instead of taking a specific problem pattern defined by the user, the system iterates over all of the predetermined problem patterns in the query plan knowledge base and provides matching solutions to known problems.

We report the results of this experiment in Figure 11. Our framework adapts well, with linear dependence over the number of recommendations in the knowledge base. The linear dependence allows our system to scale well to large knowledge databases. Our tool can process a 1000 query workload against 250 problem patterns and recommendations in around 70 minutes.

### 3.3 Comparative User Study

In the last experiment, we measure the time to perform pattern search both manually by experts and automatically with OptImatch. We also looked at the search quality. For each of the three patterns, we provide users the workload with 100 distinct QEP files. Out of 100 QEP files, 15, 12 and 18 QEP files match the three prescribed patterns #1, #2, and #3, respectively. Three experts participated in this experiment. We report the aggregated average statistics.

The purpose of this experiment is to quantify the benefits of our automatic approach against cumbersome manual search by experts that is prone to human error. The time comparison is shown in Figure 12. It can be observed that our tool drastically reduces the time to search for a pattern against even a sample of the query workload. (We perform this experiment over a sample of the query workload due to the limited time experts could spend to participate in the experiment.) Overall, our tool is around 40 times faster than the manual search by IBM experts. To simulate real world environment during the manual search for patterns experts were



**Table 1 Precision for manual search**

Pattern #	#1	#2	#3
Precision	88%	71%	81%

allowed to access the tools that they use in their daily problem determination tasks. An example of this includes the *grep* command-line utility for searching plain-text data sets for lines matching a regular expression.

When we measure the running time for automatic search with our tool, we include the time both for specifying the pattern using graphical interface in our tool (on average around 60 seconds), as well as, performing the actual search by our system. Based, on this experiment, it can be inferred that manual search for a larger query workload (1000 queries) would take approximately 5 hours, whereas, with our tool this can be performed in around 2 minutes (around 150 times faster). Note that an automatically searched pattern has to be specified only once by the user.

Last but not least, we report the quality of the search results in our comparative study. We measure the precision as the function of missed QEP files that contain the prescribed pattern. As predicted, manual search has been prone to human error. The precision for manual search by experts is on average 80%. Details are provided in Table 1. The common errors include misinterpreting information stored in the QEP file as well as formatting errors, e.g., using *grep* on operand value 0.001 while this information is represented in the QEP in either the decimal form or with an exponent as  $10^{-3}$ . Obviously, since our tool is fully automatic and immune to such differences, it provided 100% precision. Our system does not only perform significantly faster than a manual search but it also guarantees correctness. Often, high precision may be very important in problem determination analysis, as such, this experiment emphasizes another important benefit of OptImatch.

#### 4. RELATED WORK

The SQL programming language is declarative in nature. Therefore, it is enough to specify what data we want to retrieve, without actually specifying how to get data. This is one of the main strengths of SQL, as it means that it should not make a difference to the query optimizer how a query is written as long as the different versions are semantically equivalent. However, in practice this is only partially true, as there is only a limited number of machine-generated query rewrites that a database optimizer can perform [9]. As the complexity of SQL grows, there is an increasing need to have tools help with performance problem determination.

Many different formalisms have been proposed in query optimization. We cite here only the most pertinent references. Join, sort and group by are at the heart of many database operations. The importance of these operators for query processing has been recognized very early on. Right from the beginning, the query optimizer of System R paid particular attention to interesting orders by keeping track of indexes, ordered sets and pipelining operators throughout the process of query optimization, as described in Selinger et al. [14]. Within query plans, group-by, order-by and join operators can be accomplished either by a partition operation (such as by the use of a hash index), or by the use of an ordered tuple stream, as provided by a tree-index scan or by a sort operation (if appropriate indexes are not prescribed).

In Guravannavar et al. [11], authors explored the use of sorted sets for executing nested queries. The importance of sorted sets in query optimization has prompted the researchers to look beyond the sets

that have been explicitly generated. In Szlichta et al. [17], authors show how to use relationship between sorted attributes discovered by reasoning over the physical schema via integrity constraints to avoid potentially expensive join operator. The inference system presented in follow-up work provides a formal way of reasoning about previously unknown or hidden sorted sets [18], [19]. Based on that work, many other optimization techniques from relational query processing can also be adapted to optimize group by, order by and case expressions [2], [20].

Optimization strategies described above hold the promise for good improvement. Their weakness, however, is that often the indexes, views and constraints that would be useful for optimization for a given database and workload is not explicitly available and there is only a limited number of types of query transformations that the optimizer can perform. Therefore, problem determination tools [23], [24] offer an alternative automated way to analyze QEPs and provide recommendations, such as re-write the query, create an index or materialized view or prescribe an integrity constraints. However, existing automatic tools for query performance problem determination do not provide the ability to perform workload analysis with flexible user defined patterns, as they lack the ability to impose proper structure on QEPs (as described in details in Section 1).

#### 5. CONCLUSIONS

Query performance problem determination is a complex process. It is a tedious manual task that requires one to analyze a large number of QEPs that could span thousands of lines. It also necessitates a high level of skill and in-depth optimizer knowledge from users. Identification of even known issues is a very time and resource consuming and prone to human error.

To the best of our knowledge, we are the first to provide the system that performs interactive analysis in a structured manner of potentially a large number of QEPs in order to diagnose and match optimizer problem patterns and retrieve corresponding recommendations that are provided by experts. Our semantic system combines and applies the benefits of RDF model and SPARQL query language in query performance problem determination and QEP analysis. OptImatch is very well received and is proving to be very valuable in the IBM support of clients and database optimizer development organization by providing quick work around solutions through the use of a well-defined knowledge base.

Our methodology can certainly be applied to other general software determination problems (e.g., log data relating to network usage, security, or software compiling, as well as software debug data or sensor data relating to some physical system external to the system). Our framework can be applied to other general software problem determination, assuming that there exists automatically generated structured diagnostic information in the form of the graph that needs to be further analyzed by an expert or general user. This is the direction that we would like to explore in the future work.

#### 6. ACKNOWLEDGMENTS

The authors would like to thank members of the DB2 optimizer development and support teams for their feedback and guidance through the development of the OptImatch system. Special thanks in particular to Shu Lin, Vincent Corvinelli and Manopalan Kandiah.

#### 7. TRADEMARKS

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in

many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).

## 8. REFERENCES

- [1] Barber, R., Lohman, G.M., Pandis, I., Raman, V., Sidle, R., Attaluri, G.K., Chainani, N., Lightstone, S., Sharpe, D. Memory-Efficient Hash Joins. *PVLDB*, 8(4): 353-364, 2014.
- [2] Ben-Moshe, S., Kanza, Y., Fischer, F., Matsliah, A., Fischer, M., and Staelin, C. Detecting and exploiting near-sortedness for efficient relational query evaluation. In *ICDT*, 256-267, 2011.
- [3] Bornea, M.A., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O., Bhattacharjee, B. Building an efficient RDF store over a relational database. In *SIGMOD*, 121-132, 2013.
- [4] Chaudhuri, S., Narasayya, V.R. Self-Tuning Database Systems: A Decade of Progress. In *VLDB*, 3-14, 2007.
- [5] Cheng, Q., Gryz, J., Koo, F., Leung, T., Liu, L., Qian, X., and Schiefer, K. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. In *VLDB*, 687-698, 1999.
- [6] Dageville, B., Das, D., Dias, K., Yagoub, K., Zaït, M., and Ziauddin, M. Automatic SQL Tuning in Oracle 10g. In *VLDB*, 1346- 1355, 2004.
- [7] El-Helw, A., Ilyas, I.F., Zuzarte, C. StatAdvisor: Recommending Statistical Views. *PVLDB*, 2(2), 1306-1317, 2009.
- [8] Ghanem, T.M., Elmagarmid, A.K., Larson, P., Aref, W.G. Supporting views in data stream management systems. *ACM Trans. Database Syst.*, 35(1), 2010.
- [9] Gryz, J., Wang, Q., Qian, X., Zuzarte, C. Queries with CASE expressions. *Journal of Intelligent Information Systems*, 34(3): 345-366, 2010.
- [10] Gryz, J., Wang, Q., Qian, X., Zuzarte, C. SQL Queries with CASE Expressions. In *ISMIS*, 351-360, 2008.
- [11] Guravannavar, R., Ramanujam, H., and Sudarshan, S. Optimizing Nested Queries with Parameter Sort Orders. In *VLDB*, 481-492, 2005.
- [12] Kimball, R., and Ross, M. 2002. *The Data Warehouse Toolkit Second Edition, The Complete Guide to Dimensional modeling*. John Wiley & Sun.
- [13] Larson, P., Birka, A., Hanson, E.H, Huang, W., Nowakiewicz, M., Papadimos, V. Real-Time Analytical Processing with SQL Server. *PVLDB*, 8(12), 1740-1751, 2015.
- [14] Selinger, P., and Astrahan, M. Access Path Selection in a Relational Database Management System. In *SIGMOD*, 23-34, 1979.
- [15] Silva, Y.N., Aref, W.G., Larson, P., Pearson, S., Mohamed Ali, H. Similarity queries: their conceptual evaluation, transformations, and processing. *VLDB J.*, 22(3), 395-420, 2013.
- [16] Simmen, D., Shekita, E., and Malkemus, T. Fundamental Techniques for Order Optimization. In *SIGMOD*, 57-67, 1996.
- [17] Szlichta, J., Godfrey, P., Gryz, J., Ma, W., Pawluk, P., and Zuzarte, C. Queries on dates: fast yet not blind. In *EDBT*, 497-502, 2011.
- [18] Szlichta, J., Godfrey, P., Gryz, J. Fundamentals of Order Dependencies. *PVLDB*, 5(11): 1220-1231, 2012.
- [19] Szlichta, J., Godfrey, P., Gryz, J., and Zuzarte, C. Expressiveness and Complexity of Order Dependencies. *PVLDB*, 6(14): 1858-1869, 2013.
- [20] Szlichta, J., Godfrey, P., Gryz, J., Ma, W., Qiu, W., Zuzarte, C. Business-Intelligence Queries with Order Dependencies in DB2. In *EDBT*, 750-761, 2014.
- [21] Wang, X. and Cherniack, M. Avoiding Sorting and Grouping in Processing Queries. In *VLDB*, 826-837, 2003.
- [22] Zhu, Q., Tao, Y., Zuzarte, C. Optimizing complex queries based on similarities of subqueries. *Knowl. Inf. Syst.*, 8(3): 350-373, 2005.
- [23] Zilio, D.C, Rao, J., Lightstone, S., Lohman, G.M, Storm, A.J., Garcia-Arellano, C., Fadden, S. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB* 1087-1097, 2004.
- [24] Zilio, D.C., Zuzarte, C., Lightstone, S., Ma, W., Lohman, G.M., Cochrane, R., Pirahesh, H., Colby, L.S., Gryz, J. Alton, E., Liang, D., Valentin, Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. In *ICAC*, 180-188, 2004.
- [25] Zuzarte, C., Pirahesh, H., Ma, W., Cheng, Q., Liu, L., Wong, K. WinMagic: Subquery Elimination Using Window Aggregation. In *SIGMOD*, 652-656, 2003.
- [26] Miller, B.A., Nastacio, D., Perazolo, M. Problem determination service. Patent application, US7818338 B2, <http://www.google.com/patents/US7818338>

# Scalable Public Transportation Queries on the Database

Alexandros Efentakis  
 Research Center "Athena"  
 Artemidos 6 & Epidavrou,  
 Marousi 15125, Greece  
 efentakis@imis.athena-innovation.gr

## ABSTRACT

Recent scientific literature focuses on answering Earliest Arrival (EA), Latest Departure (LD) and Shortest Duration (SD) queries in (schedule-based) public transportation networks. Unfortunately, most of the existing solutions operate in main memory, making the proposed methods hard to scale for larger instances and difficult to integrate in a multi-user environment. This work proposes PTLDB (Public Transportation Labels on the DataBase), a novel, scalable, pure-SQL framework for answering EA, LD and SD queries, implemented entirely on an open-source database system. Moreover, we formulate four new types of queries targeting public transportation networks, namely the Earliest Arrival and Latest Departure  $k$ -Nearest Neighbor ( $k$ NN) and One-to-many queries and propose novel ways to efficiently answer them within PTLDB. Our experimentation will show that the proposed solution is fast, scalable and easy to use, making our PTLDB framework a serious contender for use in real-world applications.

## Categories and Subject Descriptors

H.2.8 [Database Applications ]: Spatial databases and GIS; G.2.2 [Graph Theory]: Graph algorithms

## General Terms

Algorithms, Design

## Keywords

Transportation networks,  $k$ NN queries, One-to-many queries

## 1. INTRODUCTION

Recent algorithmic advances have been very efficient in solving vertex-to-vertex queries on graphs, for a variety of different graph types and instances. For road networks, latest papers focused on supporting additional types of shortest-path (SP) queries, including *one-to-all* (finding SP distances

from a source vertex  $s$  to all other graph vertices) [8, 15], *one-to-many* (computing the SP distances between the source vertex  $s$  and all vertices of a set of targets  $T$ ) [11, 15], *range* (find all nodes reachable from  $s$  within a given timespan) [15], *many-to-many* (calculate a distance table between two sets of vertices  $S$  and  $T$ ) [11] and  $k$ -Nearest Neighbor ( $k$ NN) queries in [12, 17, 21]. For large-scale networks, the works of [4, 9, 20], proposed methods for solving vertex-to-vertex queries, whereas the works of [16] and [14] deal with more complex queries, such as  $k$ -Nearest Neighbor, Reverse  $k$ -Nearest Neighbor ( $Rk$ NN) and one-to-many queries, either on main memory or inside a database. Finally, for schedule-based public transportation networks recent works, either expand existing shortest-path techniques originally used for road networks, or work directly on the provided timetable. Most of the recent methods targeting public transit networks are surveyed in the latest literature overview on transportation networks of [5].

Despite the inherent different characteristics between those three types of graph networks (road, large-scale and public transportation networks), the prevailing technique that excels at all of them is the 2-hop labeling or hub labeling (HL) algorithm of [18],[6], in which we store a two-part label  $L(v)$  for every vertex  $v$ : a forward label  $L_f(v)$  and a backward label  $L_b(v)$ . These labels are then used to very fast answer vertex-to-vertex shortest-path queries. This technique has been adapted successfully to (i) road networks [2, 3, 10], (ii) undirected, unweighted graphs in [4, 9, 20] and (iii) has also been extended to public transportation networks in [7] and [23]. The HL method has also been applied successfully for one-to-many, many-to-many and  $k$ NN queries in road networks [11, 12] and  $k$ NN and  $Rk$ NN queries in the context of large-scale networks in [16].

Although hub labeling is an extremely efficient shortest-path technique on main memory, there are very few works that extend those results for secondary storage. HLDB [13] stores the precomputed labels for road networks in a commercial database system and translated the hub labeling, vertex-to-vertex distance query to plain SQL commands. Moreover, it showed how to answer  $k$ NN queries and  $k$ -best via points, again with simple SQL queries. The work of [20] proposed HopDB, a C++ customized solution that utilizes secondary storage during preprocessing for large-scale networks. Efentakis et al. proposed the COLD framework [14] that stores hub-labels for large-scale networks in an open-source database engine and answers vertex-to-vertex, one-to-many,  $k$ NN and  $Rk$ NN queries, using SQL commands.

In this work, we focus on timetable, public transporta-

tion networks and present a novel database framework that may service several route-planning queries on such networks. Our pure-SQL, *PTLDB* (Public Transportation Labels on the DataBase) framework, extends the hub labeling technique for public transportation networks, as presented in Timetable Labeling (TTL) [23] and may answer multiple variations (Earliest Arrival, Latest Departure and Shortest-Duration) of vertex-to-vertex queries, entirely within a database. Moreover, we formulate four new types of route-planning queries for public transit, namely the *Earliest Arrival* and *Latest Departure k*-Nearest Neighbor (*k*NN) queries and their *one-to-many* versions, i.e., namely the *Earliest Arrival* and *Latest Departure one-to-many* queries. These newly defined public-transit queries may be very useful for a variety of applications utilizing public transport, such as mobile travel guides, transportation and urban planning or geomarketing applications. Our experimentation will show that the proposed *PTLDB* framework may answer all those different types of queries efficiently, while its implementation with an open-source database engine makes *PTLDB* very easy to integrate into existing multi-user, real-world applications. Additionally, each type of query may be answered with a simple SQL command, making our results easily reproducible by anyone. Thus, the *PTLDB framework is the only database solution that focuses on public-transportation networks*, while ensuring excellent performance that is fast-enough for real-time applications.

The outline of the remainder of this work is as follows. Section 2 presents related work. Section 3 describes the novel *PTLDB* framework and its implementation details. Experiments establishing the benefits of *PTLDB* are provided in Section 4. Finally, Section 5 gives conclusions and directions for future work.

## 2. BACKGROUND AND RELATED WORK

In this section we will present related work, relative to the hub labeling method for directed weighted graphs  $G(V, E, w)$ , where  $V$  is the set of vertices,  $E \subseteq V \times V$  is the set of arcs and  $w$  is a positive weight function  $E \rightarrow R^+$ . We will also discuss the latest adaptations of this specific technique for public transportation networks.

### 2.1 Hub Labeling

In the 2-hop labeling or Hub Labeling (HL) algorithm of [18, 6], preprocessing stores at every vertex  $v$  a forward  $L_f(v)$  and a backward label  $L_b(v)$ . The forward label  $L_f(v)$  is a vector of pairs  $(u, dist(v, u))$ , with  $u \in V$ . Likewise, the backward label  $L_b(v)$  contains pairs  $(w, dist(w, v))$ . Vertices  $u$  and  $w$  are denoted as the *hubs* of  $v$ . The generated labels conform to the *cover property*, i.e., for any  $s$  and  $g$ , the set  $L_f(s) \cap L_b(g)$  must contain at least one hub that is on the shortest  $s - g$  path. To find the network distance  $dist(s, g)$  between any two vertices  $s$  and  $g$ , a HL query seeks the hub  $v \in L_f(s) \cap L_b(g)$  that minimizes the sum  $dist(s, v) + dist(v, g)$ . By sorting the pairs in each label by hub, this takes linear time by employing a coordinated sweep over both labels. The HL technique has been successfully adapted for road networks in [2, 3, 10]. In the case of large-scale graphs, the Pruned Landmark Labeling (PLL) algorithm of [4] orders vertices by degree and then during preprocessing, performs one BFS per graph vertex, starting from the highest-order / degree vertices. At each iteration, each individual BFS is pruned by using the hub labels cal-

culated from the previous searches. Later, Delling et al. [9] improve the suggested vertex ordering and the storage of the hub labels for maximum compression. The HL method has also been extended to one-to-many, many-to-many and *k*NN queries on road networks in [11] and [12] respectively. The recent work of [16] has also proposed *ReHub*, a novel main-memory algorithm that extends the Hub Labeling approach to efficiently handle Reverse *k*-Nearest Neighbor (R*k*NN) queries on large-scale networks.

Regarding secondary-storage solutions, Jiang et al. [20] propose their HopDB algorithm for constructing an efficient HL index when the given graphs and the corresponding index are too large to fit into main memory. Abraham et al. [1] introduced the HLDB system, which answers distance and *k*NN queries in road networks entirely within a database by storing the hub labels in database tables and translating the corresponding HL queries to SQL commands. In [14], Efentakis et al. presented the pure-SQL *COLD* framework (C*OMPRESSED* Labels on the Database) for answering multiple exact distance queries (vertex-to-vertex, *k*NN, R*k*NN and *one-to-many*) for large-scale networks, in a open-source database engine. Despite the fact that each type of query was translated to just few lines of SQL code, experiments have showed that *COLD* provides excellent performance and is thus, fast enough for real-time applications.

### 2.2 Public transportation networks

For methods targeting public transportation networks before 2015, one can refer to the latest related literature overview of [5]. In this section, we will mainly focus on the latest research works that appeared in 2015.

Recently, the hub labeling method has also been extended for public transportation networks, in Public Transit Labeling presented in [7] and Timetable Labeling (TTL) presented in [23]. Since our work builds on Timetable Labeling (TTL), we will follow the notation presented there. A schedule-based, public transportation network may be modeled as a multigraph  $G$ , where each vertex is associated with a station or stop (i.e., “distinct locations where one may board a transit vehicle” [7]) and each arc from a vertex  $u$  to a vertex  $v$  represents a trip  $b$  that departs from stop  $u$  at timestamp  $t_d$  and arrives at  $v$  at timestamp  $t_a$ . Thus, each arc  $e$  may be represented with a tuple  $\langle u, v, t_d, t_a, b \rangle$ , where  $b, t_d, t_a$  are the trip, departure time and arrival time of  $e$  respectively and  $t_a - t_d$  is the corresponding duration of the arc. The arc  $e$  is an outgoing arc for vertex-stop  $u$  and an incoming arc for vertex-stop  $v$ . Thus, there are multiple arcs connecting the same pair of vertices-stops that correspond to the different trips connecting those two stops together.

Timetable labelling (TTL) is a extension of Hierarchical Hub Labeling [3] for public transportation networks. The TTL index preprocessing, computes two sets of labels,  $L_{in}(v)$  and  $L_{out}(v)$ , for each vertex-stop  $v$ , such that each label in  $L_{in}(v)$  (or  $L_{out}(v)$ ) is a tuple describing a “fast” path that ends at (starts from)  $v$ . TTL assumes there is a strict vertex ordering, which, defines the relative importance of each vertex with respect to the others. Hence, the rank  $r(v)$  of a vertex  $v$  is an integer  $\in [1, |V|]$ . It is assumed that the vertex  $v$  ranks lower (i.e., is less important) than  $u$ , when  $r(v) > r(u)$ . When given a timetable graph  $G$  and a node order  $r$ , the TTL index can be uniquely constructed. The output of TTL preprocessing is two sets of labels,  $L_{in}(v)$  and  $L_{out}(v)$  for each vertex-stop  $v$ , whereas

each label contains tuples  $\langle hub, t_d, t_a, pivot, b \rangle$  representing a fast transit path between stops  $v$  and  $hub$ , passing through stop  $pivot$ , using trip  $b$  and departing at  $t_d$  and arriving at  $t_a$ . The pivot information is required for reconstructing the full path, when it is comprised of multiple trips and is set to *null*, when the corresponding tuple describes a direct trip between stops  $v$  and  $hub$  (i.e.,  $b \neq null$ ). Note, that the label sets  $L_{in}(v)$  and  $L_{out}(v)$  for a vertex  $v$  only contain paths between  $v$  and vertices of higher order.

The labels calculated during the TTL preprocessing may be used for answering the following three types of queries:

- *Earliest Arrival* (EA). Given two stops  $s$  and  $g \in G$  and a starting timestamp  $t$ , the earliest arrival  $EA(s, g, t)$  query seeks the path with the earliest arrival time among those paths that (i) start from  $s$  no sooner than  $t$  and (ii) end at  $g$ .
- *Latest Departure* (LD). Given two stops  $s$  and  $g$  and an ending timestamp  $t'$ , the latest departure  $LD(s, g, t')$  query seeks the path with the latest departure time among those paths that (i) start from  $u$  and (ii) end at  $v$  no later than  $t'$ .
- *Shortest Duration* (SD). Given two stops  $s$  and  $g$ , a starting  $t$  and an ending  $t'$  timestamp, the shortest duration  $SD(s, g, t, t')$  query seeks the path with the shortest duration among those that (i) start from  $s$  no sooner than  $t$  and (ii) end at  $g$  no later than  $t'$ .

During the query phase, TTL only has to visit the labels of stops  $s$  and  $g$  (without accessing the original graph  $G$ ) and select the best solution from three candidate cases: (i) Tuples in  $L_{out}(s)$  where  $hub = g$ , (ii) Tuples in  $L_{in}(g)$  where  $hub = s$  and (iii) Combining all tuples  $l_1 \in L_{out}(s)$  and  $l_2 \in L_{in}(g)$  where  $l_1.hub = l_2.hub$  and  $l_1.ta \leq l_2.td$ . The experimentation in [23], showed that TTL may answer EA, LD and SD queries in less than  $30\mu s$ , while requiring preprocessing time of less than  $17min$  for all datasets.

In this work, based on the lessons learnt from the previous COLD database framework [14], that answers multiple distance queries on large-scale graphs, we will: (i) show how to efficiently store the labels created from Timetable labelling for public transportation networks into a database, (ii) translate the corresponding EA, LD and SD queries into simple SQL commands (iii) formulate four new types of queries for public transportation networks, namely the *Earliest Arrival* and *Latest Departure k*-Nearest Neighbor ( $kNN$ ) and the *Earliest Arrival* and *Latest Departure one-to-many* queries and (iv) show how to efficiently answer those additional queries, by using simple SQL commands within the same database framework, implemented entirely on a popular, open-source database engine. The resulting *PTLDB* (Public Transportation Labels on the DataBase) framework will be described in the following section.

### 3. THE PTLDB FRAMEWORK

This section presents the *PTLDB* (Public Transportation Labels on the DataBase) database framework. *PTLDB* can answer multiple route-planning queries (Earliest Arrival [EA], Latest Departure [LD], Shortest Duration [SD] vertex-to-vertex, EA, LD  $k$ -Nearest Neighbor and EA, LD *one-to-many*) for public transportation networks using SQL commands. Since *PTLDB* builds on the Timetable Labeling

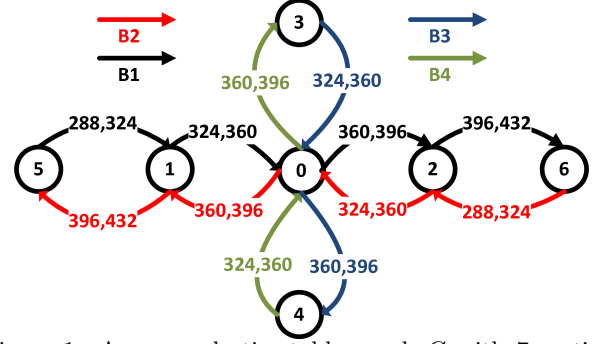


Figure 1: An example timetable graph  $G$  with 7 vertices (stops) and 4 trips (each highlighted with a different color). Timestamps are in 100s, i.e., 360=>36,000s (10:00h). Vertex 0 is the highest order vertex, followed by vertices 1,2,3,4

Table 1: The created labels for the example graph  $G$

$v$	$L_{out}(v)$	$L_{in}(v)$
0	$\langle 0, 360, 360, -1, -1 \rangle$	$\langle 0, 360, 360, -1, -1 \rangle$
1	$\langle 0, 324, 360, 0, 1 \rangle$ $\langle 1, 324, 324, -1, -1 \rangle$ $\langle 1, 396, 396, -1, -1 \rangle$	$\langle 0, 360, 396, 0, 2 \rangle$ $\langle 1, 324, 324, -1, -1 \rangle$ $\langle 1, 396, 396, -1, -1 \rangle$
2	$\langle 0, 324, 360, 0, 2 \rangle$ $\langle 2, 324, 324, -1, -1 \rangle$ $\langle 2, 396, 396, -1, -1 \rangle$	$\langle 0, 360, 396, 0, 1 \rangle$ $\langle 2, 324, 324, -1, -1 \rangle$ $\langle 2, 396, 396, -1, -1 \rangle$
3	$\langle 0, 324, 360, 0, 3 \rangle$ $\langle 3, 396, 396, -1, -1 \rangle$	$\langle 0, 360, 396, 0, 4 \rangle$ $\langle 3, 396, 396, -1, -1 \rangle$
4	$\langle 0, 324, 360, 0, 4 \rangle$ $\langle 4, 396, 396, -1, -1 \rangle$	$\langle 0, 360, 396, 0, 4 \rangle$ $\langle 4, 396, 396, -1, -1 \rangle$
5	$\langle 0, 288, 360, 1, 1 \rangle$ $\langle 1, 288, 324, 1, 1 \rangle$ $\langle 5, 432, 432, -1, -1 \rangle$	$\langle 0, 360, 432, 1, 2 \rangle$ $\langle 1, 396, 432, 1, 2 \rangle$ $\langle 5, 432, 432, -1, -1 \rangle$
6	$\langle 0, 288, 360, 2, 2 \rangle$ $\langle 2, 288, 324, 2, 2 \rangle$ $\langle 6, 432, 432, -1, -1 \rangle$	$\langle 0, 360, 432, 2, 1 \rangle$ $\langle 2, 396, 432, 2, 1 \rangle$ $\langle 6, 432, 432, -1, -1 \rangle$

(TTL) [23], we will explain the basic concepts presented there. We chose PostgreSQL [22] for our implementation, given that it is a popular, open-source RDBMS. Although we use some PostgreSQL-specific data-types (namely arrays) and SQL extensions, we use only features included in its standard installation, without any third-party extensions.

### 3.1 Vertex-to-Vertex (v2v) queries

The *PTLDB* framework uses the labels generated by the Timetable Labeling (TTL) of [23] for public transportation networks. The respective TTL implementation (and the respective datasets) were made publicly available by the authors at [24]. To highlight the results of this process, the labels for the example timetable graph  $G$  of Figure 1 are shown in Table 1. The labels  $L_{out}(v)$  and  $L_{in}(v)$  for each vertex / stop  $v$  is a vector of tuples  $\langle hub, t_d, t_a, pivot, b \rangle$  sorted by  $hub, t_d$  (see Section 2). To answer vertex-to-vertex (v2v) queries between two stop  $s$  and  $g$ , TTL only has to visit the labels  $L_{out}(s)$  and  $L_{in}(g)$  and select the best solution from three candidate cases: (i) Tuples in  $L_{out}(s)$  where  $hub = g$ , (ii) Tuples in  $L_{in}(g)$  where  $hub = s$  and (iii) Combining all tuples  $l_1 \in L_{out}(s)$  and  $l_2 \in L_{in}(g)$  where  $l_1.hub = l_2.hub$  and  $l_1.ta \leq l_2.td$ . Although selecting between those three cases is trivial for a main memory algorithm, it is complex to adapt in a database context. Thus, we need to generate for every  $v \in G$ , some extra “dummy” tuples in both

$L_{out}(v)$  and  $L_{in}(v)$  with  $hub = v$  and  $t_d = t_a$  for every DISTINCT  $(hub, t_d)$  combination existing in  $L_{out}(u)$  and for every DISTINCT  $(hub, t_a)$  combination existing in  $L_{in}(u)$ . Those dummy tuples for our example graph are highlighted in bold. Note that, those dummy tuples are only a small fraction ( $< 10\%$ ) of the total number of tuples and hence, they add minimal overhead to the PTLDB framework’s performance. By generating those dummy tuples, we can ensure that each vertex-to-vertex query may be answered by combining exactly one tuple  $l_1 \in L_{out}(s)$  and one tuple  $l_2 \in L_{in}(g)$  where  $l_1.hub = l_2.hub$  and  $l_1.ta \leq l_2.td$ , i.e., we unified the three separate cases of TTL query processing into one. Thus, the answer to the  $EA(1, 1, 324)$  query is 324 by combining the tuples  $\langle 1, 324, 324, \dots \rangle$ , present in both  $L_{out}(1)$  and  $L_{in}(1)$ .

Code 1: Vertex-to-vertex (v2v) queries for PTLDB

```

1 WITH outp AS
2   (SELECT UNNEST(hubs) AS hub,
3         UNNEST(tds) AS td,
4         UNNEST(tas) AS ta
5   FROM lout WHERE v=s),
6 inp AS
7   (SELECT UNNEST(hubs) AS hub,
8         UNNEST(tds) AS td,
9         UNNEST(tas) AS ta
10  FROM lin WHERE v=g)
11 /* For EA queries */
12 SELECT MIN(inp.ta)
13 /* For LD queries */
14 SELECT MAX(out.td)
15 /* For SD queries */
16 SELECT MIN(inp.ta-outp.td)
17 FROM outp,
18      inp
19 WHERE outp.hub=inp.hub AND outp.ta<=inp.td
20 /* For EA,SD queries */
21 AND outp.td>=t
22 /* For LD,SD queries */
23 AND inp.ta<=t'

```

After generating the additional dummy tuples for simplifying the TTL vertex-to-vertex queries, we need to store the respective  $L_{out}(v)$  and  $L_{in}(v)$  labels in the database, as two separate DB tables denoted *lout* and *lin*, respectively. Similar to the previous work COLD [14], we take advantage of the fact that PostgreSQL features an array data type that allows columns of a DB table to be defined as variable-length arrays. Hence, in PTLDB we store hubs, departure timestamps  $t_d$  and arrival timestamps  $t_a$  for a vertex (all ordered by  $hub, t_d$ ) as arrays in three separate columns (i.e., hubs, tds and tas) in a single row. The resulting *lout* and *lin* DB tables are shown in Tables 2 and 3. Similar to COLD, this approach has considerable advantages: (i) The *lout* and *lin* DB tables have exactly  $|V|$  rows (ii) Each of those DB tables has the column  $v$  as primary key, minimizing the size of the respective index. (iii) For any v2v query, PTLDB needs to access exactly two rows, regardless of the sizes of  $|L_{out}(s)|$  and  $|L_{in}(g)|$ , thus minimizing the secondary-storage utilization, even working inside a database. In case of timetables changing depending on the weekday (e.g., weekdays vs weekends) or the time of the year (e.g., on holidays) in PTLDB we would need to have different versions of the *lout* and *lin* DB tables, for servicing each different period. Also note

that in PTLDB, we do not need to store the pivot or the trip information, since if we wanted to reconstruct the full path, it would make more sense to store on the database the expanded path for each tuple generated by the TTL preprocessing. After all, this is another advantage of databases, also suggested by previous efforts like [1].

The resulting SQL commands for all types (Earliest Arrival, Latest Departure and Shortest Duration) of vertex-to-vertex queries for PTLDB are shown in Code 1, where the user may choose between the lines 12, 14 and 16 and lines 21, 23 depending on the specific type of query. We use Common Table Expressions (CTEs) for greater readability and we exploit the fact that PostgreSQL “guarantees that parallel unnesting” for *hubs*, *tds* and *tas* for each nested query “will be in sync”, i.e., each tuple  $\langle hub, td, ta \rangle$  is expanded correctly since for the same  $v$  the respective arrays have the same number of elements<sup>1</sup>. It is obvious that the PTLDB vertex-to-vertex query is very simple, since it is implemented with just a few lines of SQL code and at the same time, it is highly optimized since it only has to fetch only one row from each *lout* and *lin* DB tables.

THEOREM 3.1.1. *The PTLDB v2v query is correct.*

PROOF. By adding the dummy tuples to the *lout* and *lin* DB tables, we can guarantee that the solution to any vertex-to-vertex query is a combination of one tuple  $l1 \in lout$  and one tuple  $l2 \in lin$  with  $l1.hub = l2.hub$  and  $l1.ta \leq l2.td$  (Line 19). Considering the fact that PostgreSQL guarantees correct unnesting of the hubs, tds and tas arrays (line 2-4, 7-9) for the respective rows for  $u$  and  $v$  and the extra conditions specific for each type of query (Lines 12,14 and 16 and lines 21, 23) then the resulting PTLDB vertex-to-vertex query is correct.  $\square$

## 3.2 EA and LD $k$ NN queries

The  $k$ -Nearest Neighbour ( $k$ NN) query, either for Euclidean space or for network databases, is a very well-studied problem in database systems due to its wide range of applications. Unfortunately, it has not been extended yet, to public transportation networks. To this propose, we formulate the Earliest Arrival and Latest Departure  $k$ -Nearest Neighbour ( $k$ NN) queries for schedule-based timetable networks, according to the following definitions:

- *Earliest Arrival  $k$ NN query (EA- $k$ NN)*. Given a stop  $q$ , a set of target-stops  $T$  and a starting timestamp  $t$ , the earliest arrival  $EA-kNN(q, T, t, k)$  query seeks the  $k$ -distinct stops  $\in T$  with the earliest arrival time among the paths that (i) start from  $q$  no sooner than  $t$  and (ii) end in any stop  $\in T$ .
- *Latest Departure  $k$ NN query (LD- $k$ NN)*. Given a stop  $q$ , a set of target-stops  $T$  and an ending timestamp  $t$ , the latest departure  $LD-kNN(q, T, t, k)$  query seeks the  $k$ -distinct stops  $\in T$  with the latest departure time from stop  $q$  from among the paths that (i) start from  $q$  and (ii) end in any stop  $\in T$  no later than  $t$ .

Both these type of queries may be used in a wide range of useful applications, such as an tourist deciding to visit the nearest Point of Interest (POI) using public transport

<sup>1</sup><http://stackoverflow.com/a/23838131>

Table 2: The *lout* table used in *PTLDB* for the example graph *G*

v	hubs	tds	tas
...	...	...	...
1	{0, 1, 1}	{324, 324, 396}	{360, 324, 396}
...	...	...	...
4	{0, 4}	{324, 396}	{360, 396}
...	...	...	...

Table 4: The *ea\_knn\_naive* table for the example graph *G*,  $T = \{4, 6\}$  and  $k = 1$

hub	td	vs	tds
0	360	{4, 6}	{396, 432}
2	396	{6}	{432}
4	396	{4}	{396}
6	432	{6}	{432}

(EA- $k$ NN) or how a city visitor may determine his remaining time for finishing his breakfast, before reaching one of his preferred POI-destinations by 11:00 (LD- $k$ NN). To the best of our knowledge, *this is the first time that these queries have been formalized for public transit networks* and therefore we are not aware of any previous approach tackling them. Throughout this work we assume that targets are not changing, which is a reasonable assumption for public transportation networks, since, e.g., for location based services we already know the stops that are located near attractive POIs or the most visited city-landmarks. In the following sections, we will show how to efficiently solve those queries within *PTLDB*. For our example graph *G* (Figure 1) and the remainder of this section, we assume that target stops are 4 and 6, i.e.,  $T = \{4, 6\}$

### 3.2.1 Implementation

In this section, we will mainly discuss EA- $k$ NN queries. The solution to LD- $k$ NN queries will be directly analogous. Typically, to solve  $k$ NN queries with the hub labeling method, we need to group the  $L_{in}$  tuples of the targets by hub [1, 16, 14] and keep the  $k$ -best entries per hub. Unfortunately, this approach cannot be extended directly to public transportation networks, due to the condition  $l_1.ta \leq l_2.td$  that must always hold. A naive solution to this problem, would be to group the  $L_{in}$  tuples of the targets per *hub* and  $t_d$  instead, and again keep the best distinct  $k$ -entries per *hub*,  $t_d$  ordered by  $t_a$ , with ties broken arbitrarily. The results of this process would then be stored in the DB table *ea\_knn\_naive*, with the data structure shown in Table 4. As seen there, for  $k = 1$ ,  $hub = 0$  and  $td = 360$  we only need to keep the best entry that corresponds to  $v = 4$  and  $td = 396$ . The primary key of *ea\_knn\_naive* table will be the (*hub*,  $t_d$ ) combination. After building this table, the EA- $k$ NN( $q, T, t, k$ ) query may be solved by the SQL of Code 2, that combines the row of *lout* DB table that corresponds to vertex  $q$ , with the *ea\_knn\_naive* table. Therefore the EA- $k$ NN( $0, \{4, 6\}, 360, 1$ ) will have the correct answer (4, 396), i.e., the NN of vertex 0 for departure time 360 or later is the vertex 4 with arrival time 396. As showcased in [14], it makes sense to create one large *ea\_knn\_naive* table for the maximum value  $kmax$  of  $k$  (e.g., for  $k = 16$ ) that may be serviced by the DB framework and that same table will be used for all  $k$ NN queries up to  $k = kmax$ . In this case, we only need to retrieve  $k$ -entries per (*hub*,  $t_d$ ) combination and thus we only expand  $vs[1 : k]$  and  $tas[1 : k]$

Table 3: The *lin* table used for *PTLDB* for the example graph *G*

v	hubs	tds	tas
...	...	...	...
1	{0, 1, 1}	{360, 324, 396}	{396, 324, 396}
...	...	...	...
4	{0, 4}	{360, 396}	{396, 396}
...	...	...	...

(Lines 14-15) for  $k < kmax$ .

Code 2: EA- $k$ NN naive query for *PTLDB*

```

1 WITH n1 AS
2   (SELECT v, hub, td, ta
3    FROM
4     (SELECT v AS v,
5           UNNEST(hubs) AS hub,
6           UNNEST(tds) AS td,
7           UNNEST(tas) AS ta
8    FROM lout
9     WHERE v=q) n1a
10  WHERE td >=t)
11 SELECT v2, MIN(n2.ta)
12 FROM n1,
13  (SELECT hub, td,
14     UNNEST(vs[1:k]) AS v2,
15     UNNEST(tas[1:k]) AS ta
16  FROM ea_knn_naive) n2
17 WHERE n1.hub=n2.hub
18 AND n2.td>=n1.ta
19 GROUP BY v2
20 ORDER BY MIN(n2.ta), v2
21 LIMIT k;

```

THEOREM 3.2.1. *The naive EA- $k$ NN query is correct.*

PROOF. The naive EA- $k$ NN query joins the  $l_1$  tuples in  $q$  row of DB table *lout*, with the  $l_2$  tuples of *ea\_knn\_naive* DB table with  $l_1.hub = l_2.hub$  and  $l_1.ta \leq l_2.td$ . Since for each individual (*hub*,  $t_d$ ) combination the *ea\_knn\_naive* DB table stores the top- $k$  (earliest arrival) entries, this ensures that the naive EA- $k$ NN query provides correct results.  $\square$

Although the EA- $k$ NN naive query is very simple, it cannot scale well for large metropolitan networks. In a realistic setting, multiple buses or trains leave the same *hub* every few minutes and therefore for each hub we will have multiple  $t_d$  entries. Thus, the size of *ea\_knn\_naive* DB table and its primary key index will vastly increase (even after keeping only the best  $k$ -entries per *hub*,  $t_d$ ). The number of rows that should be joined will also grow, making queries too slow for real-time applications. Hence, we must further group entries per hub and create a condensed *knn\_ea* DB table. Ideally, each tuple contained in the  $q$  row of *lout* DB table should be joined with only a single row in the *knn\_ea* table, during a EA- $k$ NN query. To achieve that, we can group hub entries per hour of departure, i.e., making a separate entry per hub and hour of departure for the available timestamp ranges of  $t_d$  in *lin* DB table. The resulting *knn\_ea* table will have the data structure showcased in Table 5 and the combination of *hub*, *dephour* as a primary key. For a specific hub and hour, e.g.,  $hub = 0$ , *dephour* = 10, (i) the columns *tds-exp*, *vs-exp* and *tas-exp*, contain ALL tuples of *lin*

Table 5: The *knn\_ea* table data structure

hub	dephour	vs	tas	tds-exp	vs-exp	tas-exp
0	0	top-k entries (v) $hub = 0, hour \geq 1$	top-k entries (ta) $hub = 0, hour \geq 1$	all entries (td) $hub = 0, 0 \leq hour \leq 1$	all entries (v) $hub = 0, 0 \leq hour \leq 1$	all entries (ta) $hub = 0, 0 \leq hour \leq 1$
0	1	top-k entries (v) $hub = 0, hour \geq 2$	top-k entries (ta) $hub = 0, hour \geq 2$	all entries (td) $hub = 0, 1 \leq hour \leq 2$	all entries (v) $hub = 0, 1 \leq hour \leq 2$	all entries (ta) $hub = 0, 1 \leq hour \leq 2$
...	...	...	...	...	...	...
1	0	top-k entries (v) $hub = 1, hour \geq 1$	top-k entries (ta) $hub = 1, hour \geq 1$	all entries (td) $hub = 1, 0 \leq hour \leq 1$	all entries (v) $hub = 1, 0 \leq hour \leq 1$	all entries (ta) $hub = 1, 0 \leq hour \leq 1$
1	1	top-k entries (v) $hub = 1, hour \geq 2$	top-k entries (ta) $hub = 1, hour \geq 2$	all entries (td) $hub = 1, 1 \leq hour \leq 2$	all entries (v) $hub = 1, 1 \leq hour \leq 2$	all entries (ta) $hub = 1, 1 \leq hour \leq 2$

for targets  $T$ , with  $hub = 0$  and  $t_d$  between 10:00 and 11:00 ordered by  $t_d$ , whereas (ii) the columns *vs* and *tas* contain only the best top-k (earliest arrival) distinct entries for targets  $T$  and  $hub = 0$ ,  $t_d \geq 11:00$ .

Thus, the optimized EA-kNN query must implement those two separate cases: (i) Expanding the  $l_2$  tuples for a specific hub between e.g., 10:00 and 11:00 contained in DB columns *tds-exp*, *vs-exp* and *tas-exp* (still checking that  $l_1.ta \leq l_2.td$  for those entries) and (ii) Expanding the  $l_3$  tuples that leave the specific hub after 11:00. As showcased earlier, both cases are included in a single row per hub of the *knn\_ea* DB table. The resulting query is shown in Code 3. For combining those aforementioned cases we still have to use the UNION operator (Line 30) and for increasing performance, the JOIN between the *lout* and *knn\_ea* DB tables happens AFTER expanding the *lout* tuples for  $q$  row and BEFORE expanding the tuples in *knn\_ea* DB table (Lines 1-18). Note, that if each row in *lout* DB table contains on average  $|L_{out}|/|V|$  tuples, then the optimized EA-kNN query will always access at most  $|L_{out}|/|V|$  rows from the *knn\_ea* DB table, thus minimizing secondary storage utilization.

**THEOREM 3.2.2.** *The construction of the *knn\_ea* DB table and the corresponding EA-kNN query are correct.*

**PROOF.** For the EA-kNN query  $(q, T, t)$ , assume there is a tuple  $l_1 = \langle h, td, ta \rangle$  with  $td \geq t$  for the query vertex  $q$ , included in the *lout* DB table. The EA-kNN query will join this tuple with exactly one row in the *knn\_ea* DB table for which  $hub = h$  and  $dephour = FLOOR(ta/3600)$ . This specific *knn\_ea* row contains (i) all  $l_2$  tuples of targets  $T$  for  $hub = h$  and  $dephour$  between  $FLOOR(ta/3600)$  and  $FLOOR(ta/3600)+1$  (we still need to check for those entries that  $l_1.ta \leq l_2.td$ ) but (ii) only the best top-k (earliest arrival) distinct  $l_3$  tuples for targets  $T$  that leave  $hub = h$  after  $FLOOR(ta/3600) + 1$ . There is no need to search for any other tuples for  $hub = h$  after  $FLOOR(ta/3600) + 1$  because all other tuples will have worst arrival time than the  $l_3$  tuples. Also, there is no need to look for any other tuples for  $hub = h$  before  $FLOOR(ta/3600)$ , because the  $l_1$  trip arrives at hub  $h$  after  $FLOOR(ta/3600)$ . Since the EA-kNN query will similarly join all tuples in *lout* DB that leave query vertex  $q$  after timestamp  $t$ , the resulting EA-kNN query is correct.  $\square$

Considering the choice of an hour as the tuning parameter for grouping the *knn\_ea* DB table entries, we could have chosen any other valid time interval to that purpose. In fact, we have even experimented with other intervals (smaller or larger than an hour) but (i) when smaller intervals are used, the respective *knn\_ea* table has more rows (which makes queries slower) and (ii) when larger intervals are used (e.g., 3 hours) the number of tuples stored in *tds-exp*, *vs-exp*

and *tas-exp* columns vastly increases, which counteracts the benefit of the smaller number of total rows. Thus, a time interval of an hour seems like the best compromise between those two scenarios and worked well for all tested datasets. However, it can be tuned according to the specific use-cases and user needs for a particular public transit network.

In the case of LD-kNN queries, the corresponding *knn\_ld* table will have a similar data structure to the *knn\_ea* table. There are some main differences though: (i) Entries are grouped by hub and hour of ARRIVAL, i.e., for a specific hub and hour, e.g.,  $hub = 0, arrhour = 10$ , the columns *tds-exp*, *vs-exp* and *tas-exp*, contain all tuples of *lin* for targets  $T$ , with  $hub = 0$  and  $t_a$  between 10:00 and 11:00 ordered by  $t_a$ . Likewise, the combination of *hub, arrhour* will be the primary key. (ii) The columns *vs* and *tds* (and not *tas* as before) contain only the best top-k (latest departure) distinct entries for targets  $T$  and  $hub = 0$ ,  $t_a \leq 10:00$ . The corresponding LD-kNN query (see Code 4) will also be slightly different (e.g.,  $MIN(n3.ta)$ ,  $MIN(n2.ta)$ ) will be replaced by  $MAX(n1.td)$  or the DESC ordering) but it will still offer the same performance benefits as before.

### 3.3 EA and LD One-to-many queries

Similar to kNN, we formulate the Earliest Arrival and Latest Departure One-to-many queries for schedule-based timetable networks, according to the following definitions:

- *Earliest Arrival One-to-many query* (EA-OTM). Given a stop  $q$ , a set of target-stops  $T$  and a starting timestamp  $t$ , the earliest arrival  $EA-OTM(q, T, t)$  query seeks the earliest arrival time for all target-stops for trips that start from  $q$  no sooner than  $t$ .
- *Latest Departure One-to-many query* (LD-OTM). Given a stop  $q$ , a set of target-stops  $T$  and an ending timestamp  $t$ , the latest departure  $LD-OTM(q, T, t)$  query seeks the latest departure times for trips starting from  $q$  and end in any stop  $\in T$  no later than  $t$ .

Again the EA-OTM and LD-OTM queries have a wide range of useful applications, such as transportation planning (e.g., find faraway stops) or geomarketing applications (e.g., nearby what stop one must build a franchise store to be more easily reachable by clients). To the best of our knowledge, *this is also the first time that these queries have been formalized for public transportation*. How to efficiently solve them within *PTLDB*, will be shown in the following.

For answering the EA-OTM query, we need to build a new *otm\_ea* DB table that will have the data structure showcased in Table 6. In the *otm\_ea* table the columns *hub*, *dephour* (= hour of departure), *tds-exp*, *vs-exp* and *tas-exp* are identical to the *knn\_ea* DB table and



Code 3: EA-kNN and EA-OTM queries for *PTLDB*

```

1 WITH n1 AS
2 (SELECT v,hub, td, ta
3 FROM
4 (SELECT v,
5 UNNEST(hubs) AS hub,
6 UNNEST(tds) AS td,
7 UNNEST(tas) AS ta
8 FROM lout
9 WHERE v=q) n1a
10 WHERE td >=t),
11 n1b AS
12 (SELECT n1bb.*,
13 n1.ta AS n1_ta
14 n1.td AS n1_td
15 /* EA-$k$NN query */
16 FROM knn_ea n1bb,n1
17 /* EA-OTM query */
18 FROM otm_ea n1bb,n1
19 WHERE n1bb.hub=n1.hub
20 AND n1bb.dephour=FLOOR(n1.ta/3600))
21 SELECT v2,MIN(ta)
22 FROM (
23 (SELECT v2,MIN(n3.ta) AS ta
24 FROM
25 (SELECT
26 /* EA-$k$NN query */
27 UNNEST(tas[1:k]) AS ta,
28 UNNEST(vs[1:k]) AS v2
29 /* EA-OTM query */
30 UNNEST(tas) AS ta,
31 UNNEST(vs) AS v2
32 FROM n1b) n3
33 GROUP BY v2
34 ORDER BY MIN(n3.ta),v2
35 /* EA-$k$NN query */
36 LIMIT k
37 )
38 UNION
39 (SELECT n2.v2,MIN(n2.ta) AS ta
40 FROM
41 (SELECT n1_ta,
42 UNNEST(tds_exp) AS td,
43 UNNEST(vs_exp) AS v2,
44 UNNEST(tas_exp) AS ta
45 FROM n1b) n2
46 /* Check for l1.ta<=l2.td */
47 WHERE n1_ta<=n2.td
48 GROUP BY n2.v2
49 ORDER BY MIN(n2.ta),v2
50 /* EA-$k$NN query */
51 LIMIT k
52 )) S53
53 GROUP BY v2
54 ORDER BY MIN(ta),v2
55 /* EA-$k$NN query */
56 LIMIT k;

```

Code 4: LD-kNN and LD-OTM queries for *PTLDB*

```

1 WITH n1 AS
2 (SELECT v,hub,td,ta
3 FROM
4 (SELECT v,
5 UNNEST(hubs) AS hub,
6 UNNEST(tds) AS td,
7 UNNEST(tas) AS ta
8 FROM lout
9 WHERE v=q) n1a),
10 n1b AS
11 (SELECT n1bb.*,
12 n1.ta AS n1_ta,
13 n1.td AS n1_td
14 /* LD-$k$NN query */
15 FROM knn_ld n1bb,n1
16 /* LD-OTM query */
17 FROM otm_ld n1bb,n1
18 WHERE n1bb.hub=n1.hub
19 AND n1bb.arrhour=FLOOR(t/3600))
20 SELECT v2,MAX(td)
21 FROM (
22 (SELECT v2,MAX(n3.n1_td) AS td
23 FROM
24 (SELECT n1_td, n1_ta,
25 /* LD-$k$NN query */
26 UNNEST(tds[1:k]) AS td,
27 UNNEST(vs[1:k]) AS v2
28 /* LD-OTM query */
29 UNNEST(tds) AS td,
30 UNNEST(vs) AS v2
31 FROM n1b) n3
32 WHERE n3.td>=n1_ta
33 GROUP BY v2
34 ORDER BY MAX(n3.n1_td) DESC, v2
35 /* LD-$k$NN query */
36 LIMIT k
37 )
38 UNION
39 (SELECT n2.v2,MAX(n2.n1_td) AS td
40 FROM
41 (SELECT n1_td,n1_ta,
42 UNNEST(tds_exp) AS td,
43 UNNEST(vs_exp) AS v2,
44 UNNEST(tas_exp) AS ta
45 FROM n1b) n2
46 WHERE n2.td>=n1_ta
47 AND n2.ta<=t
48 GROUP BY n2.v2
49 ORDER BY MAX(n2.n1_td) DESC, v2
50 /* LD-$k$NN query */
51 LIMIT k
52 )) S53
53 GROUP BY v2
54 ORDER BY MAX(td) DESC, v2
55 /* LD-$k$NN query */
56 LIMIT k;

```

Table 6: The *otm\_ea* table data structure

hub	dephour	vs	tas	tds-exp	vs-exp	tas-exp
0	0	best entry per target (v) $hub = 0, hour \geq 1$	best entry per target (ta) $hub = 0, hour \geq 1$	all entries (td) $hub = 0, 0 \leq hour < 1$	all entries (v) $hub = 0, 0 \leq hour < 1$	all entries (ta) $hub = 0, 0 \leq hour < 1$
0	1	best entry per target (v) $hub = 0, hour \geq 2$	best entry per target (ta) $hub = 0, hour \geq 2$	all entries (td) $hub = 0, 1 \leq hour < 2$	all entries (v) $hub = 0, 1 \leq hour < 2$	all entries (ta) $hub = 0, 1 \leq hour < 2$
...	...	...	...	...	...	...
1	0	best entry per target (v) $hub = 1, hour \geq 1$	best entry per target (ta) $hub = 1, hour \geq 1$	all entries (td) $hub = 1, 0 \leq hour < 1$	all entries (v) $hub = 1, 0 \leq hour < 1$	all entries (ta) $hub = 1, 0 \leq hour < 1$
1	1	best entry per target (v) $hub = 1, hour \geq 2$	best entry per target (ta) $hub = 1, hour \geq 2$	all entries (td) $hub = 1, 1 \leq hour < 2$	all entries (v) $hub = 1, 1 \leq hour < 2$	all entries (ta) $hub = 1, 1 \leq hour < 2$

the combination (*hub*, *dephour*) again serves as the primary key. The only difference is in the *vs* and *tas* columns, where we must store the best tuple (earliest arrival) per vertex for the following hours, instead of only the top-*k* entries over all targets (as in *knn\_ea* table), i.e., the *vs* and *tas* columns will store at-most  $|V|$  tuples per row, instead of only *k*. Although this makes the resulting EA-OTM query slower, its SQL implementation is practically identical to the EA-*k*NN query (see Code 3). We only need to replace the *knn\_ea* table with *otm\_ea* (Line 16), remove the LIMIT *k* clauses (Lines 36,51,56) and use UNNEST(*tas*), UNNEST(*vs*) (Lines 30,31) instead of the lines 27,28.

Likewise, for LD-OTM queries we need to build the corresponding *otm\_ld* DB table that follows the same structure as *knn\_ld* DB table, except that in the *vs* and *tds* columns, we must store the best tuple (latest departure) per vertex for the PREVIOUS hours, instead of only the top-*k* entries over all targets (as in *knn\_ld* table). The respective LD-OTM query is very similar to the previous LD-*k*NN query, as showcased in Code 4.

Conclusively, for any query vertex *q* (containing on average  $|L_{out}|/|V|$  tuples), then the proposed *k*NN and *One-to-many* queries will always access at most  $|L_{out}|/|V|$  rows from the respective *knn* or *otm* DB tables. Thus, it will be hard to achieve better secondary storage utilization inside a database. It is important to note that once we load the TTL labels and create the *lout* and *lin* DB tables, all the auxiliary DB tables within *PTLDB* (namely the *knn\_ea*, *knn\_ld*, *otm\_ea* and *otm\_ld*) may also be created by simple SQL commands (the corresponding queries were omitted due to space restrictions). This fact also demonstrates that *PTLDB* is truly a pure-SQL framework for servicing multiple route-planning queries on public transportation graphs. In the following experimentation section, we will benchmark *PTLDB*'s performance for various real-world datasets.

## 4. EXPERIMENTAL EVALUATION

To assess the performance of *PTLDB* on various public transportation graphs, we conducted experiments on a workstation with a 4-core Intel i7-4771 processor clocked at 3.5GHz and 32Gb of RAM, running Ubuntu 14.04. In our experiments, we use the same 11 public transportation networks from [19], as in Timetable Labeling (TTL) [23], where “each dataset records the timetable of the public transportation network of a major city or country on a weekday” [23]. The characteristics of these networks and the necessary TTL preprocessing (using the vertex ordering files provided by its authors) for creating the labels are presented in Table 7. The graphs’ average degree is between 53 and 413 and the TTL algorithm creates 630 – 7,230 tuples per vertex, requiring 4.5 – 353.6s for the labels’ construction.

Table 7: Public transportation graphs statistics

Graph	V	E	Avg degr.	HL / V	TTL Preproc. Time (s)
Austin	2K	317K	119	1,600	11.3
Berlin	12K	2,081K	153	1,734	184.7
Budapest	5K	1,446K	252	2,486	54.4
Denver	10K	7,11K	75	1,190	27.3
Houston	10K	1,113K	113	2,196	72.6
Los Angeles	15K	1,928K	127	2,572	194.5
Madrid	4K	1,913K	413	7,230	338.5
Roma	9K	2,281K	258	4,370	353.6
Salt Lake City	6K	330K	53	630	4.5
Sweden	51K	4,072K	76	775	179.1
Toronto	10K	3,300K	305	2,987	262.1

*PTLDB* was implemented in PostgreSQL 9.3.6, 64bit with the same settings used in [14] (8192Mb *shared buffers* and 64Mb *temp buffers*). We conducted experiments belonging to the following query types: (i) Earliest Arrival (EA), Latest Departure (LD) and Shortest Duration (SD) *vertex-to-vertex*, (ii) Earliest Arrival (EA) and Latest Departure *k*NN and (iii) Earliest Arrival (EA), Latest Departure (LD) *one-to-many*. For each experiment, we used 1,000 random start vertices (and goal vertices for vertex-to-vertex queries), reporting the average running time. Starting timestamps for EA and SD queries are randomly selected from the first quarter of timestamp ranges, whereas ending timestamps for LD and SD queries are randomly selected from the fourth quarter of timestamp ranges, to ensure that in the majority of the cases we actually get trip results that service a particular type of query. Contrarily, selecting timestamps randomly from all available ranges would significantly lower query times, since a significant percent of those queries would provide no results (no trip would fill the suggested criteria). Before each experiment, we restart the PostgreSQL server for clearing its internal query cache and we also clear the operating system’s cache for accurate benchmarking. All *k*NN and one-to-many charts are plotted in logarithmic scale. Note that (i) *PTLDB* is the only pure-SQL framework tailored for servicing public transportation queries and (ii) to the best of our knowledge there is no previous work or any working system trying to tackle EA, LD *k*NN and one-to-many queries for such networks. Thus, we only present our results, since there is no previous secondary-storage work for comparison.

### 4.1 Performance on HDD

In our first round of experiments, we ran experiments on an HDD, specifically a SATA3, ST3000DM001, 7200rpm Seagate Barracuda disk with 64Mb cache.

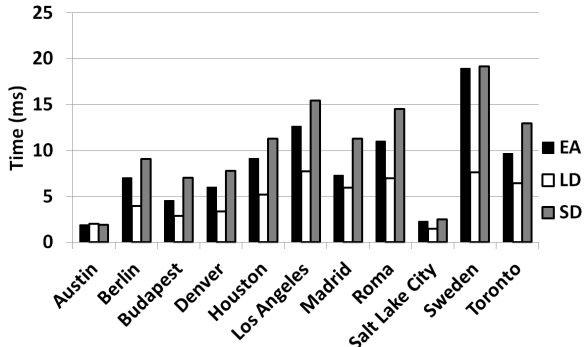


Figure 2: EA, LD and SD v2v queries on a HDD

#### 4.1.1 Vertex-to-vertex queries

Figure 2 shows results for vertex-to-vertex (v2v) queries for *PTLDB*. Results show that LD queries are 35% faster than EA queries, because in the LD queries we select timestamps from the fourth quarter of timestamp ranges where there are less trips than the beginning of the day (as in EA queries). SD queries are on average 26% slower than EA queries, due to the increased complexity of the query. In all cases, EA and SD queries take less than 19.2ms and LD queries take less than 7.7ms, which is an considerable achievement, since even main memory solutions (before TTL) would require a few ms for vertex-to-vertex queries and the suggested datasets [23]. Moreover, we may answer such queries with a simple SQL command inside a database, which ensures scalability, regardless of the numbers of users or the size of the datasets and with a performance that is fast enough for real-time online applications.

#### 4.1.2 kNN queries

In this section, we provide the *PTLDB*'s results for EA and LD *kNN* queries. Similar to previous works [14], we will experiment with varying values of  $k$  and *target density*  $D$ , i.e., the ratio  $|T|/|V|$ , where  $T$  is the set of target-stops in the graph and  $|V|$  is the total number of vertices. As explained in Section 3.2.1, for database frameworks it makes sense to create large *knn* tables for the maximum value  $k_{max}$  of  $k$  that will be serviced by the respective framework. Thus, we have created two versions of *kNN* DB tables for *PTLDB*, one for  $k_{max} = 4$  and one for  $k_{max} = 16$ . Then, the *kNN* DB table for  $k_{max} = 4$  is used for answering *kNN* queries for  $k = 1$ ,  $k = 2$  and  $k = 4$  and the *kNN* table for  $k_{max} = 16$  is used for answering *kNN* queries for  $k = 8$  and  $k = 16$ .

In our first set of *kNN* experiments, we compare our optimized EA-*kNN* and LD-*kNN* queries (see Codes 3, 4) in comparison to the corresponding naive *kNN* implementation (Code 2) for varying values of  $k$ . Results are presented in Figure 3. Results show that the optimized versions are 11 – 53× faster than their naive counterparts. Thus, it really pays off to group tuples in the *knn\_ea* (and *knn\_ld*) DB tables by departure (arrival) hour. For the remainder of the paper, we will only provide results for the optimized EA-*kNN* and LD-*kNN* queries, since those queries provide significantly superior performance.

Figure 4 shows the absolute times of optimized EA-*kNN* and LD-*kNN* queries for the same scenario, i.e., for  $D = 0.01$  and varying values of  $k$ . Results show that the EA-*kNN* queries require <64ms for all values of  $k$  (except the highest ratio  $|HL|/|V|$  dataset of Madrid and  $k = 8, 16$ ). LD-*kNN*

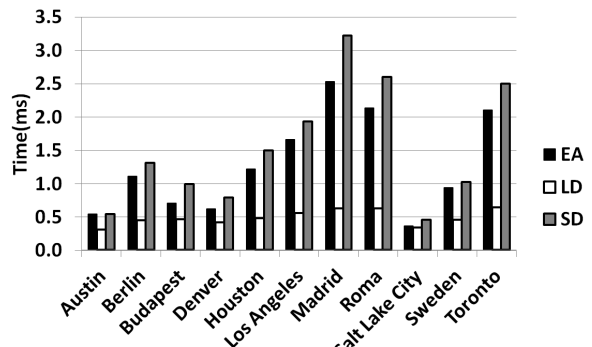


Figure 7: EA, LD and SD v2v queries on a SSD

queries are even faster, requiring less than 32ms on all cases.

In our third set of *kNN* experiments, we assess the performance of *PTLDB* for varying values of  $D$ . For each value for  $D$ , we have build separate versions of *knn\_ea* and *knn\_ld* DB tables for  $D \cdot |V|$  targets selected at random from each dataset and  $k_{max} = 4$ . Figure 5 shows results for  $k = 4$  and  $D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$ . Results show, that although *PTLDB*'s performance degrades for larger values of  $D$ , *kNN* queries may still be answered in less than 128ms (with the exception of Madrid for EA queries and Toronto for LD queries and  $D = 0.1$ ). For the smaller datasets (Austin, Berlin, Budapest, Denver, Houston, Los Angeles, Salt Lake City, even Sweden) *kNN* queries always take less than 32ms. Moreover, EA-*kNN* queries are more robust to increasing values of  $D$  than LD-*kNN* queries that perform significantly worse for denser targets (i.e., for  $D = 0.1$ ). Conclusively, the *PTLDB* framework provides excellent *kNN* query performance for all values of  $D$  and  $k$ .

#### 4.1.3 One-to-Many queries

In our third round of experiments, we assess the performance of *PTLDB* for one-to-many queries. Figure 6 presents the corresponding results for varying values of  $D$  ( $D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$ ). *PTLDB* answers EA-OTM queries in less than 512ms for all datasets and values of  $D$  (except the Madrid and Toronto datasets that require 1084ms and 751ms respectively for  $D = 0.1$ ). For LD-OTM queries *PTLDB* requires less than 256ms for all datasets and values of  $D$  (except the Madrid, Roma and Toronto datasets that require 303ms, 325ms and 349ms respectively for  $D = 0.1$ ). Note, that for such high values of  $D$ , the *one-to-many* query almost degrades to the *one-to-all* query and hence, it cannot get any faster on a secondary storage device.

## 4.2 Performance on SSD

Having established *PTLDB*'s performance in the HDD, we repeat most previous experiments on a SSD (a SATA3 Crucial CT512MX100SSD1 MX100 512GB 2.5") to measure how the secondary-storage device type impacts results.

#### 4.2.1 Vertex-to-vertex queries

Results for all variants of vertex-to-vertex queries on the SSD are shown in Figure 7. Results show that by using the SSD, *PTLDB* is 3 - 20× faster for EA, 6 - 17× faster for LD and 3 - 19× faster for SD vertex-to-vertex queries. Thus, EA queries may now be answered in less than 2.5ms, SD queries may now be answered in less than 3.2ms and LD queries may now be answered in less than 0.6ms. Conclusively, the usage

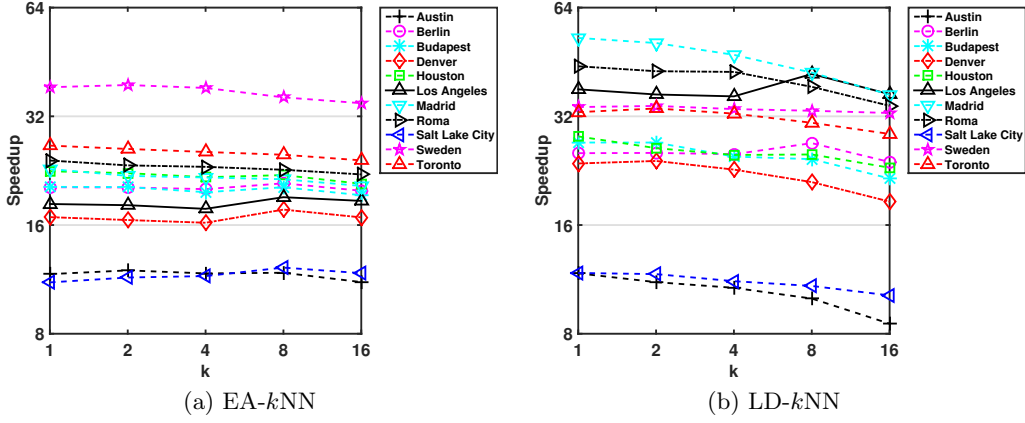


Figure 3: Speedup of optimized  $k$ NN queries, in comparison to the naive versions for  $D = 0.01$  and varying values of  $k$

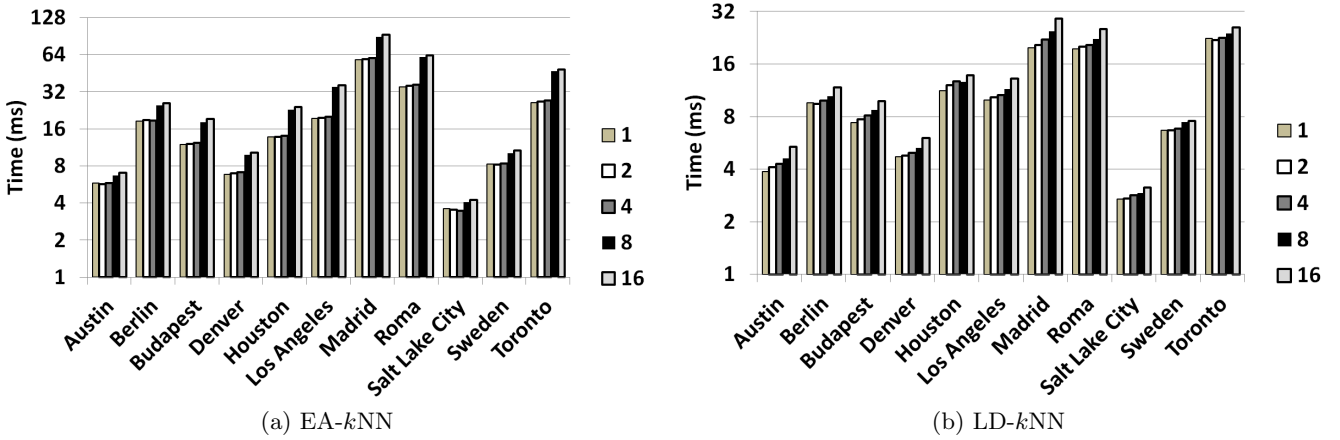


Figure 4:  $k$ NN queries for  $D = 0.01$  and varying values of  $k$

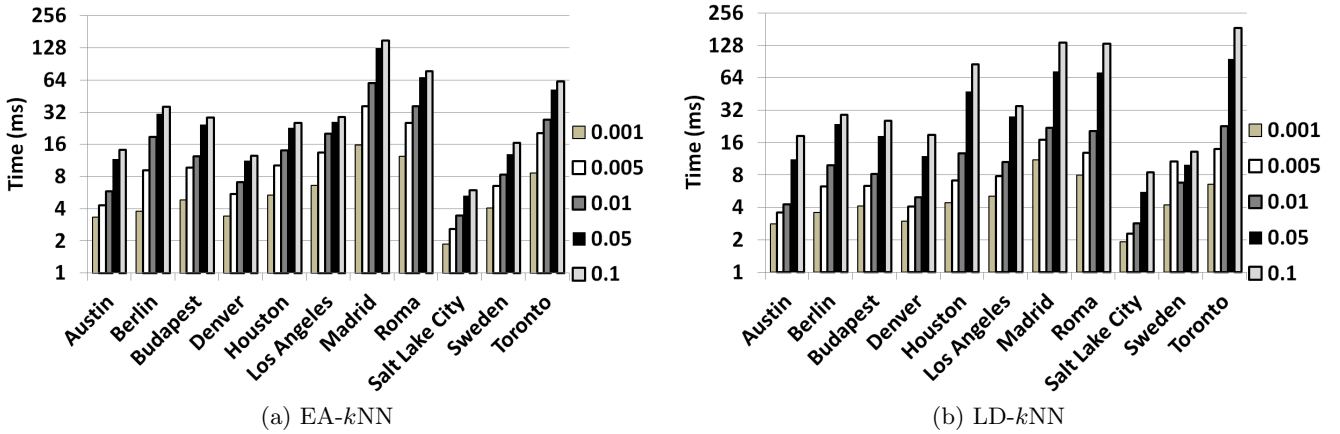


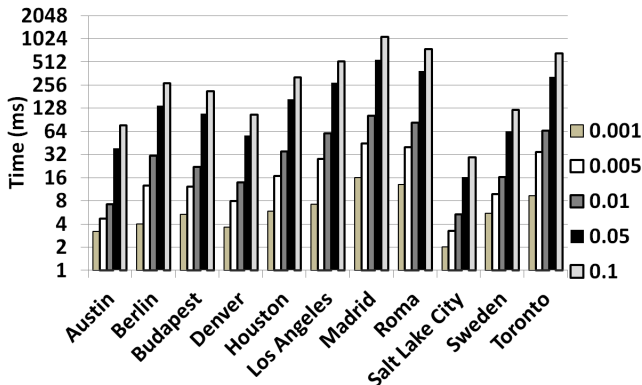
Figure 5:  $k$ NN queries for  $k = 4$  and varying values of  $D$

of SSD benefits significantly all vertex-to-vertex variations within *PTLDB* and therefore *PTLDB* may easily be used for public-transit real-time applications and such queries, since query times always require less than  $3.2ms$ .

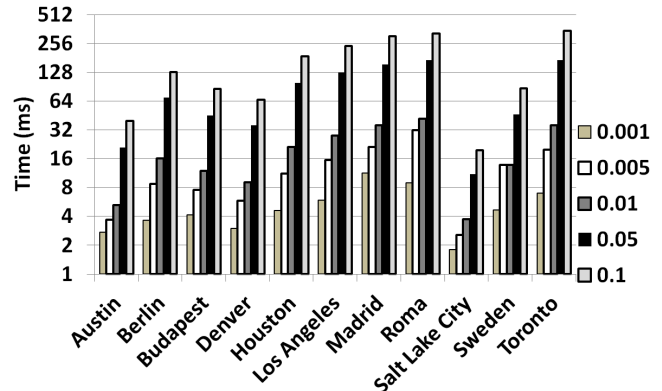
#### 4.2.2 $k$ NN and One-to-many queries

In this section, we repeated all the  $k$ NN and one-to-many experiments performed in Sections 4.1.2 and 4.1.3 on the SSD. Results for  $k$ NN queries,  $D = 0.01$  and varying values

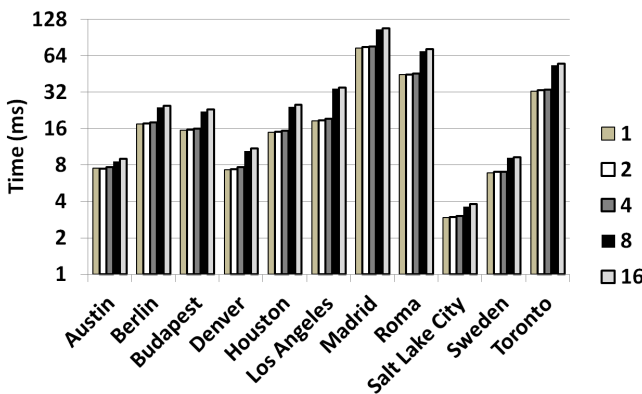
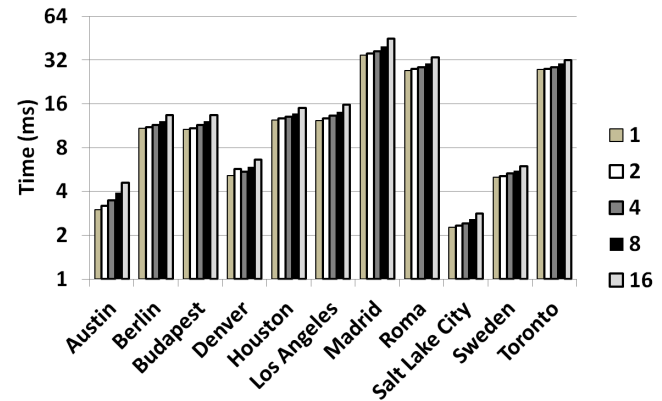
of  $k$  are presented in Figure 8. Results show that for  $k$ NN queries, the usage of the SSD does not provide any further benefits (in fact sometimes the SSD performs slightly worse), meaning that in *PTLDB* we have effectively minimized secondary storage utilization for  $k$ NN queries and thus, adding a faster storage medium adds no performance benefits. The same pattern was encountered on all experiments, for different values of  $D$  or  $k$ , including the respective one-to-many queries and therefore the resulting figures are omitted.



(a) EA-OTM



(b) LD-OTM

Figure 6: One-to-many queries for varying values of  $D$ (a) EA- $k$ NN(b) LD- $k$ NNFigure 8:  $k$ NN queries for  $D = 0.01$  and varying values of  $k$  on the SSD

### 4.3 Summary

Our experimentation has shown that our proposed *PTLDB* framework provides excellent performance for all public transportation planning queries. Using HDDs, *PTLDB* may answer vertex-to-vertex queries in less than  $19.2ms$ . For SSDs, this time drops down to  $3.2ms$ . For the newly formulated EA and LD  $k$ NN queries, *PTLDB* requires less than  $64ms$  and  $32ms$ , for  $k = 16$  and  $D = 0.01$  for the vast majority of the tested datasets. Even the EA and LD One-to-many queries require less than  $512ms$  and  $256ms$  respectively, for most datasets and varying values of  $D$ . Regarding memory requirements, *PTLDB* is very modest, since all DB tables and primary key indexes, including the *lout*, *lin*, *knn\_ea*, *knn\_ld* (for all values of  $D$  and  $kmax = 4, 16$ ) and the *otm\_ea*, *otm\_ld* tables (for all available values of  $D$ ) for all tested datasets, require less than  $12GB$ . Hence, *PTLDB* may easily scale to even significantly larger datasets. Overall, not only *PTLDB* is the only pure-SQL framework tailored for multiple public-transportation queries, offering excellent performance for real-time applications but the simplicity of its SQL queries, makes its integration with existing real-world applications very easy and seamless.

### 5. CONCLUSION

This work presented *PTLDB*, a novel SQL framework for answering multiple route-planning queries for public transportation graphs on a database. Our results showed that *PTLDB* provides excellent query performance, minimum secondary storage utilization and maximum scalability. Moreover, we have extended  $k$ NN and one-to-many queries for public transportation networks and proposed how to efficiently answer them within *PTLDB*, with a few lines of SQL code. This establishes *PTLDB* as a competitive database-driven solution for querying public transportation networks.

The paper gives the complete design and implementation details of *PTLDB* using a popular, open-source database engine, along with the exact SQL queries used in our implementation. This easily allows the replication of our results and might provide the necessary foundation for other researchers to expand the *PTLDB* framework towards handling additional types of queries and novel use-cases. In terms of future work, currently the *PTLDB* framework aims at optimizing travel times, without taking the number of transfers as an additional optimization criterion. Integrating this additional constraint would further improve the use-cases and marketability of the *PTLDB* framework.

## Acknowledgments

This work was partially funded by the project “Research Programs for Excellence 2014-2016 / CitySense-ATHENA R.I.C.” The author would also like to thank the authors of Timetable Labeling (TTL) [23].

## 6. REFERENCES

- [1] I. Abraham, D. Delling, A. Fiat, A. V. Goldberg, and R. F. Werneck. Hldb: Location-based services in databases. In *SIGSPATIAL GIS*. ACM, November 2012.
- [2] I. Abraham, D. Delling, A. Goldberg, and R. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In P. Pardalos and S. Rebennack, editors, *Experimental Algorithms*, volume 6630 of *Lecture Notes in Computer Science*, pages 230–241. Springer Berlin Heidelberg, 2011.
- [3] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In L. Epstein and P. Ferragina, editors, *Algorithms – ESA 2012*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer Berlin Heidelberg, 2012.
- [4] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, USA*, pages 349–360, 2013.
- [5] H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. *CoRR*, abs/1504.05140, 2015.
- [6] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '02*, pages 937–946, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.
- [7] D. Delling, J. Dibbelt, T. Pajor, and R. Werneck. Public transit labeling. In E. Bampis, editor, *Experimental Algorithms*, volume 9125 of *Lecture Notes in Computer Science*, pages 273–285. Springer International Publishing, 2015.
- [8] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. Phast: Hardware-accelerated shortest path trees. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11*, pages 921–931, Washington, DC, USA, 2011.
- [9] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Robust distance queries on massive networks. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 321–333, 2014.
- [10] D. Delling, A. V. Goldberg, and R. F. Werneck. Hub label compression. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, pages 18–29, 2013.
- [11] D. Delling, A. V. Goldberg, and R. F. F. Werneck. Faster batched shortest paths in road networks. In *ATMOS*, pages 52–63, 2011.
- [12] D. Delling and R. F. Werneck. Customizable point-of-interest queries in road networks. In *21st SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2013, Orlando, FL, USA, November 5-8, 2013*, pages 490–493, 2013.
- [13] D. Delling and R. F. F. Werneck. Better bounds for graph bisection. In *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 407–418, 2012.
- [14] A. Efentakis, C. Efstathiades, and D. Pfoser. Cold revisiting hub labels on the database for large-scale graphs. In C. Claramunt, M. Schneider, R. C.-W. Wong, L. Xiong, W.-K. Loh, C. Shahabi, and K.-J. Li, editors, *Advances in Spatial and Temporal Databases*, volume 9239 of *Lecture Notes in Computer Science*, pages 22–39. Springer International Publishing, 2015.
- [15] A. Efentakis and D. Pfoser. GRASP. Extending graph separators for the single-source shortest-path problem. In A. S. Schulz and D. Wagner, editors, *Algorithms - ESA 2014*, volume 8737 of *Lecture Notes in Computer Science*, pages 358–370. Springer Berlin Heidelberg, 2014.
- [16] A. Efentakis and D. Pfoser. Rehub. extending hub labels for reverse k-nearest neighbor queries on large-scale networks. *CoRR*, abs/1504.01497, 2015.
- [17] A. Efentakis, D. Pfoser, and Y. Vassiliou. Salt. a unified framework for all shortest-path query variants on road networks. In E. Bampis, editor, *Experimental Algorithms*, volume 9125 of *Lecture Notes in Computer Science*, pages 298–311. Springer International Publishing, 2015.
- [18] C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '01*, pages 210–219, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.
- [19] GoogleTransitDataFeed. PublicFeeds. List of publicly-accessible transit data feeds [online]. <https://code.google.com/p/googletransitdatafeed/wiki/PublicFeeds>, 2015.
- [20] M. Jiang, A. W. Fu, R. C. Wong, and Y. Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *PVLDB*, 7(12):1203–1214, 2014.
- [21] B. Liao, L. U, M. Yiu, and Z. Gong. Beyond millisecond latency k nn search on commodity machine. *Knowledge and Data Engineering, IEEE Transactions on*, 27(10):2618–2631, Oct 2015.
- [22] PostgreSQL. The world’s most advanced open source database [online]. <http://www.postgresql.org/>, 2015.
- [23] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou. Efficient route planning on public transportation networks: A labelling approach. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15*, pages 967–982, New York, NY, USA, 2015. ACM.
- [24] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou. Timetable labelling [online]. <http://sourceforge.net/projects/tt12015/>, 2015.

# Characterizing Home Device Usage From Wireless Traffic Time Series

Katsiaryna Mirylenka\*  
IBM Research - Zurich  
Rüschlikon, Switzerland  
kmi@zurich.ibm.com

Vassilis Christophides†  
INRIA Paris -  
Rocquencourt, France  
vassilis.christophides@inria.fr

Themis Palpanas  
Paris Descartes University,  
France  
themis@mi.parisdescartes.fr

Ioannis Pefkianakis†  
Hewlett Packard Labs  
Palo Alto, CA, USA  
ioannis.pefkianakis@hpe.com

Martin May  
Technicolor Research &  
Innovation Center  
Rennes, France  
martin.may@technicolor.com

## ABSTRACT

The analysis of temporal behavioral patterns of home network users can reveal important information to Internet Service Providers (ISPs) and help them to optimize their networks and offer new services (e.g., remote software upgrades, troubleshooting, energy savings). This study uses time series analysis of continuous traffic data from wireless home networks, to extract traffic patterns recurring within, or across homes, and assess the impact of different device types (fixed or portable) on home traffic. Traditional techniques for time series analysis are not suited in this respect, due to the limited stationary and evolving distribution properties of wireless home traffic data. We propose a novel framework that relies on a *correlation-based similarity* measure of time series, as well as a notion of *strong stationarity* to define *motifs* and *dominant devices*. Using this framework, we analyze the wireless traffic collected from 196 home gateways over two months. The proposed approach goes beyond existing application-specific analysis techniques, such as analysis of wireless traffic, which mainly rely on data aggregated across hundreds, or thousands of users. Our framework, enables the extraction of recurring patterns (motifs) from traffic time series of individual homes, leading to a much more fine-grained analysis of the behavior patterns of the users. We also determine the best time aggregation policy w.r.t. to the number and statistical importance of the extracted motifs, as well as the device types dominating these motifs and the overall gateway traffic. Our results show that ISPs can exceed the simple observation of the aggregated gateway traffic and better understand their networks.

\*Work done while visiting Technicolor R&I Center.

†Work done while at the Technicolor R&I Center.

## Keywords

Wireless Traffic, Motif Extraction, Similarity Measurement

## 1. INTRODUCTION

The increasing diversity of home devices and network technologies have added layers of complexity to the connected home environment. Residential gateways (RGWs), Smart TVs, smartphones and tablets are just a few of the devices in today's connected home. Furthermore, several "over-the-top" services, such as video streaming (e.g., Netflix) and conferencing (e.g., Skype), are being delivered to a variety of devices and users using wireless home networks. Managing the connected home environment, and indeed optimizing the Quality of Experience (QoE) of residential users, emerges as a critical differentiator for Internet and Communication Service providers (ISPs and CSPs, respectively) and heavily relies on the analysis of home networks.

Although RGWs technology has been considerably improved to deliver IP-based, whole-home services as well as advanced WiFi capabilities as "Community WiFi" hotspots, ISPs still have little information about what lies beyond the subscribed RGW and how home network resources (e.g. bandwidth) are actually consumed by the underlying device ecosystem. For this reason, RGWs have been extended with continuous home network measurement capabilities under normal service operation and provide fine-grained connectivity, usage and performance data of home networks and devices<sup>1</sup>. Mining recurring patterns from these home traffic time series can enable a *data-driven* paradigm in network management [13] in order, for instance, to reduce the cost for *servicing* and *diagnosing* remotely home networks [3, 22, 11], or even to improve residential QoE via home-specific *bandwidth sharing* [1] and *prioritization* [21] policies.

More precisely, home networks troubleshooting is almost always reactively initiated by residential users, requires a human intervention and as a consequence is a time consuming (e.g., 38 min. average time of a technical support call [26]) and not always feasible task (e.g., user's problem is solved in only 14% of technical support calls, partially solved in 24%

<sup>1</sup>Home network monitoring is of course subject of different low restrictions in different countries, and thus we are interested in non-intrusive passive probing techniques that guarantee anonymity.

and not solved at all in 62% [26]). One of the reasons of low efficiency of remote technical support is that technicians cannot completely understand the problem and home network settings, mainly due to limited and sometimes inaccurate information that residential users provide. Extracting previously unknown recurring patterns (aka motifs [19, 7]) from residential traffic time series will bring strong evidence of regular user activity in homes that can be contrasted to the trouble description reported by users. In particular, traffic patterns enriched with detailed home device information is a valuable input for root cause diagnosis. Moreover, in their majority, ISPs typically broadcast firmware and software updates to all gateways at nights (some operators even on a daily basis). This may cause service outages, given that some gateways may exhibit an active network usage during night time. A fine-grained temporal characterization of residential bandwidth consumption will enable ISPs to differentiate RGWs firmware update policies according to the least cumbersome time window per home, thus, improving the overall QoE of residential users.

In addition, home network resources (bandwidth) are shared not only among residents using an increasing number of online applications (e.g., social networking, gaming, uploading/downloading, etc.) and real time services (TV on demand, teleconferencing), but also with guests, neighbors, or even the occasional passersby. Existing methods for bandwidth sharing and traffic prioritization are static and coarse. ISPs usually allocate a fixed percentage of home bandwidth to non-residential users, while traffic prioritization in commodity gateways is at best based on the network port on which traffic is sent or received. We believe that behavioral patterns extracted by gateway traffic time series can be used to support dynamic policies for sharing home bandwidth that consider the online habits of residential users. For example, in-home traffic congestion can be avoided by ordering the traffic patterns of different devices observed especially during afternoon and weekends. These patterns reveal the bandwidth consumption behavior of different groups of residential users (adults and children employ different devices during the same time-slots) while the comparison of traffic domination help us to distinguish between residents and guests (pattern-specific vs global traffic dominant devices).

To the best of our knowledge, this paper presents the first thorough analysis of traffic dynamics of heterogeneous wireless (WiFi) devices connected to 196 real RGWs, which are subscribers of a major European ISP. We focus on a time-oriented analysis of *continuous traffic data* to extract *previously unknown patterns* recurring of internet consumption that happen within, or across homes. We also assess the impact of different types of devices, such as laptops, desktops (classified as “fixed” devices), and tablets, smartphones (classified as “portables”), on these patterns. Unsupervised learning techniques are used for patterns discovery as the ground truth data regarding home activities are not available. Rather than partitioning homes or devices into distinct behavioral clusters, we are looking to extract *informative motifs of bandwidth consumption* within or across homes. Different from a previous analysis of the same dataset [23], which focused on coarsely aggregated gateway traffic, in this paper we conduct various types of time series analysis, including a per-device analysis, motif extraction, and search of dominant devices.

In our study, we develop a framework for analyzing the

distribution properties of traffic data, the stationarity and predictability of usage patterns within and across homes, the similarity of device specific traffic to the overall home traffic, and others. We demonstrate that traditional techniques of time series analysis [20] are not suitable in our setting due to restricted stationarity of traffic time series. This is caused also by the fact that low-valued non-active traffic occupies the most of the probability mass of the traffic distribution, while the values of active traffic are detected as outliers. In contrast to Euclidean Distance and Dynamic Time Warping (DTW), the proposed approach better fits the requirements of our applications: (a) it correctly identifies similar trends, both when absolute values are important and when they are not; (b) it restricts the matches to time-aligned sequences; and (c) it provides a similarity measure, whose values (between  $-1$  and  $1$ ) we know how to interpret based on the theory of statistics.

The main contributions of this paper are:

1) We propose a novel analysis framework for wireless home traffic data, namely: (a) a *correlation-based similarity* measure, which exploits the evolution characteristics, rather than the absolute traffic values, and is invariant to scaling; (b) a notion of *strong stationarity* that in addition to the similarity of data distributions imposes a correlation similarity across non-overlapping time windows; and (d) a definition of *dominant devices* based on the correlation similarity, that enables an intuitive and statistically grounded interpretation of the results.

2) We evaluate the effectiveness of the proposed framework using real data of wireless traffic observations and report the main findings: (a) there are many repetitive patterns within and across RGWs which describe the intrinsic user behavior of users and valuable to ISPs; (a) as networking time series are not stationary certain aggregation should be performed in order to find statistically significant patterns. The best time windows to aggregate home traffic data is found to be 8 hours for weekly patterns and 3 hours for daily patterns; (b) frequent weekly patterns correspond to heavy bandwidth usage both during weekdays and weekends, and frequent daily patterns correspond to (mostly) evening usage, (c) weekend usage tends to rely on portable devices, weekday usage relies more on fixed devices, while discontinuous usage within a day (mostly active in the evening or the morning) is still due to portable devices; and (d) almost every RGW involves a device that dominates its overall traffic, thus the behavior of this device should be mainly considered by ISPs while planning the updates.

The paper is organized as follows. Sections 2 and 3 describe the related work and our dataset. Section 4 shows the preliminary analysis of the wireless traffic. Section 5 describes the proposed methodology. Section 6 correlates the wireless traffic with home devices. Section 7 presents in detail the time aggregation and motifs analysis. Section 8 concludes the paper.

## 2. RELATED WORK

We consider several directions of related work that cover the specifics of analysis of wireless devices in home.

**Wireless traffic analysis.** The analysis of wireless traffic dynamics has been widely used to provide energy savings [12, 24], to build collaborative wireless networks [27], and to accommodate traffic offloading [16]. For example, recent proposals seek to power off idle Access Points (APs) [12]



or cellular base stations [24] to save energy. SEAR [12] forms clusters of WiFi APs and powers on/off APs of the same cluster based on the traffic demand that the cluster needs to serve. In a similar fashion, the system proposed in [24] powers off under-utilized cellular base stations when their traffic load is light, and power them on when the traffic load becomes heavy. Collaboration among APs has been explored in [27] to offer energy savings and load balancing in WiFi APs. The above designs though, are based on two key assumptions, which may not hold in RGWs. First, they are based on wireless traffic stationarity to predict idle times (e.g., the traffic volume is stable over short-term (2 hours), which can remain stable over several consecutive days [24]). Second, a set of devices (e.g., APs or base stations) show very similar temporal traffic characteristics. Our analysis shows that these assumptions do not hold in our case.

**Human behavior.** Multiple works demonstrate that the behavior of humans can be described through recurrent activity patterns. This is shown in the work [4] for social network data, in the work [31] for user behavior in microblog and in the work [10] for moving trajectories using mobile phone data. According to the results of work [10] the patterns of human mobility behavior are very regular. Home traffic data which we focus on is also determined by human behavior, but unlike previous studies it is not defined by a single individual but by a group of individuals who share one home. This leads to less repetability an large variety of possible activity patterns, making the analysis of RGWs more challenging.

Work [14] analyzes human correspondence behavior via mobile phone data. As in the other human activity studies [15, 9, 5], the data show high inhomogeneous in activities, meaning that periods of active events are much shorter than inactivity silence. This leads to the bursty time series with long tails in the probability density function. We observe this property for our data as well while pattern extraction is needed to take place on more regular data. Study [14] checks whether the inhomogeneous of correspondence behavior is due to daily or weekly periodical patterns or it is due to the nature of the behavior of human task execution. According to the results of this study, even after de-seasoning when daily and weekly periodical patterns are excluded, the data remains inhomogeneous, which means that the character of human activities is one of the main sources of bursty time series. Unlike mobile phone data, where long silence periods were observed in night time and during weekends, our networking data for certain homes have peak activities exactly in this “silence” periods. This means that in our case de-seasoning is not applicable as there are no strong daily and weekly silence period for all homes into consideration. Thus, we safely assume that the heterogeneity of our data is caused by human activities even to larger extend than for phone call data. In our work we concentrate on reducing inhomogeneous data characteristics by other means, such as by excluding background traffic from consideration and by specially devised distance measure for traffic time series. Then, we extract recurrent activity patterns of usage behavior using the technique of motifs.

**Motifs.** Mining of motifs or sequential patterns is an important task in time series analysis [19], [7]. As far as we know this kind of analysis has not been applied to the internet traffic data before as most of the studies use aggregated traffic values instead of time series. For this task, we

considered several state-of-the-art tools for motif discovery, such as GrammarViz [17] and VizTree [18]. However, these tools are not suitable for our analysis for the following reasons. (a) These tools (and many other techniques available online) use Symbolic Aggregate approxIimation (SAX) to represent time series, assuming that the distribution of time series values is normal [19]. This is not true for our traffic time series though, as their values follow the Zipf’s law (we note that, contrary to the claims in [19], z-normalization does not lead to normal distribution if the initial distribution of the time series is not normal). At the same time we do not have ground truth data about the motifs in order to tune the alphabet size of SAX, which assigns more symbols near the value of zero, while in our case this region should have been coded with only one symbol. (b) GrammarViz seeks to discover motifs of different lengths, and exploits grammar distance for this. On the other hand, our data has clear time semantics, and we would like to discover motifs for week- and day shifting (i.e., non-overlapping) windows of fixed length, that is not enabled with GrammarViz.

### 3. DATASET DESCRIPTION

We analyze wireless traffic data collected from 196 residential gateways under normal service operation, involving subscribers of a large European ISP that are distributed over a large geographic area spanning 10 cities. The residential subscribers participate on a voluntary basis to our large scale data collection campaign. For privacy reasons, we do not collect data regarding running applications of home devices, demographics and activities users are engaged in.

The gateway platforms of our deployment have the following specifications: (i) ADSL2+ modem or fiber WAN access link, (ii) 4 ethernet ports, (iii) a WiFi access point enabled by a Broadcom 802.11b/g/n 2x2 radio. The 802.11 interface operates at the 2.4GHz band and supports PHY rates up to 300 Mbps. The most (67%) of our deployment’s gateways are fiber (92% of the fiber plans provide 100/10 Mbps downstream/upstream speed, and for the rest it is 30/3 Mbps) and the rest are ADSL (with 24/1 Mbps downstream/upstream speeds). Each gateway logs the traffic counters at all network interfaces on the IP layer, and reports in bytes the cumulative outgoing (transmitted) and incoming (received) traffic of each connected device to the gateway. The focus of this work is the WiFi traffic. The gateway further reports the aggregated gateway traffic, which is the sum of the corresponding outgoing and incoming traffic of all its devices. These data measurements are automatically reported every minute by each gateway to a central server. Note that the wireless traffic reported by a gateway depends on the running applications’ data rates and is bounded by the wireless effective throughput or the access link throughput (for traffic exiting/entering the home). A recent study though [23] has shown that wireless (and wired) network throughput is rarely the bottleneck.

Our dataset includes more than 20 million measurement reports collected over a 2-month period (starting from March 17, 2014). We were able to identify a total of 2147 distinct wireless devices (a device is defined by its MAC address). Using a heuristic-based algorithm [25], we were able to infer the type of a wireless device. The heuristic algorithm leverages the device MAC address (revealing manufacturer name) and the device names typically assigned by the user, which are reported by the gateway. For example, “Nintendo

Co., Ltd.” is known to produce game consoles, “EPSON” – peripheral devices, while “Katy’s-iPhone”, indicates a smartphone manufactured by Apple. We have validated the effectiveness of the algorithm using ground truth data collected from surveys at 49 homes of our deployment. All ‘light’ devices such as smartphones, tablets and others are labeled as “portable”, while laptops and desktops fall under the “fixed” category. There is also a category of “network equipment” that includes devices such as WiFi extenders, and additionally there is a small amount of “game consoles”.

## 4. STANDARD DATA ANALYSIS

In this section, we study the main data characteristics (distribution and stationarity) of the traffic time series observed by the RGWs in our deployment, and discuss the challenges arising in this task when using traditional analytical techniques.

### 4.1 Traffic Data Distribution

We first conduct a preliminary analysis using the wireless traffic data of the 10 most representative gateways with the highest number of observations for a single week period. Our analysis aims to answer the questions: (a) What are the main properties of the distribution of traffic values? Are probability density functions of traffic counters similar across gateways? (b) Which kind of traffic (outgoing, incoming, or overall) provides the most meaningful description of a gateway? To answer these questions, we exploit the following methods:

- 1) *boxplots* in order to visualize general probability distribution and outliers of time series, and
- 2) *estimation of probability density function (PDF)* using Kernel Density Estimators in order to assess and compare the probability distributions of time series.

The above methods lead to the following results: (a) The distribution of incoming and outgoing traffic of gateways follows Zipf’s law (see Figure 1a), which means that the concentration of low traffic values is much larger than the amount of medium and high traffic values. The periods of really active traffic are very small, and thus detected as outliers in data distribution plots and boxplots. As an example, we show a typical time series in Figure 1b, an approximation of its PDF using Kernel Density Estimation zoomed around 0 of the y-axis in Figure 1a, and its boxplots with and without outliers in Figures 1c and 1d. This phenomena is also called inhomogeneity of data and as we mentioned in Section 2 it is typical for data describing human activities.

(b) According to our results, there is a strong correlation between the incoming and outgoing traffic (mean = 0.92, median = 0.95, stddev = 0.08) of the gateways in our deployment. Since they are strongly correlated, we consider that the overall traffic of a gateway reflects the active behavior of a user without artifacts.

**Summary.** Since low traffic values account for most of the probability mass, the traffic values reported when devices are actually used are essentially detected as outliers in data distribution plots and boxplots considered by traditional time series analysis techniques. This motivates us to characterize the background traffic (Section 6.1) and remove it when looking for recurring patterns of active internet consumption. In this context, z-normalization alone does not help, as we want to consider also similarity of rankings of traffic values.

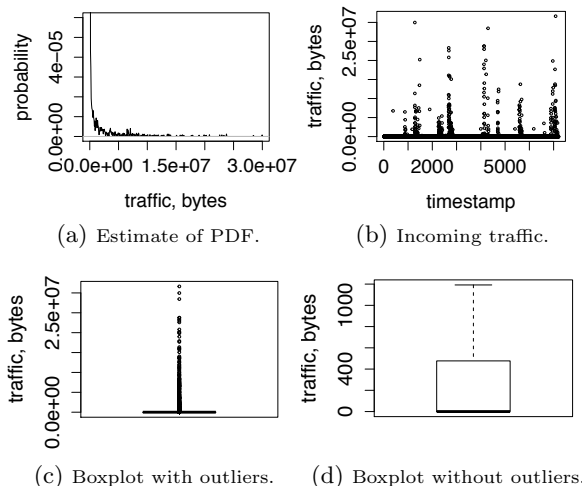


Figure 1: Statistical analysis of a typical gateway.

### 4.2 Traffic Correlation and Stationarity

We now turn our attention to the following questions regarding the characteristics of the data: (a) Is the behavior of traffic counters similar among gateways? (b) Is there significant autocorrelation in the traffic of a gateway, or significant cross (lag-)correlation between gateways, or in general, what is the predictive power of the time series under examination? (c) Is the traffic of a gateway stationary in a short term period? (d) Is there a relationship between the number of connected devices and the traffic values? (e) Are home traffic time series sensitive to time aggregation? We use the following standard analysis techniques:

- 1) *correlation coefficients* – to measure similarity of RGWs;
- 2) *autocorrelation coefficients* – to evaluate how strong the connections between the values of a single time series are, and what their predictive power is;
- 3) *cross-correlation coefficients* – to measure how strong the connections between values of a pair of time series shifted in time are, and what their predictive power is;
- 4) *stationarity tests* (KPSS unit root test, Augmented Dickey-Fuller (ADF) test and others) – to check if time series are wide-sense stationary.

In all our experiments, when statistical tests are exploited, we use a significance level of  $\alpha = 0.05$ . For correlations tests, we use Pearson’s, Spearman’s, and Kendall’s correlations, and interpret the strength of the correlation as follows: [0.0; 0.1) → No Correlation, [0.1; 0.3) → Low Correlation, [0.3; 0.5) → Medium Correlation, [0.5; 1] → Strong Correlation. This interpretation is widely accepted [2], [6], [29], though the borders may slightly vary depending on the application domain, for example, in medicine higher borders for strong correlation are usually required [28].

The above methods revealed the following results:

(a) There are gateways, for which we can make predictions about their future behavior due to low, but statistically significant autocorrelations of their traffic time series. The example with the highest autocorrelation is shown in Figure 2(left). In this figure, the y-axis defines the value of Autocorrelation Function (ACF) that depends on the time lag between the time series values (x-axis). We note that no gateway exhibits a seasonal behavior. There is also some predictive power of one gateway given another, as some cross correlations with lags across gateways are significant. An example of a high cross-correlations is depicted in Figure 2(right). Even-though these observations suggest some predictive power, due to the significant amount of silence or

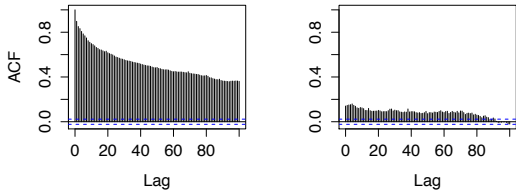


Figure 2: Autocorrelation (on the left) and cross-correlation (on the right) of gateways.

background traffic, ARIMA modeling for this time granularity cannot yield useful results, as it is not able to predict the rare bursts of the active traffic.

(b) Wireless traffic data is not stationary in the traditional sense, as all stationarity tests were rejected. This means that the data distribution characteristics change over time. For example, the covariance function of the time series is not constant in sliding window. We have also noticed that the Zipfian distribution of time series also evolves over time, meaning that the time series are also not stationary in the general sense.

(c) For all the gateways we checked, the correlation between the overall traffic time series and the number of connected devices time series was statistically significant, but low (mean = 0.37, median = 0.38, stddev = 0.21). This is an interesting result, indicating that traffic at a gateway depends more on the user behavior, rather than on the number of connected devices.

(d) For the time-aggregated time series with larger time binning, patterns become more visible as traffic peaks become more similar. At the same time, when excluding many points of low traffic, the essential information about the high traffic periods persists. Correlation and distribution heavily depend on time aggregation:

- The smaller the aggregation period is, the more different the data distribution within the week is. Almost all Kolmogorov-Smirnov tests were rejected for the smallest aggregations. For higher aggregation periods distributions become more similar.

- The smaller the aggregation period is, the lower the correlations between time series are. At the same time, for larger aggregation periods correlations either significantly grow, or disappear completely.

**Summary.** Our preliminary analysis of gateway traffic reveals that traffic time series are not stationary, neither in the general, nor in the wide sense: both the probability density function and the main time series characteristics (e.g., mean and covariance) change over time. Consequently, time series with current one minute binning are highly irregular, there are no stationary gateways, and similarity between different gateways of our deployment is very low. Hence, extracting meaningful patterns of bandwidth usage both across time and gateways requires to adapt new analytical methodology.

## 5. TRAFFIC ANALYSIS FRAMEWORK

We now define and describe the key concepts, on which we base our proposed analysis framework.

**Similarity.** The core issue when comparing traffic time series of residential gateways is to define a suitable similarity measure. As discussed earlier, absolute values of traffic volume are not helpful to understand seasonal usage of home devices within or across homes. Instead, we consider similarity in terms of correlation, which takes into account the monotonicity of traffic volume changes, rather than their ab-

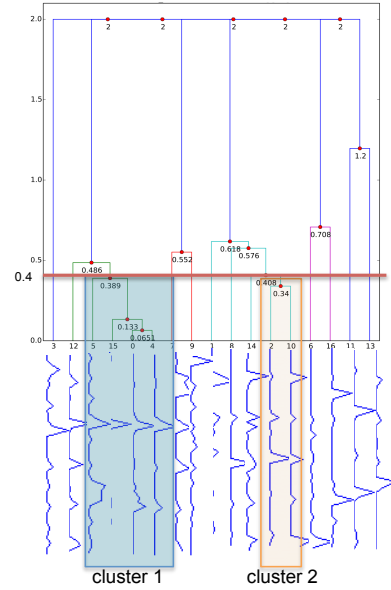


Figure 3: Hierarchical clustering of time series based on correlation similarity measure.

solute values, thus, providing invariance to scaling. We use three popular correlation coefficients as all kinds of dependencies between traffic time series described by these coefficients are important. *Linear* dependency is provided by Pearson’s correlation, and *monotonicity* and *ranking-based* dependencies are provided by Spearman’s and Kendall’s coefficients. The correlation coefficients are not directly comparable but they have the same domain and semantic interpretation of strength, allowing us to use the highest possible value. More formally, the similarity measure is defined as follows:

**DEFINITION 1. Correlation similarity measure**  $cor(X, Y)$  between two time series  $X = \{x_i\}_{i=1}^n$  and  $Y = \{y_i\}_{i=1}^n$  is a maximum correlation coefficient among statistically significant Pearson’s  $r(X, Y)$ , Spearman’s  $\rho(X, Y)$  and Kendall’s  $\tau(X, Y)$  correlation coefficients:

$$cor(X, Y) = \max[r_{p_v < \alpha}(X, Y), \rho_{p_v < \alpha}(X, Y), \tau_{p_v < \alpha}(X, Y)].$$

If none of the three correlation coefficients are statistically significant  $cor(X, Y) = 0$ .

The significance of the corresponding correlation coefficient test is defined w.r.t. the zero-hypothesis  $H_0$  is that there is no correlation, or the coefficient is equal to zero. The significance level  $\alpha$  is (as before) set to 0.05.

A correlation-based similarity measure also leads to meaningful threshold values for time-series similarity. For example, in the hierarchical clustering of traffic time series illustrated in Figure 3, the distance measure is set to  $1 - cor(\cdot, \cdot)$ . As the correlation  $\geq 0.6$  is considered to be high, the corresponding threshold on a distance measure 0.4 is used to detect similarity clusters.

Spearman’s and Kendall’s correlation coefficients are insensitive to z-normalization of the data, while the Pearson correlation coefficient is normalization dependent. As we will see in Section 6.2, our correlation-based similarity measure allows to better grasp the actual device usage compared to similarity measures based on absolute values, such as the popular Euclidean distance. Euclidean and DTW distance also do not meet the needs of similarity measurement in the work [30]. This work extracts the patterns of human behavior but in terms of time series of item popularity over

online media and also proposes a distance measure invariant to scaling. But, unlike our case, they also consider the patterns to be similar if the peaks of activities are shifted in time. In case of behavioral patterns that are valuable to ISPs it is important that the traffic is active simultaneously or within the same aggregated time periods.

**Stationarity.** Since our goal is to extract time-evolving patterns and characterize these patterns across different dimensions of interest, we define a new notion of stationarity adapted to the peculiarities of our traffic time series. Our traffic data has very strong time semantics — traffic depends highly on the day of the week and on time of usage, so we cannot expect that the data distributions during the weekend and working days are the same. Thus, we are interested in time-framed patterns (from one day to another, or from week to another) and consider regularity of behavior in terms of non-overlapping time windows. In this respect, we measure the correlation similarity of the time series with itself, comparing each window of the chosen period with each other, in order to measure the entire stationarity of a period of interest. We also check whether the traffic data distribution changes significantly from one period to another using a non-parametric comparison test for arbitrary probability distributions (i.e. Kolmogorov-Smirnov test). More formally:

**DEFINITION 2.** *A time series of a gateway is **strongly stationary** for a particular window size if:*

- *it has a correlation similarity measure  $> 0.6$  among all non-overlapping windows in consideration;*
- *the Kolmogorov-Smirnov test (that checks if the distributions of two time series is the same) is not rejected for all possible window pairs.*

The main difference between our ‘strong stationarity’ notion and the classical stationarity is that instead of using sliding windows, we use non-overlapping windows. Furthermore, apart from the similarity of the distributions, we also check the correlation similarity between the windows, which makes this a ‘strong’ notion of stationarity. Asserting that the time series of a gateway is strongly stationary, ensures that the underlying bandwidth usage is regular and can be described by a repetitive usage pattern.

**Time aggregation.** We use the notions of correlation similarity measure (Definition 1) and strong stationarity (Definition 2) in order to formulate optimization criteria for choosing the best time periods for aggregating traffic values, or the best binning of time series. More formally the problem is defined as follows:

**DEFINITION 3.** *Given a mapping function  $W$  of nonoverlapping time windows of length  $g$  defined over a set of times series  $U$ , the **best aggregation granularity**  $g_{best} \in G$  is defined as*

$$g_{best} = \arg \max_{g \in G} E[cor(x(g), y(g))],$$

where  $x(g), y(g) \in S$ , a set of time series  $S$  defined from  $U$  through  $W$ :  $S = W(U)$ ,  $x(g)$  and  $y(g)$  are aggregated traffic volume values according to the time binning  $g$ .

Mean is used as an unbiased estimate of  $E[\cdot]$ .

As we will see in Section 7.2, deciding which is the best traffic aggregation binning is crucial for extracting unknown meaningful recurring patterns of medium-term (week) and short-term (day) usage behaviors of a residential gateway. These patterns are called **motifs**.

**Dominant devices.** We also need to detect dominant devices per gateway, that is, the devices that have traffic time series very similar to the overall traffic of a gateway. We define a dominant device as follows.

**DEFINITION 4.** *Device  $d$  is  $\phi$ -dominant per a gateway if the correlation similarity between its traffic and gateway traffic is larger than a threshold  $\phi$ .*

Besides determining dominant devices of the traffic reported by the gateways of our deployment (Section 6.2), this definition will enable us to better characterize the motifs in terms of types of devices that contribute to the traffic of the motif the most (Section 7.2).

## 6. GATEWAY DEVICE ANALYSIS

In this section, we are interested in identifying the active usage traffic generated when residents actually run online applications, as well as in detecting the devices that contribute the most to the overall traffic.

### 6.1 Active Device Traffic

As we have seen in Section 4.1, the majority of the traffic volume values reported by the gateways are rather low. This background traffic is essentially attributed either to the control traffic generated by the operating systems of devices (e.g., during sleep mode or application software updates), or to low traffic generated by light applications running in the background (e.g., when a mail server checks for new emails, or when twitter updates the message list). Background traffic has its own patterns and fluctuations that influence our time-series analytics given that the majority of wireless devices in our data, such as tablets and smartphones, are frequently in the idle state and use their wireless radio rarely in order to increase the battery life.

In order to extract recurring patterns from active usage traffic generated when residents actually run online applications, we need to exclude the background traffic. Background traffic can be separated from active traffic by setting a threshold  $\tau$  on the number of bytes in traffic time series. To obtain active traffic time series, all the values which are lower than  $\tau$  are set to zero. Deciding on an appropriate threshold  $\tau$  for background traffic is far from trivial, given the lack of a ground truth (both on the operating systems for particular sleep policies and the applications running on devices). Instead, we can exploit a general statistical technique based on the probability distribution of the traffic time series, as the boxplots described in Section 4.1. Given that the interval in which most of data values of time series belong falls between the whiskers of the plot, we use the upper whisker of a boxplot in order to define  $\tau$ . This is supported by the fact that background traffic values are the most frequent in our data, while active traffic is sparse.

As this threshold is device specific, we estimate it for each device, per outgoing and incoming traffic separately. We study the background traffic for four weeks of data. For this period, we observed the traffic for 934 devices connected to user gateways. According to the histograms for outgoing and incoming traffic (refer to Figure 4), the background threshold for most of the devices is below 5000 bytes per minute (i.e., less than 1 Kbps), while for almost all the devices  $\tau$  is lower than 40,000 bytes for both outgoing and incoming traffic, which corresponds to a rate of 5.3 Kbps. There are

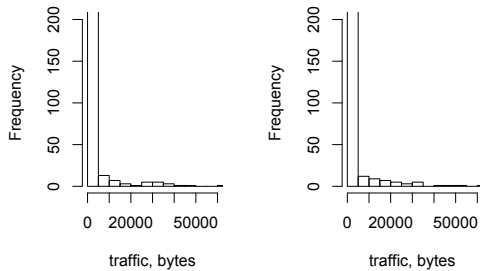


Figure 4: Distribution of  $\tau$  of outgoing (on the left) and incoming (on the right) traffic.

24 devices with  $\tau > 40000$  bytes for incoming traffic, and 15 devices with  $\tau > 40000$  bytes for outgoing traffic.

We checked if there is a dependency on the type of device and the distribution of traffic it produces. We grouped the devices by  $\tau$  as follows. 'Small' group corresponds to  $\tau \leq 5000$ , 'medium' to  $\tau \in (5000, 40000]$ , and 'large' to  $\tau > 40000$ . We use the types of devices defined in Section 3. In the small and medium groups portable devices dominate, while in the last group fixed devices are the most popular as on PCs and laptops much more applications can run simultaneously in non-active mode. Thus, background traffic can be a significant feature for device type classification.

In summary, to exclude background traffic from consideration, we use a threshold per device of  $\tau_{back} = \min(\tau, 5000)$ . This value is an upper border of the background traffic for the majority of the devices, as illustrated in Figure 4. Our threshold of 5000 bytes per minute for background traffic is also consistent with the previous works [25], [23], which set it to 1 kbps, thus making our threshold more tight.

As we will see in Section 7, background traffic removal reveals more regularity in traffic time series. The ability to automatically detect the background traffic of a device will also help ISPs to improve the energy saving policies *without* using data aggregation from multiple homes as has been proposed in the literature [8].

## 6.2 Dominant Devices

A device is considered to be dominant if it characterizes the general behavior of a gateway over time with respect to the overall traffic. As before, we only consider wireless traffic and wireless devices.

Definitions 1 and 4 are used in order to detect  $\phi$ -dominant devices. As we are interested in high time series similarity we have chosen  $\phi$  threshold to be 0.6, as before only statistically significant correlations were reported. We perform the search of dominant devices for all the gateways that have at least one observation per week for each week of consideration, we have observed 153 such gateways. The data contains the time series of all devices that were connected to a gateway after March 17, 2014 together with its overall traffic. If there are several dominant devices detected, we rank them in descending order of their correlation similarity.

According to the results, 7 gateways had 3 dominant devices, 43 gateways had 2 dominant devices, 99 gateways had 1 dominant device, and only 4 gateways did not have any dominant device. In most of the cases, there is at least one dominant device per gateway, meaning that the bandwidth consumption of the gateway is determined by the usage of the device. There might be several dominant devices, which can indicate that the number of residents regularly using network is higher than one. There are at most 3 dominant devices per gateway, we ranked them in the descending order of correlation similarity value, hence, first dominant devices

has traffic time series that is the most similar to the overall traffic time series of a gateway.

We also checked what type of devices are dominant per gateway. Overall, among dominant devices we detected 74 fixed, 67 portable, 53 unlabeled, 9 network equipment devices, and 3 game\_consoles. The distribution of the different device types, depending on the ranking of dominance, is shown in Figure 5. The plot shows that there are many gateways, for which the dominant devices for all the ranks are fixed devices. This is attributed to the fact that fixed devices produce in general more traffic and are usually connected for longer periods to the gateway. Still among dominant devices there is a significant number of portable devices, which are increasingly being used nowadays.

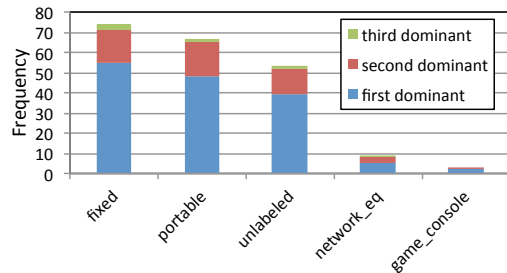


Figure 5: Distribution of types of dominant devices given their ranking.

The knowledge of dominant devices over long periods (one month in our case) is of great importance to ISPs, as it can provide a high level profiling of gateways.

**Using Other Distance Metrics.** For the sake of completeness we have compared the dominant devices obtained using our correlation-based similarity measure with those obtained when using the Euclidean distance or simply the absolute traffic volume used in work [23]. For the Euclidean distance computation, we consider the time series of a gateway  $X = \{x_i\}_{i=1}^n$  and time series of a device  $Y = \{y_i\}_{i=1}^n$ , where  $n$  is the number of observations for four weeks of data. The Euclidean distance is computed using the formula:  $dist_{Eucl} = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$ . Alternatively, the devices which produce the highest volume of traffic are considered to be dominant traffic-wise.

As there are no clear thresholds for Euclidean and traffic-based dominances, we compare the results of our correlation-based dominant devices, where we used meaningful threshold, with the devices that are ranked using the Euclidean distance (ascending order) and the traffic volume (descending order). Using the correlation dominance we have detected 206 dominant devices. For some gateways there can be two or three dominant devices. For each gateway we detect its correlation-based dominant devices and obtain a ranking of dominant devices based on the other two measures. Then if the devices in the ranking are the same, meaning that the first device in one ranking is also the first in the second ranking and so on, we say that the devices are detected equally using the two measures.

Among the 206 dominant devices, 182 (88%) are ranked the same as Euclidean-based dominant devices, and 151 (73%) are ranked the same as traffic-based dominant devices. Nevertheless, there are many cases, where dominant devices have lower overall traffic (around 15%), even though they closely follow the traffic time series of the gateway, with an exception of a few bursts (3 or 4). Our similarity measure is able to detect these devices, which cannot be detected

using the Euclidean and traffic-based distances.

We have also tried more strict  $\phi$  threshold for dominance,  $\phi = 0.8$ . Even with very tight constraint on dominance, there is still large amount of gateways (67%) that have at least one dominant device and the ratio of fixed devices among the dominant devices is even larger.

**Dominant Devices and Number of Residents.** Having the results of a recent user survey over a subset of 49 gateways in our deployment, which contained information on the number of users per gateway, we checked if the number of dominant devices is correlated with the number of residents. The result of this analysis showed that there is no evidence of significant correlation. This may be due to the fact that different users are active during different periods of time, and in case of multiple users (and therefore multiple devices) the number of overall dominant devices is lower.

On the other hand, in the gateways with one user, there is always one dominant device detected. In the case of two users (9 gateways), 2 dominant devices are detected in 5 gateways (56%) and 1 dominant device is detected in 4 gateways (44%) and there are no three dominant devices detected. We have calculated the correlation coefficient between the number of dominant devices and the number of users only for gateways with 1 and 2 users, and we obtained a statistically significant correlation value = 0.53. In the case where the number of users  $> 3$ , the effect of multiple devices, discussed in the previous paragraph, is present again, and only 1 or 2 dominant devices are detected.

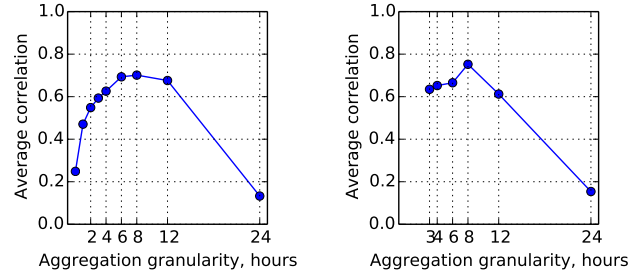
**Summary.** The most interesting findings of our analysis of dominant devices per home are:

- 1) Each gateway has at least one dominant device.
- 2) The majority of dominant devices are fixed devices.
- 3) Correlation based dominance is beneficial because it identifies dominant devices that are missed by Euclidean, or traffic-based distance notions.
- 4) The number of dominant devices provides a lower bound for the number of residents per gateway.

## 7. TIME AGGREGATIONS AND MOTIFS

In this section, we are interested in investigating which time aggregations of gateway traffic better reveal regular user behavior in houses. For example, whether the produced traffic is more significant in the morning rather than in the evening, during the weekdays rather than the weekends, etc. By checking various meaningful time aggregations ranging from 30 minutes to 12 hours, we have experimentally verified that the larger the aggregations are, the more correlations are observed within or across gateways. This is due to the fact that the periods of non-active traffic become smaller, while traffic peaks become more similar. If at the same time we exclude the points of background traffic, the actual usage patterns of home gateways become more visible.

Since there is no golden standard as to which aggregation should be used, the choice is usually application driven. In our work we rely on the notion of strong stationarity (see Definition 2) capturing repetitive usage patterns of gateways to systematically determine the right aggregation level. If a time series is strongly stationary, then the patterns found for this time series are stable and we can generalize the results of the analysis over this time series. This is not always truth if standard binning such as morning, working hours, late afternoon, evening, night is used, because usually the borders of this binning, number of bins and their length is



(a) Average correlation for stationary gateways starting at midnight

(b) Average correlation for stationary gateways starting at 2am

Figure 6: Aggregation curves for weekly patterns.

not experimentally verified but just arbitrary set.

We consider two kinds of windows: daily-windows starting from midnight to reveal short-term patterns of usage behavior and weekly windows starting from Mondays to reveal medium-term patterns. For four weeks of data and a weekly period with 3 hours aggregation 7% of gateways appeared as stationary. Thus, though there are strongly stationary gateways, still most of them change their behavior from week to week. Finally after removing the background traffic (see Section 6.1) 11% of gateways were detected as stationary. In the next section we choose the best aggregation period according to maximization of time series correlation.

### 7.1 Best Aggregation Period

In this section, we discuss how to choose the best aggregation period, according to the maximization of the time series correlations across time. The problem is formally stated in Definition 3. Background traffic is removed from all time series, as described in Section 6.1.

#### 7.1.1 Weekly Patterns

First, we consider the best aggregation for medium-term patterns of weekly behavior. As traffic depends heavily on the time of the day from day to day we considered all time aggregations starting at midnight that are factors of 24 hours, namely 1, 2, 3, 4, 6, 8, 12, 24 hours and additionally we considered initial time series aggregation, which is 1 minute. We also try aggregation granularities which are larger than 2 hours, starting from 2am and 3am. For the analysis, we consider all the user gateways that have at least one traffic observation every week during the 4 weeks of interest. The total number of such gateways is 153.

An aggregation period is considered to be the best if it reveals the highest correlation of traffic time series of a gateway values from one week to another.

In order to compare aggregations, we calculate the average correlation among all the week-week pairs separately for each gateway, and for strongly stationary gateways (Definition 2). The plots of the average correlation values are shown in Figure 6.

The maximum points for strongly stationary gateways are reached at granularity periods of 6, 8 and 12 hours for aggregations from midnight (Figure 6a) and 8 hours for aggregations starting from 2am (Figure 6b). When considering all the gateways, large correlation points are reached for 3, 4 and 6 hours of aggregations starting at midnight, while 8 hour aggregations starting at 2am is still a maximum point. Since the 8 hours aggregation period starting at 2am is an absolute winner for weekly patterns, we use this aggregation for our further analysis. Note that this aggre-

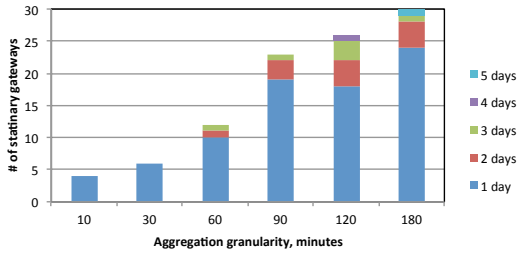


Figure 7: Stairway gateways per aggregation window.

gation has a meaningful semantic interpretation: each day is divided into 3 periods, namely, "morning" between 2am-10am, "working hours" between 10am-6pm, and "evening" between 6pm-2am. The traffic behavior of gateways for this aggregation is also of interest to ISPs.

### 7.1.2 Daily Patterns

As the initial observations are received at a rate equal to 1 minute, and for daily patterns we need to have a reasonable amount of points, we try the following aggregation periods: 1, 5, 10, 30, 60, 90, 120 and 180 minutes. We do not consider aggregations larger than 180 minutes as it is not desirable to have less than 8 data points per pattern. All the binning values are factors of 1440 minutes, which constitute a day.

As before we define the aggregation period to be the best if it reveals the highest correlation of traffic time series of a gateway values for daily patterns. Unlike weekly patterns, we do not require every day to be similar to each other, but we expect that Mondays should be highly correlated with Mondays, Tuesdays with Tuesdays, etc. For the analysis, we consider all the user gateways that have at least 1 traffic observation every day during the 4 weeks of interest. Their number is 100.

For daily patterns we also studied strongly stationary gateways, where the behavior of the same days of the week is stationary (in the sense of Definition 2). Note that for stationarity we require not only highly correlated observations among the corresponding days of the week (e.g., all Mondays should be correlated with each other), but we additionally require that the probability distributions of all instances of that weekday should be similar. The number of stationary gateways per aggregation granularity is shown in Figure 7. We also decompose the total number of stationary gateways to the number of gateways which have one stationary day of the week, two stationary days, and so on.

Figure 7 shows that the number of stationary gateways grows with the aggregation granularity. Additionally, more days are stationary within the same gateways if the granularity is larger.

In order to compare the aggregation granularities we calculate the average correlation among all the pairs of the same day of the week separately for all gateways and for gateways that appeared to be strongly stationary. The plots of average correlation are shown in Figure 8.

The results show that small aggregation periods correspond to low regularity in the data; moreover, there are no gateways with at least 1 stationary day of week for aggregation granularities of 1 and 5 minutes. The correlation value for all the gateways grows significantly up to 1 hour aggregation, then it becomes stable up to the level of 180 minutes or 3 hours. At the same time, the average correlation of the stationary gateways keeps growing with the larger aggregation granularity, and the highest value is reached for the 3 hours aggregation granularity, which is the aggregation pe-

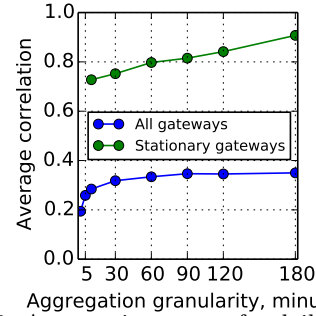


Figure 8: Aggregation curves for daily patterns.

riod we use for daily patterns in the rest of this study.

## 7.2 Motif Discovery

As mentioned in Section 5, for motif extraction we only consider patterns that correspond to medium-term (week) and short-term (day) usage behaviors across time and across gateways. Patterns within a particular gateway only or of a longer period can also be identified following the proposed methodology.

The following definition of a motif is used in our work:

**DEFINITION 5.** Given a set of times series  $U$ , a mapping function  $W$  of non-overlapping windows, which extracts periods of interest  $S$  from  $U$  according to a window length and its starting point synchronizes with the corresponding timestamp of  $U$  (beginning of a day or a week),  $S = W(U)$ , **motif** is a set  $M \subseteq S$ ,  $M = (m_i)_{i=1}^k$ , where  $k$  is a support of a motif.  $M$  has the following properties:

1. **individual similarity:**  $\forall i \in \{1, 2, \dots, k\}, \exists j \in \{1, 2, \dots, k\} : cor(m_i, m_j) \geq \phi$ ,
2. **group similarity:**  $\forall i, j \in \{1, 2, \dots, k\}, i \neq j : cor(m_i, m_j) \geq 3/4\phi$ ,

Thus, when a new subsequence is included in a motif it is very similar to at least one existing subsequence in  $S$  and it is reasonably similar to all the rest  $s_i, i = 1, \dots, k$ . In our case  $\phi = 0.8$ .

In other words, two time series are considered to constitute a motif if the correlation distance between them is very high. The threshold we have chosen is 0.8. Several motifs can be combined if all the time series that comprise the motifs have high correlations with each other. In this case, the correlations should be  $\geq 0.6$ .

Motif extraction enables us to enrich traditional analysis of the aggregated traffic reported by a gateway. As a matter of fact, we can detect detailed behaviors within the same house that can be attributed to different residents (e.g., adults or children), or to different habits (i.e., daily or weekly patterns). Identification of diverse behaviors inside a single house goes beyond the current state of the art. Furthermore, to provide additional information about the obtained motifs, we analyze them across the following dimensions:

1. How many dominant devices per gateway contributed to the motif? In this case we consider dominance for the corresponding time period of time series that formed the motif.
2. How do the dominant devices per motif and gateway relate to the overall dominant devices of a gateway detected for a period longer than 4 weeks?
3. What is the distribution of the dominant devices per motif? We consider portable, fixed devices, network equipment and others as discussed in Section 6.1.
4. Are there daily motifs, which are more common among weekends than working days and vice versa?

## 5. What gateways contribute the most to the motifs?

For the analysis we concentrate on significant motifs with high support values, so called motifs of interest.

### 7.2.1 Weekly Motifs

In order to find weekly motifs we use 8 hour aggregation period starting at 2am, as it is the best time aggregation according to the experimental results in Section 7.1.1. The motif search was done on user gateways that have at least one observation for each week, out of the six weeks starting from March 17th. The number of such gateways is 147.

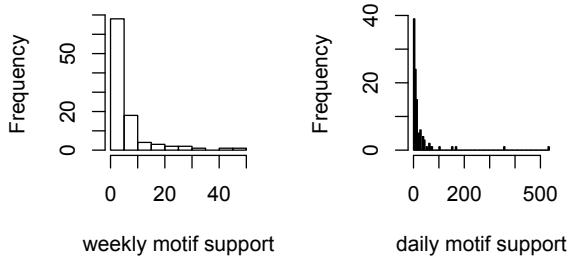


Figure 9: Distributions of support values for weekly (left) and daily (right) motifs.

As a result, 101 motifs are discovered from 882 (147\*6) weeks of observations. Out of those, 14 motifs have support  $\geq 10$ . The distribution of the support values is shown in Figure 9. For weekly motifs, the participation of gateways in motifs is rather low, but at least one week time series per gateway contributed to the motif construction. The top gateways among the gateways with the highest contribution provided at least 6 time series for weekly motifs while on average number of the distinct motifs per gateway is 2.76. The distribution of the number of distinct weekly motifs per gateway can be seen on Figure 10.

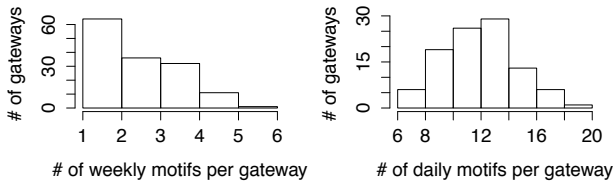


Figure 10: Distributions of the number of motifs, where a gateway participated in.

Some of the motifs of interest are shown in Figure 11.

In general, all the detected weekly motifs correspond either to the patterns of everyday evening usage, or to the patterns where certain days are the most influential.

We further elaborate on weekly motifs of interest, as shown in Figure 11. We label the motif in Figure 11a - motif1, Figure 11b - motif2, and Figure 11c - motif3. The distribution of the number of dominant devices for these motifs is shown in Figure 12a. We observe that these motifs have one or two dominant devices, and most of them correspond to the overall dominant devices of a gateway according to Figure 12b. Nevertheless, there are some devices, which are dominant per motif time slot, but not dominant for a gateway overall. The number of these devices is at least 2.5 times smaller than the number of common dominant devices.

We notice that motif1 and motif3 are mainly observed for portables (Figure 13), while motif2 is observed for fixed devices. This can be attributed to portable devices being used in the evenings. But, more regular users, like the ones contributing to motif2, tend to use fixed devices.

### 7.2.2 Daily Motifs

We have extracted daily motifs for the aggregation period that is considered to be the best in terms of the highest number of correlated patterns, as discussed in Section 7.1.2. This is the three hours aggregation which gives 8 points for each daily time series.

The daily motifs were extracted from time series of 100 user gateways (out of 196 gateways), which have at least one observation per day in raw time series for the observation period of four weeks. In total 112 motifs were extracted, with 48 motifs having support larger than 10. The distribution of the support values is shown in Figure 9. The most popular (with the highest support) daily motifs are connected to various evening usages, while there are still motifs with daily behaviors and mixed morning and evening behaviors. Surprisingly, for daily motifs each gateway contributed with at least 16 time series. The top gateways (among the gateways with the highest contribution) provided at least 28 time series for daily motifs. At the same time the average number of distinct motifs per gateway is 12.5.

The distribution of the number of distinct daily motifs per gateway is shown in Figure 10. The support of the daily motifs is more repetitive between the same homes than in the case of the weekly motifs. This can be attributed to the fact that more days per gateway were considered. 28 data windows are used for daily motifs, while 6 data windows are used for weekly motifs.

**Analysis.** We analyze in detail 4 representative daily motifs shown in Figure 14. The distribution of the number dominant devices is illustrated by Figure 15a.

We observe that motifs usually have one or two dominant devices. Unlike the weekly motifs, many of them do not correspond to the overall gateway's dominant devices (Figure 15b). However, the majority still coincides with the overall dominant. This higher ratio of new dominant devices is attributed to the small time intervals (i.e., a single day), while overall dominance is determined for 4 weeks of data.

The distribution of dominant device types is shown in Figure 16a. Motifs A, B and C which correspond to high usage behavior in the morning or/and in the evening are mainly created by portable devices, while motif D, with all day active usage (and similar daily motifs we detected), is more generated by fixed devices. In general, daily patterns have higher percentage of portable devices as dominant devices, especially in the cases of not continuous usage behaviors.

Most of the motifs correspond to both working days and weekends, but all day usage (motifD) contains more working days (Figure 16b). The relative support of working-day motifs is larger than that of weekends, as working days are more frequent in the training set.

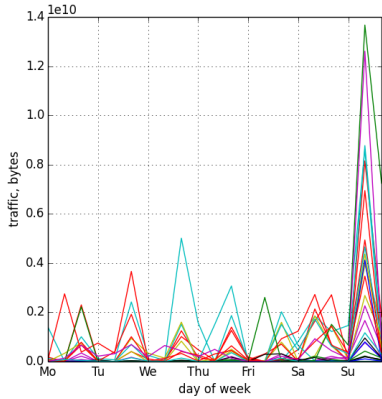
#### Summary.

1) Portable devices are sources of short-term morning or evening activities, while fixed devices generate day and night long-term patterns.

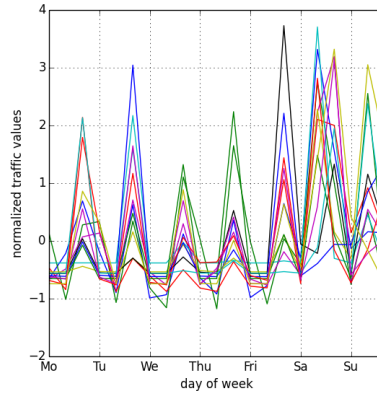
2) Gateways participate in several motifs. This supports our hypothesis that a gateway involves multiple distinct behavioral patterns. Our approach is able to reveal those, thus leading to an accurate characterization of the gateways, which in turn provides valuable insights to ISPs.

We observe that the discovered motifs reveal fine-grained regular behaviors that can be exploited in order to better manage residential networks. For example, our analysis identifies groups of users, irrespective of their location and

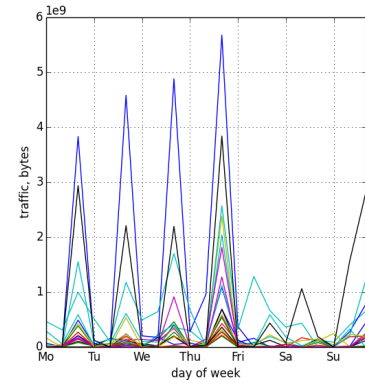




(a) Heavy Weekend Users: Support = 26 time series (23 % occur during different weeks in the same gateways).

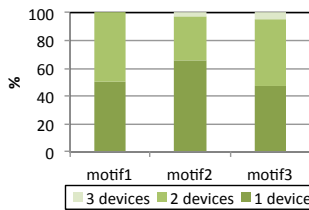


(b) Everyday Users: Support = 13 time series (15 % occur during different weeks in the same gateways).

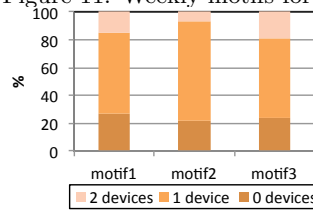


(c) Workdays Users: Support = 21 time series (29 % occur during different weeks in the same gateways).

Figure 11: Weekly motifs for 8 hours aggregation granularity.



(a) Distribution of the number of dominant devices.



(b) Distribution of intersections with overall dominant.

Figure 12: Dominant devices for weekly motifs.

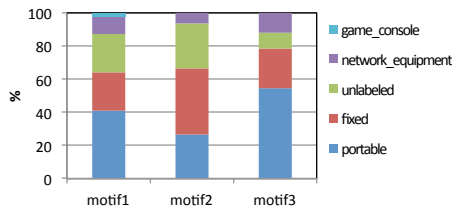


Figure 13: Distribution of types of dominant devices for weekly motifs.

demographics, that share similar time periods of low activity (in different parts of the day, or night). These can be used by ISPs and CSPs to schedule maintenance processes in a way that minimally interferes with the user activities.

## 8. CONCLUSION

In this work, we analyze the wireless traffic time series of 196 home gateways. We describe a similarity measure suitable for capturing the characteristics of traffic *evolution* within and across gateways. We propose a notion of stationarity that, in addition to the similarity of data distributions, also imposes a correlation similarity across non-overlapping time windows. This work is a first step towards understanding fine-grained regularities on residential traffic consumption. ISPs and CSPs could leverage such analytics to enable remote maintenance services such as: (i) troubleshooting and firmware/software updates of RGWs, and (ii) hotspot resource management and collaborative networks, which require *fine time-scale identification* of gateways' and home devices' active and idle times. Existing methods rely heavily on wireless traffic stationarity for such predictions, which do not hold in home networks (cf. Section 2). Our analytics framework uncovers the best aggregation of home traffic values, in order to identify motifs and to detect gateways with similar/different traffic patterns. We are currently working

towards integrating our time series correlation and motif extraction, in a streaming big data analytics platform, such as Apache Storm or Amazon Kinesis.

## Acknowledgments

We would like to thank Gevorg Poghosyan, Augustin Soule, Pascal Le Guyadec, Henrik Lundgren, Jaideep Chandrashekar and Christophe Diot for their precious help in making sense of wireless home traffic data. This work was partially funded by the European ICT FP7 User Centric Networking project (grant no. 611001).

## References

- [1] M. Chetty, R. Banks, R. Harper, T. Regan, A. Sellen, C. Gkantsidis, T. Karagiannis, and P. Key. Who's hogging the bandwidth?: The consequences of revealing the invisible in the home. In *CHI 2010*. Association for Computing Machinery, Inc., April 2010.
- [2] G. W. Corder and D. I. Foreman. *Nonparametric statistics for non-statisticians: a step-by-step approach*. 2009.
- [3] L. DiCioccio, R. Teixeira, and C. Rosenberg. Measuring home networks with homenet profiler. In *Passive and Active Measurement*, 2013.
- [4] N. Eagle and A. S. Pentland. Eigenbehaviors: Identifying structure in routine. *Behav Ecol Sociobiol*, 2009.
- [5] J.-P. Eckmann, E. Moses, and D. Sergi. Entropy of dialogues creates coherent structures in e-mail traffic. *National Academy of Sciences*, 101(40):14333–14337, 2004.
- [6] S. B. Eom, M. A. Ketcherside, H.-H. Lee, M. L. Rodgers, and D. Starrett. The determinants of web-based instructional systems' outcome and satisfaction: An empirical investigation. *Instructional techn.: Cognitive asp. of online programs*, 2004.
- [7] P. Ferreira and P. Azevedo. Evaluating deterministic motif significance measures in protein databases. *ALMOB*, 2(1), 2007.
- [8] E. Goma, M. Canini, A. Lopez Toledo, N. Laoutaris, D. Kostić, P. Rodriguez, R. Stanojević, and P. Yagie Valentin. Insomnia in the access: Or how to curb access network related energy consumption. *SIGCOMM Comput. Commun. Rev.*, 41(4):338–349, 2011.
- [9] B. Gonçalves and J. J. Ramasco. Human dynamics revealed through web analytics. *Phys. Rev. E*, 78:026123, Aug 2008.

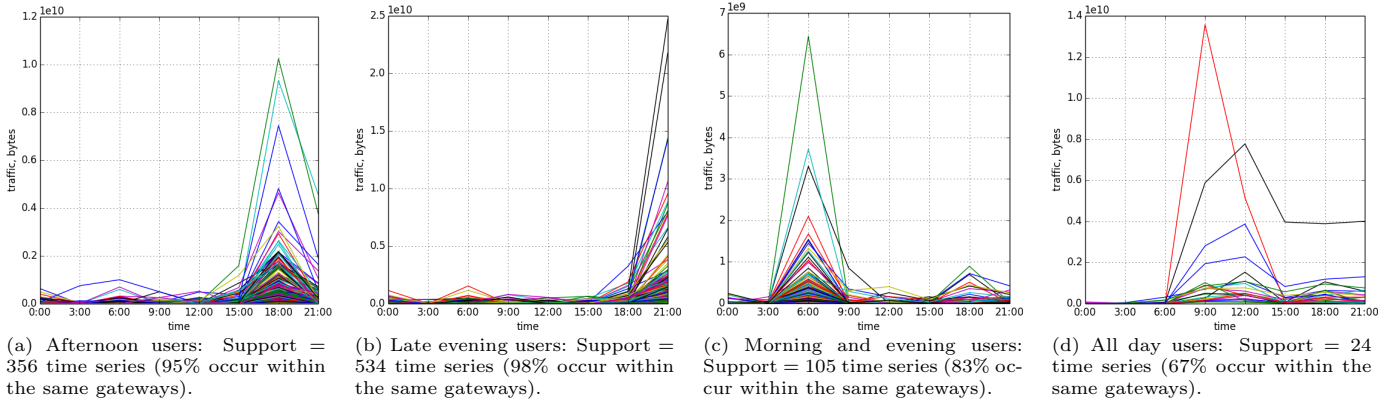


Figure 14: Daily motifs for three hours aggregation granularity.

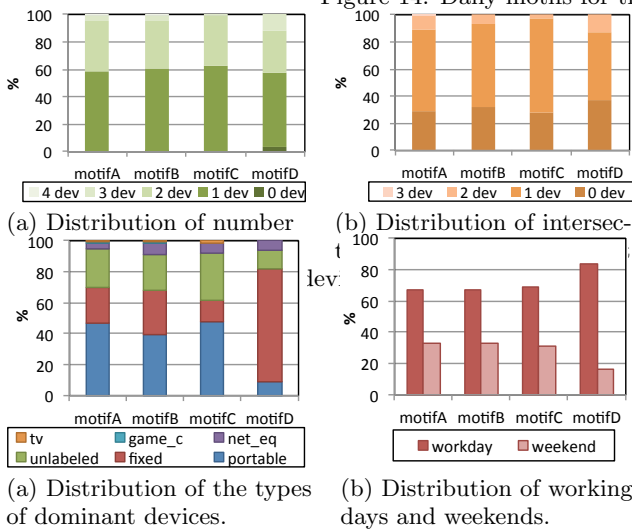


Figure 16: Dominant devices and days. Daily motifs.

[10] M. C. Gonzalez, C. A. Hidalgo, and A.-L. Barabasi. Understanding individual human mobility patterns. *Nature*, 2008.

[11] S. Grover, M. S. Park, S. Sundaresan, S. Burnett, H. Kim, B. Ravi, and N. Feamster. Peeking behind the nat: an empirical study of home networks. In *IMC Conference*, 2013.

[12] A. P. Jardosh, K. Papagiannaki, E. M. Belding, K. C. Almeroth, G. Iannaccone, and B. Vinnakota. Green w lans: On-demand wlan infrastructures. *Mob. Netw. Appl.*, 14(6):798–814, 2009.

[13] Y. Jennifer. The data-driven approach to network management: Innovation delivered, 2010.

[14] H.-H. Jo, M. Karsai, J. Kertész, and K. Kaski. Circadian pattern and burstiness in mobile phone communication. *New Journal of Physics*, 14(1):013055, 2012.

[15] M. Karsai, M. Kivelä, R. K. Pan, K. Kaski, J. Kertész, A.-L. Barabási, and J. Saramäki. Small but slow world: How network topology and burstiness slow down spreading. *Phys. Rev. E*, 83:025102, Feb 2011.

[16] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong. Mobile data offloading: How much can wifi deliver? *ACM CoNEXT*, 2010.

[17] Y. Li, J. Lin, and T. Oates. Visualizing variable-length time series motifs. In *SDM*, 2012.

[18] J. Lin, E. J. Keogh, and S. Lonardi. Visualizing and discovering non-trivial patterns in large time series databases. *IVI*, 4(2), 2005.

[19] J. Lin, E. J. Keogh, L. Wei, and S. Lonardi. Experiencing sax: a novel symbolic representation of time series. *Data Min Knowl Discov*, 15(2), 2007.

[20] S. Makridakis and M. Hibon. Arma models and the box-jenkins methodology. *Journal of Forecasting*, 16(3), 1997.

[21] J. Martin and N. Feamster. User-driven dynamic traffic prioritization for home networks. In *ACM SIGCOMM Workshop on Measurements Up the Stack*, W-MUST '12, 2012.

[22] A. Patro, S. Govindan, and S. Banerjee. Observing home wireless experience through wifi aps. In *ACM MobiCom '13*.

[23] I. Pefkianakis, H. Lundgren, A. Soule, J. Chandrashekar, P. Le Guyadec, C. Diot, M. May, K. Van Doorselaer, and K. Van Oost. Characterizing home wireless performance: The gateway view. In *accepted for IEEE INFOCOM 2015*.

[24] C. Peng, S.-B. Lee, S. Lu, H. Luo, and H. Li. Traffic-driven power saving in operational 3g cellular networks. In *MOBICOM*, pages 121–132, 2011.

[25] G. Poghosyan. Device analytics in home networks. Master's thesis, EPFL, August, 2014.

[26] E. S. Poole, W. K. Edwards, and L. Jarvis. The home network as a socio-technical system: Understanding the challenges of remote home network problem diagnosis. *Comput. Supported Coop. Work*, 18(2-3):277–299, June 2009.

[27] C. Rossi, C. Borgiattino, C. Casetti, and C. F. Chiasserini. Energy-efficient wi-fi gateways for federated residential networks. In *IEEE WoWMoM'13*.

[28] R. Taylor. Interpretation of the correlation coefficient: a basic review. *Journal of diagnostic medical sonography*, 6(1), 1990.

[29] L. Waluyan, S. Sasipan, S. Noguera, and T. Asai. Analysis of potential problems in people management concerning information security in cross-cultural environment -in the case of malaysia-. In *HAISA*, 2009.

[30] J. Yang and J. Leskovec. Patterns of temporal variation in online media. In *WSDM*, pages 177–186, 2011.

[31] C. Zhang, Y. He, and Y. Ji. Temporal pattern of user behavior in micro-blog. *Journal of Software*, 8(7), 2013.

# Parallel Duplicate Detection in Adverse Drug Reaction Databases with Spark

Chen Wang  
CSIRO  
Sydney, Australia  
chen.wang@csiro.au

Sarvnaz Karimi  
CSIRO  
Sydney, Australia  
sarvnaz.karimi@csiro.au

## ABSTRACT

The World Health Organization (WHO) and drug regulators in many countries maintain databases for adverse drug reaction reports. Data duplication is a significant problem in such databases as reports often come from a variety of sources. Most duplicate detection techniques either have limitations on handling large amount of data or lack effective means to deal with data with imbalanced label distribution. In this paper, we propose a scalable duplicate detection method built on top of Spark to address these problems. Our method uses the  $k$ NN ( $k$  nearest neighbors) classifier to identify labelled report pairs that are most useful for classifying new report pairs. To deal with the high computational cost of  $k$ NN, we partition the labelled data into clusters for parallel computing. We give a method to minimize the cross-cluster  $k$ NN search. Our experimental results show that the proposed method is able to produce robust duplicate detection results and scalable performance.

## 1. INTRODUCTION

Adverse drug reactions, or ADRs, impose significant hazards to public health. They are one of the leading causes of hospitalization, disabilities, and death around the world. ADRs incur significant costs to health-care systems [10, 19]. Post-marketing drug safety surveillance plays an increasingly important role in ADR detection in comparison with pre-marketing drug clinical trials as clinical trials have limitations on the number of patients involved and the diversity of patient groups. Post-marketing drug safety surveillance mainly uses *Spontaneous Reporting Systems (SRS)* to detect signals of potential ADRs. These signals are then further assessed by experts to establish a causal relationship between a drug and an ADR. The World Health Organization (WHO) and drug regulators in many countries, such as the FDA in the US and the TGA in Australia maintain databases for adverse drug reaction reports. ADR reports are submitted from a variety of sources including general practitioners, pharmacists, hospitals, and consumers etc. The ADR report

database is the major part of the reporting system. Many drug safety assessment methods detect potential ADR signals through the comparison of the reported ADR ratio of a specific drug and that of other drugs in the database. Disproportionality often indicates a potential ADR signal [6, 7]. As these methods are sensitive to the number of ADR reports, data quality in these databases is essential to the performance of ADR detection. One significant problem faced by such a database is *report duplication*. Duplicates often result from two sources. First, reports from different data sources have overlaps as the same ADR may be reported by different organizations through different channels. Second, the follow-up reports of the same ADR are wrongly put as separate records in the databases. Duplicates may distort the report ratio of an ADR and affect the performance of these methods significantly. Nkanza and Walop [17] reported a 5% duplication rate in vaccine adverse event data, providing an indication of the spread of the problem.

Duplicated reports in an ADR database are often not exactly the same in each field. Table 1 shows two examples. In the first example, *report A* and *report B* are duplicate, but they differ in the *reaction outcome description* field and *report description* field. In the second example, *report C* and *report D* are duplicate, but they differ in *patient age*, *ADR name* and *report description* field. The different values in the *patient age* field are likely to be an error introduced when entering a handwritten report.

A database record consists of multiple fields. Duplicate detection techniques therefore have two levels: *field matching* and *record matching*. Field matching mainly concerns comparing numerical, categorical and string values in each field. Record matching concerns whether two or more feature vectors formed by common fields belonging to different records are duplicate.

There are many existing works on duplicate detection in relational databases. Early works are referred as *record linkage* with a focus on linking together two or more separately recorded pieces of information concerning an individual case [16]. Large amount of work deals with the comparison of fields that can identify a particular record, such as name, address, and age. The values of these fields are normally short strings. Many field matching techniques concern approximate comparison of strings based on various similarity metrics. Commonly used string similarity metrics include *edit distance* [13], *Hamming distance* [8], *cosine distance* and *Jaccard coefficient* [3] etc. Field matching alone is not able to detect many duplicates, e.g., two string values in the surname field can be totally different in the case that a woman

(a)

Field Name	Report A	Report B
patient age	46	46
patient sex	M	M
patient state	-	-
onset date	-	-
reaction outcome description	Unknown	Recovered
drug name	Atorvastatin	Atorvastatin
ADR name	Rhabdomyolysis	Rhabdomyolysis
report description	Reference number xxx is a literature report received on 02-Oct-2013 pertaining to a 46 year-old male patient who experienced rhabdomyolysis while on atorvastatin for the treatment of unknown indication.	The 46-year-old male subject started treatment with atorvastatin calcium 80 mg, start date and duration of therapy unknown. In 2009,the subject presented with myalgia shoulder and hips for 2-3 weeks, minimal weakness and was diagnosed with rhabdomyolysi

(b)

Field Name	Report C	Report D
patient age	84	34
patient sex	F	F
patient state	Not Known	Not Known
onset date	30/04/2013 00:00:00	30/04/2013 00:00:00
reaction outcome description	Unknown	Recovered
drug name	Influenza Vaccine,Dtpa Vaccine	Influenza Vaccine,Dtpa Vaccine
ADR name	Vomiting,Pyrexia,Cough,Headache	Cough,Headache,Choking sensation,Chills,Vomiting
report description	On 30 April 2013, in the evening, within hours of vaccination with Boostrix, the subject experienced uncontrollable cough and felt like she was choking On the same night, the subject experienced headache.On the 01-05-2013 at 3am, the subjet experienced.	In the afternoon of 30-Apr-2013, the patient experienced uncontrollable cough for 2 hours, then started choking and had to call an ambulance. She required oxygen before she felt better and so didn't go to hospital. She then reported a headache, cold shive

Table 1: Sample duplicated reports.

changes her maiden name to her husband's surname, but they belong to the same record.

Record matching considers each field as an element of a feature vector of a record. It combines the differences among common fields of two or more records to determine whether they are duplicate. The ways of combining field similarities of records differentiates record matching techniques. Some techniques calculate the probability of difference of values in each common field, converts these probabilities to log values and add them together [16, 11, 12]. Some techniques use supervised decision tree induction and unsupervised clustering to determine if a record is likely to match a set of other data records [5]. Active learning is also used for record matching to reduce the amount of training data [20].

However, the effectiveness of existing duplicate detection techniques on ADR databases is not well investigated mainly due to the slow progress of the industry's adopting information technologies. We collaborate with the Therapeutic Goods Administration (TGA) in Australia and use the ADR reports they collected during 6 month period to carry out this work. In this paper, we study the duplicate detection problem in ADR databases with a focus on detection performance and system scalability that is important for fast growing data in this area. Our contributions are as below:

1. To identify duplicates in ADR reports, pairwise distances between these reports often need to be calculated. The distribution of duplicate pairs and non-duplicate pairs is highly imbalance. This poses a significant challenge when selecting report pairs from a large dataset to train a classifier for exploiting useful information. The classification results are highly influenced by the overwhelming majority of non-duplicate report pairs. There is no generally applicable classification method to address this issue. We develop a

$k$ NN based method to address this problem for ADR reports.  $k$ NN is helpful for the classifier to learn from individual reports and makes the classification results easy to understand. It also offers flexibility through assigning weights to the information carried in the neighbors in decision making. Our extensive experiments show that our method produces more robust detection results in comparison to SVM based classifiers.

2. A drug regulator database in a country with a relatively small population may already contain a large number of ADR reports that easily overwhelms the computing capacity for effective duplicate detection. In addition,  $k$ NN is both data- and compute-intensive when the dataset grows large. The MapReduce model is a convenient way to scale up the duplicate detection. However, its Google and open source implementation are mainly optimized for batch jobs so far [22]. Duplicate detection for ADR databases has a potential use-case for interactive and fast detection of duplicates for a specific report. Apache Spark [24] has the potential to support interactive data analytics. We implement a Spark based parallel system to support fast and scalable  $k$ NN classification for duplicates. Taking advantage of the distributed memory management and parallel data processing support offered by Spark, the system is able to handle large amount of ADR reports in a scalable manner.

The rest of this paper is organized as follows: Section 2 introduces the background technologies of our work; Section 3 defines the problem; Section 4 describes the system and our Fast  $k$ NN classification method; Section 5 gives the evaluation results of the performance of the system; Section 6 summarizes related works; and Section 7 concludes the paper.

## 2. BACKGROUND

### 2.1 $k$ NN Classification

In a  $D$ -dimensional space  $\mathcal{D}$ ,  $s$  and  $t$  are two vectors representing two data objects. We use  $d(s, t)$  to denote the distance between  $s$  and  $t$ . Consider  $S$  and  $T$  are two sets of such vectors, for a vector  $s \in S$ , we denote its  $k$  nearest neighbors according to a given distance function in  $T$  as  $knn(s, T, k)$ . We assume that each  $t \in T$  is associated with a label  $L_t, L_t \in \{-1, +1\}$ . The labels of vectors in  $S$  are unknown. We use  $knn^+(s, T, k)$  to denote the vectors in  $knn(s, T, k)$  with label “+1” and  $knn^-(s, T, k)$  to denote vectors with label “-1”.  $k$ NN classifier assign a label to  $s$  according to the following equation (note that the number of vectors in  $knn(s, T, k)$  is an odd number):

$$L_s = \begin{cases} +1, \sum_{t \in knn^+(s, T, k)} L_t + \sum_{t \in knn^-(s, T, k)} L_t > 0 \\ -1, \sum_{t \in knn^+(s, T, k)} L_t + \sum_{t \in knn^-(s, T, k)} L_t < 0 \end{cases} \quad (1)$$

Assigning labels to each  $s \in S$  according to their nearest neighbors requires a  $k$ NN *join* operation between  $S$  and  $T$  to identify  $k$  nearest neighbors of set  $S$  in  $T$ , denoted as  $S \times_{knn} T$ .

$$S \times_{knn} T = \{(s, knn(s, T, k)) | \forall s \in S\} \quad (2)$$

### 2.2 Spark

Spark [24] is a cluster computing framework that supports iterative and interactive data processing. It provides a level of data abstraction called *resilient distributed datasets* (RDDs) [23] to represent a set of immutable data objects. These data objects can be partitioned among a number of cluster nodes. RDDs are fault-tolerant and can be reconstructed when their hosting nodes fail.

There are two types of operations that can be applied to a RDD in the Spark framework: *transformations* and *actions*. A transformation contains operations that produces a new RDD from an existing RDD while an action returns a value after operating on a RDD. Operations are often executed in parallel on a RDD. In addition to support *map*, *reduce* and *aggregate* operations on a RDD, Spark also offers operations such as *join*, *union* and *cartesian* to manipulate multiple RDDs. Different to MapReduce, RDDs where *map* and *reduce* operate on can be persistent in memory in nodes of the cluster, which greatly improves the efficiency of iterative and interactive applications. A duplicate detection system like the one discussed in this paper is an iterative process that contains data processing of multiple stages and it fits the Spark framework well.

## 3. THE PROBLEM

An adverse drug reaction report database  $\mathcal{A}$  stores reports continuously collected by a regulator. We consider that a set of new reports, denoted by  $R$  arrive in the database may contain duplicates among themselves as well as with existing reports in the database. The problem is to identify the following set of report pairs:

$$Dupe(R, \mathcal{A}) = \{(r, h) | sim(r, h) < \epsilon, \forall r \in R, \forall h \in \mathcal{A} \cup R - r\} \quad (3)$$

in which, *sim* is a scoring function that measures the similarity between two reports and  $\epsilon$  is a threshold that determines whether two reports are duplicate.

Duplicate detection within database  $\mathcal{A}$  can be seen as a recursive process in which reports are sorted according to their arrival time to the database and reports with later arrival time are checked for duplication against those with earlier arrival time.

Note that even though an ADR database may contain 5% of reports that have at least one duplicate in the database, when it comes to the number of duplicated report pairs, the rate of duplicates is much lower. This is because the number of report pairs grows quadratically with the number of reports and non-duplicate report pairs grows much faster than duplicate report pairs. This results in highly imbalanced distribution of duplicate and non-duplicate report pairs in the dataset derived from  $\mathcal{A}$ .

## 4. THE SYSTEM

### 4.1 The Workflow

The workflow for duplicate detection in ADR database is shown in Figure 1. It contains the following major components:

- Report database: The report database stores reports collected by a regulator and new reports are continuously added to this database.
- Text processing module: Our system contains a text processing component to clean up text data in a report using common natural language processing techniques. Free text plays an increasingly important role in ADR reports as consumers increasingly participate in reporting drug side effects to regulators in recent years. Free text in a report not only is helpful for understanding drug side effects, but also contains useful information to identify duplicated reports. Compared to string fields such as name and address in existing duplicate detection systems, a free text field such as “report description” is significantly longer with majority of them being 250 and 300 characters long. This requires NLP techniques to extract useful information from the text for duplicate detection. On the other hand, regulators also increasingly monitor various data sources that contain large amount of text for ADR related information.
- Pairwise distance computing module: The module computes the pairwise distance among a set of reports including selected reports from the database and new reports arriving to the system.
- Training datasets (labelled datasets): The system maintains two temporary databases for duplicate detection: one contains report pairs that are known to be duplicate; the other contains samples of non-duplicate report pairs. The initial duplicate/non-duplicate labelling is done manually by domain experts from drug regulators, in our case, the TGA. Afterwards, the collection of report pairs in the two temporary databases are dynamically adjusted when new duplicates and non-duplicates are identified. Note that the duplicate report pair database stores all known duplicates while the non-duplicate report pair database only keeps a subset of known non-duplicates. The difference is due to the highly imbalanced distribution of duplicate and non-duplicate pairs.

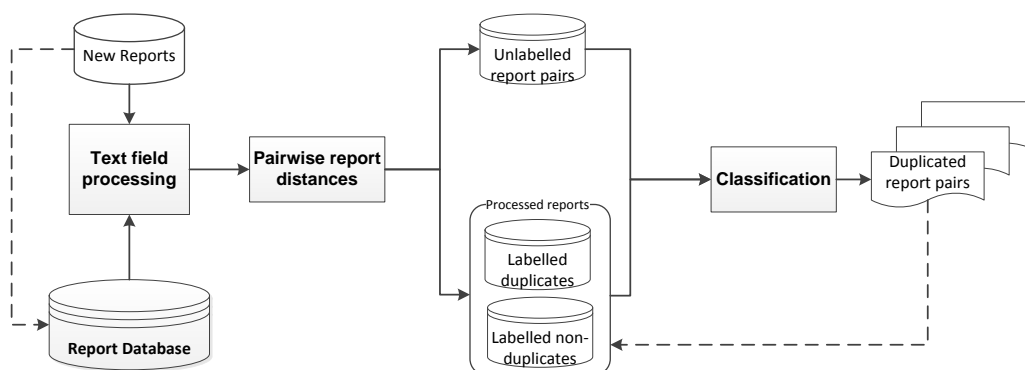


Figure 1: The workflow of the system – the dashed line represents that the source data becomes part of the target data when the processing finishes.

- Classification module: The report pairs are fed into the classification module that computes the scores for each pair and generates a list of duplicate pairs given a score threshold. Many classification algorithms fit into this system framework. We use  $k$ NN classifier as the default one for this application area. One advantage of  $k$ NN is that the classification results are easy to explain with human intuition and the basis of decision making can be justified clearly. This characteristics is particularly useful when the training dataset is highly imbalanced and global algorithms are difficult to separate them with general models.

## 4.2 Report Distance Calculation

A typical ADR report contains fields as shown in Table 2. Due to different missing data rates in different fields as well as schema inconsistency in data sources, common practices choose a subset of fields as input for duplicate detection. The fields used in our method are highlighted in bold fonts. The selection of fields is based on the WHO system as described in [18]. In the selected fields, *patient age* (“calculated age”) is numerical type. *Patient sex*, *residential state* and *onset date* are treated as categorical data type. *ADR name* (“MedDRA PT code”) and *drug name* (“generic name description”) have string type. Different methods deal with string type differently, e.g., they are treated as categorical type in probability based methods and programming level string type in SVM based methods. As mentioned, free text becomes increasingly important in ADR reporting systems, we therefore include the *report description* field in our duplicate detection system and treat it as string type.

For a numerical field, if the values of two reports in the field is the same, the distance is 0, otherwise 1. The same calculation applies to categorical field types. For fields of string type, we use *Jaccard similarity coefficient* to measure the distance between two values as below:

$$d(S_1, S_2) = 1 - \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \quad (4)$$

in which,  $|S|$  is the size of set  $S$ . The free text field is of string type, but as mentioned earlier, majority of values in the free text field are between 250 and 300 characters long. In order to eliminate the impact of typographical errors or

different ways of constructing sentences, we apply common techniques to *tokenize* the content in the *report description* field, *remove stop words*, and then *stem tokenized words* to their root forms before computing their distances.

The distances of values in these selected fields of any two reports form a distance vector between the report pair. The comparison between two distance vectors, i.e., measuring how similar a report pair is in comparison to another pair of reports, is based on the Euclidean distance between the two distance vectors of the two pairs.

Information	Fields
Case Details	case number, report date
Patient Details	<b>calculated age</b> , sex, weight code, ethnicity code, <b>residential state</b>
Reaction Information	<b>onset date</b> , date of outcome, <b>reaction outcome code</b> , reaction outcome description, severity code, severity description, <b>report description</b> , treatment text, hospitalisation code, hospitalisation description, MedDRA Low Level Term (LLT) code, LLT name, <b>MedDRA Preferred Term (PT) code</b> , PT name
Medicine Information	suspect code, suspect description, trade name code, trade name text, trade name description, generic name code, <b>generic name description</b> , dosage amount, unit proportion code, dosage form code, dosage form description, route of administration code, route of administration description, dosage start date, dosage halt date
Reporter Details	reporter type, report type description

Table 2: Data fields of an ADR report in TGA data. The bold font indicates fields used in our duplicate detection method.

### 4.3 Fast $k$ NN Classification

With the pairwise distances calculated, a  $k$ NN join is applied to labelled report pairs, denoted by  $T$ , and report pairs containing new reports, denoted by  $S$ . The classification of a report pair  $s \in S$  is based on the score computed from its nearest neighbors in  $T$ . Due to the imbalanced distribution of positive and negative labels, the negative report pairs easily overwhelm the positive ones. We therefore normalize the score using the distance between two pairs as below.

$$score_s = \sum_{t \in knn^+(s, T, k)} \frac{1}{sim(s, t)} - \sum_{t \in knn^-(s, T, k)} \frac{1}{sim(s, t)} \quad (5)$$

The label of  $s$  is therefore determined by the following equation, in which  $\theta$  is a given threshold:

$$L_s = \begin{cases} +1, & score_s \geq \theta \\ -1, & score_s < \theta \end{cases} \quad (6)$$

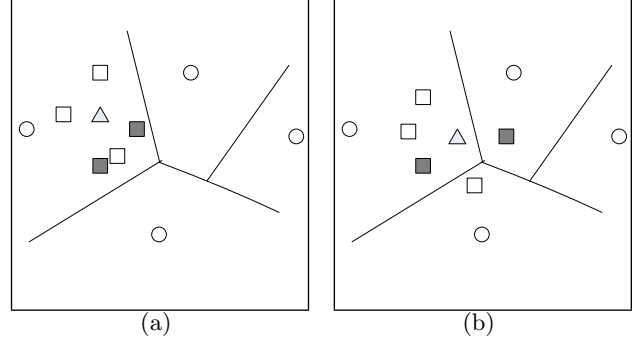
#### 4.3.1 Parallelization Strategy

Consider the number of report pairs in  $T$  is  $n$  and the number of report pairs in  $S$  is  $m$ , the computing complexity of  $k$ NN classification is  $O(m \cdot n)$  for join and  $O(m \cdot k)$  for score calculation.  $n$  exhibits quadratic growth with the number of reports. The amount of data to process easily overwhelms a single server. To make the classification scalable, we partition  $T$  and  $S$  into a set of clusters, denoted by  $\{T_1, T_2, \dots, T_b\}$  and  $\{S_1, S_2, \dots, S_c\}$  respectively. The cluster size in a partition is adjusted to fit into the memory capacity of a computing node. A naive parallelization strategy is to apply  $k$ NN join for each partition group  $\{(T_i, S_j) | 1 \leq i \leq b, 1 \leq j \leq c\}$  and then merge the nearest neighbors from each partition group. This approach does not reduce the overall computing complexity and incurs high data transfer cost as each partition in  $S$  needs to compare with all partitions in  $T$ . The merge of intermediate nearest neighbors may potentially become another bottleneck that limits the scalability.

To address this problem, we exploit the locality of report pairs in  $T$  in partitioning. We first partition report pairs using  $k$ -means clustering to obtain  $c$  clusters. The center of each cluster is calculated and stored in memory. Note that clusters produced by  $k$ -means form a *Voronoi diagram* where each report pair in a cluster is closer to the center of the cluster it belongs to than to any other cluster centers. We then assign each report pair  $s \in S$  to a cluster whose center is the closest to  $s$  comparing to other cluster centers. It is likely that most of the  $k$  nearest neighbors of  $s$  are within the cluster it is assigned, i.e., most of report pairs in  $knn(s, T, k)$  can be found in  $knn(s, T_i, k)$  where  $T_i$  is the cluster to which  $s$  is assigned. Certainly, there are chances that some of the  $k$  nearest neighbors of  $s$  are in clusters sharing borders with the cluster. The two scenarios are illustrated in Fig. 2.

Our method therefore consists of two stages to deal with the two scenarios. In the first stage, we compute the  $k$  nearest neighbors within a partition to which each  $s \in S$  is assigned. In the second stage, the cross-cluster comparison is performed for those testing report pairs falling into the second scenario in Fig. 2. The key technical challenge is how to determine whether it is necessary to check neighbor partitions when identifying the  $k$  nearest neighbors of a testing report pair.

#### 4.3.2 Observations



**Figure 2:  $k$ NN under partitioned dataset: the circle represents the cluster center of a partition; the triangle represents a testing report pair  $s$ ; the dark square represents a positive report pair in  $knn(s, T, k)$  and the light square represents a negative report pair in  $knn(s, T, k)$ . (a)  $knn(s, T, k)$  are all in one partition; (b)  $knn(s, T, k)$  are in different partitions.**

To address this problem, we develop an optimization algorithm that intends to prune unnecessary cross-cluster comparisons. The algorithm is based on the following observations:

1. The number of positively labelled report pairs is small and it incurs low computational cost to calculate distances between these positive report pairs and report pairs in testing dataset.
2. When the  $k$  nearest neighbors of  $s \in S$  are all labelled negative, there is no ground to classify  $s$  as a duplicate report pair.
3. When the distance between  $s$  and its nearest positive neighbor is greater than that between  $s$  and the  $k$ -th nearest negative neighbor in a subset of  $T$ , it is clear that there is no positive report pair in the  $knn(s, T, k)$ .
4. As mentioned above,  $k$ -means produces Voronoi partitions on the training report pairs, hence the hyperplane, denoted by  $h$  that separates two partitions is in the middle between the two cluster centers of the two partitions. If the distance between  $s \in S$  and its  $k$ -th nearest neighbor, denoted by  $s_k$  in the partition to which it is assigned is less than its distance to the hyperplane, the distance between  $s$  and  $s_k$ , denoted by  $d(s, s_k)$  is certainly less than distances from  $s$  to any report pairs in the other partition.

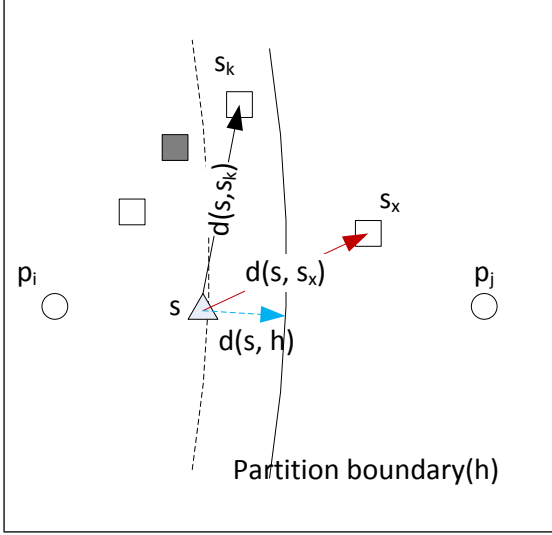
The scenario of observation 4 is shown in Fig. 3, where the distance between  $s$  and the partition hyperplane is  $d(s, h)$ , the distance between  $s$  and the closest report pair  $s_x$  in the other partition is  $d(s, s_x)$ .  $d(s, h)$  can be derived according to [9] as below:

$$d(s, h) = \frac{d(s, p_j)^2 - d(s, p_i)^2}{2 \cdot d(p_i, p_j)} \quad (7)$$

in which,  $p_i$  and  $p_j$  denote the center of partition  $T_i$  and  $T_j$  respectively. As  $d(s, b)$  is the shortest distance between  $s$  and the hyperplane  $b$  and connecting  $s$  and  $s_x$  needs to cross the hyperplane, we have  $d(s, s_x) \geq d(s, b)$  according to

the triangle inequality. Therefore when  $d(s, s_k) \leq d(s, b)$ , it is not necessary to include the partition  $T_j$  with center  $C_j$  in the searching for  $knn(s, T, k)$ .

Algorithm 1 describes the steps for selecting additional partitions to be included  $kNN$  computing for a report pair  $s$ . Line 2 – 5 is based on observation 1 – 3 and line 6 – 12 is based on observation 4.



**Figure 3: Additional partition selection for  $kNN$  under partitioned datasets:**  $p_i$  and  $p_j$  represent two partition centers; the triangle represents a testing report pair  $s$ .

---

#### Algorithm 1 Additional Partition Selection

---

**Require:**  $s \in S$   
**Require:**  $knn(s, T_i, k)$   
**Require:**  $\min(s, T^+)$ : the minimal distance between  $s$  and positive report pairs in  $T$   
**Require:** The centers of partitions :  $\{p_j | 1 \leq j \leq b\}$

- 1: partitions =  $\{\}$
- 2:  $d(s, s_k) = \max(knn(s, T_i, k))$
- 3: **if**  $d(s, s_k) \leq \min(s, T^+)$  **then**
- 4:     **return** partitions
- 5: **end if**
- 6: **for**  $1 \leq j \leq b, j \neq i$  **do**
- 7:     compute  $d(s, h_{ij})$  using Equation 7.  $h_{ij}$  denotes the hyperplane separating  $T_i$  and  $T_j$ .
- 8:     **if**  $d(s, s_k) > d(s, h_{ij})$  **then**
- 9:         partitions = partitions  $\cup T_j$
- 10:     **end if**
- 11: **end for**
- 12: **return** partitions

---

#### 4.3.3 The Classification Algorithm

Algorithm 2 gives main steps of implementing the duplicate detection method described above with Spark primitives of transformations and actions. The stage 1 (intra-cluster comparison stage) is performed in line 6 – 8. The stage 2 (cross-cluster comparison stage) is performance in line 9 – 16. Stage 2 first computes the distances of the testing report pair to all positive training pairs. It skips

cross-cluster comparison if the top  $k$  most similar report pairs obtained so far are all negative, indicating there will not be any positive training report pairs closer than the  $k$ -th nearest negative report pairs. Otherwise, cross-cluster comparisons are performed in line 12 – 15. The output score of each report pair is used to assign a label to the pair according to Equation 6.

---

#### Algorithm 2 Fast $kNN$ Classification.

---

**Require:** A training dataset  $T$  containing report pairs labelled as duplicate or non-duplicate  
**Require:** A testing dataset  $S$  containing unlabelled report pairs  
**Require:**  $b$  – the number of clusters for partitioning  $T$ ;  $c$  – the number of partitions for  $S$

- 1: use  $k$  – means to partition  $T$  into  $b$  clusters:  $\{T_1, T_2, \dots, T_b\}$  with cluster centers of these partitions denoted by  $P = \{p_1, p_2, \dots, p_b\}$
- 2: run *map* operation to compute distances between  $P$  and  $s \in S$
- 3: assign  $s$  the partition  $i$  where  $dist(s, p_i), (1 \leq i \leq b)$  is minimal for vectors in  $P$ .
- 4: randomly split  $S$  into  $c$  partitions :  $\{S_1, S_2, \dots, S_c\}$
- 5: **for**  $i = 1$  to  $c$  **do**
- 6:     run *join* operation on  $S_i$  and  $T^-$  based on cluster IDs ( $T^-$  denotes the negative report pairs in  $T$ )
- 7:     run *map* operation to compute the similarity between joined report pairs
- 8:     run *aggregate* operation to obtain the top  $k$  most similar report pairs for each  $s \in S_i$  based on the output of the previous step
- 9:     run *map* operation to compute distances between  $s \in S_i$  and  $T^+$  ( $T^+$  denotes the subset of positive report pairs in  $T$ )
- 10:     run *map* operation to combine the results from step 8 and step 9 to update the top  $k$  most similar reports pairs to  $s$
- 11:     **if** the output of step 10 contains at least one positive report pair **then**
- 12:         run *map* operation on  $\{(s, knn(s, T, k))\}$  for  $s \in S_i$  to compute a set of additional partitions to compare using Algorithm 1
- 13:         run *join* operation on  $S_i$  and additional partitions for  $s \in S_i$  to compare
- 14:         run *map* operation to compute the similarity between joined report pairs
- 15:         run *union* and *reduce* operation to merge top  $k$  nearest neighbors in each partition for  $s \in S_i$
- 16:     **end if**
- 17:     run *map* to calculate the score for  $s \in S_i$  according to Equation 5
- 18: **end for**
- 19: **return** all  $s \in S$  and corresponding scores

---

#### 4.3.4 Further Pruning of the Testing Dataset

To further reducing the execution time of  $kNN$  classification, we can prune some report pairs from the testing dataset before applying the classifier. As shown in Equation 6, a pre-defined  $\theta$  determines the label of a report pair. When the distance between a testing report pair and its positively labelled  $k$ -nearest neighbor is further than a threshold, the neighbor offers little hint to the label of the testing report

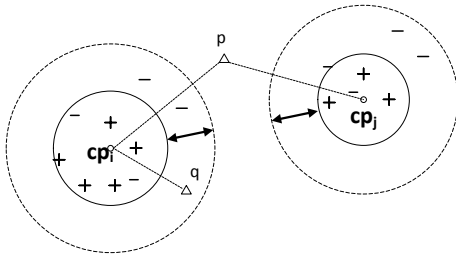


pair because of the low similarity between the two pairs. This observation can be used to prune the testing dataset.

When the testing dataset is large and the positive training pairs accumulate along time, it is necessary to speedup the computation of distances between each report pair in the testing dataset and each labelled positive report pair in the training dataset. We cluster the report pairs labelled as positive and use the cluster centers to determine whether a report pair in the testing dataset should be included in the classification. Assume the distance threshold between two report pairs is a function of  $\theta$ , denoted as  $f(\theta)$ , The process is described as below:

- Step 1. Cluster the positive report pairs into  $l$  clusters using  $k$ -means. We denote the cluster centers as  $cp_i (0 < i < l)$ ;
- Step 2. Compute the distance of the furthest report pair to its center in each cluster. We denote these distances as  $dcp_i (0 < i < l)$ ;
- Step 3. For each report pair  $t$  in the testing report pair set, do the following:
  - (a) For each  $0 < i < l$ , calculate  $dist(t, cp_i)$ ;
  - (b) if any  $dist(t, cp_i) \leq dcp_i + f(\theta)$ , include  $t$  into the testing set;

Fig. 4 shows an example of the process in 2-dimensional space.  $cp_i$  and  $cp_j$  are the centers of two clusters of positive report pairs. For simplicity, we assume the positive report pairs are partitioned into only two clusters. The inner circles are formed using the distance between the furthest report pair and the center in each cluster. The shortest distance between the dashed circle and its corresponding inner circle is  $f(\theta)$ . In Fig. 4,  $p$  and  $q$  are testing report pairs.  $p$  is outside of both dashed circles and those positive report pairs are considered not helpful to decide the label of  $p$ .  $p$  is therefore pruned from the testing dataset. On the other hand,  $q$  is close enough to  $cp_i$  (within its dashed circle) and the positive report pairs in the  $cp_i$  cluster may carry useful information for labelling  $q$ .  $q$  is therefore included in the testing dataset for classification.



**Figure 4: Pruning the testing dataset:**“+” represents a positive report pair and “-” represents a negative report pair. The triangle represents a testing report pair.

## 5. EVALUATIONS

We implement the duplicate detection system in Java with Spark 1.2.1 API. We evaluate the performance of our system in a cluster consisting of 14 physical nodes. Each node

has 2 x Intel Xeon E5-2660@2.20GHz CPU (8 cores) and 128GB physical RAM, in which 96GB is allocated to containers. The connection among nodes is via Infiniband networks. The OS in each node is Debian Wheezy. Cloudera CDH5 (5.0.0) with Hadoop 2.3.0 is installed with Yarn mode on. We run multiple *executors* on these nodes.

### 5.1 Datasets

We obtain ADR report data from TGA Australia. TGA maintains a database to store ADR reports submitted by various parties and collected by themselves. The sources that submit reports to the database include pharmaceutical companies, hospitals, general physicians, patients etc. TGA provides us 10,382 ADR reports they collected for a period of six months from July 2013 to December 2013. These reports consist 286 pairs of reports labelled as known duplicates. These duplicates were annotated by officers of TGA. Table 3 summarizes the dataset.

Report Period	1 Jul. 2013 - 31 Dec. 2013
Number of cases	10,382
Number of fields per report	37
Number of unique drugs	1,366
Number of unique ADRs	2,351
Known duplicate pairs	286

**Table 3: Summary of TGA dataset.**

The fields in each report in this dataset are listed in Table 2.

### 5.2 Fast $\kappa$ NN Performance

#### 5.2.1 Baseline

Many classification methods are applicable to duplicate detection problem where distance vectors of report pairs are classified as similar or different. Support vector machine (SVM) is a popularly used one in duplicate detection [2, 20]. In our evaluation, we use a SVM classifier as the baseline for comparison.

SVM based methods take distance vectors between each pair of reports as input and map them into a high-dimensional space. These methods then use a hyperplane to separate distance vectors that represent duplicate report pairs and those representing non-duplicate report pairs. The hyperplane is obtained through learning from a training dataset containing labelled duplicates and non-duplicates. The hyperplane maximizes its margins to points belonging to the two different classes. With the hyperplane, new report pairs are considered duplicates if their distance vectors fall into the match side of the hyperplane with a large margin; otherwise, they are considered non-duplicates.

#### 5.2.2 Precision and Recall

We measure the classification performance using the *area under precision and recall curve (AUPR)*. *Precision* and *recall* are defined as below in our case:

$$precision = \frac{\text{number of correctly identified duplicate pairs}}{\text{number of total identified duplicate pairs}}$$

$$recall = \frac{\text{number of correctly identified duplicate pairs}}{\text{number of total true duplicate pairs}}$$

*AUPR* shows how the precision values vary with different recall values. *AUPR* is able to visualize the difference of algorithms compared to other metrics and suitable for highly imbalanced datasets [4]. The goal to improve an algorithm with the precision-recall curve metric is to move the curve towards the upper-right corner.

Fig. 5 compare the performance of our Fast  $k$ NN algorithm and SVM. Fig. 5(a) and Fig. 5(b) show *AUPR* curves under different training dataset sizes. It is clear that in both cases, our algorithm significantly outperforms SVM based method. The main reason is that with highly imbalanced datasets, it is difficult to build a consistent model using SVM while large number of negative report pairs are surrounding few positive report pairs.

One way to improve the consistency of SVM classifier is to sample representative report pairs into the training dataset in hope that the model is applicable of a wide range of testing dataset. We implement an improved SVM classifier called *SVM clustering* by clustering training set and make sure report pairs in small clusters are included in the training dataset. Fig. 5(c) shows the actual area size varies with training dataset sizes under the three classification methods. It is easy to see that sampling a variety of report pairs into the training dataset does not have significant impact to SVM performance. Our method improves the classification performance by 19.1% in average in comparison to SVM classifier.

### 5.2.3 Effect of $k$

We also examine the impact of  $k$  on the classification performance and execution time. The results are shown in Fig. 6. We vary  $k$  from 5 to 21 and the variation of *AUPR* values is not significant, as shown in Fig. 6(a). This is due to that the score calculation takes the distance of a neighbor to the report pair being classified into account in Equation 5, which eliminates the impact of neighbors that are far away from the report pair to classify.

On the other hand, increasing  $k$  does increase the execution time of the Fast  $k$ NN classifier. As shown in Fig. 6(b), the execution time grows by 31% when  $k$  is increased from 5 to 21. This is mainly due to that a larger  $k$  potentially increases the number of partitions to compare.

### 5.2.4 Effect of cluster number $b$

The parallelism is affected by the number of clusters in the  $k - means$  step in Algorithm 2, which determines the number of training dataset partitions as well as the number of partitions of joined testing and training datasets. Fig. 7 shows how the system performance is affected by the setting of the cluster number. Fig. 7(a) shows that as the number of clusters increases, the overall number of intra-cluster comparisons decreases in general, while the number of additional clusters to check in the next phase increases proportionally as shown in Fig. 7(b). The increase of cluster number results in smaller number of report reports in each cluster, which leads to the reduction of the total number of intra-cluster comparisons. However, the trend stops when the cluster number increases to 70 and the total number of intra-cluster comparisons slightly increases. This is due to the cluster sizes are uneven and the probability that a large portion of report pairs in the testing dataset is assigned to a large cluster increases. Therefore the overall comparison number increases.

Note, the total number of additional clusters to check increases in the second stage does not mean the total number of cross-cluster comparisons increases. The shrinking cluster size also reduces the number of comparisons in each cluster in stage 2. As shown in Fig. 7(c), the total number of cross-cluster report pair comparisons shows a decreasing trend as the number of cluster increases. Similar to intra-cluster comparison stage, the trend stops when the number of clusters increases to 70. It is also due to the uneven distribution of cluster sizes.

The computational complexity of the cross-cluster comparison stage is low compared to the intra-cluster comparison stage. As shown in Fig. 8(a), the total number of cross-cluster comparisons varies from 1.4% to 1.9% of the total number of intra-cluster comparisons. This also indicates that further reducing the number of cross-cluster comparisons is not able to have big impact on overall execution time.

The execution time change with different cluster numbers reflects the change of overall comparison number. Fig. 8(b) shows that the execution time has a decreasing trend when the number of clusters increases. When the cluster number is set to a number below 25, the memory of each executor can not accommodate joined partitions and frequent swapping triggers a few task failures due to timeout. The automatic retries significantly stretches the execution time. When the cluster number increases from 25 to 55, the execution time is reduced by 31%. When the cluster number becomes 70, the execution time slightly increases by 5.7% comparing to that when the cluster number is 55.

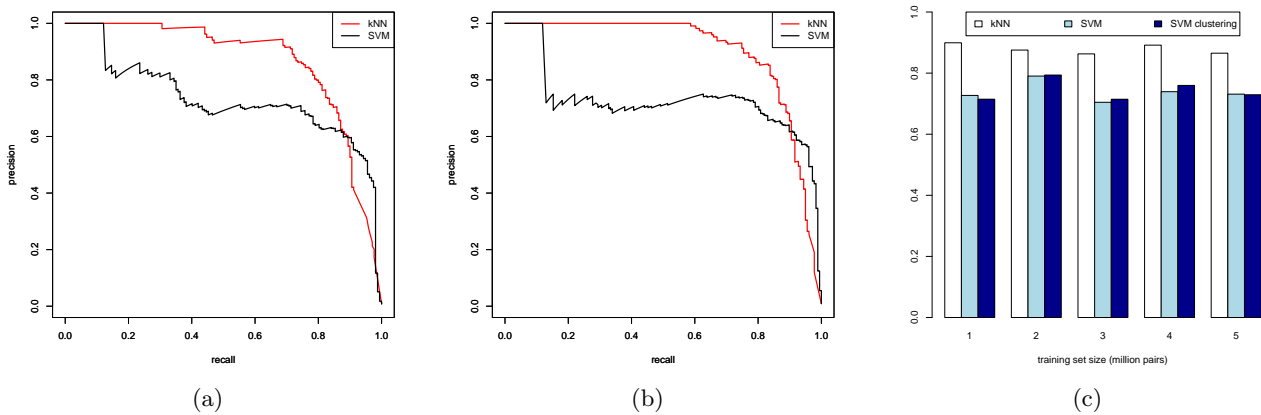
### 5.2.5 Scalability

We measure the scalability of Fast  $k$ NN from two aspects: firstly, we examine how it scales with the size of training dataset; secondly we investigate how it scales with the number of executors. As shown in Fig. 9, with different partition number of the testing dataset, the execution time increases proportionally with the increase of the training dataset size. The execution time increases 1.4 – 2.1 times when the size of training dataset increases 5 times.

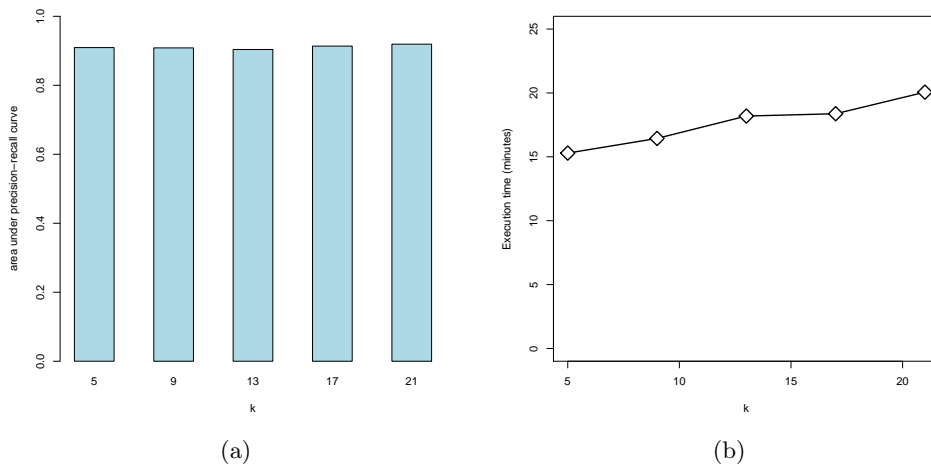
Fig. 10(a) shows the execution time change with the number of executors. For different training dataset sizes, the increase of executor number leads to the decrease of execution time. The decrease trends become flatter as the executor number increases. This is due to that the overhead of data shuffle gradually increases while more nodes participate the computation. For comparison purpose, we show the pairwise distance computing time separately in Fig. 10(b). The input data for pairwise distance computing is relatively small and the data distribution cost is low in this step. As a result, its speedup is significant when the number of executors further increases. Fig. 10(a) and Fig. 10(b) also show that the time used in the pairwise distance computing step is only a small portion of the overall execution time.

### 5.2.6 Effect of Testing Set Pruning

In the following, we measure the effectiveness of the testing set pruning method. We use 204,736 randomly selected report pairs as testing dataset. The training data contains 1,000,000 report pairs, in which 266 pairs are duplicate. Fig. 11 shows the result. When the distance threshold is set to 0.9, nearly 100% of the testing pairs are included in the classification phase. When the threshold is set to 0.7,



**Figure 5: Comparison of area under precision and recall curve:  $k$ NN vs. SVM. Total number of testing pairs – 20,000. (a) Total number of training pairs – 5 millions; (b) Total number of training pairs – 1 millions; (c) Change of area sizes under precision and recall curves: the number of clusters in SVM clustering is set to 8.**



**Figure 6: Effect of  $k$ : Total number of training pairs – 3 millions; Total number of testing pairs – 10,000. (a) Area under precision-recall curve (AUPR) comparison; (b) The execution time comparison.**

about 75% of testing pairs are included in the classification phase. Setting the threshold to 0.5 does not produce significant pruning and 73% of testing pairs are included. When the threshold is set to 0.3, 65% of testing pairs remain. The pruning ratio is not exactly proportional to the threshold setting because of the non-uniform distribution of both positive training report pairs and testing report pairs. Note that all these threshold settings enable the duplicate report pairs in the testing dataset being included for classification.

On the execution time of classification aspect, the reduction is significant. The threshold setting of 0.3, 0.5 and 0.7 reduces the execution time to 35%, 65% and 61% of the classification time without pruning. This is mainly due to the reduced data transfer and memory use. Even though setting the threshold to 0.5 slightly prunes more testing pairs than setting it to 0.7, the classification time under threshold 0.5 is slightly longer than that under threshold 0.7. It is related to the report pair distribution and how balance the workload

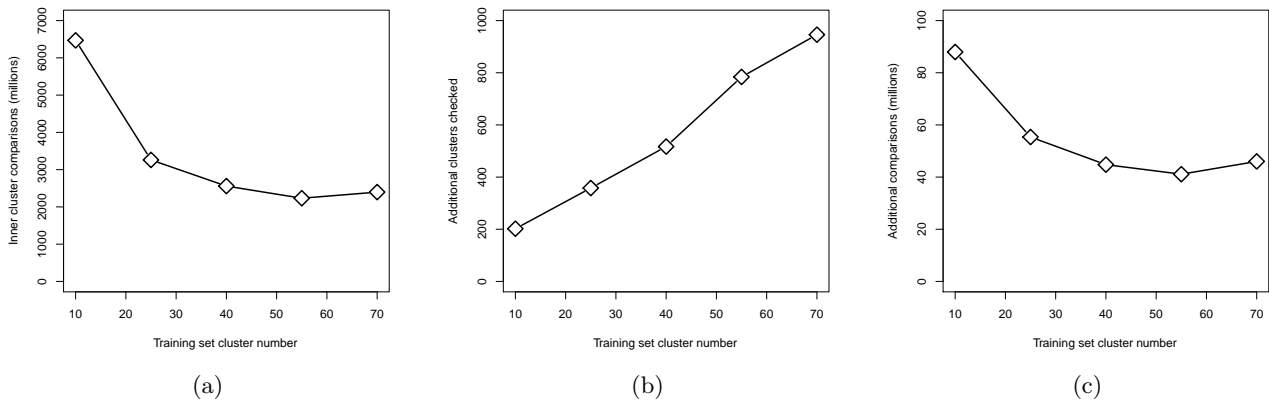
of comparing these report pairs is among Spark data nodes that store them.

The setting of the threshold directly affects the performance improvement of testing set pruning. Potentially, the setting can be learned from the labelled data, which we leave as our future work.

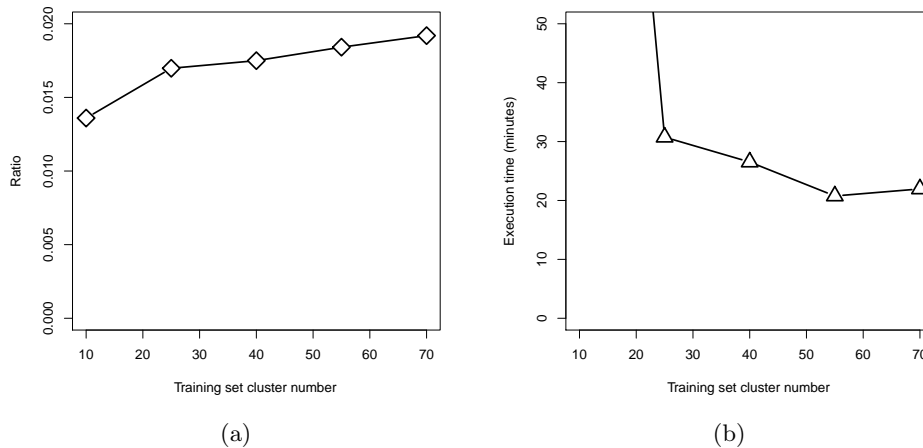
## 6. RELATED WORK

We compare our method with closely related works on parallelizing  $k$ NN join as well as approaches for handling datasets with imbalanced label distribution.

C. Zhang et.al [25] describes a basic Block based data partitioning method for  $k$ NN join using Hadoop. It then proposes to build an index using R-tree for each local block in a dataset. The built-in  $k$ NN functionality of R-tree is able to speedup the  $k$ NN search in the local block. The method does not reduce the overall computing and communication complexity.



**Figure 7: Impact of the cluster number: Total number of training pairs – 4 millions; Total number of testing pairs – 10,000. (a) The number of intra-cluster comparisons; (b) The number of additional clusters to check for each element in the testing set; (c) The number of cross-cluster comparisons.**



**Figure 8: Impact of the cluster number of training set on cross-cluster comparison: Total number of training pairs – 4 millions; Total number of testing pairs – 10,000. (a) Ratios of cross-cluster/intra-cluster comparison number; (b) The execution time change with the cluster number: memory size of each executor is 32GB.**

W. Lu et.al [15] gives an improved algorithm for parallelizing  $k$ NN join using MapReduce and aims to reduce the search space. It partitions datasets into a Voronoi diagram. The differences between our approach and [15] lie in the following aspects: firstly, our method uses  $k$  – means to partition the training dataset while [15] uses it as an option in data pre-processing to select pivots (partition centers). A *map* operation is then applied to use the selected pivots to partition datasets and collect partition statistics. As  $k$  – means produces Voronoi sets already, it is not necessary to run another data partitioning operation. Secondly, our approach uses the characteristics of imbalanced datasets to reduce the cross-cluster comparison rather than introducing another statistics collection step to achieve this goal. Our experimental results show that the cross-cluster comparison cost is low. Thirdly, our approach makes use of distributed memory management of Spark to cache data partitions in comparison to the ad-hoc caching mechanism in [15].

In addition, there are works on set-similarity join using MapReduce [21] and Voronoi partitioning for MapReduce [1]. These works focus on identifying the nearest neighbors. Our work differ from them in using  $k$ NN as a means for classifying highly imbalanced datasets, which gives us additional information for reducing the search space, e.g., when the distance between a report pair and its nearest positively labelled neighbor is further than a given distance threshold, the report pair is likely to be non-duplicate and therefore pruned from the search space.

W. Liu and S. Chawla [14] illustrates the problem of imbalanced label distribution in classification and proposes a weighted method for handling imbalanced data in  $k$ NN classifier. Our results show that  $k$ NN classifier is more robust and less affected by over-represented negative data than SVM on detecting highly imbalanced duplicate/non-duplicate report pairs. Further improving the classification performance of  $k$ NN require careful analysis of similarity of report

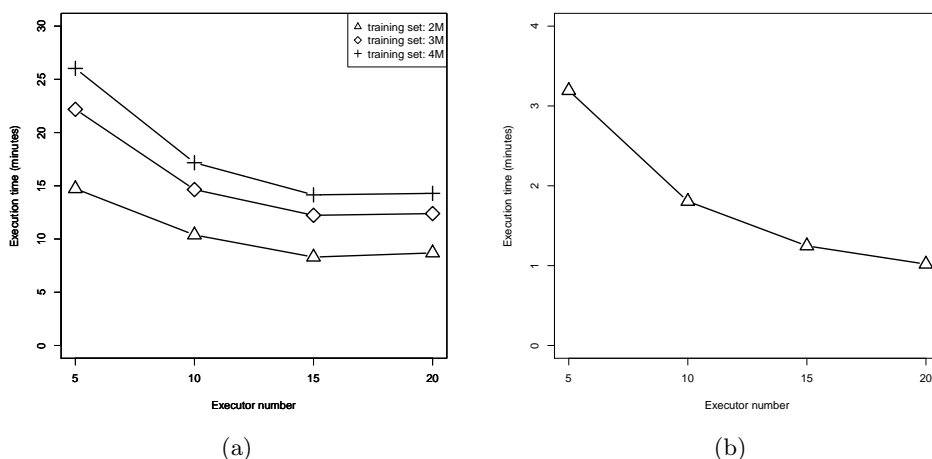


Figure 10: Execution time change with the number of executors: testing set size – 10,000; cluster number of the training set – 48; block number – 5; executor memory size – 32GB; executor core – 1. (a) Overall execution time; (b) Pairwise distance computing time (total number of reports – 10,382).

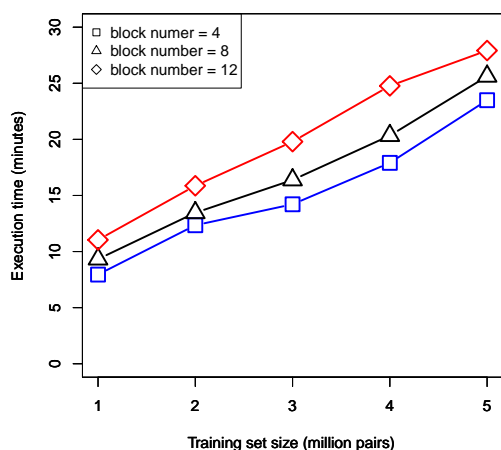


Figure 9: Scalability with the size of training dataset: testing set size – 10,000; cluster number of the training set – 32; total number of executors – 25 (32GB memory and 1 core).

pairs by taking additional domain knowledge into account, which is our future work. Fortunately, the simplicity of  $k$ NN provides flexibility to accommodate new models.

## 7. CONCLUSIONS

In this paper, we studied duplicate detection in adverse drug reaction report databases. We proposed a Fast  $k$ NN classification method to deal with highly imbalanced label distribution in the dataset. Comparing to some datasets used for evaluating  $k$ NN join methods, there were relatively small number of fields chosen to calculate the pairwise distance vector between reports, however, the ADR report database contained a long text field with non-trivial distance calcula-

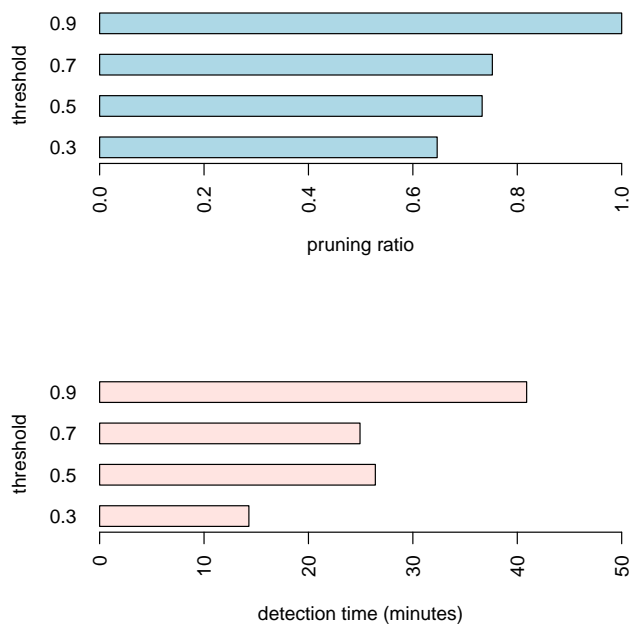


Figure 11: Effectiveness of pruning the testing dataset: training set size – 1,000,000; cluster number of the training set – 200; testing set size – 204,736; cluster number of the testing set – 30;  $f(\theta)$  (threshold) is set to 0.3, 0.5, 0.7 and 0.9; executor number – 20; number of cores per execution – 4.

tion complexity. We showed that our method is effective in detecting duplicates in real ADR data and significantly outperforms SVM based classifier. The system we implemented using this method is capable of handling large amount of adverse drug reaction report data in a scalable way. We built the system using Spark. We effectively reduce the comput-

ing complexity by exploiting label imbalance and Voronoi partitioning of the training dataset. We also gave a method to prune the testing dataset to further improve the performance. We are not aware of other parallel duplicate detection system in practical use in this domain with rapidly growing data and increasing importance. Our future work will focus on load balancing among executors for better scalability.

## 8. REFERENCES

- [1] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. Voronoi-based geospatial query processing with mapreduce. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 9–16. IEEE, 2010.
- [2] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 39–48. ACM, 2003.
- [3] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings Of the 22nd International Conference On Data Engineering*, pages 5–17, Atlanta, GA, 2006.
- [4] J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.
- [5] M. Elfeiky, V. Verykios, and A. Elmagarmid. TAILOR: A record linkage toolbox. In *Proceedings Of the 18th International Conference On Data Engineering*, pages 17–28, San Jose, CA, 2002.
- [6] S. Evans, P. Waller, and S. Davis. Use of proportional reporting ratios (PRRs) for signal generation from spontaneous adverse drug reaction reports. *Pharmacoepidemiology and Drug Safety*, 10(6):483–486, 2001.
- [7] D. Fram, J. Almenoff, and W. DuMouchel. Empirical Bayesian data mining for discovering patterns in post-marketing drug safety. In *PKDD*, pages 359–368, Washington, DC, 2003.
- [8] R. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.
- [9] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces (survey article). *ACM Trans. Database Syst.*, 28(4):517–580, Dec. 2003.
- [10] B. Hug, C. Keohane, D. Seger, C. Yoon, and D. Bates. The costs of adverse drug events in community hospitals. *Joint Commission Journal On Quality and Patient Safety*, 38(3):120–126, 2012.
- [11] M. A. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406):pp. 414–420, 1989.
- [12] M. A. Jaro. Probabilistic linkage of large public health data files. *Statistics in medicine*, 14(5-7):491–498, 1995.
- [13] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966.
- [14] W. Liu and S. Chawla. Class confidence weighted knn algorithms for imbalanced data sets. In *Proceedings of the 15th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining - Volume Part II, PAKDD'11*, pages 345–356, Berlin, Heidelberg, 2011. Springer-Verlag.
- [15] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proceedings of the VLDB Endowment*, 5(10):1016–1027, 2012.
- [16] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records: Computers can be used to extract “follow-up” statistics of families from files of routine records. *Science*, 130(3381):954–959, 1959.
- [17] J. Nkanza and W. Walop. Vaccine associated adverse event surveillance (VAAES) and quality assurance. *Drug Safety*, 27(12):951–952, 2004.
- [18] G. N. Norén, R. Orre, and A. Bate. A hit-miss model for duplicate detection in the WHO drug safety database. In *KDD*, pages 459–468, Chicago, Illinois, 2005.
- [19] E. Roughead and S. Semple. Medication safety in acute care in Australia: Where are we now? part 1: A review of the extent and causes of medication problems 2002-2008. *Australia and New Zealand Health Policy*, 6(1):18, 2009.
- [20] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *The 8th ACM SIGKDD International Conference On Knowledge Discovery and Data Mining*, pages 269–278, Edmonton, Alberta, Canada, 2002.
- [21] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 495–506. ACM, 2010.
- [22] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Fast and interactive analytics over hadoop data with spark. *USENIX; login*, 37(4):45–51, 2012.
- [23] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.
- [24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.
- [25] C. Zhang, F. Li, and J. Jestes. Efficient parallel knn joins for large data in mapreduce. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 38–49. ACM, 2012.

# e#: Sharper Expertise Detection from Microblogs

Thibault Sellam  
CWI, the Netherlands  
thibault.sellam@cwi.nl

Martin Hentschel\*  
Snowflake  
hemartin@snowflake.net

Vasilis Kandylas  
Microsoft  
vakandyl@microsoft.com

Omar Alonso  
Microsoft  
omalonso@microsoft.com

## ABSTRACT

Microblogging platforms such as Twitter provide low cost access to an immense reserve of authoritative professionals, opinion leaders and hobbyists for a wide range of topics. Yet, as microposts are short and incredibly diverse, many of these experts are hidden. In this paper, we present *e#*, a system to retrieve experts automatically for a given set of keywords. Our design targets exhaustivity: *e#* can detect previously undetectable experts. The core idea is to enhance a state-of-the-art expert detection algorithm with a graph of expertise domains. Our system produces this graph from hundreds of Gigabytes of Web search query logs and behavioral data, processed in a distributed, parallel fashion. We provide a detailed description of our architecture, including an original SQL-based community detection algorithm. We then benchmark our system with 750 queries, using crowdsourcing. We observe that *e#* finds many more experts than a state-of-the-art baseline.

## Keywords

Expert detection, query expansion, clustering

## 1. INTRODUCTION

Microblogging offers a mighty, low-cost means to disseminate and consume knowledge. Platforms such as Twitter let political analysts comment elections, sports journalists explain why their favorite team fell short, and technology fans criticize the weight of a new phone. In this paper, we investigate the problem of *expertise detection*: we want to retrieve experts from microblogs, given a topic expressed as a set of keywords. For example, suppose that we wish to learn more about American football from Twitter. Given a set of keywords such as *49ers* or *NFL* as input, can we return a list of all the authoritative Twitter accounts?

Expertise detection systems must achieve high *precision*

\*Martin Hentschel was affiliated with Microsoft at the time of this work

and high *recall*. *Precision* measures the purity of the results. It is the proportion of experts returned by the system which are relevant to the topic. On Twitter, achieving high precision is challenging because the data contains an extravagantly large range of topics and vocabulary: it contains spam, fake accounts, but also many ambiguities. In our example, the simple term *football* designates a different sport in Europe and America. *Recall* measures the exhaustivity of the results. It is the proportion of relevant experts on the whole microblogging platform detected by our system. Recall is challenging because tweets are short. An expert in *49ers* is likely to be an expert in *West Coast football* too, because the *49ers* is a popular football team from the US West Coast. Yet, as tweets cannot contain more than 140 characters, the chance to have both expressions in the same post is low. Therefore, a search for *49ers* may miss the experts for *West Coast football*.

Expertise detection has been studied for decades, but in a very different context: initially, it focused on finding experts from enterprise documents, in order to smoothen collaboration between employees. The corpora were small, heterogeneous and the queries were very specific (e.g., *FORTRAN developer*). With social media, the context is different: the topics of interest can be narrow (e.g., *49ers draft*) or broad (e.g., *sports*). The corpora are homogeneous (all messages have the same format), but their scale is massive. Also, the requirements in terms of precision and recall are different. In enterprise settings, the aim was to initiate professional collaborations, thus false positives were very costly. Most studies on expertise retrieval targeted precision, recall came as distant second [4]. In contrast, our users are looking for information sources, not collaborators. Hence, they value depth and variety, while false positives are relatively cheap. This shifts the balance towards recall. Unfortunately, achieving high recall is also much harder on social media, because microposts have a short length and an immense vocabulary.

How can we detect experts with both high recall and high precision? We present *e#*, a system to detect previously undetectable experts. Our strategy is based on *query expansion*, well known in the context of document search but seldom used for expert detection. We operate in two steps, offline and online. Offline, we build a collection of linked topics of expertise from Web data. Online, we exploit this collection to *augment* incoming queries, and feed the result to a precision-based expert detector. We obtain a variety of high quality experts.

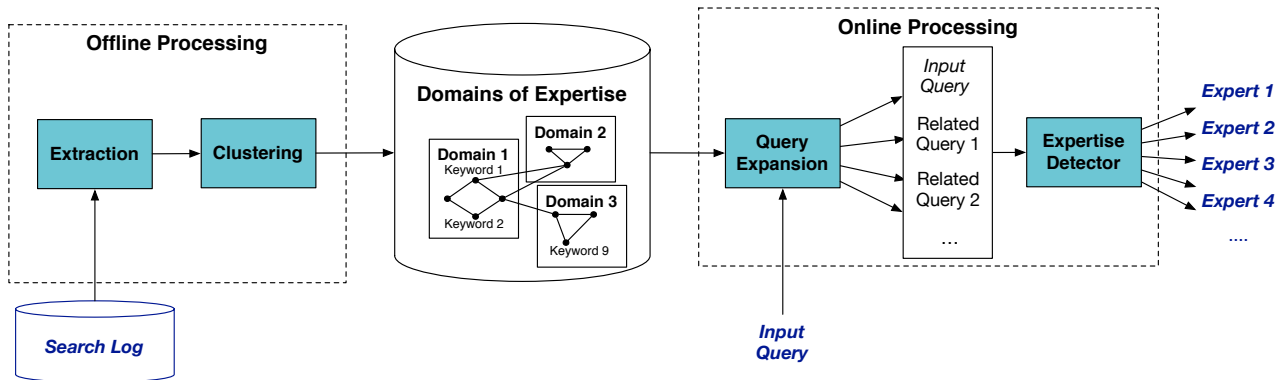


Figure 1: Overview of e# - our system augments an input query with related queries, inferred from the search log of a commercial search engine.

Two questions remain. First, how do we collect and link topics of expertise? We propose to exploit the search query log of a commercial search engine. This source gives us a massive, time-relevant collection of keywords. We infer the semantic associations between the terms with search and click behavior. Second, how do we exploit this collection? Our approach is to partition the search terms into *communities*, that is, groups of strongly related keywords. We then use these groups to enrich the queries. Because of the scale of the datasets involved, the engineering effort is non trivial. We present an original implementation of *modularity maximization*, a framework to detect communities in graphs. The advantage of our approach is that it can directly be implemented in (parallel) declarative languages such as Hive, Pig, Microsoft’s SCOPE or even SQL.

To summarize, here are our contributions:

- We present our pipeline e#, which combines search query log analysis, community detection and query expansion at scale.
- We introduce a parallel, distributed algorithm to infer clusters of related keywords from large Web search logs.
- We describe a complete experimental evaluation, with real-life examples and a crowdsourcing study. We present our results on 750 queries from many different topics.

The rest of this paper is organized as follows. In the following section, we give an overview of e#. We then present the base expertise detection algorithm on which we built e#. In the fourth section, we detail how e# builds collections of related keywords to expand the queries. We expose how our system matches queries and expertise domains in the fifth section. We present our experiments in the sixth section. We then describe related works and conclude.

## 2. OVERVIEW

The main idea behind e# is to enhance an existing expert detection algorithm with a collection of expertise domains. Figure 1 gives an overview of our pipeline. It depicts two stages: an offline stage, during which we build the collection, and an online stage, during which we exploit it.

The offline stage can itself be decomposed in two steps. First, we process a search query log. Using search terms

and clicks, we build a weighted graph, in which each vertex represents a keyword and each edge represents a semantic association. Then, we detect *communities* in this graph, with a custom SQL-based algorithm. Each of the communities we obtain describes a topic of expertise, exploitable for query augmentation.

During the online stage, we run the actual query augmentation: we match the query with a topic of expertise from the database, and append the corresponding keywords. We then run a detection algorithm presented previously in the literature [14]. Section 3 presents the algorithm.

Thanks to the query log, our collection of domains is inherently current. For instance, at the time of writing, it contained keywords related to new technological products (**smart watches** or **VR glasses**) or upcoming media events (e.g., **Star Wars VII**). This is particularly useful when dealing with social media. Also, entries often come in many variants (e.g., **football**, **foftbal**, **foot**, etc...). This variety improves the robustness of our system at little CPU cost.

## 3. PRELIMINARIES - BASELINE

In this section, we present the algorithm on which we built e#. Expertise detection involves two main challenges: *candidate selection* and *expertise ranking*. Candidate selection is the problem of finding candidate experts for a given topic. Expertise ranking is the problem of determining the strength of expertise given textual evidence. To solve both problems, we use a method proposed recently by Pal and Counts [14], shown to be competitive for Twitter data. The framework was simplified for production purposes, it currently runs in a commercial environment.

We implemented candidate selection on Twitter as follows. A candidate expert is either an author of a tweet, or a person mentioned in a tweet. In both cases, the tweet must match the query. By default, a tweet matches a query if it contains all of its terms after lower-casing [4, 14].

For expertise ranking, we first compute features of textual evidence, and then rank the candidates on these features. In their paper, Pal and Counts evaluate a dozen features. We kept those which they present as important: the topical signal (*TS*), the mention impact (*MI*), and the retweet impact (*RI*). These features are defined as follows:



$$TS = \frac{\#tweets\ by\ user\ on\ topic}{\#tweets\ by\ user}$$

$$MI = \frac{\#mentions\ of\ user\ on\ topic}{\#mentions\ of\ user}$$

$$RI = \frac{\#retweets\ of\ user's\ tweets\ on\ topic}{\#retweets\ of\ user's\ tweets}$$

The first two features,  $TS$  and  $MI$ , measure how much the user is specialized in the topic of interest. The third feature,  $RI$ , measures the influence of the user.

Before we perform the ranking, we normalize and aggregate the features. To normalize the features, we compute their z-score. For instance, if  $\mu_{TS}$  is the average of  $TS$  and  $\sigma_{TS}$  its standard deviation, we compute  $z_{TS} = \frac{x - \mu_{TS}}{\sigma_{TS}}$ . In practice, the features appear to be log-normally distributed. Therefore, we take their logarithm to obtain Gaussian distributions. To aggregate the scores, we used a weighted sum, using the authors' guidelines.

In their paper, Pal and Counts propose an optional filtering step, based on cluster analysis. This step is computationally expensive, and it is contrary to our objective of improving recall. Therefore, we discarded it in our implementation.

## 4. COLLECTING TOPICS OF EXPERTISE

In this section, we describe how we build our collection of expertise domains. During the *extraction* phase, we derive a graph of semantic relationships from the search query log. During the *clustering* phase, we detail how to decompose this graph into communities, using a parallel, modularity-based approach.

### 4.1 Extracting Semantic Relationships

To build our collection of related topics, we exploit the search log of a commercial search engine. We chose this source because it is intrinsically current and exhaustive.

How can we infer semantic connections between terms from a search log? We propose to exploit the URLs clicked for each keyword. This approach lets us detect non-obvious semantic associations, and it is practical to implement [1]. Consider a vector space where each dimension represents a URL from the query log. In this space, we associate each query to a vector. Each component of the vector represents the number of clicks on the URL. To obtain the similarity between two terms, we can compute the cosine distance between the two vectors which represent them. If we compute the distance between every possible pair of terms, we obtain a term similarity graph. In this weighted, undirected graph, each vertex represents a query, and the edges describe their similarity. We illustrate this operation with Figure 2. This graph gives us the material for our next step: the community detection.

In practice, a few adjustments are necessary. For instance, we remove all the queries which appear less than 50 times per month, to reduce noise and save space. Even after this operation, the same term can appear with dozens, sometimes hundreds of variants (e.g., `san francisco`, `#sanfrancisco`, `sf`, ...). We leave these queries unchanged (no stemming, or correcting), in order to capture as many different cases as possible.

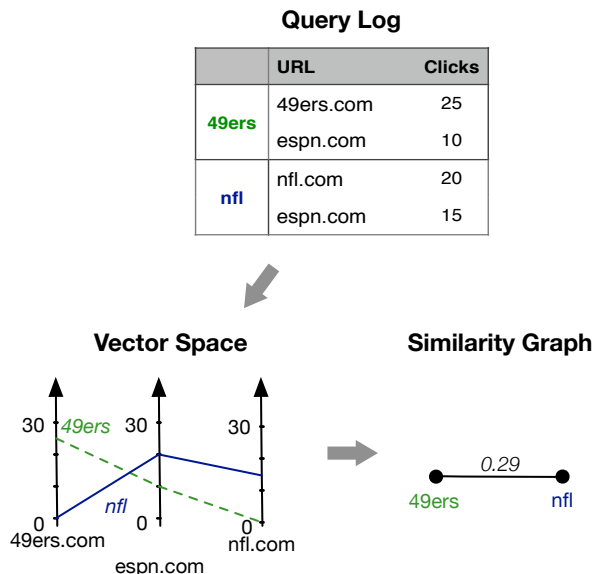


Figure 2: Extracting the similarity between terms from the search log.

## 4.2 Detecting High-Level Domains

Once the similarity graph is built, our next step is to create groups of related keywords. We solve this problem with *community detection*. The idea is to identify groups of queries which are densely connected to each other, but loosely connected to the rest of the graph. We assume that if a group of keywords obeys such a property, then we can use it to expand queries. The network analysis literature contains dozens of ways to formalize this notion [10]. We base our system on modularity maximization [13], which is simple and widely studied. We first present the original sequential algorithm, proposed by Newman et al., then we introduce our parallel variant.

### 4.2.1 Modularity Maximization

**Overview.** Consider an undirected graph  $G = (V, E)$ . For the sake of presentation, we consider that this graph is not weighted, but that more than one edge can connect two nodes<sup>1</sup>. Consider a set of vertices  $C \subset V$ . The modularity measures how densely connected  $C$  is. To compute it, we count the number of edges within the set, and compare to what we would expect if the edges were drawn randomly between  $G$ 's vertices, preserving the vertex degrees. The modularity is the difference between these two terms. Let  $E$  describe the expected value:

$$Modularity = \#edges - E[\#edges] \quad (1)$$

Partition  $G$ 's vertices into  $p$  partitions  $C_1, \dots, C_p$ . If we sum the modularities of each of these partitions, we obtain the *total modularity*:

$$TMod(\{C_1, \dots, C_p\}) = \sum_{i \in 1..p} Mod(C_i) \quad (2)$$

<sup>1</sup>We can convert the similarity graph described in the previous section into this representation. To do so, we rescale and discretize the weights to obtain integers. Then, we create one edge for each unit.

This is our objective function. A high value indicates that we found many dense communities. A low value means that either the graph does not contain any community, or that the partitioning is sub-optimal.

**Computing the Modularity.** We defined the modularity as the difference between the number of edges within a set of vertices and the expected number of edge within this set. To obtain the first quantity, we can simply count. Now, how do we obtain the second one?

For a set of vertices  $C$ , the variable  $m_C$  describes the number of edges and  $E[m_C]$  the expected number of edges. We have:

$$Mod(C) = m_C - E[m_C] \quad (3)$$

We want to compute  $E[m_C]$ . Draw an edge at random between two vertices of  $G$ . Let  $P_C$  describe the probability that the edge connects two vertices of  $C$ , and let  $m_G$  denote the number of edges in the graph. We obtain:

$$E[m_C] = m_G * P_C \quad (4)$$

Let's compute the probability  $P_C$ . Let  $D_G = 2 * m_G$  represent the sum of all the degrees of all the vertices of the graph, and let  $D_C$  represent the sum of all the degrees of the vertices in  $C$ . For a given edge, the probability that one of the endpoints ends up in the set  $C$  is  $D_C/D_G$ . Therefore, the probability of having both endpoints in the community is:

$$P_C = (D_C/D_G)^2 \quad (5)$$

Putting everything together, we obtain:

$$Mod(C) = m_C - m_G * (D_C/D_G)^2 \quad (6)$$

Note that the modularity is often normalized: many authors use  $Mod(C)/m_G$  instead of  $Mod(C)$ . As  $m_G$  is a constant, this approach is equivalent to ours.

**Greedy Heuristic.** Maximizing modularity is a NP-hard problem [13]. The seminal single-machine heuristic, presented by Newman et al., operates in a greedy, bottom-up manner. We initialize the algorithm by assigning each vertex to its own community. Then, at each iteration, we find the two closest communities, and merge them. We stop when we cannot improve the score anymore, or when we have reached a satisfying number of communities.

The critical part of the algorithm is to define "closest". According to Newman, two communities are close if merging them leads to an improvement in the global modularity. Formally, if  $C_1$  and  $C_2$  describe these communities, we have:

$$\Delta Mod = Mod(C_1 \cup C_2) - Mod(C_1) - Mod(C_2) > 0 \quad (7)$$

Instead of computing the three terms of this equation separately, we can use a computational shortcut [13]. If  $m_{1 \leftrightarrow 2}$  represents the number of edges between  $C_1$  and  $C_2$ , we have:

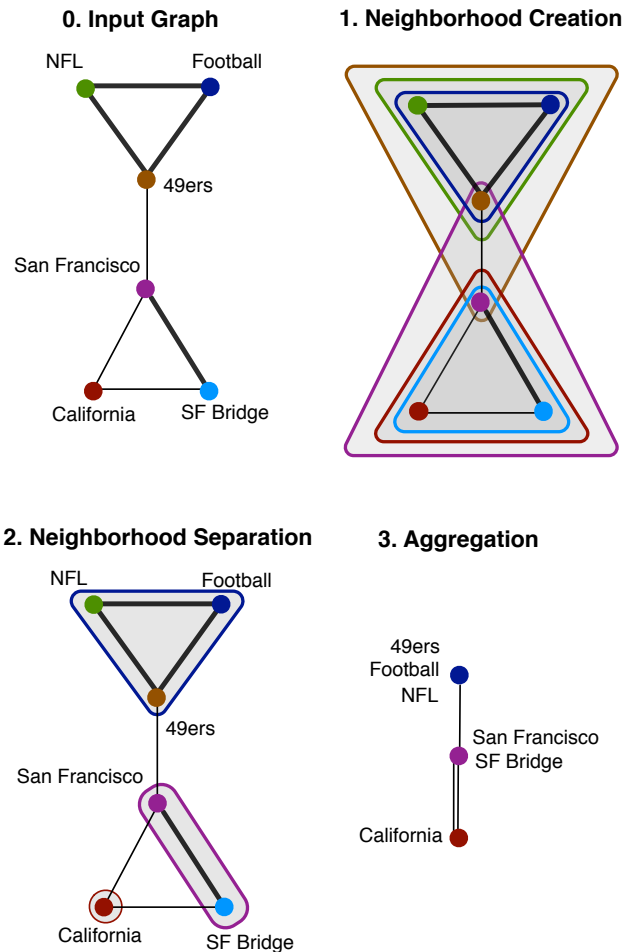
$$\Delta Mod = m_{1 \leftrightarrow 2} - E[m_{1 \leftrightarrow 2}] \quad (8)$$

Let  $D_1$  and  $D_2$  represent the sum of degrees of  $C_1$  and  $C_2$ 's vertices. We obtain the second term as follows:

$$E[m_{1 \leftrightarrow 2}] = \frac{D_1 * D_2}{2 * m_G} \quad (9)$$

#### 4.2.2 SQL-based Modularity Maximization

To deal with the scale of commercial search engine query logs, we developed a custom variant of Newman's procedure.



**Figure 3: First iteration of our modularity optimization algorithm on a fictive example. The shaded triangles represent neighborhoods.**

Compared to previously published frameworks such as [17], our approach can be directly implemented in a SQL-like language such as Hive, Microsoft's SCOPE or Pig. Therefore, we can parallelize it with standard map-reduce relational operators [5].

As previously, we initialize the algorithm by assigning each vertex to a community. Then, we repeat the following three steps:

1. For each community, list all the *neighbor* communities. Two communities are neighbors if (a) they are connected and (b) if we union them, the total modularity increases ( $\Delta Mod > 0$ ). We obtain several *neighborhoods*, one for each community.
2. The neighborhoods found in Step 1 are overlapping: one community may belong to several neighborhoods. To remediate this, take each community, list all the neighborhoods to which it belongs and keep the closest one ( $\Delta Mod$  is as large as possible).
3. For each neighborhood, aggregate all the communities into one large, new community

```

neighbors = select c1.query as query1,
                 c2.query as query2,
                 distance
           from graph
           inner join communities c1 on query2
           inner join communities c2 on query1
           where ModulGain(query1,query2) > 0;

partitions = select query2,
                  argmax(distance, query1)
           from neighbors
           group by query2;

communities = select query1 as comm_name,
                   query2 as query;

```

Figure 4: Body of the community detection algorithm in pseudo-SQL. The table **Graph(query1, query2, distance)** represents the graph, and **Communities(comm\_name, query)** represent the communities (the foreign key relationships are underlined).

We illustrate these steps in Figure 3, and present the pseudo-code in Figure 4.

#### 4.2.3 Parallelization and Optimization

We presented our algorithm in pseudo-SQL. This approach is *declarative*: we rely on the data management system to effectively parallelize the code. We now discuss a few query processing methods to achieve good performance.

The most time consuming operation is the join between the communities and the graph, necessary to list the neighborhoods (the first query in Figure 4). If the nodes have enough main memory, we can speed it up with a replicated join: we replicate and index the **communities** table at each node. Then, we split the **graph** table, broadcast the partitions, and execute the join at each node. If this is not possible, we must chain two map-side joins. We cluster the tables **communities** and **graph** on the join keys (first **query1**, then **query2**), send each partition to a node, then perform the join at each node.

The following two operations (grouping and renaming) are much simpler, and can be executed in one map-reduce pass. The mappers emit the tuples with the key **query2**, then the reducers perform the aggregation and the renaming.

## 5. QUERY MATCHING

We now describe how to retrieve a community for a given query. Our approach is based on exact match: we find the community which contains the query terms exactly and in order, after lower-casing. Once we identified the relevant community, we run the expert search for all the related terms separately. We then union the results and rank the experts. This approach is purposely conservative, and it is straightforward to implement.

An advantage of production query logs is that terms often come in hundreds of variants, with alternative spellings and mistakes. This improves the flexibility of the matching at little computational cost.

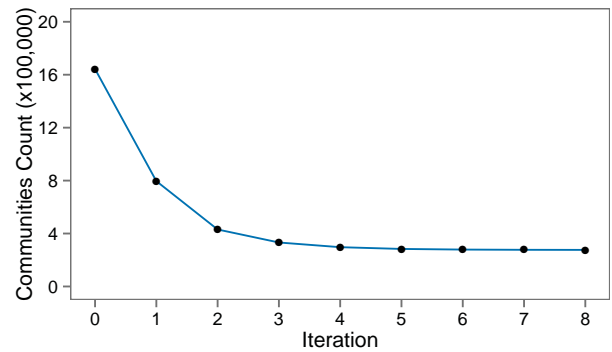


Figure 5: Convergence of the community detection algorithm.

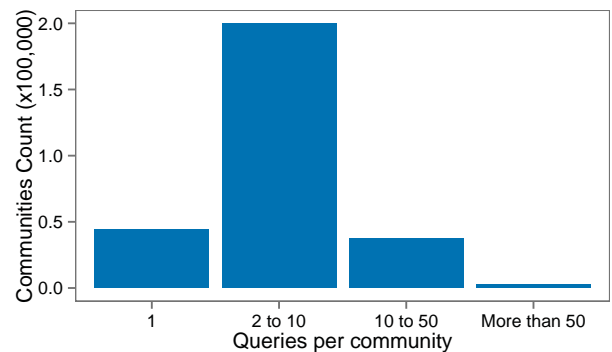


Figure 6: Distribution of the community sizes.

## 6. EXPERIMENTS AND EVALUATION

We now describe our experiments with *e#*. We first present the topics we extracted from a month of search queries. We then demonstrate *e#*'s effectiveness with a crowdsourcing study.

### 6.1 Topics of Expertise

We described how to extract topics of expertise with community detection. In this section, we illustrate our methodology with real world data. We use a full month of web search query logs (May 2014, US only, 998 GB). The graph we obtain contains approximately 60 million edges (1.45 GB). We describe our hardware setup in Section 6.3.

Figure 5 shows how many topics our algorithm finds after each iteration. We see that it starts with lots of tiny communities, then the count decreases very fast. Roughly, the procedure converges after 6 iterations. Figure 6 shows the distribution of the topic sizes. We observe that a large majority of communities contains between 2 and 10 queries (around 60%). We also found around 20% of orphans. We have very few communities with more than 50 items.

Figure 7 shows three groups of related keywords. To obtain this figure, we plotted the community which contains the term **49ers** (in dark blue), along with its three closest communities - in light blue, light green and dark green. The **49ers** community contains many non-trivial keywords: alternative spellings (**niners**), related activities

Set Name	Count	Examples
Sports	100	49ers, hernandez, buffalo bills, nascar, baltimore ravens
Electronics	100	bluetooth, ipad mini, garmin, xbox, vacuum cleaners
Finance	100	nasdaq, dow futures, msft, quotes, bloomberg
Health	100	scoliosis, asthma, diabetes, bmi, bulimia
Wikipedia	100	world war II, aashiqui 2, lycos, beyonce, albert einstein
Top 250	250	sarah palin, mapquest, honda, antonov225, saudi arabia

Table 1: Queries used for our crowdsourcing study.

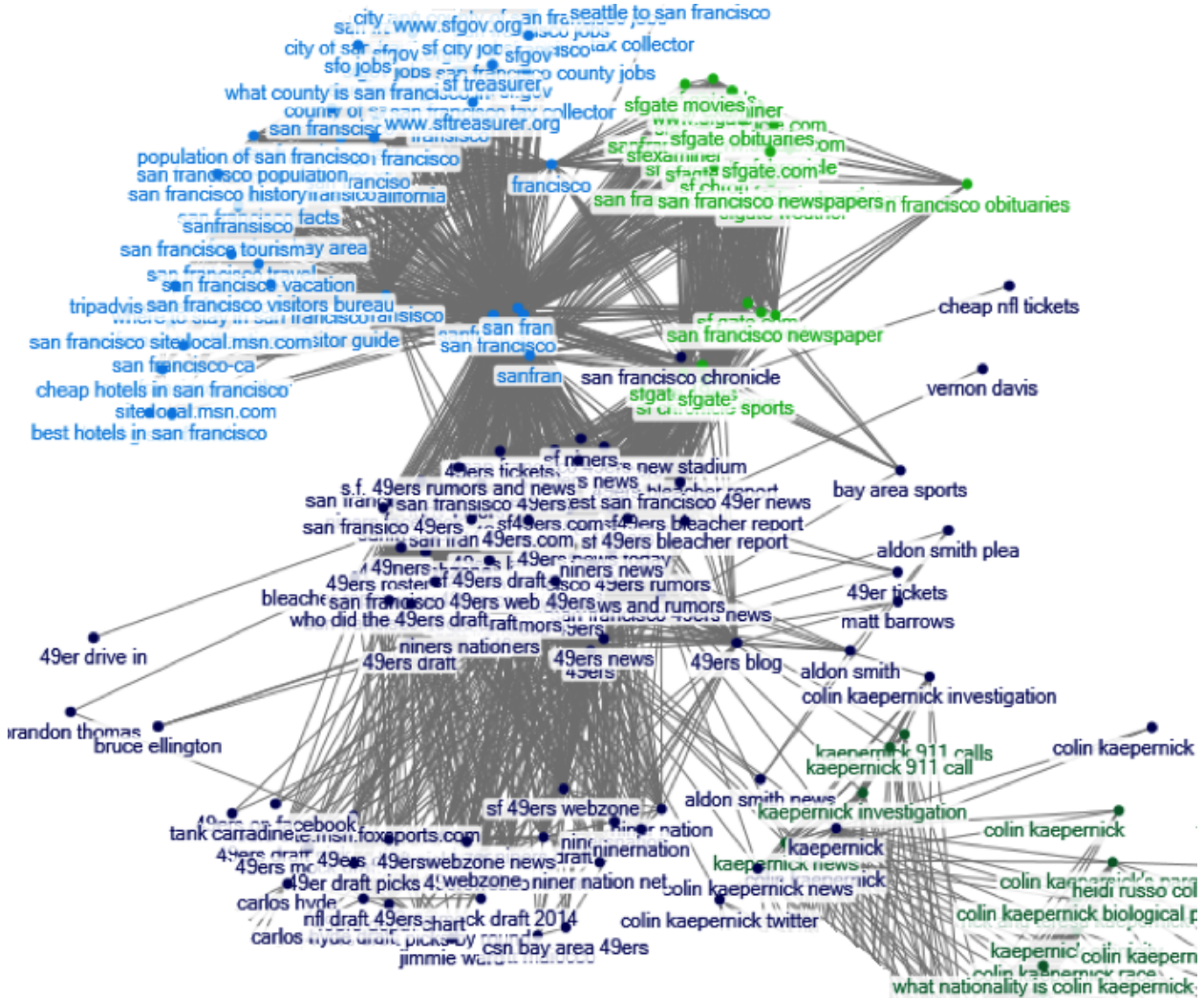


Figure 7: Graph and communities around the term “49ers”.

(49ers draft), or players (bruce ellington, vernon davis). We see that the query-log distance lets us detect semantic associations - we could not have detected these relations with a string-based distance. The three other groups contain topics which are somehow related to the 49ers, but not closely enough to be used in query expansion. The light blue community contains topics related to San Francisco tourism. This is not surprising, because the 49ers is the official team of the city. The light green one mentions “SF Gate”, which is a popular San Francisco newspaper (with a thick sports section). The dark green set focuses on Colin Kaepernick, a star player in the 49ers.

## 6.2 Impact on Expertise Retrieval

In this section, we demonstrate  $e\#$ 's effectiveness on Web data with a crowdsourcing study.

### 6.2.1 Experimental Setting

We compare two algorithms: Pal and Counts' algorithm, detailed in the second section of this paper, and  $e\#$ . To test the algorithms, we used queries which reflect popular interest in many different domains. Our assumption is that if a topic is popular on the Web in general, then it is likely to be popular on social media too. We used six sets, described in Table 1. The sets Sports, Electronics, Finance

Algorithm	Screen Name	Description	Verified	Followers
Baseline	SF49ersAllNews	All news about San Francisco 49ers	False	1,821
	Tim Kawakami	Tim Kawakami is a Mercury News sports columnist	True	45,924
	Matt Barrows	Matt (that’s me) covers the 49ers for The Sacramento Bee.	True	36,271
e#	Tre9er	49ers/NFL/Draft Tweets. Host of NinersNation.com ..	False	4,651
	NinersGoldRush	Your source for all breaking 49ers news ...	False	4,135
	Red n Gold	Huge #49ers fan. LET’S GO #NINERS!	False	537

**Table 2: Selected experts for the query 49ers. The flag Verified comes from Twitter, it attests the authenticity of a popular account.**

Algorithm	Screen Name	Description	Verified	Followers
Baseline	Huawei Club	Official Twitter handle for Huawei Club India...	False	1,589
	The Internet Patrol	The Internet Patrol is your source for Internet news.	False	179
	TekspezczDotnet	[...] where technology is not a passion, but an obsession.	False	87
e#	LuguLake	LuguLake believe tech shouldn’t drift apart people ...	False	3,215
	HavitAworldnet	HAVIT is specialized in PC and entertainment ...	False	879
	Bluesound	High-res Music. Wireless. Everywhere.	False	1,308

**Table 3: Selected experts for the query bluetooth speakers.**

Algorithm	Screen Name	Description	Verified	Followers
Baseline	Arthur Hogan	Chief Market Strategist -Dad-SOX Fan	False	409
	CNBC Newsroom	... recognized world leader in business news	True	92,014
	WorldEconRecon	Breaking Financial News   Investment Analysis	False	7,470
e#	ET Commodities	Your most trusted resource for timely news...	True	15,733
	MarketWatch	Tracking the pulse of the markets...	True	1,119,485
	The Exchange	We promote financial literacy, through Hip Hop.	False	3,830

**Table 4: Selected experts for the query dow futures.**

Algorithm	Screen Name	Description	Verified	Followers
Baseline	Amer. Diabetes Assn.	Leading the fight to #StopDiabetes ...	True	73,905
	Diabetes 101	Get Educated About Type 1 Diabetes.	False	3,829
	DiabetesNews.com	The Most Comprehensive Diabetes News ...	False	47,402
e#	amidiabetic (Stuart)	Stuart #T1 diabetic, trying to help others	False	58,451
	Eliot LeBow	Psychotherapist & Certified Diabetes Educator	False	1,813
	Diabetesview	We deliver the latest Diabetes news everyday	False	6472

**Table 5: Selected experts for the query diabetes.**

Algorithm	Screen Name	Description	Verified	Followers
Baseline	National Interest	... premier international-affairs magazines.	False	12,340
	Franz-Stefan Gady	Foreign Policy Analyst, Occasional Reporter ...	False	1,054
	EmperorTigerstar	...YouTube channel about maps and history	False	116
e#	ProjectBugle	The First World War Commemoration Project	False	36
	Wales Remembers	1914-1918 Sharing stories, history, information	False	1,166
	WWI in Africa	What happened in Africa should not stay in Africa.	False	392

**Table 6: Selected experts for the query World War I.**

Algorithm	Screen Name	Description	Verified	Followers
Baseline	Ron Devito	Sarah Palin supporter; LAN Infrastructure PM	False	171
	Sarah Palin News	Palin news and opinion from Palin-focused sites.	False	19,897
	Jer	A Governor @SarahPalinUSA Conservative Supporter!	False	821
e#	Sarah Palin News	All news about Sarah #Palin	False	1,651
	TheDean’sReport	.. issues of the day from an honest [...] point of view	False	7,108
	Truthyism News	Truthyism is a news and media organization	False	177

**Table 7: Selected experts for the query Sarah Palin.**

Data set	Baseline	e#	Improvement
Sports	0.87	0.96	10%
Electronics	0.89	0.98	10%
Finance	0.94	0.97	3.1%
Health	0.82	0.98	19%
Wikipedia	0.83	0.87	4.8%
Top 250	0.64	0.86	35%

**Table 8: Proportion of queries for which at least one candidate expert was found, before and after query expansion.**

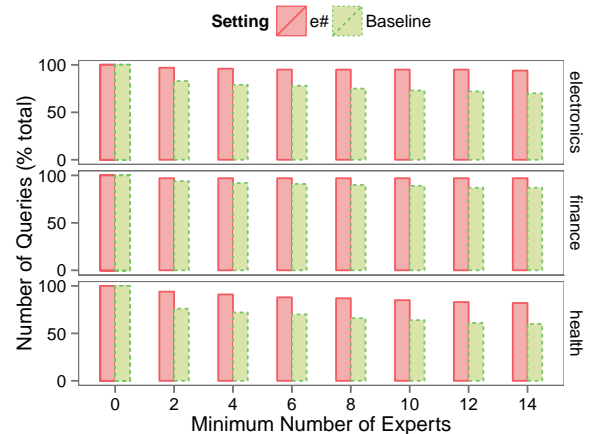
and **Health** contain the 100 most popular search terms from a commercial search engine, for each category. The set **Wikipedia** contains the title of the top 100 Wikipedia pages visited in 2014. It gives us an alternative view of popular interests. To increase diversity, we added the set **Top 250**, which contains the top 250 queries of a commercial search engine for July 2014. In total, we used 750 different queries.

We provide some examples of experts for the queries **49ers**, **bluetooth**, **dow futures**, **diabetes**, **World War I**, and **Sarah Palin** in Tables 2, 3, 4, 5, 6 and 7 respectively. We observe that the experts are diverse: among others, we notice journalists (**CNBC Newsroom**), individuals (**Arthur Hogan**), support groups (**Diabetes 101**) and local associations (**Wales Remember**).

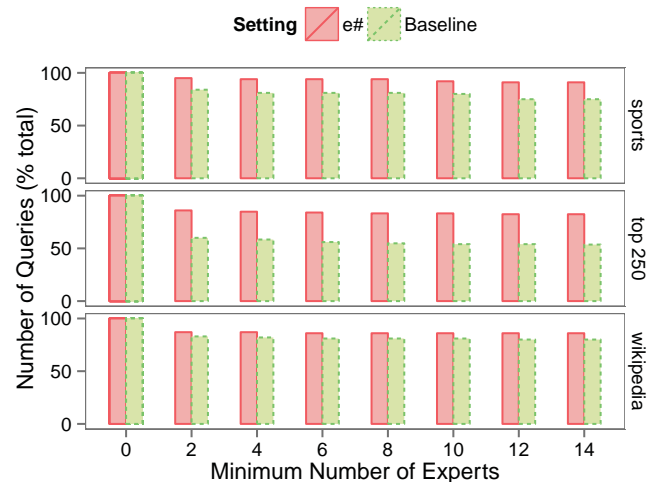
To assess the quality of the results, we were assisted by 64 crowdworkers, provided by a commercial third party. Evaluating expertise is a challenge for two reasons. First, the workers themselves must have some knowledge of the topic to recognize other experts. Second, the task is somehow subjective. We strived to incorporate these considerations in our experimental design. For each query, we generated up to 15 experts per algorithm and interleaved the results. To avoid worker fatigue, we chunked the resulting sets into smaller sets of at most 6 experts. We also randomized the order to prevent the position bias. We asked the workers to spot “non-experts”, that is, accounts from which they could *not* get any objective information about the topic of interest. We chose to exclude “non-experts”, rather than validate experts, because we assumed that the former task requires less background knowledge than the latter. We gave examples, encouraged high response times, presented links to the Twitter pages, and gave crowdworkers the option to ignore questions for which they were not confident. We filtered spammers with trivial preliminary questions. We set up the experiments such that each expert was reviewed by 3 different workers, and aggregated the results with majority voting.

### 6.2.2 Impact on Recall

In Table 8, we present the impact of query expansion on the size of the result sets. We show the number of queries for which at least one expert was found, before and after expansion. We note that in all six cases, we obtain a neat improvement. We notice the smallest performance gain with the **Finance** set: the baseline results are already very high, and **e#** only brings a 3% improvement. We observe the most dramatic effects with **Top 250**: **e#** answers 35% more queries. This result is not surprising: we trained **e#** on the search log from which the queries come from, therefore we expected it to perform well.

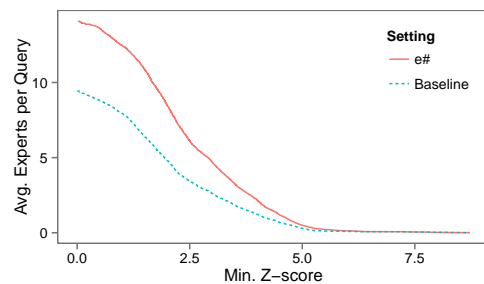


(a) Sets electronics, finance, and health.



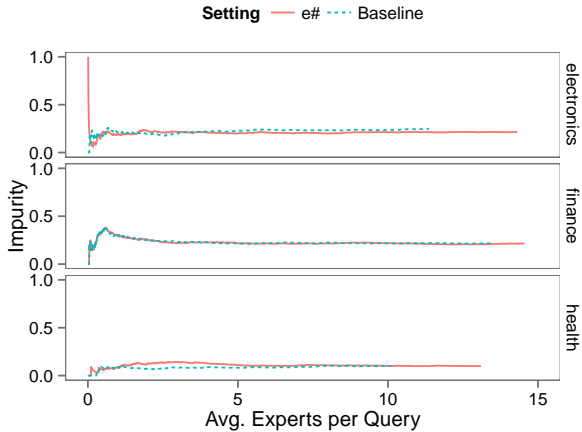
(b) Sets commerce, top 250, and Wikipedia.

**Figure 8: Effect of the query expansion on the number of experts per query.**

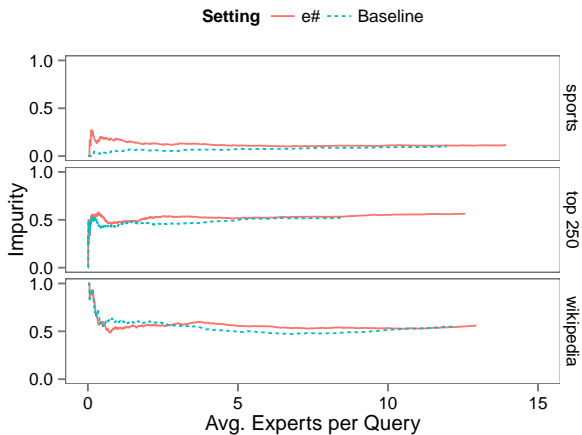


**Figure 9: Impact of the z-score on the number of experts for the set Top 250.**

Figure 8 present a finer view of **e#**’s impact on the number of experts retrieved for each query set. It presents the number of queries for which the algorithms return  $n$  experts or more, with  $n$  varying between 0 and 14. For instance, the leftmost bars show that 100% of queries have 0 hits or more. The rightmost bars show the number of queries for which our algorithms found 14 experts or more. In almost



(a) Sets electronics, finance, and health.



(b) Sets commerce, top 250, and Wikipedia.

**Figure 10: Size vs. quality trade-off. The impurity is the proportion of results marked as non relevant by the judges.**

every cases, we observe that query expansion improves the number of experts found (in average of about 10%, up to 30%). We conclude that our expansion strategy works: it does improve recall.

### 6.2.3 Impact on Precision

The major challenge in query expansion is to enhance recall without hurting precision. Indeed, query expansion may weaken the quality of the retrieved experts. This degradation has multiple sources: it comes from noise in the corpora, noise in the clustering, or errors in the expansion (e.g., disambiguation problems). In this section, we study the extent of this penalty.

Before we present our results, recall that our algorithm needs to be tuned. The users must choose a minimum z-score, under which the experts are rejected. This threshold defines a trade-off between precision and recall: a low value will lead to many low quality experts, a high value will lead to a few excellent experts. We illustrate this effect for **Top 250** in Figure 9.

Figure 10 compares the quality of the experts found for different levels of recall. For a given number of experts per query, it returns the *impurity*, that is, the proportion of ex-

Step	VMs	Runtime	Read	Write
Extraction	65	38 min.	998 GB	2.6 GB
Clustering	65	2 hours	2.6 GB	94 MB
Expansion	1	< 100 ms.		
Detection	1	< 1 sec.		

**Table 9: Resource consumption for one iteration (September 2015)**

perts marked as non relevant by the crowdworkers. We observe that the difference between the algorithms is very subtle. It is maximal for the first few results of the set **Sports**, it is almost imperceptible for **Finance** and **Health**. In the dataset **Electronics**, *e#* performs slightly better than its competitor. In conclusion, the accuracy penalty incurred by *e#* is minimal, if not negligible. We can improve expert detection recall with minimal losses in precision.

## 6.3 Resource Consumption

We now provide hints about the resource consumption of *e#*. The offline part of our system runs weekly on a production cluster. Table 9 reports statistics for one iteration (September 2015). It must be noted that our environment is completely virtualized, and thus performance depends heavily on availability: a relational operator can use between one and hundreds of virtual machines, depending on its nature and the cluster’s workload. Besides, we have no control over the underlying hardware. The data center comprises servers with 12 x86-64 cores, between 32 and 64 GB main memory and SSD drives with 1 to 3 TB. But each of these servers can handle dozens of virtual machines. The collection of expertise topics produced by *e#* weighs about 100 MB. We store and index it in SQL Server 2014, which allows us to query it in a few milliseconds. We refer the reader to Pal and Counts [14] for more details about the final query detection.

## 7. RELATED WORK

Our work bridges two fields of study: expertise retrieval and query expansion. It is, to our knowledge, the first attempt to augment expert search with an external thesaurus.

### 7.1 Expertise Retrieval.

Researchers have been interested in detecting experts for several years now, in particular since the problem was introduced at TREC in 2005 [2]. An extensive review was written by Balog et al. in 2012 [4]. The two oldest approaches are the *candidate model* and the *document model*. In the candidate model, a textual profile is created offline for each candidate (for instance, by aggregating all the documents authored by the candidate). Then, these profiles are ranked with traditional IR model [7]. The document model operates the other way around. First, a set of relevant documents is identified for the query. Then, the algorithms find the associated people and rank them according to the relevance of the documents and their degree of association [15]. More recently, authors have proposed alternative models. Discriminative models such as the Arithmetic Mean Discriminative model are robust and they can integrate heterogeneous, arbitrary features [9]. Graphs models have also gained popularity. For instance, Serdyukov et al. have shown the effectiveness of random walks on “expertise graphs” [18].

Jianshu et al. is, to our knowledge, the first team to have published work about expert detection on Twitter [20]. Their system is based on a graph describing the topical similarity between the users. To detect authorities, they run a variant of PageRank on this graph for each topic. An alternative was proposed by Pal et al. [14]. We introduce a production version of their framework. Recent work has studied how to incorporate location data into expertise retrieval, focusing on “local” experts rather than “general” experts [6].

As our framework is based on query expansion, we do not compete with any of these approaches. Our system can work with any Expertise Retrieval system.

## 7.2 Query Expansion.

Authors have proposed query expansion methods for decades in document search. Researchers were already building “classes of similar terms” to improve search before 1960 [16]. To measure the proximity between terms, they used co-occurrence in the training documents. Qiu et al. proposed a notable improvement with “concept-based” query expansion [16]. The main idea was to represent the terms by points in a vector space, where each dimension represents a document. From this representation, it was possible to build a so-called similarity thesaurus. Recent publications have shown that external source of knowledge can also improve search, such as WordNet [19] or ontologies [12].

Our work differs from all of the above because we use a query log. This source of data is relevant for two reasons. First, it is relatively easy to manipulate: we do not have to process the whole collection of documents (in our case, this would mean the whole Web). Second, it is constantly renewed; thus, we believe that the microblogging vocabulary is better captured by this source than by existing ontologies. Other authors have used query logs for query expansion, such as Cui et al. [8]. They observe that if a set of documents is frequently selected for a certain keyword, then their terms are probably strongly correlated to this keyword. However, they still use the underlying documents.

It seems that little work was presented on query expansion in the context of expertise retrieval. Macdonald et al. have mostly focused on local query expansion, i.e., using top ranked documents for pseudo-relevance feedback [11]. Balog et al. have presented ways to incorporate external evidence of expertise into language models [3]. These lines of work are complementary to ours, but they are not overlapping.

## 8. CONCLUSION

We introduced an approach for expertise detection on social media that emphasizes recall. We showed that finding related domains of interests can be expressed as a graph community detection problem. We presented a parallelized implementation and showed the evaluation results on a large Twitter data set. Our findings demonstrated that  $e\#$  can increase the number of experts without losing quality. Future work includes expanding into other social networks such as Quora and Facebook, and exploring different community detection paradigms.

## 9. ACKNOWLEDGMENTS

Thibault Sellam performed this work during an internship at Microsoft Corporation. His PhD is supported by the Dutch national program COMMIT.

## 10. REFERENCES

- [1] R. Baeza-Yates and A. Tiberi. Extracting semantic relations from query logs. *Proc. SIGKDD*, page 76, 2007.
- [2] P. Bailey, A. D. Vries, N. Craswell, and I. Soboroff. Overview of the TREC 2007 Enterprise Track. *TREC*.
- [3] K. Balog and M. de Rijke. Non-local evidence for expert finding. In *Proc. CIKM*, pages 489–498, 2008.
- [4] K. Balog, Y. Fang, M. de Rijke, P. Serdyukov, and L. Si. Expertise Retrieval. *Foundations and Trends in Information Retrieval*, pages 127–256, 2012.
- [5] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB*, pages 1265–1276, 2008.
- [6] Z. Cheng, J. Caverlee, H. Barthwal, and V. Bachani. Who is the barbecue king of texas?: a geo-spatial approach to finding local experts on twitter. In *Proc. SIGIR*, pages 335–344, 2014.
- [7] N. Craswell, D. Hawking, A.-M. Vercoustre, and P. Wilkins. Panoptic expert: Searching for experts not just for documents. In *Ausweb Poster Proc.*, 2001.
- [8] H. Cui, J.-R. Wen, J.-Y. Nie, and W.-Y. Ma. Probabilistic query expansion using query logs. In *Proc. WWW*, pages 325–332, 2002.
- [9] Y. Fang, L. Si, and A. P. Mathur. Discriminative models of integrating document evidence and document-candidate associations for expert search. In *Proc. SIGIR*, pages 683–690, 2010.
- [10] S. Fortunato. Community detection in graphs. *Physics Reports*, pages 75–174, 2010.
- [11] C. Macdonald and I. Ounis. Expertise drift and query expansion in expert search. In *Proc. CIKM*, pages 341–350, 2007.
- [12] R. Navigli and P. Velardi. An analysis of ontology-based query expansion strategies. In *Proc. ECML, Workshop on Adaptive Text Extraction and Mining*, pages 42–49, 2003.
- [13] M. E. J. Newman. Modularity and community structure in networks. *Proc. National Academy of Sciences*, 2006.
- [14] A. Pal and S. Counts. Identifying topical authorities in microblogs. *Proc. WSDM '11*, page 45, 2011.
- [15] D. Petkova and W. B. Croft. Hierarchical language models for expert finding in enterprise corpora. *IJAIT*, pages 5–18, 2008.
- [16] Y. Qiu and H.-P. Frei. Concept based query expansion. In *Proc. SIGIR*, pages 160–169, 1993.
- [17] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader. Parallel community detection for massive graphs. In *Parallel Processing and Applied Mathematics*, pages 286–296, 2012.
- [18] P. Serdyukov, H. Rode, and D. Hiemstra. Modeling multi-step relevance propagation for expert finding. In *Proc. CIKM*, pages 1133–1142, 2008.
- [19] E. Voorhees. Query Expansion using Lexical-semantic Relations. In *Proc. SIGIR*, pages 61–69, 1994.
- [20] J. Weng, E.-P. Lim, J. Jiang, and Q. He. Twitterrank: finding topic-sensitive influential twitterers. In *Proc. WSDM*, pages 261–270, 2010.



# DECT: Distributed Evolving Context Tree for Mining Web Behavior Evolution

## Industrial Paper

Xiaokui Shu<sup>\*</sup>  
Department of Computer  
Science, Virginia Tech  
Blacksburg, VA USA  
subx@cs.vt.edu

Nikolay Laptev  
Yahoo! Labs  
701 First Avenue  
Sunnyvale, CA USA  
nlaptev@yahoo-inc.com

Danfeng (Daphne) Yao  
Department of Computer  
Science, Virginia Tech  
Blacksburg, VA USA  
danfeng@cs.vt.edu

### ABSTRACT

Internet user behavior models characterize user browsing dynamics or the transitions among web pages. The models help Internet companies improve their services by accurately targeting customers and providing them the information they want. For instance, specific web pages can be customized and prefetched for individuals based on sequences of web pages they have visited. Existing user behavior models abstracted as time-homogeneous Markov models do not provide efficient support for modeling user behavior variation through time. This paper presents DECT, a scalable time-variant variable-order Markov model. DECT digests terabytes of user session data and yields user behavior patterns through time. We realize DECT using Apache Spark. Our implementation is being open-sourced and we deploy DECT on top of Yahoo! infrastructure. We demonstrate the benefits of DECT with anomaly detection and ad click rate prediction applications. DECT enables the detection of higher-order path anomalies that are masked out by existing models. DECT also provides insights into ad click rates with respect to user visiting paths.

### Keywords

Markov Model; Context Tree; Distributed Computing; Time Series; Anomaly Detection; Link Prediction

## 1. INTRODUCTION

Understanding Internet user behavior is a key to the optimization of Internet services and software. A web browser or server can prefetch or prepare webpages for a user, if the system knows the user will visit the page in the short future [23]. A service provider can customize clickable ads for

<sup>\*</sup>The work was mostly done while the first author was an intern at Yahoo! Labs.

a user, if the provider knows which ads the user is likely to click [17]. Service providers can also design search engines to fit human browsing dynamics [13].

*Markov model* (first-order, time-homogeneous) is commonly adopted for Internet user behavior modeling [5]. It is, however, amnesiac; the probability of the next user visit is purely based on the current status of the user. Higher-order Markov models cure the amnesia issue by digesting historical visiting sites of users [15]. Variable-order Markov models improve higher-order Markov models by pruning away unnecessarily higher-order paths for space saving purposes [3].

While the community has developed a string of advanced Markov models to describe Internet user behavior patterns, one strong assumption is constantly kept in all existing models: user behavior patterns do not change over time.

The above assumption, however, does not hold in the real world. New products are releasing; UI of existing websites are changing; cyber attacks occur; breaking news happen. *The Internet is evolving, and the observed Internet user behavior patterns should reflect the changes.*

This paper presents DECT (distributed evolving context tree), a time-variant model for efficiently describing Internet user behavior patterns and their changes through time. DECT is a time-variant variable-order Markov model. It improves the state of the art variable-order Markov models by releasing its assumption of static time-invariant user behavior patterns. DECT is designed to handle large volumes of user session data and can be efficiently constructed via distributed computing.

Time-variant variable analysis, e.g., visit counts of services, has been widely used in industry to detect anomalies like attacks, failures, and bugs. However, these commonly used variables are stateless or only first-order with respect to Markov models.

In contrast, DECT enables higher-order time-variant visiting path analysis. DECT yields both regular time series of individual path visiting probabilities and high-dimensional time series for a set of related paths, e.g., paths that share the same prefix. We demonstrate in Section 4.1 that DECT can produce deep signals for anomaly detection. It helps reveal stealthy attacks, e.g., application layer DDoS attack [21] and browsing mimicry attack [22]. First-order Markov models, in contrast, could mix these signals into noises. We demonstrate in Section 4.2 that DECT distinguishes ad click probability variations based on historical web pages a user visits, while existing first-order prediction is blind to differ-

**Table 1: Symbols, Terms and Definitions**

Term	Definition
$s$	site the primitive unit to record user behavior (e.g., a URL, a web service, a website)
$E$	session a sequence of sites that a user visits (every session has a beginning and an end)
$\bar{p}$	path a substring of a session
$\tau$	target the next site a user is going to transit to
$\bar{c}$	context a sequence of visited sites prior to $\tau$

ent types of users who come from diverse paths.

The contributions of our work are summarized as follows.

- We design DECT to digest large volumes of user session data and construct time-variant user behavior models in a distributed manner.
- We explain the benefits of a time-variant user behavior model and showcase application examples of DECT in anomaly detection and ad click prediction.
- We realize DECT using Apache Spark and demonstrate its performance processing terabytes of real-world user session datasets.

## 2. DECT

We discuss two major features of DECT in this section: *variable-order* and *time-variant*. These features are realized through a *flattened context tree*, which embeds time series information in its leaf nodes.

### 2.1 Definitions and Overview

We study user behavior in terms of their visiting paths on the Internet. A (desktop or mobile) user session is recorded as a sequence of visits to a set of Internet *sites* – resources that users are visiting. Sites vary from specific URLs to domains<sup>1</sup>. We define related variables that are used to express user visiting paths in Table 1.

Higher-order Markov models have been proved effective in modeling static user behavior [5]. Given a path  $\bar{p}$ , a higher-order Markov model can be trained to predict the last transition of  $\bar{p}$  based on previously visited sites in  $\bar{p}$ . In this setup, we refer to the last transition in  $\bar{p}$  as the target  $\tau$ , and the sites prior to  $\tau$  as the context  $\bar{c}$ .

The key question we aim to answer is how target transition probabilities change over time. When considering the higher-order Markov model as a weighted directed graph  $G_M = (V_M, E_M)$ , we construct our model to keep track of:

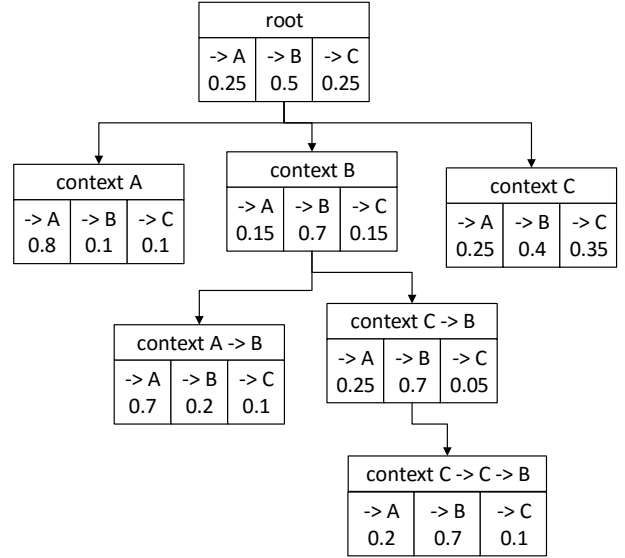
- change of  $V_M$ : new and obsoleted nodes
- change of  $E_M$ : transition matrix variation

DECT enables the tracking of both changes, and it provides two features to handle large amounts of data and mitigate exponential space explosion caused by regular higher-order Markov model:

- variable-order context-target probability
- fine-grained parallel path computing and pruning

We realize the two features through *flattened context tree* – a new parallel and concise data structure for building distributed time-variant variable-order Markov model.

<sup>1</sup>The granularity of sites is a data collection parameter.



**Figure 1: Example of regular context tree (pruned to represent a variable-order Markov model).**

### 2.2 A New Context Tree Structure

Context tree is a common data structure for constructing variable-order Markov model [2,5,6]. We first give a brief description of regular context tree and its operations. Then we present our flattened context tree structure for distributed tree construction and fine-grained parallel pruning.

#### 2.2.1 Regular Context Tree

A regular context tree  $T_C$  is a  $k$ -ary tree where  $k$  is the number of all possible sites.  $T_C$  records static transition probabilities of any target given a limited length context. The limited context length is the depth of the tree.

We give an example of a regular context tree in Figure 1. A site  $s \in \{A, B, C\}$ . Each node maps to one context that is recorded. A node  $n_{\bar{c}}$ , which corresponds to context  $\bar{c} = (s_{-y}, \dots, s_{-2}, s_{-1}, s_0)$ , positions at depth  $y$  in  $T_C$ . Its parent is the node with context  $\bar{c}' = (s_{-(y-1)}, \dots, s_{-2}, s_{-1}, s_0)$  at depth  $y - 1$ . Its children are nodes with context  $\bar{c}_{c_i} = (s_{-(y+1)_i}, \dots, s_{-2}, s_{-1}, s_0)$  at depth  $y + 1$  where  $0 \leq i \leq \xi_{\bar{c}} \leq k$ .  $\xi_{\bar{c}}$  is the total number of children of  $\bar{c}$ .

$n_{\bar{c}}$  stores the target probability distribution with respect to the context  $\bar{c} = (s_{-y}, \dots, s_{-2}, s_{-1}, s_0)$ , i.e.,  $P(\tau_j | \bar{c})$  where  $0 \leq j \leq \kappa_{\bar{c}} \leq k$ .  $\kappa_{\bar{c}}$  is the total number of reachable targets given the context  $\bar{c}$ .

**Pruning** Pruning a regular context tree of a higher-order Markov model results in a variable-order Markov model. The standard  $T_C$  pruning strategy is a bottom-up process: pruning away  $n_{\bar{c}}$  if both criteria are satisfied:

- $n_{\bar{c}}$  is a leaf node.
- The distance, e.g., KL divergence, between the target probability distribution of  $n_{\bar{c}}$  and that of its parent node  $n_{\bar{c}'}$  is less than a predefined threshold  $\mathcal{T}_{pv}$ .

**Time-variant capability** Unfortunately, regular context tree is designed to accommodate static transition probabilities. The transition matrix of the corresponding Markov model is fixed when the model is built. Updating the tree to reflect a time-variant model is expensive. It requires to

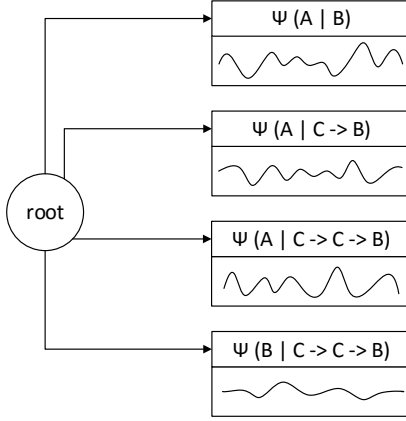


Figure 2: Example of flattened context tree with change of transition probabilities in time series.

Table 2: Flattened Context Tree vs. Regular Context Tree

	FCT*	RCT*
Node semantics	(context, target)	context
Tree depth	1	highest order
Probability time series	embedded	none

\*FCT/RCT: flattened/regular context tree

recalculate node probabilities and reevaluate previous pruning procedures for pruned nodes.

### 2.2.2 Flattened Context Tree

We present a *flattened context tree*. We define the pruning strategy to facilitate distributed tree operations for our time-variant Markov model. We prove that our parallel pruning strategy preserves the tree structure: one branch can be pruned only if all its children are pruned.

A flattened context tree  $T_F$  only has a depth of two: depth 0: root, and depth 1: all data nodes. Each depth-1 node  $n_{\bar{p}}$  corresponds to a path  $\bar{p} = (\bar{c}, t) = (s_{-y}, \dots, s_{-2}, s_{-1}, s_0, t)$  and it records a time series of transition probability  $\Psi(\tau|\bar{c}) = \{P_t(\tau|\bar{c}) : t \in T\}$ . In comparison, a node  $n_{\bar{c}}$  in  $T_C$  stores the distribution of transition probabilities according to context  $\bar{c}$ . Table 2 shows the most significant differences between our flattened context tree and regular context tree. We illustrate the structure of flattened context tree in Figure 2.

The advantage of the flattened tree structure is that each node can be processed independently of other nodes, which enables fine-grained parallel probability computing and pruning for each  $(\bar{c}, \tau)$  pair. Furthermore, different nodes in  $T_F$  can be processed on a different processing unit in a distributed manner to scale out the process.

**Pruning** The purpose of pruning is to transform a higher-order Markov model to a variable-order one. Pruning of  $T_F$  is performed for individual nodes in parallel. A node  $n_{\bar{p}}$  is pruned away if  $\bar{p}$  is rarely visited through a large segment of the monitored time period. The criteria can be measured as the total number of *path visits* or the total number of *path appearances* during the overall monitored time period. The latter yields *True* or *False* in each inspection window and

sums the total number of *True* for the overall time period.

**THEOREM 1.** In a flattened context tree  $T_F$ , a node  $n_{\bar{p}}$  records the target transition probability time series of path  $\bar{p}$ .  $\bar{p}'$  denotes a suffix string of  $\bar{p}$ .  $n_{\bar{p}}$  in  $T_F$  can be pruned if  $n_{\bar{p}'}$  (node corresponding to  $\bar{p}'$ ) can be pruned.

**PROOF.** If a path  $\bar{p}$  is visited, its suffix path  $\bar{p}'$  is visited. And two different paths  $\bar{p}_1$  and  $\bar{p}_2$  can share the same suffix path  $\bar{p}'$ . So  $V(\bar{p}') \geq V(\bar{p})$  where  $V(\bar{p})$  is the number of path  $\bar{p}$  visits. Given  $\mathcal{T}_{pv}$  as the pruning threshold for path visits,  $V(\bar{p}) < \mathcal{T}_{pv}$  holds if  $V(\bar{p}') < \mathcal{T}_{pv}$ .  $\square$

If one restructures a flattened context tree  $T_F$  back to a regular context tree  $T_C$ , Theorem 1 guarantees that all children of a branch node are pruned away before the branch node is pruned. It is consistent with the standard pruning strategy presented in Section 2.2.1.

**Time-variant capability** Time series information is embedded into each node  $n_{\bar{p}}$  in  $T_F$ , so  $T_F$  reflects the change of the corresponding Markov model through time. Change of  $E_M$  (discussed in Section 2.1) is distributed across  $\Psi(\tau|\bar{c})$  in each node. Changes of  $V_M$  (discussed in Section 2.1) are also stored in new or obsolete nodes if not pruned.

## 2.3 Growing the Flattened Context Tree

A flattened context tree  $T_F$  grows through time. We use a sliding window  $w$  to aggregate sessions through time and yield transition matrices of our time-variant Markov model at different times. Time series yielded from nodes in the flattened context tree are extended when new sessions are consumed and the tree has grown.

### 2.3.1 Session Batch

*Session batch* is a set of sessions. It is the smallest sliding unit for  $w$ . Sessions are batched according to the timestamp of its first visited site. A session may last across several batch time periods, but the *entire* session is recorded only once in the first batch it appears<sup>2</sup>. The timestamp of the session batch is the start of the session batch.

Session batches do not interference with each other, and they can be preprocessed in parallel to facilitate the tree construction. In each session batch:

- i) All paths at different lengths are identified (through  $n$ -gram with variable- $n$ ) and parsed into tuples  $(\bar{c}, \tau)$ .
- ii) The counts of each tuple are accumulated.
- iii) A set of 4-tuples  $(\bar{c}, \tau, t_b, \eta_{(\bar{c}, \tau)})$  is yielded as the session batch digest where  $t_b$  is the session batch timestamp and  $\eta_{(\bar{c}, \tau)}$  is the count of tuple  $(\bar{c}, \tau)$  in the batch.

### 2.3.2 Sliding Window

The sliding window  $w$  covers a fixed number of session batches and each slide takes in a new session batch and abandons the earliest batch in the previous  $w$ .

In each sliding position, three operations are performed:

- i) 4-tuples  $(\bar{c}, \tau, t_w, \eta_{(\bar{c}, \tau)})$  are accumulated from session batch digests where  $t_w$  is the timestamp of the earliest session batch in the window.
- ii) 3-tuples  $(\bar{c}, t_w, \eta_{\bar{c}})$  are accumulated from the 4-tuples where  $\eta_{\bar{c}}$  is the count of context  $\bar{c}$  in the window.

<sup>2</sup>Our current design does not support streaming because it requires the entire user session to finish before it can be sessionized and batched.

Table 3: Description of Spark Runtime Stages of our DECT Prototype

Functionality	#(SS)	Description
Input handling	1	reading user session data from HDFS
Modeling	19	$n$ -gram generation, batching, in-window aggregation, probability calculation, pruning
Time series generation	9	assembling time series and storing them onto HDFS
Statistics generation	3	generating and yielding statistics throughout the entire processing procedure

#(SS): number of Spark runtime stages

- iii) A set of 4-tuples  $(\bar{c}, \tau, t_w, P(\tau|\bar{c}))$  is yielded as the window digest where  $P(\tau|\bar{c}) = \frac{\eta(\bar{c}, \tau)}{\eta_{\bar{c}}}$ .

### 2.3.3 Flattened Context Tree Update

Digests of  $w$  at different positions are aggregated according to tuple  $(\bar{c}, \tau)$  in the window digest. Each tuple forms a depth-1 node  $n_{\bar{p}}$  in the flattened context tree  $T_F$ . The time series at each node  $(\Psi(\tau|\bar{c}))$  at node  $n_{\bar{p}}$  where  $\bar{p} = (\bar{c}, \tau)$  is extended when a new window position is processed.  $w$  at different positions of a single node can be computed in parallel if all session batches are known.

Pruning of  $T_F$  can be performed at any time based on the generated time series at nodes. As discussed in Section 2.2.2, pruning is performed at nodes that are rarely visited. If  $n_{\bar{p}}$  is pruned,  $\Psi(\tau|\bar{c})$  prior to the pruning action is lost.  $n_{\bar{p}}$  can be added back to  $T_F$  if it becomes popular in the future, but without the segment of  $\Psi(\tau|\bar{c})$  when it is pruned away.

### 2.3.4 Time Series Production and its Applications

$T_F$  records the user behavior pattern evolution and it can be consumed by a variety of time series analysis tools (e.g., anomaly detection, path prediction).

**User behavior evolution data generation.** The user behavior evolution data are yielded in three major forms for post-processing and analytics:

- $\Psi(\tau|\bar{c})$ : individual time series for each  $(\bar{c}, \tau)$  pair
- $\{\Psi(\tau_i|\bar{c}) \mid 0 \leq i \leq \kappa_{\bar{c}}\}$ : high-dimensional time series for all targets of a context  $\bar{c}$  where  $\kappa_{\bar{c}}$  is the total number of reachable targets of context  $\bar{c}$ .
- $\{\Psi(\tau|\bar{c}_i) \mid 0 \leq i \leq \varrho_{\tau}\}$ : a collection of time series for a target  $\tau$  where  $\varrho_{\tau}$  is the total number of contexts which can reach the target  $\tau$ .

$\{\Psi(\tau_i|\bar{c}) \mid 0 \leq i \leq \kappa_{\bar{c}}\}$  forms a high-dimensional time series where each dimension is  $\Psi(\tau_i|\bar{c})$ . We write  $\{\Psi(\tau|\bar{c}_i) \mid 0 \leq i \leq \varrho_{\tau}\}$  as a collection because each  $\Psi(\tau|\bar{c}')$  is not independent of  $\Psi(\tau|\bar{c})$  where  $\bar{c}'$  is a suffix of  $\bar{c}$ , yet it is quite useful to compare  $\Psi(\tau|\bar{c})$  with  $\Psi(\tau|\bar{c}')$ .

**User behavior evolution data analysis.** Three most important components of an aggregated user behavior time series are *trend*, *seasonality*, and *irregular component*.

**Trend**  $\Psi_T(\tau|\bar{c})$  describes long-term movement without calendar related and irregular effects.

**Seasonality**  $\Psi_S(\tau|\bar{c})$  characterizes regular cyclic movements influenced by seasonal factors.

**Irregular component**  $\Psi_I(\tau|\bar{c})$  records non-systematic and unpredictable component(s) after trend and seasonal components are removed from the signal.

Many user behavior time series  $\Psi(\tau|\bar{c})$  can be very well

decomposed into the three components as described in (1).

$$\Psi(\tau|\bar{c}) = \Psi_T(\tau|\bar{c}) + \Psi_S(\tau|\bar{c}) + \Psi_I(\tau|\bar{c}) \quad (1)$$

$\Psi_S(\tau|\bar{c})$  can be further divided into daily and weekly seasonal components as found in our experiments.

Two major applications of our model are described next.

**Anomaly Detection** aims to discover anomalous user behavior with respect to specific visiting paths. A *spike* or a *ravine* in a time series could indicate breaking news, flash crowds, Denial-of-Service attacks, service failures, etc. A plateau appearing in the *trend* component of a time series may indicate a persistent attack or a test for a new feature. User behavior evolution data  $\Psi(\tau|\bar{c})$  and  $\{\Psi(\tau_i|\bar{c}) \mid 0 \leq i \leq \kappa_{\bar{c}}\}$  are yielded for anomaly detection.

**Ad Click Prediction** is an application to predict how likely a user will click an ad on her current visiting site given her visiting path of the current session. Different visiting paths leading to the same site may give different ad click rates, and the probability trends of different paths may be different. User behavior evolution data  $\Psi(\tau|\bar{c})$  and  $\{\Psi(\tau|\bar{c}_i) \mid 0 \leq i \leq \varrho_{\tau}\}$  are yielded for ad click prediction.

## 3. IMPLEMENTATION

We implement DECT via Apache Spark using Scala. We deploy DECT on top of Yahoo! infrastructure to support anomaly detection and other services on Yahoo! network.

Our DECT implementation is open-sourced on github [19]. The implementation takes advantage of scalable and robust transformations on resilient distributed dataset (RDD) in Spark, e.g., `mapValues` and `join`. DECT compiles to 32 Spark stages at JVM runtime (shown in Table 3).

Our implementation consumes plaintext session data stored on HDFS where each line records a user session<sup>3</sup>. A user session consists of a timestamp  $t_s$  and a sequence of visited sites  $E = \{s_0, s_1, \dots\}$ . DECT digests the plaintext session data and yields two types of information: *i*) time series harvested from the flattened context tree, stored on HDFS, and *ii*) statistics on processed data, e.g., total number of  $i$ th-order time series, printed to Spark log.

Our realization is optimized from the following aspects:

1. Session and path ( $n$ -gram) data are aggregated at early stages to minimize unnecessary duplicate data processing. For example, before generating and counting  $n$ -grams in each session  $E$ , same  $E$  with the same session batch timestamp  $t_b$  are counted and deduplicated.
2. Compact data structures are used to reduce storage and transmitting overhead, e.g., a context as a single JVM string, instead of an array of sites (JVM strings).

<sup>3</sup>We employ a Pig script to retrieve, sessionize and store raw user event data from HCatalog onto HDFS prior to DECT.

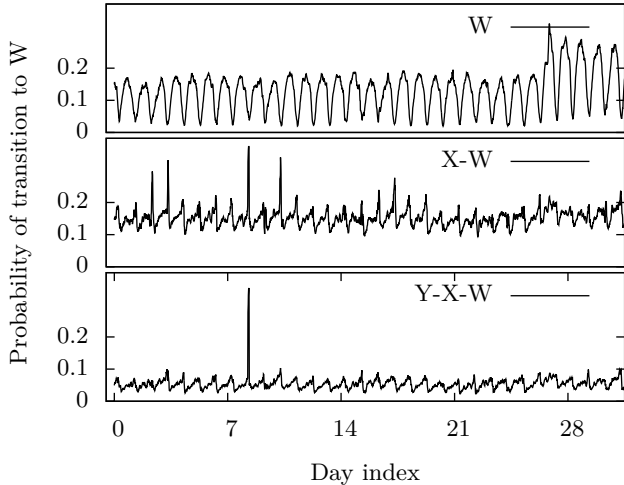


Figure 3: Higher-order path time series anomaly detection<sup>5</sup>.

3. Job parameters are broadcasted, e.g., string splitter.
4. Partitioning strategies are manually specified to reduce data movement among worker nodes.

## 4. EVALUATION

We conduct experiments on two Yahoo! daily user session datasets to answer the following key questions:

1. What do we benefit from our time-variant user behavior model over existing static stochastic models?
2. Is our design scalable to handle enterprise-wide tasks consisting of billions of sessions?

We evaluate DECT with all data collected through Yahoo! data highway. We analyze Yahoo! user activities within the first half of 2015. In the evaluation, we focus on user sessions within a single product, e.g., Yahoo! mail. Visits to alien sites during sessions are ignored<sup>4</sup>.

We process user session data of Yahoo! US websites (English version) within two products separately: Yahoo! mail and Yahoo! finance<sup>5</sup>. Each site in a session is roughly a view in the model-view-controller (MVC) web architecture, and it has a unique URL.

### 4.1 Case Study: Anomaly Detection

Time series of site visits are commonly used as anomaly detection signals. However, if no context information is specified, anomaly signals of specific visiting paths are masked out by other signals. Therefore, it causes false negatives.

We pick a typical Yahoo! mail site  $W$  (i.e.,  $\tau$  in Section 2)<sup>5</sup> and show that anomalies in higher-order path signals are significant and can be revealed by DECT. We use DECT to compute visiting probabilities of  $W$  for all contexts  $\{\bar{c}\}$  that exist. We then fed time series  $\{\Psi(\tau_i|\bar{c}) \mid 0 \leq$

<sup>4</sup>DECT can be deployed at the client/browser side to model and analyze Internet-wide user behavior.

<sup>5</sup>According to Yahoo! data privacy requirements, *i*) detailed data statistics are not provided; *ii*) probabilities in figures are disguised while their relative positions are preserved.

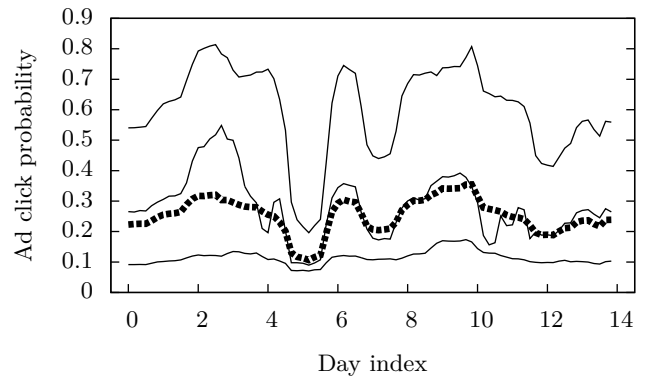


Figure 4: Ad click probabilities given different paths. The bold dotted line denotes the overall ad click rate of users on a site. Each thin line denotes ad click rates of users on this site coming from one specific visiting path<sup>5</sup>.

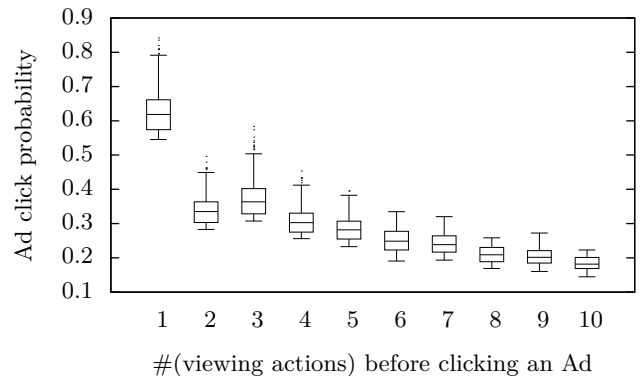


Figure 5: Ad click probability of a wanderlust<sup>5</sup>.

$i \leq \kappa_{\bar{c}}\}$  yielded by DECT into EGADS for anomaly detection. EGADS is a generic and scalable framework for automated anomaly detection on large scale time-series data [10]. The entire anomaly detection application consists of DECT (time-variant user behavior modeling) and EGADS (time series anomaly detection).

Fig. 3 shows two higher-order anomalous time series identified by EGADS. The upper subfigure, a common seasonal time series across a month, is the visiting probability of  $W$  across all Yahoo! mail sites. One anomalous time series (site  $X$  to  $W$ ) is detected during that month (several spikes in the middle subfigure). Another anomaly is found on the 8th day of visiting path “ $Y$  to  $X$  to  $W$ ” in the lower subfigure. Any anomaly (spike) with respect to a higher-order path may be hidden in the time series of its suffix path.

### 4.2 Case Study: Ad Click Prediction

Existing ad click prediction techniques do not take historically visited paths into account. We run DECT on the Yahoo! finance dataset to show that such information is useful in distinguishing probabilities of ad clicks.

We draw the overall ad click rate on a Yahoo! finance site<sup>5</sup> in Fig. 4 with the bold dotted line. We then use DECT to

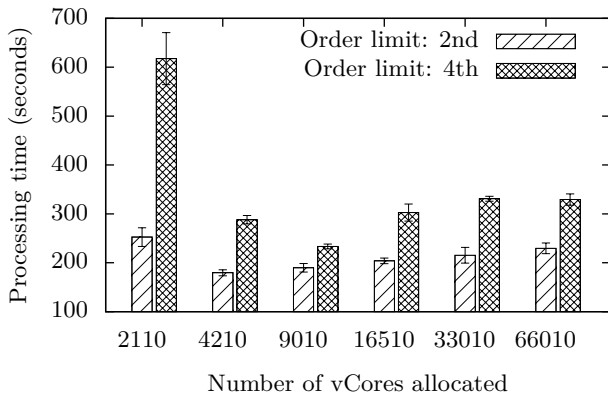


Figure 6: Performance pivot discovery of DECT.

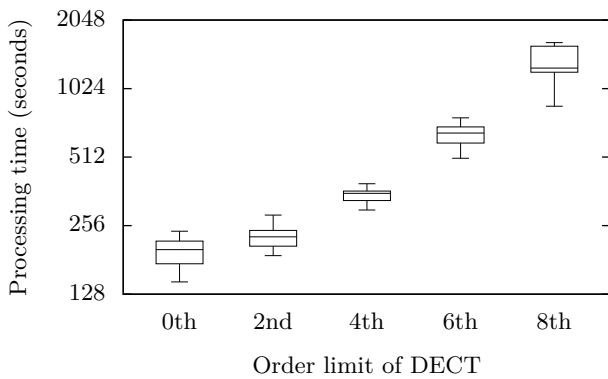


Figure 7: Order impact on computation complexity.

investigate three ad click rate time series, each of which has a site previously visited (one-time context) before the target site. Fig. 4 shows that the click rate of users coming from one site can be 5 times higher than that of another.

Besides the finding that *ad click rates are related to user visiting paths*, another interesting conclusion we reached is that *the more a user views articles on a site, the less likely she will click an ad on that site*. We illustrate the decrease of ad click rates on a Yahoo! finance site<sup>5</sup> in Fig. 5. We explain the phenomenon that frequent readers tend to continuously consume target information, e.g., stock values, and ignore ads. Ads could be less effective and more annoying to frequent readers than normal visitors. This work is being deployed at Yahoo! for better ad-targeting.

### 4.3 Performance analysis

We demonstrate the performance of our implementation by processing a subset of Yahoo! mail data (around 0.1 billion user sessions, 10GB storage size on HDFS)<sup>5</sup>.

**Scalability and Performance Pivot.** Our realization of DECT is based on the scalable Apache Spark framework. A distributed system results in an increasing amount of communicating/scheduling overhead when scaling out. We are interested in discovering performance pivots and parameter tuning on real-world datasets.

We conduct several groups of experiments with DECT

running on different numbers of worker nodes. We measure the degree of parallelism via the maximum number of processing units (vCore)<sup>6</sup> concurrently allocated at any execution stage. Fig. 6 shows that our implementation scales out well before reaching a performance pivot. Performance pivots are reached for DECT with order limit 4 at 9010 vCores and order limit 2 at 4210 vCores. Increasing the number of processing units after the pivots wastes more in overhead than gaining better performance. The more complex the computation is, e.g., higher order limit, the larger amount of processing units are required to reach the pivot.

**Magnitude of frequently visited higher-order paths.** DECT is a variable-order Markov model. It employs a pruning procedure to remove time series of rarely visited higher-order paths. We are interested in the order impact on the overall computational complexity.

We execute DECT with various order limits. Because the total number of possible paths is exponential in path order, the results in Fig. 7 show that: *i*) the processing time increases exponentially with the increase of the order limit, and *ii*) the pruning procedure reduces the number of time series *by a constant factor* on Yahoo! mail dataset.

## 5. RELATED WORK

Our work is motivated by time-homogeneous Markov user behavior modeling, time series analysis, and evolutionary network analysis.

**Time-homogeneous Markov modeling.** Web user behavior has been studied for various purposes, such as PageRank [13], link prediction [17], document prefetching [23]. A variety of time-homogeneous Markov models have been tested to describe Internet user behavior [5]. The time-homogeneous indicates that the transition matrix of the Markov model does not change through time. We list some existing models classified by their Markov orders below.

- First-order Markov model: [12, 13]
- Second-order Markov model: [23]
- Higher-order Markov model: [15]
- Variable-order Markov model: [2, 5, 6]

Variable-order Markov models compute different orders for different paths to reduce storage expenses. The idea was proposed by Bühlmann and Wyner [3]. There exist two generic approaches to construct variable-order models.

**Pruning-based approach:** starting with a complete higher-order model and iteratively pruning low-entropy branches to get an incomplete tree, e.g., [6].

**Growing-based approach:** starting with a first-order Markov model and expanding leaves with inconsistent distribution into branches, e.g., [2].

Our design follows the former approach for straightforward parallel design. The operations of growing higher-order paths, i.e., slicing and clustering, are computational heavy and the results cannot be efficiently reused over time.

**Time series analysis.** A time series denotes the change of a variable over time [8]. Time series analysis has been applied to many fields including signal forecasting [9], data feature extraction [7] and anomaly detection [4, 10, 11, 20].

Time series analysis is widely used to detect anomalous

<sup>6</sup>The number of workers is linear to the number of vCores.

user events in the industry. However, studied variables in existing systems are mostly primitive, e.g., counts of site visits. They only represent zeroth-order or first-order (one-hop) paths [10]. The prediction is fast but loses rich context information. DECT, in contrast, utilizes historical site visiting information to provide more detailed signals for anomaly detection and ad click rate prediction as shown in Section 4.

**Evolutionary network analysis.** Dynamic networks appear in social networks, wireless sensor networks, Internet of Things, and the Web. The analysis of evolving networks provides a comprehensive understanding of such networks [1] and enables applications such as link prediction [18] and anomaly detection [16].

Graphs are generic models for dynamic network representation [1]. More specifically, dynamic networks usually generate complex cyclic graphs, and evolutionary network analysis heavily relies on unique properties of such graphs, e.g., community discovery [14]. Compared to cyclic graphs, variable-order Markov models are tree-equivalent structures. In our model, we bring some concepts from evolutionary network analysis, e.g., *change of  $V_M$*  and *change of  $E_M$* . But, in general, it is currently unclear how evolutionary network analysis methods can be applied to dynamic web user behavior modeling.

## 6. CONCLUSIONS AND FUTURE WORK

This paper presents DECT, a scalable time-variant web user behavior model. It characterizes the changing nature of Internet user behavior with a time-variant variable-order Markov model. DECT can be efficiently realized on scalable distributed frameworks, e.g., Apache Spark, to process large volumes of user behavior data. DECT enables time series analysis on individual or related sets of long (higher-order) user paths. We open-sourced DECT and deployed it at Yahoo! to support path time series analysis such as anomaly detection, click probability prediction and path trend discovery. In the future work, we plan to work on streaming pruning strategies to enable streaming user behavior processing using DECT.

## 7. REFERENCES

- [1] C. Aggarwal and K. Subbian. Evolutionary network analysis: A survey. *ACM Computer Surveys*, 47(1):10:1–10:36, May 2014.
- [2] J. Borges and M. Levene. Evaluating variable-length Markov chain models for analysis of user web navigation sessions. *IEEE Transaction on Knowledge and Data Engineering*, 19(4):441–452, April 2007.
- [3] P. Bühlmann and A. J. Wyner. Variable length Markov chains. *The Annals of Statistics*, 27(2):480–513, April 1999.
- [4] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computer Surveys*, 41(3):1–58, July 2009.
- [5] F. Chierichetti, R. Kumar, P. Raghavan, and T. Sarlos. Are web users really Markovian? In *Proceedings of World Wide Web Conference*, pages 609–618, New York, NY, USA, 2012.
- [6] M. Deshpande and G. Karypis. Selective Markov models for predicting web page accesses. *ACM Transactions on Internet Technology*, 4(2):163–184, May 2004.
- [7] P. Esling and C. Agon. Time-series data mining. *ACM Computing Surveys*, 45(1):12, 2012.
- [8] J. D. Hamilton. *Time series analysis*, volume 2. Princeton university press Princeton, 1994.
- [9] R. J. Hyndman and A. B. Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, 22(4):679–688, 2006.
- [10] N. Laptev, S. Amizadeh, and I. Flint. Generic and scalable framework for automated time-series anomaly detection. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1939–1947, 2015.
- [11] N. Laptev, R. Hyndman, and E. Wang. Large-scale unusual time series detection. In *Proceedings of IEEE International Conference on Data Mining*, 2015.
- [12] Z. Li and J. Tian. Testing the suitability of Markov chains as web usage models. In *Proceedings of Annual International Computers Software and Applications Conference*, pages 356–361, November 2003.
- [13] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.
- [14] S. Parthasarathy, Y. Ruan, and V. Satuluri. Community discovery in social networks: Applications, methods and emerging trends. In C. C. Aggarwal, editor, *Social Network Data Analytics*, pages 79–113. Springer US, 2011.
- [15] P. Piroli and J. Pitkow. Distributions of surfers’ paths through the World Wide Web: Empirical characterizations. *World Wide Web*, 2(1-2):29–45, 1999.
- [16] S. Ranshous, S. Shen, D. Koutra, S. Harenberg, C. Faloutsos, and N. F. Samatova. Anomaly detection in dynamic networks: a survey. *Wiley Interdisciplinary Reviews: Computational Statistics*, 7(3):223–247, 2015.
- [17] R. R. Sarukkai. Link prediction and path analysis using Markov chains. *Computer Networks*, 33(1):377–386, 2000.
- [18] R. R. Sarukkai. Link prediction and path analysis using Markov chains. *Computer Networks*, 33(1–6):377–386, 2000.
- [19] X. Shu. Distributed evolving context tree (DECT), <https://github.com/subbyte/DECT>.
- [20] O. Vallis, J. Hoehenbaum, and A. Kejariwal. A novel technique for long-term anomaly detection in the cloud. In *Proceedings of USENIX HotCloud Workshop*, pages 15–15, Philadelphia, PA, June 2014.
- [21] Y. Xie and S. zheng Yu. Monitoring the application-layer DDoS attacks for popular websites. *IEEE/ACM Transactions on Networking*, 17(1):15–25, Feb 2009.
- [22] S. Yu, G. Zhao, S. Guo, Y. Xiang, and A. Vasilakos. Browsing behavior mimicking attacks on popular web sites for large botnets. In *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 947–951, April 2011.
- [23] I. Zukerman, D. W. Albrecht, and A. E. Nicholson. Predicting users’ requests on the WWW. In *Proceedings of the 7th International Conference on User Modeling*, pages 275–284, Secaucus, NJ, USA, 1999. Springer.

# Strudel: Framework for Transaction Performance Analyses on SQL/NoSQL Systems

Junichi Tatemura  
NEC Labs America  
tatemura@nec-labs.com

Zheng Li  
Univ. of Massachusetts Lowell  
zli@cs.uml.edu

Oliver Po  
NEC Labs America  
oliver@nec-labs.com

Hakan Hacigümüş  
NEC Labs America  
hakan@nec-labs.com

## ABSTRACT

The paper introduces Strudel, a development and execution framework for transactional workloads both on SQL and NoSQL systems. Whereas a rich set of benchmarks and performance analysis platforms have been developed for SQL-based systems (RDBMSs), it is challenging for application developers to evaluate both SQL and NoSQL systems for their specific needs. The Strudel framework, which we have released as open-source software, helps such developers (as well as providers of NoSQL stores) to build, customize, and share benchmarks that can run on various SQL/NoSQL systems. We describe Strudel's architecture and APIs, its components for supporting various NoSQL stores (e.g., HBase, MongoDB), example benchmarks included in the release, and performance experiments to demonstrate usefulness of the framework.

## 1. INTRODUCTION

As a large number of web applications adopt cloud computing platforms, various types of “NoSQL” systems have emerged and been employed as scalable and elastic data stores. They are expected to serve transactional workloads<sup>1</sup> of an application that interacts with a large and varying number of users on top of commodity server resources (typically in the cloud).

Now that application developers have many choices of NoSQL systems as well as SQL systems (i.e., RDBMSs), they face various questions (which we would call “SQL-or-NoSQL questions”): When should we use a NoSQL store instead of a traditional RDBMS? How can we choose a NoSQL system that suits for our purpose? With a particular NoSQL system, what kind of trade-off do we face between scalability/elasticity gain and the cost of reduced consistency/integrity support? What about other alternatives such as

<sup>1</sup>We focus on user-facing transactional application workloads instead of analytic ones.

purchasing a parallel RDBMS product or sharding (partitioning) open-source RDBMSs?

Standard benchmarks (such as TPC-C) have been very helpful for evaluating and choosing RDBMS products. On the other hand, the effort on benchmarking for NoSQL does not seem catching up with the evolution of NoSQL systems. Currently, YCSB [16, 12] is the most commonly used benchmark for NoSQL, but it mainly focuses on micro-benchmarking of key-value read/write operations. As NoSQL supports various features such as transaction and more complicated queries, more benchmarks are needed to capture the requirements behind such features.

It is challenging to develop benchmarks, especially to compare SQL-based systems and NoSQL systems together, given many different query APIs. It is also challenging to cover the wide range of the application needs. A transactional data store is just part of a larger application system, and its role and requirements are significantly different among applications.

Thus, we can hardly expect that a limited number of standard benchmarks are enough for transactional workloads over SQL/NoSQL systems. Given the variety of data stores and the variety of application needs, we need a way to efficiently develop a large and evolving suite of benchmarks, from micro-benchmark level to application-level, with minimum engineering effort in terms of (1) developing a new benchmark on existing SQL/NoSQL systems and (2) supporting a new SQL/NoSQL system for existing benchmarks.

In this paper we introduce our development and execution framework, called Strudel, for transactional benchmark workloads with expectation to contribute to the benchmarking effort in the community.

The design philosophy of the Strudel framework is to provide composability and reusability with (1) decomposing benchmark implementations into small components with multiple abstraction layers and (2) employing a configuration description language to combine these components into a specific workload in a reproducible and shareable manner. In order to bridge the gaps among various data stores, the framework provides multiple abstraction layers: most notably Entity DB API and Session Workload framework. A configuration description language is adopted in the framework to enable developers to compose a system and workload with custom properties in XML.

We have used and kept extending the framework for years through our research and product development of elastic relational stores (SQL engines on top of KVS [25]). As it



has matured as a generic and extensible framework, we recently released it (including benchmarks and SQL/NoSQL supports as described later) as open source software[9] for wider development purposes.

In this paper, we describe the design and architecture of the Strudel framework, its support on various NoSQL systems, benchmark examples we have developed, and performance experiments to demonstrate usefulness of the framework.

## 2. RELATED WORK

**YCSB** The Yahoo! Cloud Serving Benchmark (YCSB) [16, 12] is the most commonly used benchmark for NoSQL systems. However, from our objective to conduct performance studies on transactional aspects of SQL and NoSQL systems, the original YCSB has very limited support of transactional workloads. Researchers have extended YCSB to experiment transactions over multiple key-value objects (e.g., [17]). With an appropriate framework, we should be able to share such custom efforts to serve for more general application performance analyses.

As NoSQL systems evolve from a simple Key-Value store, there are increasing needs of performance studies with higher-level (i.e., application-level) workloads, which are especially important to compare SQL systems and NoSQL system from the application developers' viewpoints. YCSB++ [23] extends the original YCSB to evaluate advanced features of NoSQL systems (HBase and Accumulo). The features that are relevant to transactional workload evaluation include (1) pushing filtering to the data store, (2) measurement of data staleness due to weak consistency. One possible idea is to integrate such features with the Strudel framework to evaluate advanced NoSQL features not only for micro-benchmarks but also for application-level benchmarks.

**OLTP-Bench** OLTP-Bench [18, 6] is an extensible testbed for benchmarking RDBMSs for (primarily) transactional workloads. Our framework and OLTP-Bench are not mutually exclusive but complimentary. We focus on a special class of OLTP problems where developers have "SQL or NoSQL" questions. OLTP-Bench, aiming for more general OLTP use cases, provides a lot of useful features that our framework misses, such as sophisticated (e.g., more realistically skewed) data and workload generation and a rich set of SQL workloads (including traditional ones that we do not focus on). A skilled developer may reuse these features combined with the Strudel framework.

**Performance studies** There have been various performance studies reported on NoSQL systems, including the the original work of YCSB [16]. For an example of performance studies from the application's viewpoint, Klein et al.[22] report their performance evaluation of NoSQL systems for a healthcare application. Their customer (a healthcare provider) requests them to evaluate NoSQL technologies for a new electronic healthcare record system to replace the current version that uses an RDBMS. They developed evaluation tests by modifying the code of YCSB to fit the application's data model. With an appropriate development framework provided, their development could have been easier. In addition, whereas their study excludes RDBMSs (according to the customer requirements), it would need more engineering effort, in general, to compare NoSQL with RDBMSs.

Floratou et al. [20] report comparative performance stud-

ies between SQL and NoSQL systems, including comparison between SQL Server and MongoDB for YCSB workloads. We hope Strudel is useful to extend such studies to cover more NoSQL systems and various benchmarks that capture application-level requirements.

## 3. ARCHITECTURE

Figure 1 illustrates the layered architecture of the Strudel framework that provides composability and reusability in benchmark application development. The abstraction layers are visualized as orange boxes with underscored italic words (representing the names of APIs). Among them, Java Persistence API (JPA) is a Java standard for object-relational mapping (that converts Java object manipulation to SQL queries). The Strudel framework provides the other APIs.

Entity DB is a simplified data access API that covers transactional data access features that are common in various NoSQL systems as well as relational databases (Section 4). Basic implementation of Entity DB on a NoSQL system would not be very difficult (we also provide another API, Transactional KVS, to make it easier). If a benchmark is implemented on Entity DB API, it can run on various NoSQL systems as well as RDBMSs that support the JPA standard.

The Session Workload is a framework that helps developers to implement benchmark application on different data access APIs (Entity DB, JPA, and native NoSQL APIs) by reusing the code as much as possible (Section 5).

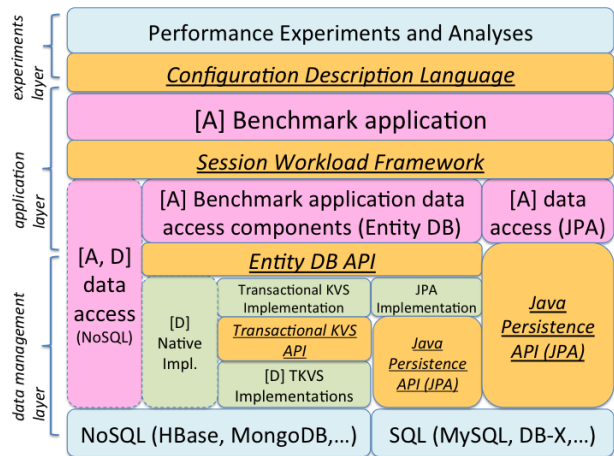


Figure 1: Layered Architecture of Strudel

In Figure 1, red boxes labeled with [A] are components a developer needs to implement for each benchmark application, and green boxes with [D] are implemented for each NoSQL system. The label [A,D] indicates a component to be implemented for each pair of a benchmark and a NoSQL system, and our goal is to minimize such components.

A developer can conduct a specific experiment by combining these components with a particular set of configuration parameters. We employ a configuration description language for such experiments to make experiments reproducible and individual components reusable across different experiments (Section 6).

Strudel also provides workload management and execution engines for experiments in a cluster environment in an

automated manner (Section 7).

## 4. ENTITY DB API

Entity DB API is one of the abstraction layers that are useful to develop workloads that can run on various data stores. It employs a subset of JPA (Java Persistence API) to fill the gap between SQL and NoSQL systems. JPA is a standard Java API for Object-Relational Mapping, providing a way to map Java objects (entities) and relational tables and a way to access data in a relational database through such Java objects.

JPA provides a basis for us to abstract out the details of underlying data stores so that application developers can focus on data handling in an object-oriented manner. However, to most NoSQL systems, JPA is not applicable directly since the concept of object-relational mapping relies on expressive power and declarativeness of SQL and the relational model.

Thus, we have designed a simplified version of APIs that consists of standard Java annotations (e.g., `@Entity`, `@Id`) of JPA and extended annotations as well as simplified data access methods. Whereas this API is not meant for application production use (missing various features required in production)<sup>2</sup>, it would support simplified application prototypes to quickly compare data store alternatives before the real application version is developed.

### 4.1 Entity Group Annotations

One of the key difference of NoSQL systems from the traditional RDBMS is that not all the data items are equal in terms of transactional data access. Distributed transactions are often expensive in a commodity cluster (especially cloud) environment in order to handle a large number of concurrent read/write accesses with high availability under a short response time requirement of interactive applications. Thus, most NoSQL systems provide a way to compromise transactional consistency for scalability to avoid distributed transaction as much as possible. In a typical case, a NoSQL system allows ACID data access only on a data set associated with a single key. HBase [2] (open source implementation of Bigtable [15]) provides an atomic check-and-update operation on a single row of a (big)table. For transactions over multiple rows, we need to use an external transaction manager (e.g., Omid [19]) to implement concurrency control and recovery.

In order to provide a common API to incorporate such transaction support, we adopt the concept of *entity groups*. Helland [21] argued that, in order for an application to be truly scalable, it must forgo expensive distributed transactions; instead, each transaction must operate on a uniquely identifiable collection of data (i.e., *entity groups*) that lives on a single machine. Google Megastore [14] supports entity group as a way to associate multiple entities to a group key and guarantee efficient ACID transactions within a single group.

We introduce a set of annotations to specify entity groups in a similar way to Megastore. In addition to standard annotations (`@Entity`, `@Id`, `@IdClass`), we introduce the following annotations: `@Group`, `@GroupId`, `@GroupIdClass`.

<sup>2</sup>For production use, there is an open source product, for example, DataNucleus that supports common APIs for Java data persistence on some of NoSQL systems: <http://www.datanucleus.org/>

We illustrate how these annotations are used in an example benchmark (which emulates an auction application) in Figure 2. The code uses the JPA standards to define Bid Java class as an entity (`@Entity`) with a compound key (`sellerId`, `itemNo`, `bidNo`) (as annotated with `@Id`), which is packaged as one object of a class BidId (`@IdClass`).

```
@Group(parent = AuctionItem.class)
@Entity
@Indexes({
    @On(property = "auctionItemId"),
    @On(property = "userId")
})
@GroupIdClass(ItemId.class)
@IdClass(BidId.class)
public class Bid {
    @GroupId @Id private int sellerId;
    @GroupId @Id private int itemNo;
    @Id @GeneratedValue
    private int bidNo;
    private double bidAmount;
    private long bidDate;
    private int userId;
}
```

Figure 2: Example code with annotations

With extended annotations, a benchmark developer can associate two entity classes together (as a parent-child relationship) in one group by specifying `@Group` annotation at a child class to indicate its parent. In this example, Bid is associated with AuctionItem (so that we can consistently access all the bids on one particular auction item and update the item when the maximum bid price changes). A group id (`@GroupId`) is a member of a compound key (a group key) that specifies a group instance. A set of group ids on an entity class must be a subset of the set of ids (i.e., a (compound) primary key) that are annotated with `@Id`.

### 4.2 Data Access Operations

**CRUD Operations** Entity DB API supports basic CRUD (Create-Read-Update-Delete) operations: (1) create (2) get, (3) update, and (4) delete operations (Figure 3). They (roughly) correspond to persist, find, merge, and remove operations of EntityManager, a data access interface of JPA<sup>3</sup>.

**Secondary Key Access** Unlike JPA, the current Entity DB API does not support a SQL-like query language or automatic retrieval of related entities with a join column (i.e., annotations such as `@OneToMany`, `@ManyToOne`). Instead, it provides a way to read multiple instances of the same entity class by specifying one of the entity's property as a *secondary key* (`getEntitiesByIndex` in Figure 3).

**Group Transactions** The current version of the framework only supports a transaction within an entity group (designing API to indicate a "global" transaction is a plan for a future version). A transaction starts with a given group key and commits after multiple CRUD operations. Figure 4 shows an example of transaction execution. The application code gives an instance of EntityTask interface to the EntityDB API ("`edb.run()`"), then the underlying Entity

<sup>3</sup>A subtle difference from JPA is that there is no concept of attachment/detachment in the current version: an application always needs to use an update operation to apply changes in an entity Java object to the database.

```

<T> T get(Class<T> entityClass, Object key);
void create(Object entity);
void update(Object entity);
void delete(Object entity);
<T> List<T> getEntitiesByIndex(
    Class<T> entityClass, String property,
    Object key);

```

Figure 3: EntityDB Interface (partial)

DB implementation runs the instance of EntityTask by giving an EntityTransaction object (`run(EntityTransaction tx)`). The entity task (i.e., the application code) uses this transaction handler (`tx`) to issue multiple CRUD operations.

The reason behind this rather convoluted API design (instead of providing usual begin/commit operations) is to abstract out transaction retry handling, especially for optimistic concurrency control. A transaction of applications for NoSQL systems is often simple and lightweight. It is easy to retry the entire transaction when a commit request fails under optimistic concurrency control. In order to analyze the impact of transaction conflict in different NoSQL systems, we want experiment different retry policies without changing the application code.

```

BidResult r = edb.run(Bid.class, itemId,
    new EntityTask<BidResult>() {
        public BidResult run(EntityTransaction tx) {
            AuctionItem item =
                tx.get(AuctionItem.class, itemId);
            if (item == null) {
                return BidResult.NONE;
            }
            if (bid.amount() <= item.getMaxBid()) {
                return BidResult.LOST;
            }
            tx.create(bid);
            item.setMaxBid(bid.amount());
            tx.update(item);
            return BidResult.SUCCESS;
        }
    });

```

Figure 4: Example of group transaction

### 4.3 Auxiliary Data Maintenance

Entity DB API supports two features that involve maintenance of auxiliary data in the underlying data store: (1) values for automatic unique key generation and (2) indices.

**Auto Key Generation** Entity DB API lets the developers specify the standard `@GeneratedValue` annotation to use a generated unique value for an id. Just as the standard JPA, an underlying implementation can choose a way to generate unique values based on the data store’s capability.

The major difference from the standard specification is the uniqueness requirement: In JPA, `@GeneratedValue` is used to generate a unique value within a table. In Entity DB, it can only be *locally* unique: it is required that the compound group/primary keys that include this id as a member are unique. For example, in Figure 2, the id `bidNo` is automatically generated when a Bid entity instance is created. In fact, `itemNo` is also a generated value specified in the AuctionItem class definition. The value of `itemNo` must be unique

within a particular user (identified with `sellerId`), and the value of `bidNo` must be unique within a particular auction item.

This local uniqueness requirement gives the underlying data store more freedom to use a scalable and efficient way to generate values.

**Indices** To use secondary-key data access, the developers need to explicitly specify an index on a property of an entity. Notice that the role of an index at logical design level is different from the case with relational databases where selection of an index is logically transparent from queries. Thus, we introduce a special annotation (`@Indexes`) separated from JPA’s index annotation (`@Table.indexes`).

One non-trivial semantics of this index-based entity access is its consistency under a specific entity group design. In terms of transaction isolation, an index entry can be seen another (system-defined) entity, and question is whether it is grouped together with the entities it refers to.

An index can be included (and is included by default) in the group if the compound index key (i.e., a set of ids) includes the compound group key. We call such an index an *in-group index* and otherwise we call it an *out-of-group index*. For example, in Figure 2, there are two indices on AuctionItemId (which is in fact a compound key with `sellerId` and `itemNo`) and `userId`. The former index is in-group (the index key equals to the group key) and the latter index is out-of-group.

If the index is not included in the group, we cannot prevent a *phantom read*: a transaction cannot read the all the bids on the same item (which is done through the index) in an isolated manner (e.g. it cannot be isolated from insertion of a new bid).

In the current Entity DB, we only allow an index on immutable columns: the value is specified only at creation of an entity instance and does not change until the instance is deleted.

## 4.4 Implementations

### 4.4.1 Generic Transactional KVS

Whereas Entity DB API is simplified for minimum support for entity data access, it still needs engineering efforts to develop an implementation for a particular NoSQL system. We provide yet another API for transactional key-value data access so that a provider of a NoSQL system can quickly implement this further simplified API instead of directly implementing Entity DB.

In Transactional KVS, a data record is just a pair of byte-array key and value, and records are grouped by a group key (another byte array). Data access is done by a group transaction (started with a group key) and simple put/get operations.

The framework provides an Entity DB implementation for Transactional KVS, which automates (1) mapping from entities to byte array key-value objects, (2) index management, and (3) auto key generation.

**Index and Key generation** As a baseline implementation of Entity DB API, we implemented an index and a key-generation counter as sets of key-value objects on top of the Transactional KVS data model. For each index key, we create an object with the index key and a value that encodes the pointers to the indexed entities. A counter object is created for each parent key (i.e., the compound primary

key except an ID to be generated) of an entity, and its value is a counter number.

Consider the example in Figure 2 when an application workload creates a new instance of Bid entity. Bid has two indices (`auctionItemId` and `userId`) and a generated key (`bidNo`). There are three auxiliary key-value objects updated when we create a new Bid (which is a new key-value object by itself): (a) an index object of type `auctionItemId` with key (`sellerId`, `itemNo`), (b) an index object of type `userId` with key `userId`, and (c) a counter object of type Bid with key (`sellerId`, `itemNo`). Among them only the object *a* is included in the group with the Bid entity, and creation of Bid involves a following sequence of three transactions:

1. updates a counter object *c* and acquires a new value for `bidNo`.
2. updates an index *b* to insert a key of Bid (`sellerId`, `itemNo`, `bidNo`) using the result of transaction 1.
3. updates an index *a* and creates a new key-value object for a Bid entity.

The order of these transactions is important to keep an index *b* consistent. Even if another transaction accesses the index *b* between transaction 2 and 3 (or even if transaction 3 fails), it will just read a dangling pointer in the index to a non-existent entity, which does not cause inconsistency. On the other hand, if these transactions create a Bid entity but fail to update the index *b*, it results in inconsistency (i.e., the existing entity cannot be accessed by the index). When the entity is deleted, the order of transactions will become opposite. When the entity is updated, we do not update the index (i.e., the current Entity DB assumes keys are immutable).

Notice also that the counter object *c* could have been updated in the transaction 3 if there were no out-of-group index such as *b* (that needs a new committed value of `bidNo` from *c*). In this case, creation of an entity is done by a single transaction and the counter *c* and index *a* is implemented with a single key-value object since their have the same key.

We have developed the following implementations of Transactional KVS.

**HBase** [2] To implement transactions, we employ HBase's check-and-put operation, which is in general called a *compare-and-swap* (CAS) operation and operates value comparison (read) and update (write) in an atomic manner.

Since check-and-put is applicable only to a single row, all the records (entities) that belong to the same group must be packed into one row. To do this, we implement each key-value record as a column name-value pair (HBase/Bigtable's columns are created in an ad-hoc manner).

In addition to these columns, each row has a special column that holds a transaction version. When the Entity DB starts a transaction, it will read the current value of this transaction version on a row that corresponds to an instance of entity group. All the updates are buffered at the client side during the transaction. At a commit request, the Entity DB issues a check-and-put operation that applies the buffered updates to the corresponding row if the current transaction version equals to the one at the beginning of the transaction.

**Omid** [5] Omid is a transaction server on top of HBase in order to realize ACID transactions over multiple rows [19]. It

employs optimistic multi-version concurrency control using multi-versioning of HBase (each row can have multiple versions associated with (logical or physical) timestamps). For each transaction, Omid server issues a new timestamp value, with which a transaction client puts updates to HBase rows. A commit request is sent to the Omid server and ensured after conflict check and writing a log entry for recovery. Although it is a centralized server, the required computation at the transaction server is lightweight and it will not become a bottleneck for scalability easily. For our Entity DB implementation, we also implemented *sharded* Omid servers, by applying hash partitioning over the group key and routing a transaction request to one of multiple Omid servers. In our small-scale experiments up to 10 HBase region servers, however, we did not need more than one Omid server to achieve scalability (Section 8.2).

**MongoDB** [3] MongoDB's update operation is atomic for a single document and consists of query part and update part (i.e., a more general form of the CAS operation). Similar to the HBase implementation, we pack entities of the same group into one document. In the query part of the update, we include the document id (that corresponds to a group key) and a value of a transaction version (stored as a field in a document).

**TokuMX** [10] TokuMX is an enhanced version of MongoDB. One of the enhancements is to support a multi-statement transaction over multiple documents whereas the original MongoDB only supports a single statement transaction (i.e., an update operation) over a single document. A client can begin and commit or rollback a transaction. During a transaction, a client can read and update multiple documents. Isolation is achieved by locking documents (i.e., it takes pessimistic concurrency control).

One big limitation in the current TokuMX version is it does not support a multi-statement transaction for sharded document collections (i.e., partitioned data).

Our Entity DB implementation uses a cluster of independent TokuMX servers and partition data based on the group key. It emulates sharded MongoDB with application-level request routing. Since a transaction for one group key is always executable with a single server, we can employ TokuMX's multi-statement transaction.

This implementation has a limitation when it is used in practice: it does not support rebalancing of partitions (or "chunks" in MongoDB's terminology), which is one of the most important feature of NoSQL to provide elasticity.

#### 4.4.2 Java Persistence API

The Strudel framework includes an implementation of Entity DB API using JPA so that a benchmark on Entity DB can run on any RDBMSs as long as it supports JPA. It is straightforward to implement Entity DB API using JPA since most of the features of Entity DB have the direct counterpart in JPA.

The implementation automatically translates a secondary key access to a query in JP QL, JPA's standard query language (which is then translated to SQL of a specific RDBMS).

In order to optimize physical design of the database, the developer can use any other JPA annotations. For example, selection of indices is an independent decision from the secondary key access specification (`@Indexes`) of Entity DB API: the developer specifies indices using the standard JPA (i.e., `indexes` attribute of `@Table` annotation).

### 4.4.3 Native Implementations

Our framework lets the developer implement a custom way to map entity access to a specific NoSQL system. We expect a future version of Strudel include such custom implementations for popular NoSQL systems.

For example, by mapping parent-child relationship to a specific data model supported by a NoSQL system, we can eliminate some of the indices specified in `@Indexes` as follows:

**Nested data structure** Various NoSQL systems, such as HBase and MongoDB, support a nested data structure: HBase's column family can be used to represent a set of child records (e.g., a set of bid records on a particular auction item). MongoDB's data model is a document, allowing to group entities in a flexible manner. If the secondary key to access is the parent key (e.g., `auctionItemId` index in Figure 2), we can retrieve these nested entities in one operation.

**Range key access.** HBase employs range partition to distribute a table and supports a range query on the row ID. By encoding parent-child relationship as a prefix of a row ID, we can efficiently implement a secondary key access (if the secondary key to use is a parent key).

## 5. SESSION WORKLOAD FRAMEWORK

Although Entity DB provides a common API which is reasonably implementable for many NoSQL systems, it is often too restrictive for a specific NoSQL system or an RDBMS, which have more advanced features that can contribute to higher application workload performance.

We provide another abstraction layer, *Session Workload*, at an application level for session-oriented workloads so that developers can create benchmarks that can run various data access APIs besides Entity DB API.

The Session Workload framework enables developers to build workloads that emulate interactive applications in a similar way to TPC-W [11] (emulating e-commerce) and RUBiS [8] (emulating auction) benchmarks.

Emulated user interaction for each user is called a *session*, which consists of a sequence of actions (called *interactions*). An interaction is a unit of the application's work, which is a predefined data accessing procedure without user intervention (one interaction may execute multiple transactions to perform a unit of work). A user issues a request for an interaction one by one (with optional intervals called "think time"). A user behavior is modeled as a state transition and the next interaction request is chosen based on the predefined probability and the results of the previous interactions.

The Session Workload framework makes the benchmark code reusable and customizable through the following features: (1) Interaction interface that separates data access logic and other part of benchmark code, and (2) highly configurable parameters including state transition definitions.

### 5.1 Interaction Interface

Figure 5 shows the interface for interactions. An interaction must implement three parts: `prepare`, `execute`, and `complete`. When an execution engine runs one interaction, it calls these three methods in this order.

The `prepare` operation is to generate a parameter that indicates a specific action that the interaction will take in the next `execute` operation. Typically, this operation emulates a thinking process of a human for this interaction (i.e. not

the application side procedure). For example, an auction benchmark emulates how a bid price is decided given the current session state (e.g., information on the auction item retrieved in the past interactions).

The `execute` operation implements the actual action that accesses the data. Given the parameter (`param`) generated by `prepare` and the data access API (`db`), the method performs transactions with the data store.

The `complete` operation defines how the session state is modified based on the result of `execute` operation. For example, to emulate a human's browsing activities on a web application, the result of a browsing interaction includes a list of retrieved items. The `complete` operation may choose one of such items as "current item of interest" (i.e. part of the *state*). The modified state is used in the following interaction, which may take an action on the chosen item (e.g., placing a bid).

```
public interface Interaction<T> {
    void prepare(ParamBuilder paramBuilder);
    Result execute(Param param, T db,
                  ResultBuilder res);
    void complete(StateModifier modifier);
}
```

Figure 5: Interface for Interaction

We designed to split these methods so that we can implement a benchmark in a reusable manner for multiple data access APIs, as we describe in the following.

**Generic Interaction Interface** The Interaction interface employs Java's Generics to parameterize a data access API and reuse the benchmark code as much as possible. In Figure 5, the type variable `T` corresponds to a class of data access API (e.g., EntityDB and JPA's EntityManager). The application code can be written agnostic to a specific data access API as long as it does not need to know what `T` actually is. For example, the `prepare` method does not have to know if an interaction is used with EntityDB or any other API.

**Abstract Interaction Classes** To make a benchmark reusable for many data access methods, a developer is encouraged to create an *abstract interaction class* for each interaction in the benchmark. An abstract interaction class implements two methods of the interface, `prepare` and `complete`, and lets its sub-class implement the remaining `execute` method.

In the benchmarks we have developed, we implement both EntityDB and EntityManager (JPA) versions of interactions. These two implementations share majority of the benchmark code (e.g., entity definitions, data generation, workload parameter generation, state transition) (see Section 8.6 for details).

### 5.2 Session State Transition

A benchmark workload based on the Session Workload framework can be easily customized for a specific experiment. A state transition model that emulates a user behavior is given at run-time as an XML data. Figure 6 shows an example of an XML element (`session`) that contains state transitions (`transitions`). The `session` element typically contains various other parameters that take part of the session state in order to customize behavior of the interactions.

```

<session>
  <packageName .../>
  <Transitions>
    <transition name="START">
      <next name="HOME"/>
    </transition>
    <transition name="HOME">
      <next name="SELLAUCTION.ITEM" prob="0.2"/>
      <next name="SELL.SALE.ITEM" prob="0.1"/>
      <next name="VIEW.AUCTION.ITEMS.BY.SELLER" prob="0.1"/>
      <next name="VIEW.SALE.ITEMS.BY.SELLER" prob="0.1"/>
      <next name="VIEW.AUCTION.ITEMS.BY.BUYER" prob="0.1"/>
      <next name="VIEW.SALE.ITEMS.BY.BUYER" prob="0.1"/>
      <next name="VIEW.BIDS.BY.BIDDER" prob="0.1"/>
      <next name="VIEW.WINNING.BIDS.BY.BIDDER" prob="0.1"/>
      <next name="END" prob="0.1"/>
    </transition>
    <transition name="SELLAUCTION.ITEM">
      <next name="HOME"/>
    </transition>
  </Transitions>
</session>

```

Figure 6: State transition in XML

### 5.3 Benchmarks

The current Strudel also includes example implementations of benchmarks for micro-level and application-level experiments on top of the Session Workload framework.

**Micro Benchmark** The Micro benchmark emulates a simplified user content management application in order to serve as a microbenchmark. The data and workload scale in terms of user IDs. To represent different patterns of user data access, the user content consists of the following four types of entities:

- *personal items* represent content privately owned by individual users. Each user has a set of items as one entity group (i.e., the number of groups scales as the number of users). An item is only read and written by its owner.
- *shared items* represent shared content written and read by the users. Items are grouped into multiple entity groups associated with set IDs (which give another scaling factor besides the user IDs).
- *public items* represent individual users' content that are open to the public for reading. An item is only written by its owner but can be read by other users.
- *message items* represent content exchanged from one user to another, having a sender ID and a receiver ID.

The benchmark defines various read-write and read-only interactions for each type of entities. A developer can compose a workload by creating a state transition that includes any subset of these interactions. By mixing interactions on these four types of entities, a developer can emulate the need of a specific application to some degree without coding a new benchmark.

In Section 8, we use *personal* items and *shared* items to demonstrate various scenarios of transaction performance analyses.

**Auction Benchmark** For application-level benchmarks, we have implemented an auction benchmark, which is similar to AuctionMark in OLTP-Bench [6] and RUBiS benchmark [8] but customized to use entity groups. The Bid entity in Figure 2 is part of this benchmark (shown after omitting some detailed code).

## 6. CONFIGURATION DESCRIPTION LANGUAGE

The Strudel framework provides abstraction layers to separate a benchmark application into various customizable pieces from a data access API implementation of a specific data store to a parameter generation of a benchmark workload. In order to combine these pieces together as one specific benchmark experiment, we employ a configuration description language that is similar to ones used for system component deployment in Grid and cloud infrastructures [13, 24, 7]. We have separately released this language as open source software called Congenio [1].

Our XML-based language supports the following features: (1) inheritance (**@extends** attribute), (2) document unfolding (**foreach** element), (3) reference resolution (**@ref** attribute), and (4) value expression (See the web site [1] for more details of the language).

With **@extends**, an experiment document can refer to existing templates (that define various components such as benchmarks and data stores) and customize the default values of these templates. With **foreach** elements, an experiment document can generate a set of documents, each of which corresponds to one workload execution with a specific set of parameters. Figure 7 illustrates such an experiment document to run an auction benchmark on HBase with different number of data servers (5, 10) and different workload scales (i.e. the number of users and worker servers).

```

<jobSuite>
  <foreach name="scale">
    <s><w>1</w><u>10000</u></s>
    <s><w>2</w><u>20000</u></s>
    <s><w>4</w><u>40000</u></s>
  </foreach>
  <foreach name="server" sep=" " >5 10</foreach>
  <job extends="auction-hbase">
    <workerNum ref="scale/w"/>
    <serverNum ref="server"/>
    <userNum ref="scale/u"/>
  </job>
</jobSuite>

```

Figure 7: Job definition in XML

The definition of an experiment in Figure 7 refers to a specific job template as illustrated in Figure 8. A job template combines various components including a workload (benchmark), database (access to data stores), and cluster (worker servers that run workloads).

```

<job>
  <workerNum>1</workerNum>
  <serverNum>1</serverNum>
  <userNum>10000</userNum>
  <threadsPerWorker>200</threadsPerWorker>
  <cluster extends="cluster">...</cluster>
  <database extends="tkvs-hbase">
    <name>auction</name>
    ...
  </database>
  <workload>
    <session extends="session-auction">
      <numOfThreads ref="threadsPerWorker"/>
      ...
    </session>
    <measure>...</measure>
  </workload>
  <report>...</report>
</job>

```

Figure 8: Example of job composition

When the execution platform (Section 7) runs an experiment with a given job definition, it records a document after inheritance resolution along with other information

(measured results, etc.). After inheritance resolution, the document includes all the information imported from other documents (referred to by `@extends`). This is very useful to reproduce the same experiments. In our lab, we use a version control system (git) to commit this document and experiment results together in the same version.

## 7. EXECUTION PLATFORM

Strudel’s execution platform consists of the workload manager and a cluster of workers.

The workload manager starts with a given job definition XML file (in the configuration description language) and interacts with worker servers as well as servers of SQL/NoSQL systems. The workload manager has the following features:

- server configuration and start-up (invoking external scripts for NoSQL/SQL systems).
- data generation and population
- workload control and workflow management
- performance monitoring (JMX) and aggregation of the reports from workloads.
- performance reporting as JSON files.

The Strudel framework does not include the individual scripts to configure and start/stop servers since it depends on the infrastructure (e.g., whether the system is deployed on the cloud platform, a Hadoop cluster, or a simple cluster servers mounting a shared file system).

The actual workload is run by a cluster of worker nodes that receive a workload definition from the workload manager. A worker is a workload execution engine that is deployed on a cluster of server machines. The workload manager coordinates a cluster of workers to run a benchmark workload in a scalable manner (a large number of threads) to put enough load on a scalable data store.

The Session Workload is one type of workloads the workers can run. It can run any custom workloads if they implement a workload interface defined in the Strudel framework. For example, it should be easy to develop a special workload that runs the YCSB benchmark.

## 8. DEMONSTRATION

In this section, we demonstrate some use cases of Strudel to conduct performance experiments. Notice that the objective of the following experiments is not a formal performance study to state any conclusive claims on a particular data store but a demonstration of the features of our framework.

### 8.1 System Settings

In our experiments, we use the following settings.

**HBase** [2] We use HBase version 1.1.1 on top of Hadoop version 2.7.1. HBase servers consist of a single master server (which manages the entire system and metadata) and a set of region servers (which manages data partitions (i.e., regions)). In the experiments, we mean the number of region servers by the number of data servers. A region server is collocated with Hadoop HDFS data node (to maximize locality of I/O). The name node of HDFS is located separately in a dedicated server. HBase also requires ZooKeeper processes to achieve coordination across servers. We use 3 ZooKeeper

processes collocated with the master server and two of the region servers.

**Omid** [5] We use version 0.8.0. Omid works with HBase and we use the same setting for HBase as the HBase-only data store. Omid supports multiple ways to persist transaction status for recovery. We use the default of the current version: storing the states on HBase. We use the same HBase cluster with the application (benchmark) workloads. In the experiment, we use only one Omid server and the number of data server refers to the number of region servers as in the case of the HBase-only setting.

**MongoDB** [3] MongoDB’s version is 3.0.5. To employ sharding (horizontal data partitioning), we need to deploy 3 config servers (just like ZooKeeper for HBase) and a set of shard servers (which we refer to by “data servers” in the experiments). We also need “mongos” servers that route applications’ requests to appropriate shard servers. We deploy one mongos server for each worker server as it is a common use case to collocate a mongos server with an application server.

**TokuMX** [10] For TokuMX we use version 2.0.1. As mentioned in Section 4.4, when we use multi-statement transaction with TokuMX, we cannot use sharding (automated partitioning). So we deploy a set of independent single-node TokuMX servers (which we call “data servers”), and let our EntityDB implementation route data access to these servers (emulating the application-level sharding).

**MySQL** [4] For experimenting JPA-based implementation of benchmarks, we use MySQL (Ver 14.14 Distrib 5.1.73) with default settings. We only use a single server MySQL in this demo.

**Server machines.** We use cluster machines in our lab with the following features: CentOS 6.6 (Linux 2.6.52), Intel Xeon E5620 2.40 GHz 16 core CPU, 16GB 1333 MHz RAM, Intel Pro2500 SATA SSD 240GB. Some OS parameters (e.g., the maximum number of open files) are set as data store providers recommend.

### 8.2 Data Store Scalability

First, we demonstrate a simple workload running on various data stores and show how these stores scale with an increasing number of data servers.

Based on the Micro benchmark, we composed a workload executing a single interaction that updates 4 *personal* items in the same group (which is randomly chosen from 1 M groups). We configure the workload so that transactions never conflict with each other: Each execution thread randomly chooses one user ID from an individual pool that is disjoint from the pools of other threads and uses the ID to choose a group (that belongs to this user).

We measure the throughput of the workloads using 1600 session concurrency (16 workers each of which runs 100 threads that keep running the update interaction without think time) for different data stores (except Omid) with changing the number of data servers (3, 5, 10). We made sure that the throughput is saturated (i.e., increasing session concurrency does not increase the throughput).

For Omid, we needed a larger number, 10800 (36 workers and 300 threads per worker), of concurrency to saturate the same number of data stores: Because one interaction takes longer time, a larger concurrency is needed to generate a sufficient number of read/write operations on the data stores. Using the Omid transaction server with HBase adds some

overhead (longer response time for each interaction and a larger number of data servers to achieve a throughput number) but it does not limit scalability at least for 10 region servers.

For MySQL we show the result of the JPA-based implementation (the figure has only one bar for a single server MySQL execution). We also ran the same workload on the Entity DB-based implementation but it did not show any significant performance difference for this simple workload (hence, it is omitted). In fact, these two implementations would generate the same SQL queries. Notice that the throughput of MySQL is good as a single data server: A 3-node NoSQL system does not achieve the same throughput *per data server*. This observation is consistent with [20], which reports the superior efficiency of an RDBMS compared to NoSQL stores. If the application workload can fit with a single data server, an RDBMS might be the most cost effective approach. If elasticity (dynamic re-balancing of partitions) is not required, purchasing a parallel RDBMS product may pay off for its efficiency.

In this demonstration, we did not cover parallel RDBMS products but it would be easy to run the same workload as long as the product is given with JPA support.

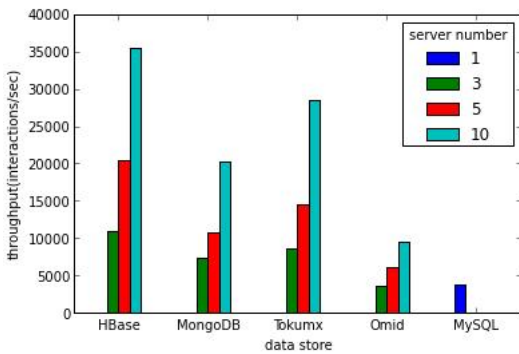


Figure 9: Throughput of item update interactions (4 items per interaction, 1600 concurrent sessions) on different number of data servers.

### 8.3 Transaction Concurrency

The result of the previous experiment demonstrates efficiency of a lightweight implementation of entity-group transaction with a simple check-and-update (HBase) compared to an approach with an additional transaction server (Omid). One drawback of this approach is that there is no concurrency allowed within a group (i.e., concurrent updates on the same group will fail). In many applications, this may not be a problem: when a group is associated with an individual user, a single user would not issue a large number of concurrent transactions. However it would not be always the case (e.g., auction bidding).

The next experiment we demonstrate is to see the trade-off between HBase (single-row transaction) and Omid (multi-row transaction) in terms of transaction concurrency. We use a workload that updates one *shared* item randomly chosen from a randomly chosen group. We fix the total number of items (80K) and change the size of group (400, 40, 4 items per group). From the viewpoint of the Omid transaction manager, these cases are identical (no difference be-

tween intra-group and inter-group). But for HBase, the total number of groups will decide the concurrency limit of the workloads (if two transaction updates different items in the same group, they will conflict with each other and only one can be successful). The result with 3200 concurrent sessions on 5 data servers (region servers) is shown in Figure 10. As the number of groups becomes smaller, HBase’s throughput degrades and becomes worse than Omid.

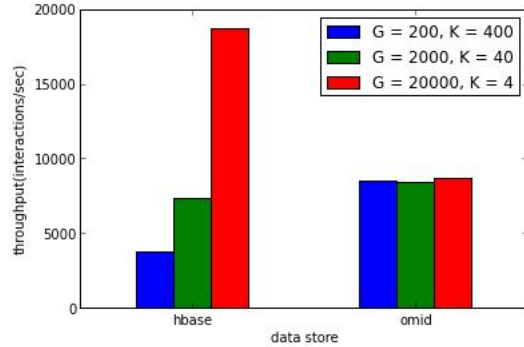


Figure 10: Throughput under different transaction concurrency: 3200 concurrent sessions, 5 data servers, 1 update/interaction, 80K items in G shared groups (K items/group)

It is beneficial to use a transaction server when the concurrency within a group needs to be high, even if it adds significant extra overhead (additional commit processing) compared to the main part of transaction (amount of read/write),

In a real application development setting, developers will need to manage the trade-off by building workloads to emulate the application’s needs and conducting similar experiments.

### 8.4 Transaction Conflict

Recall that TokumX is an enhanced version of MongoDB and supports a multi-statement transaction based on locking of documents (data items). An application developer would wonder how and when this feature should be used. One interesting experiments using our framework would be comparison between optimistic concurrency control with MongoDB (single-document transactions) and pessimistic concurrency control with TokumX (multi-document transactions).

At high-level, we know a rule of thumb, which is to take a pessimistic approach when conflict will likely happen in order to avoid unnecessary re-computation. But it always depends on a specific case.

In this example, a workload consists of a single interaction that updates 4 items in a randomly chosen group. The difference from the experiments in Figure 9 is that there are (varying degrees of) conflicts. We use 3200 concurrent sessions that update items in 3200 groups under the following three conditions: (1) 400 *personal* items per group: each thread will keep updating its own group (i.e., no conflict), (2) 400 *shared* items per group: each thread will randomly choose one of 3200 groups and choose 4 from 400 items (i.e., mild conflict), (3) 40 *shared* items per group: each thread



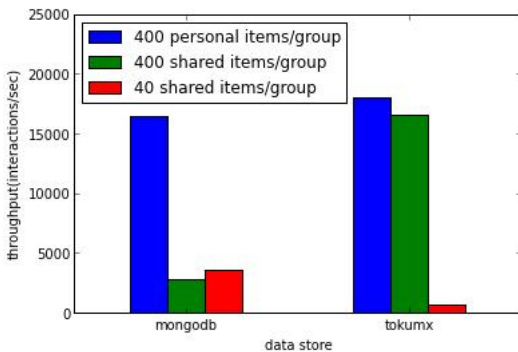
will randomly choose one of 3200 groups and choose 4 from 40 items (i.e., heavy conflict). The results are shown in Figure 11.

We notice the difference besides the concurrency control in the two versions: they are also different in allowed transaction concurrency (just like HBase and Omid). The MongoDB version of entity group transaction cannot have concurrency within a group. Hence, we do not see the difference between the case 2 and 3 for MongoDB. Their throughput values are almost equal to each other and are much lower than the throughput in the case 1.

On the other hand, the TokumX version employ a lock for each item (i.e., document) and it looks very effective in the case 2 showing only slight degradation from the case 1.

However, the behavior of the TokumX version is quite different in the case 3, showing a very low performance. In fact most of the transactions fail due to either deadlock or failure to acquire a lock, and these transaction will keep retrying until they finish successfully.

To compare optimistic and pessimistic concurrency control under heavy conflict, there is a fundamental difference in the cost of retrying transactions. In this particular case of optimistic concurrency control using CAS operations, the conflict relationship among transactions is very simple and there will be no deadlock: i.e., at least one of the conflicting transactions will “win.” Although retrying involves inefficiency, there is always progress in the computation. On the other hand, the pessimistic concurrency control may suffer from deadlock, in which case nobody wins. Thus, to ensure progress of the computation, the execution threads need to *back off* and wait longer time before retrying. In fact, the above result is after tuning the back-off policy using configuration options provided by the Strudel framework.



**Figure 11: Throughput under different degree of conflict: 3200 concurrent sessions, 5 data servers, 4 updates/interaction over 3200 shared/personal groups**

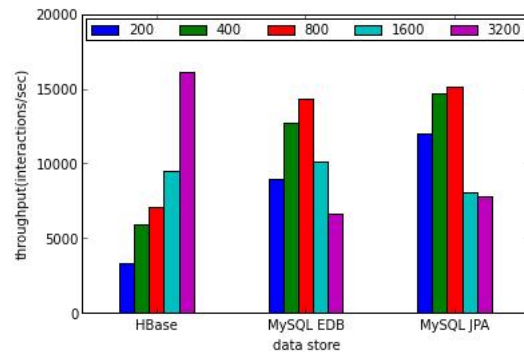
When it is very cheap to retry a transaction, the optimistic concurrency control can be an easier approach. In a practical setting, employing pessimistic concurrency control might be tricky in a cluster environment (especially when the system is built with open-source components and deployed on the cloud platform). Careful performance analysis is necessary to validate if it is really worth employing. The best approach would depend on the requirement of a specific application, and our tool can help the developer to explore

various options.

## 8.5 Application-level Performance

To demonstrate a scenario of an application-level performance analysis, we compare HBase and MySQL using the auction benchmark. For MySQL we use two benchmark implementations based on Entity DB API and JPA, respectively.

The JPA version of auction benchmark uses join queries when they are applicable. For example, in an interaction that shows the information on all the bidding by a particular bidder, the tables of items and bids are joined together. In the auction workload, all the interactions that use join are read-only, and the number of tables joined is always 2.



**Figure 12: Throughput of auction benchmark with different session concurrency on different data stores**

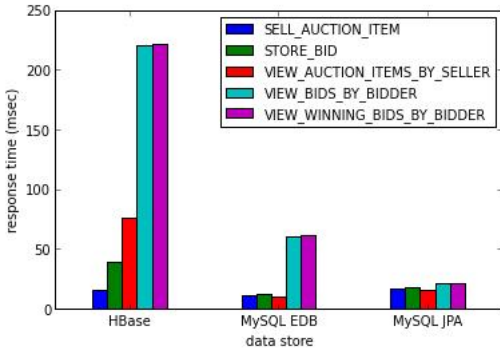
In this experiment, we increase the session concurrency from 200 to 3200 (200 threads per worker server) without think time on the same number of data servers (10 for HBase and 1 for MySQL). The number of users (and the size of the data set) is made proportional to the session concurrency (50 users per thread). The throughput of the workload is visualized in Figure 12.

As expected, HBase is scalable for an increasing number of concurrent user sessions. One observation, however, is that its throughput values are lower than the values of a single MySQL server when the number of concurrent sessions is small. This implies that MySQL’s execution of interactions with SQL is more efficient than executing the same interactions with put/get operations of HBase.

Another observation is that the JPA-based version performs better than Entity DB-based version on MySQL when the session concurrency is small, whereas the upper limit of throughput does not seem much different between these two implementations.

To see more detail of the efficiency of interaction execution, Figure 13 visualizes the average response time of individual interactions when the session concurrency is small (200). For the purpose of presentation, we only visualize 5 interactions picked up from 15 interactions used in the workload.

First, we observe the response time of two read-write interactions: sell-auction-item and store-bid. One noticeable point is that the store-bid interaction takes much longer time than the sell-auction-item on HBase (whereas the sell-auction-item performs similarly among three data stores).



**Figure 13: Response time of different interaction types in auction benchmark on different data stores (session concurrency = 200)**

The store-bid interaction creates one Bid entity and updates one AuctionItem entity in one transaction (i.e. updating one row). In fact, however, creating one Bid involves two additional row updates for out-of-group auxiliary data items: a key-generation counter and an index on the bidder id. On the other hand, key-generation and index maintenance are internal operations for MySQL, adding only negligible overhead.

We see much larger difference between HBase and MySQL for read-only interactions. We picked up three read-only interactions to represent three types of queries: (1) *view-auction-items-by-seller* gets auction items with a secondary key (the user ID of a seller). It illustrates different use of an index in Entity DB and JPA, (2) *view-bids-by-bidder* gets bids by a particular bidder as well as the corresponding auction items. The JPA-version uses a two-table join query with Bid and AuctionItem, (3) *view-winning-bids-by-bidder* gets bids by a particular bidder that *won* the auction items. The JPA-version uses a two-table join query with additional filtering conditions.

The view-auction-items-by-seller interaction reveals the difference in HBase and MySQL Entity DB: MySQL uses its internal index mechanism to implement Entity DB’s secondary key access, which is more efficient than an index object implemented on top of HBase. For this interaction, MySQL uses the same SQL for Entity DB version and JPA version (hence similar performance).

The view-bids-by-bidder interaction takes much longer time in Entity DB versions (MySQL and HBase) compared to the JPA-based implementation that uses a join query. However, the JPA-based implementation did not gain further benefit by adding filtering conditions for the view-winning-bids-by-bidder.

In a real development case, we need to take response time requirements for individual interactions to choose an implementation strategy. For example, 200 milliseconds for the view-bids-by-bidder interaction of HBase in the figure might not be acceptable for an interactive web application. The current implementation of this interaction executes a nested loop of get operations to emulate a join of Bid and AuctionItem. A possible improvement is to issue get operations asynchronously to hide latency of individual get responses.

## 8.6 Code Reusability

In addition to the above experiment scenarios, we also demonstrate the reusability of the code enabled by the Strudel framework.

Table 1 shows the size of components to implement the Entity DB interface for each NoSQL store. Each cell contains the lines of code and the number of classes (in a parenthesis). In the table, TKVS refers to the code of Transactional KVS (Section 4.4) that is commonly used by every implementation.

**Table 1: The size of store components: lines of code (number of classes)**

TKVS	HBase	Omid	MongoDB	TokuMX
3130 (36)	796 (6)	454 (4)	680 (4)	507 (4)

Table 2 shows the size of components to implement Auction and Micro benchmarks. The labels entity, param, and base correspond to definition of entity objects, parameters used in session interactions, and abstract interaction classes (Section 5), respectively. The remaining two columns, Entity DB and JPA, are components specific to data access APIs. The table does not include XML files that define session state transitions, which are part of configuration the developer can customize for specific experiments. The session state transition is agnostic to data access APIs.

**Table 2: The size of benchmark components: lines of code (number of classes)**

	entity	param	base	Entity DB	JPA
Auction	943 (9)	202 (3)	1346 (17)	1090 (18)	1043 (17)
Micro	681 (8)	212 (4)	1004 (19)	931 (19)	985 (19)

Notice that a more important point than the number of lines is *separation of concerns* achieved by the framework. For example, the benchmark components that are specific to data access APIs only need to implement individual data reads and writes that appear in the interactions.

## 8.7 Other Scenarios

Besides the scenarios the above demonstration covers, we have also used the Strudel framework for our research and development in a more customized manner. We developed custom components for our proprietary systems to run various experiments, including: (1) elasticity analysis to evaluate dynamic server scaling out (using a custom workflow that invokes various scripts to control data migration while a workload is running), (2) evaluation of bulk-loading APIs of NoSQL systems (using a custom workload that is not based on the session workload framework).

Especially, the elasticity analysis is essential to evaluate NoSQL systems. In a future version, we plan to include a generalized version of our custom components in the framework.

## 9. FUTURE WORK

We consider the following items in the future version of Strudel:

- Extended Entity DB API as a larger subset of JPA to incorporate more powerful query functionality of NoSQL (e.g. MongoDB) such as mapping parent-child entity relationship to a nested document (which enables retrieving parent and children together in one operation).
- Supporting multi-entity-group transactions on Entity DB API in a generic way to cover various solutions of multi-key transactions on NoSQL systems.
- Native EntityDB support of representative NoSQL systems such as HBase and MongoDB (Section 4.4.3).
- Better support of online analyses (e.g., an additional framework for scale-out analysis)
- Various data/workload generation (e.g. integration with such features from YCSB, OLTP-Bench).
- Better and easier integration with underlying infrastructure (e.g., software containers (e.g., Docker), resource managers (Hadoop YARN), and cloud platforms) as well as software configuration and deployment tools (e.g., Puppet [7]).

In actual applications, scalable transaction support is only part of the data management support. There are other data management features that must be considered: (1) entity search, (2) integration with analytic workloads. In either case, the developer has to choose if these functionality should be achieved by the same data store that serves transactions or done by external systems (search engines or analytic stores). Choice of SQL and NoSQL systems must take such features into account, which are beyond the scope of the current framework.

## 10. CONCLUSION

We introduce Strudel, a development and execution framework for transactional workloads both on SQL and NoSQL systems. Entity DB API provides a way to develop a benchmark using a common access API that is reasonably implementable on various NoSQL systems as well as RDBMS (through JPA). Session Workload framework provides another abstraction layer to decouple logic on data access (with a particular access API) from other logic in the benchmark (such as session state transition and parameter generation). We have implemented Entity DB API for various NoSQL systems by introducing a lower level API for transactional key-value access. A future version of the framework will explore custom EntityDB implementation on individual NoSQL systems to exploit advanced features of these systems (such as a query on a nested data structure).

## 11. REFERENCES

- [1] Congenio: Configuration generation language. [github.com/tatemura/congenio](https://github.com/tatemura/congenio).
- [2] HBase. [www.hbase.apache.org](http://www.hbase.apache.org).
- [3] MongoDB. [www.mongodb.org](http://www.mongodb.org).
- [4] MySQL. [www.mysql.com](http://www.mysql.com).
- [5] Omid. [github.com/yahoo/omid](https://github.com/yahoo/omid).
- [6] Otlp-bench. [oltpbenchmark.com](http://oltpbenchmark.com).
- [7] Puppet. [puppetlabs.com](http://puppetlabs.com).
- [8] RUBiS: Rice university bidding system. [rubis.ow2.org](http://rubis.ow2.org).
- [9] Strudel. [github.com/tatemura/strudel](https://github.com/tatemura/strudel).
- [10] TokuMX. [www.percona.com/software/mongodb-database/percona-tokumx](http://www.percona.com/software/mongodb-database/percona-tokumx).
- [11] TPC-W. [www.tpc.org/tpcw](http://www.tpc.org/tpcw).
- [12] Ycsb. [github.com/brianfrankcooper/YCSB](https://github.com/brianfrankcooper/YCSB).
- [13] CDDL configuration description language specification version 1.0. [www.ogf.org/documents/GFD.85.pdf](http://www.ogf.org/documents/GFD.85.pdf), 2006.
- [14] J. Baker, C. Bond, J. Corbett, and J. J. Furman et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.
- [15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.
- [16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154, 2010.
- [17] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.*, 38(1):5:1–5:45, Apr. 2013.
- [18] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.
- [19] D. G. Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 676–687, 2014.
- [20] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang. Can the elephants handle the nosql onslaught? *Proc. VLDB Endow.*, 5(12):1712–1723, Aug. 2012.
- [21] P. Helland. Life beyond distributed transactions: an apostate’s opinion. In *CIDR*, pages 132–141, 2007.
- [22] J. Klein, I. Gorton, N. Ernst, P. Donohoe, K. Pham, and C. Matser. Performance evaluation of nosql databases: A case study. In *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems, PABS ’15*, pages 5–10, New York, NY, USA, 2015. ACM.
- [23] S. Patil, M. Polte, K. Ren, W. Tantisiroroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. Ycsb++: Benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing, SOCC ’11*, pages 9:1–9:14, New York, NY, USA, 2011. ACM.
- [24] R. Sabharwal. Grid infrastructure deployment using smartfrog technology. In *Proceedings of the International Conference on Networking and Services, ICNS ’06*, pages 73–, Washington, DC, USA, 2006. IEEE Computer Society.
- [25] J. Tatemura, O. Po, W.-P. Hsiung, and H. Hacigümüs. Partiqle: an elastic sql engine over key-value stores. In *SIGMOD Conference*, pages 629–632, 2012.

# GROM: a General Rewriter of Semantic Mappings

Giansalvatore Mecca<sup>1</sup> Guillem Rull<sup>2</sup> Donatello Santoro<sup>1</sup> Ernest Teniente<sup>3</sup>

<sup>1</sup> Università della Basilicata – Potenza, Italy

<sup>2</sup> Universitat de Barcelona — Barcelona, Spain

<sup>3</sup> Universitat Politècnica de Catalunya — Barcelona, Spain

## ABSTRACT

We present GROM, a tool conceived to handle high-level schema mappings between semantic descriptions of a source and a target database. GROM rewrites mappings between the virtual, view-based semantic schemas, in terms of mappings between the two physical databases, and then executes them. The system serves the purpose of teaching two main lessons. First, designing mappings among higher-level descriptions is often simpler than working with the original schemas. Second, as soon as the view-definition language becomes more expressive, to handle, for example, negation, the mapping problem becomes extremely challenging from the technical viewpoint, so that one needs to find a proper trade-off between expressiveness and scalability.

## 1. INTRODUCTION

Many applications benefit from the availability of a *semantic schema* over a database, i.e., a set of views over the base tables that provide a richer description of the semantic relationships among the underlying data and a more accurate definition of the constraints. The use of such semantic views has been thoroughly studied for the purpose of query languages [6], data integration [1], and data access [2], but there are little studies of how the presence of these views impacts *data exchange* [4] applications.

Data exchange consists of moving data from a source database to a target database. This task is usually performed by developing *schema mappings*, i.e. executable transformations that specify how an instance of the source repository can be translated into an instance of the target.

In this paper, we present GROM [9, 8], a system conceived to support the management of mappings among view schemas. GROM was designed to handle mapping scenarios in which a semantic description is available over the target database, and possibly over the source database. It allows data architects to develop mappings among the two semantic schemas, rather than the underlying database schemas. Studying this variant of the problem is important for several

reasons:

- (i) The semantic web has increased the number of data sources on top of which such descriptions are developed.
- (ii) Views play a key role in information integration since they are used to give clients a global conceptual view of the underlying data, which may come from external, independent and heterogeneous information systems [7].
- (iii) Many of the base transactional repositories used in complex organizations often undergo modifications during the years, and may lose their original design. It is important to be able to run the existing mappings against a view over the new schema that does not change, thus keeping these modifications of the sources transparent to the users.

More generally, semantic schemas help to improve the overall design of the original schemas, and emphasize important semantic relationships and constraints that would not be apparent otherwise. Therefore, designing rich, high-level mappings between these schemas has often significant advantages. However, semantic schemas are virtual and mappings between them are not directly executable.

GROM solves the important problem of making these high-level mappings executable over the original database instances. It rewrites mappings between the two virtual semantic schemas under the form of standard mappings over the underlying concrete databases, in order to execute them and generate an instance of the target database from an instance of the source database. Under appropriate hypothesis, discussed in the next sections, the whole process happens in a completely transparent way, thus greatly simplifying the overall data-translation task.

Essential to this problem is the trade-off between expressiveness and complexity. In fact, the rewriting is fairly straightforward if views are conjunctive queries – it reduces to the standard view unfolding algorithm. However, conjunctive queries have a limited expressive power, unable to capture many semantic relationships between the data. Negation, for instance, is crucial to capture disjointness constraints and many classification rules.

The main concern behind the design of GROM was to provide an expressive view language that can truly benefit data architects in defining rich semantic abstractions. To this end, we adopt the language of non-recursive Datalog with negation. This makes the rewriting significantly more complex, as we discuss in Section 3.

In the following we describe how we plan to organize the demonstration of GROM. We outline the kind of mapping scenarios that will be considered with the help of a running example introducing the main features of the system. Given

©2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0  
EDBT 2016

the focus of this proposal, we have chosen to omit many of the technical details that are in published papers [9]. We concentrate on a description of the system from the user perspective and illustrate the main technical challenges and what an attendee may learn by playing with it.

The system is available under an open-source license at the following URL: <http://db.unibas.it/projects/grom/>.

## 2. MAPPING REWRITING

Assume we have the two relational schemas below and we need to translate data from the source to the target.

Source schema:  $S\text{-Product}(id, name, store, rating)$   
 $S\text{-Store}(name, location)$

Target schema:  $T\text{-Product}(id, name, store)$   
 $T\text{-Store}(id, name, address, phone)$   
 $T\text{-Rating}(id, product, thumbsUp)$

Both schemas refer to the same domain, which includes data about products, stores, and ratings. Due to the different organization of the two databases, it is not evident how to define the source-to-target mapping. In particular, it is difficult to relate tuples in the  $T\text{-Rating}$  target table to those in the source. Suppose now that a richer semantic schema has been defined over the target, as shown in Figure 1. To simplify things, in this example we consider that only the target database comes with an associated semantic schema; we discuss the more general case in the next section.

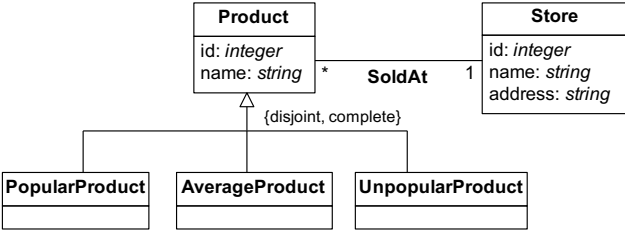


Figure 1: A Simple Target Semantic Schema.

The semantic schema distinguishes among popular, unpopular, and average products. Each concept and association is defined in terms of the database tables by means of a set of views, as follows (we use different fonts for semantic concepts and relational tables; in addition, source tables have a  $S$ -prefix in their name, and target tables a  $T$ -prefix; we use values 0 and 1 for the  $thumbsUp$  attribute):

$$\begin{aligned}
 v_1 : \text{Product}(id, name) &\Leftarrow T\text{-Product}(id, name, store) \\
 v_2 : \text{PopularProduct}(pid, name) &\Leftarrow \\
 &T\text{-Product}(pid, name, store), \neg T\text{-Rating}(rid, pid, 0) \\
 v_3 : \text{AvgProduct}(pid, name) &\Leftarrow \\
 &T\text{-Product}(pid, name, store), T\text{-Rating}(rid, pid, 1), \\
 &\neg \text{PopularProduct}(pid, name) \\
 v_4 : \text{UnpopularProduct}(pid, name) &\Leftarrow \\
 &T\text{-Product}(pid, name, store), \\
 &\neg \text{AvgProduct}(pid, name), \neg \text{PopularProduct}(pid, name) \\
 v_5 : \text{SoldAt}(pid, std) &\Leftarrow T\text{-Product}(pid, pname, std) \\
 v_6 : \text{Store}(id, name, addr) &\Leftarrow T\text{-Store}(id, name, addr, phone)
 \end{aligned}$$

We adopt the expressive language of non-recursive Data-log with negation. Notice how negation is crucial to capture the semantics of this example and it may either correspond to negated base tables (view  $v_2$ , table  $T\text{-Rating}$ ) or even to negated views ( $v_3$ ,  $\text{PopularProduct}$ ). Views can also be defined as unions of queries (not shown in the example).

The important observation here is that in many cases semantic concepts are closer to source data than physical target tables, and therefore the task of defining mappings is considerably simplified. In our example, notice how the views hide table  $T\text{-Rating}$ . As a consequence, the classification of a product in the target semantic schema is easily derived from ratings in the source database as follows: products with ratings consistently above 4 stars (out of 5) are the popular ones, those always graded less than 2 are considered to be unpopular, and the rest are average.

As is common [4], we use *tuple generating dependencies* (tgds) and *equality-generating dependencies* (egds) to express the mapping. In our case, the *source-to-semantic* translation can be expressed by using the following tgds with comparison atoms:

$$\begin{aligned}
 m_0 : \forall pid, name, store, rating : \\
 &S\text{-Product}(pid, name, store, rating), rating < 2 \\
 &\rightarrow \text{UnpopularProduct}(pid, name) \\
 m_1 : \forall pid, name, store, rating : \\
 &S\text{-Product}(pid, name, store, rating), \\
 &rating \geq 2, rating < 4 \rightarrow \text{AvgProduct}(pid, name) \\
 m_2 : \forall pid, name, store, rating : \\
 &S\text{-Product}(pid, name, store, rating), \\
 &rating \geq 4 \rightarrow \text{PopularProduct}(pid, name) \\
 m_3 : \forall pid, name, store, rating, location : \\
 &S\text{-Product}(pid, name, store, rating), \\
 &S\text{-Store}(store, location) \\
 &\rightarrow \text{SoldAt}(pid, sid), \text{Store}(sid, store, location)
 \end{aligned}$$

Intuitively, tgd  $m_0$  specifies that, for each tuple in  $S\text{-Product}$  such that the value of  $rating$  is lower than 2, there should be an  $\text{UnpopularProduct}$  in the semantic schema. Similarly for  $m_1$  and  $m_2$ . Mapping  $m_3$  relates products and stores in the source to instances of  $\text{SoldAt}$  association in the semantic schema.

The mapping designer can also express a number of constraints about the target semantic schema under the form of egds.<sup>1</sup> The egd below corresponds to a key constraint on  $\text{PopularProducts}$ : it states that whenever two popular products have the same name, their id must also be the same:

$$e_0 : \forall id_1, id_2, n : \text{PopularProduct}(id_1, n), \\
 \text{PopularProduct}(id_2, n) \rightarrow id_1 = id_2$$

In addition to being more natural, designing mappings over semantic schemas has another important benefit to the data architect. By taking advantage of the semantics of the views, the mapping designer does not need to care about the physical structure of the data in the target schema. As an example, s/he does not need to explicitly state in  $m_0$ ,  $m_1$ ,  $m_2$  that popular, average, and unpopular products are also products. The class-subclass relationships are encoded within the view definitions, and we expect their semantics to carry on into the mappings.

This, however, is true provided that we are able to translate such a *source-to-semantic* virtual mapping into a classical, executable source-to-target mapping among the two physical databases. This is the main task performed by GROM, as discussed in the following section.

## 3. OVERVIEW OF THE SYSTEM

The main technical problem addressed by GROM, depicted in Figure 2, can be stated as follows. Assume we are given:

<sup>1</sup>Previous papers [9] discuss how to handle foreign-key constraints as well.

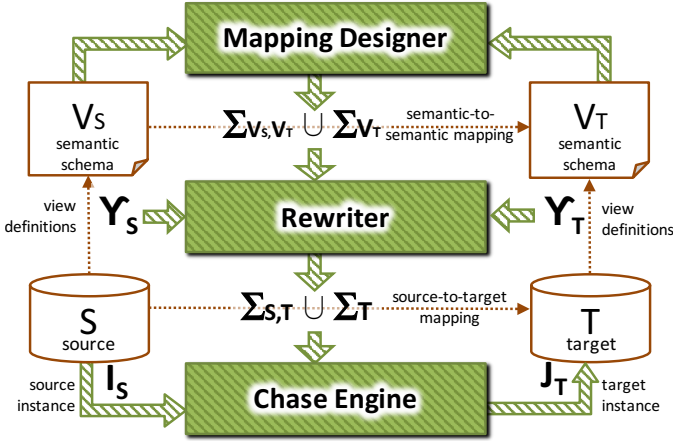


Figure 2: Architecture of the System.

- (i) a source relational schema,  $\mathbf{S}$ , and a target relational schema  $\mathbf{T}$ ;
- (ii) a source semantic schema,  $\mathbf{V}_S$ , and a target semantic schema,  $\mathbf{V}_T$ , defined by means of sets of view definitions,  $\Upsilon_S$  and  $\Upsilon_T$ , over  $\mathbf{S}$ ,  $\mathbf{T}$  respectively. View definitions may involve negations over base and derived atoms, as discussed in Section 2;
- (iii) a set of target constraints,  $\Sigma_{V_T}$ , i.e. target egds to encode key constraints and functional dependencies over the semantic schema;
- (iv) finally, a semantic-to-semantic mapping,  $\Sigma_{V_S, V_T}$ , defined as a set of s-t tgds over  $\mathbf{V}_S$  and  $\mathbf{V}_T$ .

As it can be seen from Figures 2 and 3, the system is composed of various modules. Users develop the semantic mappings using a graphical mapping-designer and view browser. The GROM rewriter takes as input  $\mathbf{S}$ ,  $\mathbf{T}$ ,  $\mathbf{V}_S$ ,  $\mathbf{V}_T$ ,  $\Upsilon_S$ ,  $\Upsilon_T$ , and the semantic-based mappings,  $\Sigma_{V_S, V_T} \cup \Sigma_{V_T}$ . It rewrites these as a new set of source-to-target dependencies  $\Sigma_{S, T} \cup \Sigma_T$ , from the source to the target database. These, in turn, are fed to the chase-engine module, along with an instance  $I_S$  of the source database, to be executed and generate an instance  $J_T$  of the target. A few observations are in place.

**Variants of the Problem.** First, this general version of the rewriting problem easily reduces to a simplified variant in which only a target semantic schema is available, and no source one, as in our running example in Section 2. In fact, assume we know how to rewrite source-to-semantic mappings  $\Sigma_{SV_T}$ , i.e., mappings designed from the source schema  $\mathbf{S}$  to the target semantic schema  $\mathbf{V}_T$ . Assume now we are given view definitions for the source schema,  $\Upsilon_{V_S}$ , in addition to the target ones, and a mapping  $\Sigma_{V_S, V_T}$  between the two semantic schemas. It can be seen that this case can be reduced to the simpler case by using the composition of two steps [9]: (i) first, we apply the source view definitions in  $\Upsilon_{V_S}$  to the source instance,  $I_S$ , to materialize the extent of the source views,  $\Upsilon_{V_S}(I_S)$ ; (b) then, we consider this materialized instance as a new source database, and solve the source-to-semantic mapping problem.

**The Mapping Language.** A second, important observation is concerned with the output of the rewriting engine. It is known [1] that the language of embedded dependencies (tgds and egds) is closed wrt unfolding conjunctive views, i.e. the result of unfolding a set of conjunctive view defini-

tions within a set of tgds and egds is still a set of tgds and egds. Unfortunately, as we have shown in [9], this is not true when views allow for negated atoms, as in our setting. This justifies two important choices wrt the algorithm:

To start, the rewriting algorithm is sound but not complete. Informally speaking, given mappings  $\Sigma_{SV_T} \cup \Sigma_{V_T}$ , GROM generates a rewritten set of source-to-target mappings  $\Sigma_{ST} \cup \Sigma_T$  such that, whenever these admit a universal solution [4]  $J_T$  over  $I_S$ , then also the original source-to-semantic mappings admit solutions on  $I_S$ , and it is the case that  $\Upsilon_T(J_T)$  is a solution for  $\Sigma_{SV_T} \cup \Sigma_{V_T}$  and  $I_S$ . However, we say nothing about the cases in which  $\Sigma_{ST} \cup \Sigma_T$  fail.

Then, as we mentioned, to better handle the effects of negation in view definitions, we choose as a mapping-definition language for  $\Sigma_{ST} \cup \Sigma_T$  the one of *disjunctive embedded dependencies (deds)*. Deds generalize tgds and egds since they may have disjunctions in the conclusion. Following is a ded generated by GROM for the running example in Section 2:

$$d_0 : TProduct(pid_1, name, store_1), \\ TProduct(pid_2, name, store_2) \rightarrow (pid_1 = pid_2) \mid \\ TRating(rid, pid_1, '0') \mid TRating(rid, pid_2, '0')$$

Intuitively, this ded translates the key constraint for **name** on concept **PopularProduct** in terms of the following constraints over the target database: for each pair of tuples in  $TProduct$  with equal values of *name*, one of the following must be true: either the two product ids are equal; or one of the products is not a popular product.

Handling deds is considerably more challenging than ordinary tgds and egds. To provide an example, *universal solutions* [4] are considered the standard notion of what a “good” solution means for standard mappings composed of tgds and egds; in addition, the *chase* is a well-known, polynomial-time procedure to generate universal solutions. On the contrary, it has been shown [3] that universal solutions are no longer sufficient for ded-based scenarios, and that the more appropriate notion of *universal model set* is needed. In addition, universal model sets may have exponential size wrt to the size of the source instance. In fact, to the best of our knowledge, GROM is the first system to tackle the problem of chasing deds.

**Handling Complexity.** The strategy to avoid such a complexity blow-up is twofold. On the one side, sufficient conditions to avoid the use of deds in the output mappings have been identified under the form of restrictions on the use of negations in view definitions [9]. As a consequence, the system is able to look at the view definitions and tell whether the rewritten mappings may contain deds or not.

On the other side, when deds are unavoidable, GROM takes special care in order to tame the complexity of the chase. To start, it relies on a fast and scalable chase engine from the LLUNATIC project [5]. This guarantees good scalability in executing mappings, even on large databases. In addition, the chase engine has been extended in order to properly handle deds by implementing a greedy chase strategy for deds [9], based on the ideas of searching for solutions to a set of deds by running multiple standard scenarios made of tgds and egds derived from the given deds. Experiments confirm the effectiveness of this approach.

## 4. EXPERIENCES WITH THE SYSTEM

The demonstration will illustrate what are the main challenges in handling semantic mappings and how the system

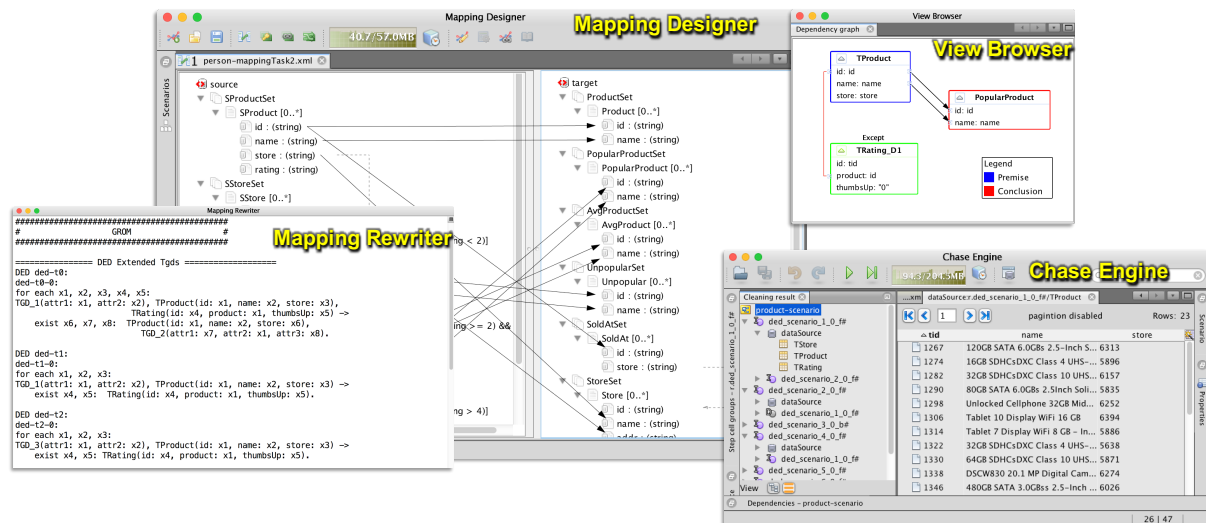


Figure 3: GROM in Action.

solves them. Attendees will be able to interact directly with the system, in such a way that the process will resemble a hands-on tutorial. Following are the main lessons that can be learned from these experiences.

**Semantic Mappings Work!** Our experiences tell us that in many cases the availability of a view over the data may greatly simplify the mapping process. In these cases, data architects may greatly benefit from a tool like GROM.

One of the typical patterns is the one discussed in our running example: one of the data sources somehow rates source objects, and the mapping application requires to classify objects in the target based on these ratings, for example for the purpose of showing them to users under the form of web pages. Often, target relational schemas are not designed to properly address this kind of need. Being able to design a view over the target database that more closely reflects such an application requirement is a great asset in these scenarios.

Another, typical case, is the one of databases that come with poor designs, or lack integrity constraints. It is very difficult in these cases to design proper mappings. On the contrary, a clean-up view over the underlying databases may simplify things.

We intend to challenge the audience with different schemas and mapping scenarios. We will ask attendees to design mappings first using the original, relational schemas, and then over properly designed views, to let them grasp the real advantage of this approach.

**Semantic Mappings are Expensive!** At the same time, it is important to let users understand the concrete trade-off between having a flexible and expressive view-definition language, and the cost of executing the mappings.

As we mentioned, rewriting and executing the mappings is quite straightforward as soon as conjunctive queries are used as a view definition language. This, however, is not sufficient to capture the actual modeling requirements in many cases.

Negation is a powerful addition, but it comes at a cost. The full power of negation generates output mappings that include ded-ids, so that chasing them is not feasible, even on small instances. Attendees will learn what features GROM offers to solve this problem. As a first solution, the system will run its greedy chase algorithm to search for solutions to the original ded-ids. This amounts to generating several scenarios made of tgds and egds, that capture specific branches in the ded-ids. This strategy is sound, but not complete. How-

ever, it is often surprisingly quick in returning some solution.

In other cases, when the constraints are more intricate, the greedy chase will take considerably more time, due to the fact that many of the generated scenarios fail to generate a solution, and new ones need to be executed. In these cases, a possible alternative is to leverage the syntactic restrictions over the use of negation [9] that guarantee that no ded-ids are generated. In essence, the user needs to inspect the views and change them in such a way to remove perverse negation patterns that will generate ded-ids. GROM supports this process by highlighting problematic views, so that the user may consider alternative formulations.

## 5. REFERENCES

- [1] A. Cali, D. Calvanese, G. De Giacomo, and M. Lenzerini. Data integration under integrity constraints. *Inf. Syst.*, 29(2):147–163, 2004.
- [2] C. Civili, M. Console, G. De Giacomo, D. Lembo, M. Lenzerini, L. Lepore, R. Mancini, A. Poggi, R. Rosati, M. Ruzzi, V. Santarelli, and D. Savo. MASTRO STUDIO: managing ontology-based data access applications. *Proc. of the VLDB Endowment*, 6(12):1314–1317, 2013.
- [3] A. Deutsch, A. Nash, and J. Rummel. The chase revisited. In *PODS '08*, pages 149–158, 2008.
- [4] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *Theoretical Computer Science*, 336(1):89–124, 2005.
- [5] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That's All Folks! LLUNATIC Goes Open Source. *PVLDB*, 7(13):1565–1568, 2014. <http://db.unibas.it/projects/llunatic>.
- [6] A. Y. Halevy. Answering Queries Using Views: A Survey. *VLDB Journal*, 10(4):270–294, 2001.
- [7] M. Lenzerini. Data integration: a Theoretical Perspective. In *PODS*, 2002.
- [8] G. Mecca, G. Rull, D. Santoro, and E. Teniente. Semantic-Based Mappings. In *Proc. of the Int. Conf. on Conceptual Modeling (ER)*, pages 255–269, 2013.
- [9] G. Mecca, G. Rull, D. Santoro, and E. Teniente. Ontology-based mappings. *Data and Knowledge Engineering*, 98:8–29, July 2015. <http://dx.doi.org/10.1016/j.datak.2015.07.003>.

# PowerQ: An Interactive Keyword Search Engine for Aggregate Queries on Relational Databases

Zhong Zeng  
National University of  
Singapore  
zengzh@comp.nus.edu.sg

Mong Li Lee  
National University of  
Singapore  
leeml@comp.nus.edu.sg

Tok Wang Ling  
National University of  
Singapore  
lingtw@comp.nus.edu.sg

## ABSTRACT

Keyword search over relational databases has gained popularity due to its ease of use. Current research has focused on the efficient computation of results from multiple tuples, and largely ignores queries to retrieve statistical information from databases. The work in [5] developed a system that allows aggregate functions to be expressed using simple keywords. However, this system may return incorrect answers because it does not consider the semantics of objects and relationships in the database. In this paper, we present an interactive keyword search engine called PowerQ to answer queries involving aggregate functions and GROUPBY. PowerQ utilizes an ORM schema graph to capture the Object-Relationship-Attribute (ORA) semantics in the database. Given a keyword query, PowerQ identifies the various interpretations of the query and applies aggregate functions and GROUPBY on the appropriate attributes of objects/relationships. Each query interpretation is denoted as an annotated query pattern, whose meaning can be described in natural language to facilitate user understanding. Through user interactions, PowerQ can determine the user's search intention, and translate the corresponding patterns into SQLs to compute the answers correctly. The PowerQ prototype is available at <http://powerq.comp.nus.edu.sg>.

## 1. INTRODUCTION

As databases increase in size and complexity, the ability for users to issue SQL queries has become a challenge. Keyword search over relational databases has gained popularity as it enables users to query the database without knowing the database schema or writing complicated SQL queries. Research on relational keyword search has focused on the efficient computation of results from multiple tuples [1, 2, 4], and largely ignores queries involving aggregates and GROUPBY. We call the latter *aggregate queries*.

Aggregate queries provide a powerful mechanism to retrieve statistical information from the database. The work in [5] designed a prototype system called SQAK to handle

aggregate queries. An aggregate query comprises of a set of terms and one of these terms is an aggregate function such as *COUNT*, *SUM*, etc. The terms in the query may match the names of relations or attributes or tuple values.

Consider the sample university database in Figure 1. Suppose we want to know the total credits obtained by the student Green, we can issue the aggregate query  $Q_1 = \{\text{Green SUM Credit}\}$ . However, we observe that incorrect answers may be returned by SQAK. For example, the term Green in  $Q_1$  matches the names of two students  $s_2$  and  $s_3$  in Figure 1. This naturally implies that we should find the sum of the credits for each of these students, that is, the total credits for  $s_2$  is 5 while the total credits for  $s_3$  is 8. However, SQAK does not distinguish between these two “different” name matches, and outputs a total credits of 13 for students called Green, which is incorrect.

Student			Course			Enrol			Teach		
Sid	Sname	Age	Code	Title	Credit	Sid	Code	Grade	Code	Lid	Bid
s1	George	24	c1	Java	5.0	s1	c1	A	c1	l1	b1
s2	Green	18	c2	Database	4.0	s1	c2	B	c1	l1	b2
s3	Green	21	c3	Multimedia	3.0	s1	c3	B	c1	l2	b1
						s2	c1	A	c2	l2	b2
						s3	c1	A	c2	l2	b3
						s3	c3	B	c3	l1	b4

Textbook			Lecturer			Department			Faculty	
Bid	Tname	Price	Lid	Lname	Did	Did	Dname	Fid	Fid	Fname
b1	Programming Language	10								
b2	Discrete Mathematics	15								
b3	Database Management	12	l1	George	d1					
b4	Multimedia Technologies	20	l2	Steven	d1	d1	CS	f1	f1	Engineering

Figure 1: Sample university database

Next, suppose we issue the query  $Q_2 = \{\text{Java SUM Price}\}$  to find the total price of the textbooks that are used in the Java course. The term Java matches a course title while the term Price matches an attribute of the *Textbook* relation. This relation contains 3 foreign keys that reference the *Course*, *Lecturer* and *Textbook* relations respectively, and represents that a course can be taught by more than one lecturer using different textbooks. We see that there are 2 such textbooks, namely,  $b_1$  used by both lecturers  $l_1$  and  $l_2$ , and  $b_2$  used by lecturer  $l_1$ . But SQAK does not detect the duplicate textbook  $b_1$  by different lecturers of the Java course (i.e.,  $c_1$ ) in the *Teach* relation, and returns 35 for the total price. This answer is incorrect as students do not need 2 copies of a textbook for the same course.

In this work, we build a relational keyword search engine called PowerQ to answer aggregate queries correctly. PowerQ extends the keyword query language and utilizes the



ORM schema graph [6] to capture the Object-Relationship-Attribute (ORA) semantics in the database. Given an aggregate query, it identifies the various interpretations of the query and applies aggregate functions and GROUPBY on the appropriate attributes of objects/relationships. Each query interpretation is denoted as a graph called annotated query pattern, whose meaning is described in natural language. The query patterns that satisfy the user's search intention are translated into SQL statements to compute the answers. During the query processing, PowerQ utilizes the ORA semantics to distinguish the objects with the same attribute value and detect the duplications of objects/relationships regardless of whether the database is normalized or not. Otherwise, the aggregate function(s) cannot be computed correctly as we have shown in our example queries  $Q_1$  and  $Q_2$ .

## 2. PRELIMINARIES

The work in [6] extends the keyword query language to include the keywords that match the names of relations and attributes. These metadata keywords provide the context of subsequent keywords and reduce the query ambiguity.

PowerQ further extends the query language to incorporate aggregates and GROUPBY. Thus, a keyword query  $Q$  is a sequence of terms  $\{t_1 t_2 \dots t_n\}$  where each term  $t_i$  either matches a relation name, an attribute name, a tuple value, GROUPBY or an aggregate function  $COUNT$ ,  $SUM$ ,  $AVG$ ,  $MIN$  or  $MAX$ .

### 2.1 ORM Schema Graph

The work in [7] classifies the relations in a database into object relations, relationship relations, mixed relations and component relations. An object (or relationship) relation captures the information of objects (or relationships), i.e., the single-valued attributes of an object class (relationship type). Multivalued attributes of an object class (relationship type) are stored in object/relationship component relations. A mixed relation contains information of both objects and relationships, which occurs when we have a many-to-one or one-to-one relationship. We call these semantics the Object-Relationship-Attribute (ORA) semantics.

The Object-Relationship-Mixed (ORM) schema graph is an undirected graph that captures the ORA semantics in the database. Each node in the graph comprises of an object/relationship/mixed relation and its component relations, and is associated with a type (object, relationship and mixed). Two nodes are connected if there exists a foreign key - key reference between the relations in these two nodes.

In Figure 1, the relations *Student*, *Course*, *Faculty* and *Textbook* are object relations while *Enrol* and *Teach* are relationship relations. Relations *Lecturer* and *Department* are mixed relations because of the many-to-one relationships between lecturers and departments, and the many-to-one relationships between departments and faculties respectively. Figure 2 shows the ORM schema graph of the database.

### 2.2 Query Patterns

Since keyword queries are inherently ambiguous, [6] introduces the notion of query patterns to represent the various interpretations of a query. These query patterns are generated from the ORM schema graph of the relational database.

Figure 3 shows one of the query patterns for the keyword query  $\{\text{Code George Green}\}$ . This pattern depicts the

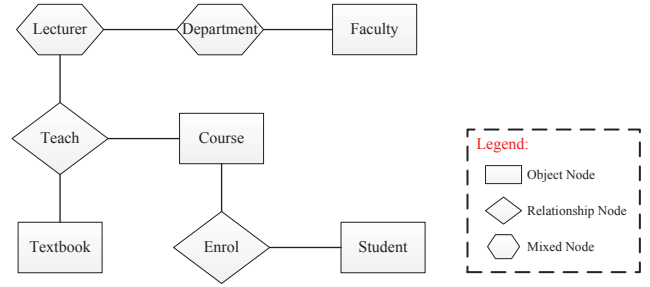


Figure 2: ORM schema graph of Figure 1



Figure 3: Query pattern of  $\{\text{Code George Green}\}$

query interpretation to find information on the course which is taught by the lecturer George and enrolled by the student Green. To generate this query pattern, we identify the matches of each term in the query. The term Code matches the name of an attribute in the *Course* relation, while the terms George and Green match the values of the attribute *Lname* in the *Lecturer* relation and the attribute *Sname* in the *Student* relation respectively. Based on these matches, we know that Code refers to a course object, George refers to a lecturer object, and Green refers to a student object. From the ORM schema graph in Figure 2, the *Course*, *Lecturer* and *Student* nodes can be connected via a *Teach* and an *Enrol* node. Hence, we create these two nodes and obtain the query pattern in Figure 3.

PowerQ utilizes query patterns to capture the interpretations of an aggregate query. However, since an aggregate query includes aggregate functions and GROUPBY, we need to annotate the patterns to indicate the objects/relationships that aggregates and GROUPBY are applicable to. We will discuss how to achieve this in the next section.

## 3. SYSTEM ARCHITECTURE

PowerQ takes as input an aggregate query, and generates a set of SQL statements for the query patterns that satisfy the user's search intention. Figure 4 shows the architecture of PowerQ. The frontend of PowerQ interacts with the user during the query processing, while the backend communicates with the database and the ORM schema graph to compute the query answers. The main components in PowerQ are *Query Parser/Analyzer*, *Query Interpreter*, *SQL Generator*, *Visualization Module* and *Normalization Module*. The following sections give the details of these components.

### 3.1 Query Parser/Analyzer

Given an aggregate query, the Query Parser/Analyzer classifies the terms in the query into basic terms and operators. A basic term matches a relation name, or an attribute name, or a tuple value in the database, while an operator matches an aggregate function or GROUPBY. For the basic terms, the Query parser/Analyzer obtains their matches and determines the objects/relationships referred to by these terms based on the ORM schema graph of the database.

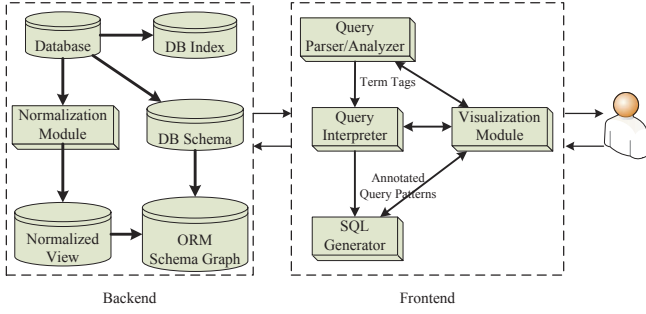


Figure 4: System Architecture

### 3.2 Query Interpreter

Next, the Query Interpreter generates a set of initial query patterns based on the basic terms of the query and the ORM schema graph of the database. Each query pattern contains a set of nodes that represents the objects/relationships referred to by the basic terms. Then, it annotates the query patterns with operators in the query. For each operator  $t_i$ , if its subsequent term  $t_{i+1}$  refers to some object or relationship, then the Query Interpreter annotates the corresponding node with  $t_i(id)$ , where  $id$  is the identifier of the object/relationship; otherwise, if  $t_{i+1}$  refers to some attribute  $a$  of an object or relationship, the Query Interpreter annotates the corresponding node with  $t_i(a)$ .

Consider the keyword query {COUNT Code George Green}. Figure 3 shows a query pattern obtained using the basic terms Code, George and Green. For the operator COUNT, since its subsequent term Code matches the name of an attribute in the *Course* relation, we will annotate the *Course* node with COUNT(Code), and obtain the annotated query pattern  $P_1$  in Figure 5. This pattern depicts the query interpretation to find the total number of courses which are taught by lecturer George and enrolled by student Green.

In an annotated query pattern, an object/mixed node with the condition  $a = t$  refers to an object such that its value of attribute  $a$  matches the basic term  $t$ . However, since this condition could be satisfied by more than one object in the database, we have two different query interpretations:

1. apply the aggregate functions(s) for every *distinct* object satisfying  $a = t$ ; or
2. apply the aggregate function(s) for *all* the objects satisfying  $a = t$ .

The Query Interpreter distinguishes these two interpretations by annotating the object/mixed node in the pattern with GROUPBY( $id$ ), where  $id$  is the identifier of the object. By applying GROUPBY on object identifiers, we can distinguish objects with the same attribute value and compute the aggregate functions for each of them.

In Figure 5, the annotated query pattern  $P_1$  contains a *Student* node that is annotated with the condition  $Sname = Green$ . From the database in Figure 1, we know that there are two students called Green. Hence, we have a second query pattern  $P_2$  that is similar to  $P_1$ , except that we annotate the *Student* node in  $P_2$  with GROUPBY(Sid). Figure 5 shows these two patterns:  $P_1$  counts the number of courses for all the students called Green, while  $P_2$  counts the number of courses for each student called Green separately.

Note that SQAk [5] does not distinguish  $P_1$  and  $P_2$ , and thus may return incorrect answers to the query.

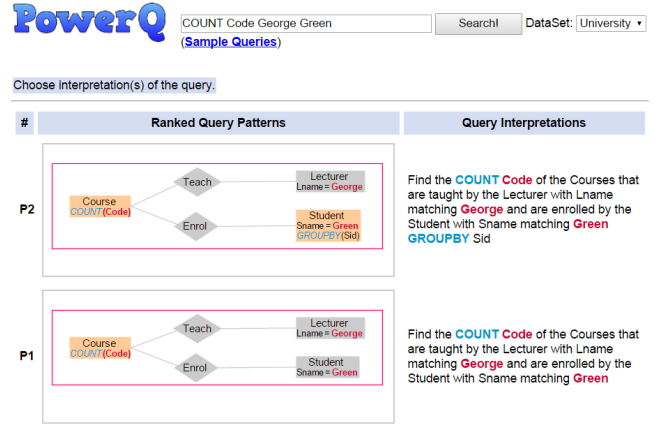


Figure 5: Screenshot of annotated query patterns

### 3.3 SQL Generator

The SQL Generator translates an annotated query pattern into an SQL statement to compute the answers. The straightforward approach is to join the relations of all the nodes, select the tuples that satisfy the conditions imposed by basic terms from the join result, and then apply aggregate(s) and GROUPBY on the selected tuples. However, this may generate an SQL that gives an incorrect answer.

Consider the query pattern  $P_2$  in Figure 5. If we simply translate  $P_2$  into an SQL that joins the relations *Course*, *Teach*, *Enrol*, *Lecturer* and *Student*, selects the tuples with conditions  $Lname = George$  and  $Sname = Green$ , and then applies the count aggregate and GROUPBY on the course code and the student id respectively, we will obtain wrong answers as the same course may be counted multiple times. This is because the *Teach* node in  $P_2$  is in fact a ternary relationship involving course, lecturer and textbook objects (see the ORM schema graph in Figure 2). The same course can be taught by a lecturer using different textbooks. In other words, the same *Lid* and *Code* are duplicated for different *Bid* in the *Teach* relation.

To avoid this problem, PowerQ examines every relationship node  $u$  in the pattern, and checks its corresponding node  $v$  in the ORM schema graph. If the pattern only contains a subset of the participating objects in relationship  $v$ , then it projects the identifiers of these objects from  $v$ . This eliminates duplicates and PowerQ replaces the relation of  $u$  with the relation obtained by this projection in the SQL.

For example, since the *Teach* node in  $P_2$  only involves course and lecturer objects, PowerQ generates a subquery “SELECT DISTINCT Lid, Code FROM Teach” to project the attributes *Lid* and *Code* in the *Teach* relation. This subquery has a “DISTINCT” keyword, thus eliminating duplicates of  $\langle Lid, Code \rangle$ . We use this subquery result to join the other relations in the FROM clause as follows:

```
SELECT S.Sid, COUNT(C.Code)
FROM Lecturer L, Course C, Enrol E, Student S
  (SELECT DISTINCT Lid, Code FROM Teach) T
WHERE L.Lid=T.Lid AND C.Code=T.Code AND
      S.Sid=E.Sid AND C.Code=E.Code
GROUP BY S.Sid
```

Note that SQAk does not detect the duplicates of courses in *Teach* relationships, and thus returns incorrect answers.

### 3.4 Visualization Module

A keyword query is inherently ambiguous. However, the user who issues the query often has some particular search intention in mind [3]. The Visualization Module represents the various interpretations of a keyword query, and actively interacts with the user to obtain the interpretations that satisfy the user’s search intention. In particular, if a term has multiple matches in the database and refers to different objects/relationships, the user is offered the opportunity to choose the matches. Further, if more than one query pattern is constructed for the query, the user is again allowed to choose his/her intended query patterns.

One feature of PowerQ is that it represents query interpretations visually and describes them in human natural language in order to facilitate users’ understanding. For instance, the annotated query pattern  $P_2$  in Figure 5 is represented as a graph annotated with the ORA semantics. The nodes with operators in the graph are highlighted to indicate the objects/relationships that aggregates are applicable to. The description of this pattern is to “Find the count of the courses that are taught by the lecturer with name matching **George** and are enrolled by the student with name matching **Green** group by Sid”. The user can easily identify the intended query interpretation by the graph structure, and verify its meaning by the description. After the user chooses a query pattern, PowerQ computes the answers and represents them according to the corresponding search intention. Figure 6 shows the screenshot of the interface which displays the query answers for the query pattern  $P_2$  in Figure 5 and the detailed information for user to verify the answers.

### 3.5 Normalization Module

Relations in a relational database are often denormalized to improve query processing performance. This denormalization process will duplicate information of objects and relationships in the database and SQAK may obtain incorrect answers for an aggregate query.

PowerQ is able to detect denormalization and keep track of the object/relationship information in the database to answer aggregate queries correctly. This is achieved by examining the functional dependencies hold on the relations. If the database is denormalized, then it generates a normalized view of the database which comprises of a minimal set of normalized relations, and obtains the mappings of relations in the normalized view and the original schema. The normalized view is used to construct the ORM schema graph of the denormalized database and build query patterns of the query, while the mappings are used to generate the SQL statements which continue to compute the answers correctly. Interested readers can refer to [8] for details.

## 4. DEMONSTRATION

In this demonstration, we will present a web-based browser interface of PowerQ, which communicates with the Java based server. The system is available at <http://powerq.comp.nus.edu.sg>. We intend to show the use of PowerQ against a number of real application scenarios such as the ACM Digital Library ([dl.acm.org](http://dl.acm.org)), and the IMDB database ([www.imdb.com](http://www.imdb.com)).

The demonstration will include three parts. First, we will run a number of sample aggregate queries against these resources. We will demonstrate how PowerQ exploits the ORA semantics in the database, distinguishes objects with the

The screenshot shows a web interface for PowerQ. At the top, it displays the keyword query: "Your keyword query is COUNT Code George Green". Below this, it provides the interpretation: "The interpretation is Find the COUNT Code of the Courses that are taught by the Lecturer with Lname matching George and are enrolled by the Student with Sname matching Green GROUPBY Sid". The results are shown in a table with columns: #, COUNT(Code), Sid, and a Verify link. The table contains two rows: (1, 1, s2, Verify) and (2, 2, s3, Verify). To the right of the table, there is a "Verify:" section with a toggle switch. Below this, it lists "Lecturer" information: Lid: 11, Lname: George, and "Student" information: Sid: s2, Sname: Green. An "OK" button is located at the bottom right of the interface.

#	COUNT(Code)	Sid	
1	1	s2	<a href="#">Verify</a>
2	2	s3	<a href="#">Verify</a>

Figure 6: Screenshot of answers to query pattern  $P_2$

same attribute value, and detects duplications of objects in relationships to answer aggregate queries correctly. The user can run queries without aggregate functions or GROUPBY to verify the answers of the aggregate queries. Next, we will run the aggregate queries on the denormalized data. We will demonstrate how PowerQ continues to process the aggregate queries correctly. Finally, the user will be free to run their own queries.

Through this demonstration, we will highlight the importance of the ORA semantics to relational keyword search. This is reflected in three aspects. First, the interpretation of keyword queries requires the system to be knowledgeable about the ORA semantics. Second, in order to answer queries involving aggregates and GROUPBY correctly, we need to distinguish objects with the same attribute value and detect duplications of objects in relationships based on the ORA semantics. Third, we need to keep track of the ORA semantics in the database, so that queries on denormalized databases can continue to be handled correctly.

## 5. REFERENCES

- [1] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, P. Parag, and S. Sudarshan. BANKS: Browsing and keyword searching in relational databases. In *VLDB*, 2002.
- [2] M. Kargar, A. An, N. Cercone, P. Godfrey, J. Szlichta, and X. Yu. MeanKS: Meaningful keyword search in relational databases with complex schema. In *SIGMOD*, 2014.
- [3] F. Li and H. V. Jagadish. Usability, databases, and HCI. *IEEE Data Eng. Bull.*, 35(3):37–45, 2012.
- [4] Y. Luo, W. Wang, and X. Lin. SPARK: A keyword search engine on relational databases. In *ICDE*, 2008.
- [5] S. Tata and G. M. Lohman. SQAK: Doing more with keywords. In *SIGMOD*, 2008.
- [6] Z. Zeng, Z. Bao, T. N. Le, M. L. Lee, and T. W. Ling. ExpressQ: Identifying keyword context and search target in relational keyword queries. In *CIKM*, 2014.
- [7] Z. Zeng, Z. Bao, M. L. Lee, and T. W. Ling. A semantic approach to keyword search over relational databases. In *ER*, 2013.
- [8] Z. Zeng, M. L. Lee, and T. W. Ling. Answering keyword queries involving aggregates and groupby on relational databases. In *EDBT*, 2016.

# Visualization through Inductive Aggregation

Parke Godfrey\*    Jarek Gryz\*    Piotr Lasek\*†    Nasim Razavi\*

York University, Canada\*  
Rzeszów University, Poland†

{godfrey, jarek, plasek, nasim}@cse.yorku.ca

## 1. INTRODUCTION

Visualization provides a powerful means for data analysis. To be useful, visual analytics tools must support smooth and flexible use of visualizations at a fast rate. This becomes increasingly onerous with the ever-increasing size of real-world datasets. First, large databases make interaction more difficult as a query across the entire data can be very slow. Second, any attempt to show all data points will overload the visualization, resulting in chaos that will only confuse the user.

Many solutions have been proposed to solve these problems,<sup>1</sup> but only one [1] addresses both of them simultaneously: *hierarchical aggregation*. Since it is not feasible to show all answers to a query, a natural way to reduce the size of the answer set is to aggregate it. We also need to support real-time interactivity; that is, to support an efficient way to move between levels of aggregation. Thus, we need a hierarchy of aggregations.

Hierarchical aggregation is not a new idea, of course. For data, it has been explored in OLAP, starting with the *data-cube* model. For images, it is only recent that new visual aggregation strategies have been developed for standard visualization techniques [1]. These strategies turn existing visualizations into multi-resolution versions that can be rendered at any desired level of detail. The *visual aggregate* can convey various information about the underlying data, such as their average, minima and maxima, and distribution.

Thus, data visualization systems face two challenges. First is an issue of *efficiency*. Most visualizations today are produced by first retrieving data from a database, and then using a specialized tool to render it. This decoupled approach results in significant duplication of functionality, while missing opportunities for cross-layer optimizations [5]. Second is an issue of *expressiveness*. Data visualization systems have not exploited modern graphics processing and rendering, due to architectural limitations, and lack of awareness. These graphics *shaders* meanwhile can significantly

improve visualization.

We combine data aggregation with visual aggregation in a tightly coupled system that provides for smooth user interaction. Our implementation is based on a hierarchical data structure we call an *aggregate pyramid*.<sup>2</sup> By interacting with the pyramid in the back-end (via the database system), the front-end visualization client can quickly filter the data to move up and down in the aggregation hierarchy. We want the visualization mantra, “overview first, zoom and filter, then details on demand,” [4] to be more like *skydiving* than gliding.

SKYDIVE’s general architecture enables us to exploit modern graphics processing and rendering in new ways that other systems have not been able to exploit. Thus, for interactive data visualization, SKYDIVE is innovative in both *expressiveness*, by flexibly enabling new rendering techniques, and *efficiency*, due to tight-coupling in its architecture.

## 2. EXPRESSIVENESS

In SKYDIVE, a *data visualization* is defined by the user in two parts:

1. the *aggregate-pyramid query*, which defines the *dataset* cut from the database the user wishes to explore; and
2. the *visual mapping*, which maps the aggregate measures of the aggregate pyramid to visual *channels* in the *data texture* the user will explore.

In the next section, we define the concept of the aggregate pyramid in more detail, and how it is used to support *efficient* data visualization and exploration. For now in overview, we consider how it supports *expressive* visualizations.

**Anatomy of the aggregate pyramid.** The aggregate pyramid represents a hierarchy of aggregation levels we call *strata*. (This is visualized in Fig. 1.) The *base* of the pyramid represents the stratum with the highest resolution of our data (the data aggregated the least); higher strata represent successively lower resolutions (the data further aggregated). As with a *data cube*, the columns of an aggregate pyramid consist of *dimensions* and *aggregates*. Each tuple in the pyramid, called a *cell*, represents the aggregates of the raw data within the cell’s area. The cells of any given stratum *tile* the dataset at the stratum’s resolution. We consider here two-dimensional pyramids, with “X” and “Y” dimension columns.<sup>3</sup>

<sup>2</sup>This concept is described in more detail in [3].

<sup>3</sup>One-dimensional pyramids are also useful for visualization, but with specific presentation models that we do not discuss here. Pyramids generalize to more than two dimensions,

<sup>1</sup>See [2] for an overview.

The aggregate columns are defined over the *measures* of the raw data. For the pyramid, an *inductive-aggregate function* is defined in two parts, a *base* and an *inductive function*. The base function aggregates over the raw data to produce the cells of the *base* stratum of the pyramid. The inductive function then aggregates over the appropriate cells of the stratum below the current to compute the aggregate values for the cells of each stratum.

For example, consider scatter-plot data of event points—say fires that have occurred in the Seattle area—that we want to visualize. One aggregate we likely will want is to count events within each cell’s area. The base function can simply be the aggregate count over the group-by into cells for the pyramid’s base stratum. The inductive function is then *sum*, to sum up the counts of the sub-cells to compute the count for the cell.

It is important that the inductive functions are only allowed to look one stratum below, for efficiency of computation. This also means a good deal of care and thought must go into defining appropriate, meaningful inductive-aggregate functions that are effective for visualization.

Even with the simple example of scatter-plot data of events that have no specific qualities—an event simply occurred at a location—there are still a number of aggregates of this information in which one might be interested. *Count-per-area* is an obvious one, as discussed above. Additionally, there are statistical aggregates that can tell us something about how the events are *distributed* in the area represented by the cell. Are they uniformly distributed across the area, or are they highly clustered in given spots? An *entropy* function can be devised that offers a measure of such *dispersion* across the area.

If the scatter-plot data is richer, then there are many more aggregates we might wish to convey. Fire events might additionally carry a measure of *intensity*. We then may wish to convey information about the intensity of events in addition to the count of events. *Maximum* may be a reasonable aggregate over intensity, to convey the highest intensity of event to have occurred in an area (the cell). Another aggregate could carry the *standard deviation* over intensity of events within the cell. Fire events will also have *time* associated with them, and so forth.

**The visualization mapping.** The second thing that must be specified for a visualization is a *mapping* of the aggregates of the pyramid into visual *channels*. If the presentation model is a 2D image, these channels are the usual suspects from image processing: e.g., *red*, *green*, and *blue* (RGB) or *hue*, *saturation*, and *lightness* (HSL), depending on how one wants to consider the color space.

The mapping consists of functions that *map* (and *normalize*) the aggregates of the pyramid to available channels in the presentation model. The mapping should be done with care to be “orthogonal”, so that each aggregate as mapped can be clearly distinguished. We envision developing a library of standard mappings to be available. SKYDIVE’s architecture, however, is a general platform that allows for devising new, novel mappings.

**The problem of channel paucity.** A critical problem is the absolute paucity of channels for visual conveyance. If we are mapping to an image, we effectively have but three channels we can use in the mapping (e.g., HSL).

albeit presentation models for these are limited.

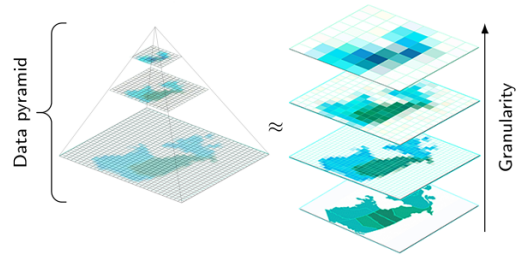


Figure 1: The Aggregate Pyramid Model.

While much work in data visualization has strived to address the problem of channel paucity—for instance, by graphical symbols, layout, and such to add effectively “channels” for conveyance—these do not work for interactive visualization in an inductive way, where one can zoom to change dynamically the degree of aggregation. Meanwhile, no work yet has taken advantage of the additional channels that the modern graphics environment afford us. SKYDIVE is designed to exploit just that, to great advantage.

**Presentation models.** The presentation model defines the “structure” that will be visualized. The model provides a set of channels that can be used by the mapping.

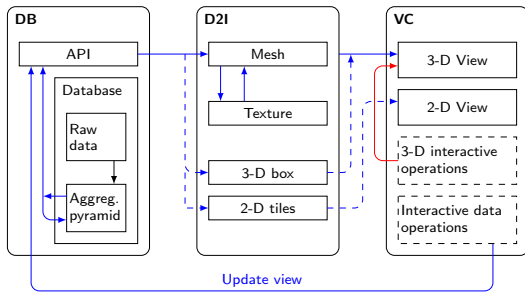
*The 2D model.* One model is that of a 2D image. The viewer manages the visualization as an image—which we call a *data texture*—within a canvas, allowing the user to zoom and pan around it. This view is, in essence, a *heat-map*. SKYDIVE’s benefit is that the data texture makes it possible to explore dynamically this heat-map view progressively in realtime. This model suffers still from channel paucity, however; it is effective only if one can live within such a constrained channel space.

*The 2<sup>1/2</sup>D model.* A second model we call 2<sup>1/2</sup>D. For this, the model is rendered in 3D. The visualization now consists of two parts: the data texture, as before; and a *terrain*—a *manifold*<sup>4</sup> rendered as a *mesh*—onto which the texture is overlaid (*UV-mapped*).<sup>5</sup> This exploits modern 3D graphics rendering, which supports meshes and UV-mapping. This offers SKYDIVE additional channels of conveyance over points in the terrain: *elevation* (Z); *specular*; and *normal*. A specular map determines how “reflective” a point is on the surface. As scenes in 3D have external lighting, this is quite noticeable. A normal map dictates deviations of the normals, the “perpendicular” of a point with respect to the surface. By perturbing the normals of a neighborhood, that part of the surface can be made to look rough; leaving them as dictated by the mesh, the surface looks smooth. These are standard in graphics processing and used in game production for making scenes look more realistic. That is, these channels visually stand out.

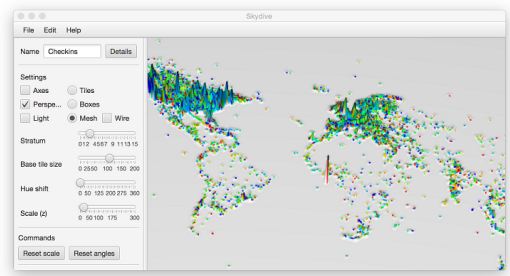
We can use the *alpha* channel additionally, as the terrain can be floated over a flat reference plane; the bleed-through of the reference through *translucency* of the terrain is readily

<sup>4</sup>A *manifold* is function that maps 2D coordinates to values. This can be rendered in 3D using *elevation*, “Z”, over the 2D plane to indicate the 2D points’ values.

<sup>5</sup>2D images are called *textures* in this context, and the mapping of textures onto the mesh surfaces is called the *UV-mapping*.



(a) SKYDIVE's architecture.



(b) SKYDIVE's main window.

Figure 2: The SKYDIVE System.

obvious. This means we have effectively *seven* channels of conveyance in the  $2^{1/2}D$  model, versus just the three in the 2D model.

*Mixed models.* SKYDIVE can mix presentation models for the same pyramid to provide simultaneous, synchronized alternative viewports into the same data. We also intend to support “cross-product” pyramids that could, for example, let one zoom and pan on  $XY$  and on  $T$  (*time*) independently.

### 3. EFFICIENCY

**Structure of the aggregate pyramid.** The idea behind this mirrors approaches taken by *progressive image formats* such as JPEG-2000. The image “pyramid” is multi-resolution data structure that represents a  $2^d \times 2^d$  image as a sequence of copies (2D arrays) of the original image, each “half” the resolution—half on the rows and half on the columns—of the next. Thus the base stratum of the pyramid is the full resolution version of the image, while the top stratum is a single pixel approximation of it.

Similarly, the *aggregate pyramid* represents  $2^d \times 2^d$  data cells at its base, with each subsequent stratum halving the “resolution” (doubling the aggregation). Construction of an aggregate pyramid can be accomplished efficiently by the database engine by building it from the base upwards. First, the base stratum is created by aggregating the raw data into the base cells. Then subsequent strata can be produced recursively by aggregating spatially the constituent quadrant cells of the stratum below. The cells can be indexed by stratum, and by Hilbert order that linearizes their order, which then can be indexed via a B+-tree. As such ordering preserves locality of sub-cells that are needed to merge for the next higher stratum, a stratum can be produced in proper order by a single scan of the cells of the stratum below it.

SKYDIVE employs the aggregate pyramid to preprocess data so the visualization process can be handled efficiently. Given a query defining the dataset to explore, the database system *materializes* the aggregate-pyramid version of the dataset query, and indexes the pyramid by stratum and Hilbert order, as discussed above.

The materialized pyramid is managed by the database engine during the visual exploration of that dataset. Interactive operations at the visualization client are then supported by querying into the aggregate pyramid at the appropriate stratum and range (bounding box), which can be handled efficiently and at real-time, interactive speeds. Thus, SKYDIVE tightly couples database support for processing the

data with the interactive visualization.

**Operations.** SKYDIVE is designed to support the following visual operations over the dataset for visualizing. Each, in turn, can be supported efficiently by the database system over the materialized aggregate pyramid.

*Resizing.* The user can change the current viewport by changing the size of the visualized data (up or down). In sizing, the visualizer may need to present a different level of resolution. For instance, in a size-up operation, the user requests a higher resolution image. As a result, the system needs to retrieve the aggregated data from a higher resolution stratum in the pyramid.

*Zooming.* The user can request to view more detail of a part of the image by specifying a *window of interest*, selecting a portion of the image by zooming in. The system maps the requested window to the stratum with high enough resolution to fit the canvas, and selects the appropriate range.

*Panning.* In panning, the user changes the viewport in the image, but within the same level of resolution. If the user pans, the system will check the availability of the visual data in the current stratum, and request the additional range from the pyramid in that stratum.

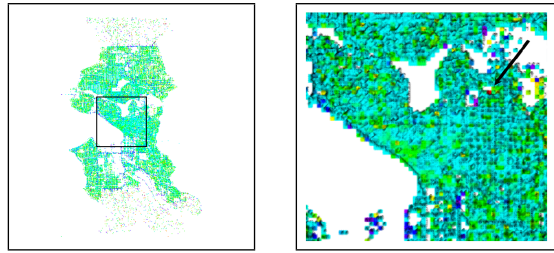
### 4. ARCHITECTURE

**Skydive's components.** SKYDIVE is composed of three main components, as shown in Figure 2:

- the Database Module (DB);
- Data-to-Image module (D2I); and
- the Visualization Client (VC).

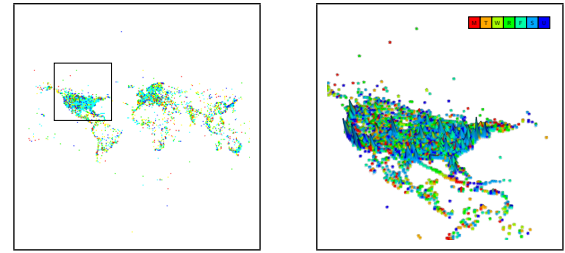
Each is designed to use a different type of *computer memory*. The DB module uses *disk* to store and manage the raw data, and materialized aggregate pyramids. The D2I module works with a small subset of the aggregated dataset, and stores data in *main memory* (RAM). The VC module uses the *graphic card's* capabilities to perform more advanced operations—such as zooming, scaling, panning, and rotation—over the graphical representation of the data.

This separation of concerns provides useful flexibility. Each component can be implemented as a separate service, deployed on a different machine. This leverages the idea of *compression* of data conveyed between the modules, letting us implement a tightly coupled visualization system. The SKYDIVE prototype is implemented as a desktop application with the three modules as described above and shown in Fig. 2a. The main window of SKYDIVE, shown in Fig. 2b, is



(a) Overview image. (b) Zoomed in, terrain view.

Figure 3: Visualizations of the Seattle 911 Dataset.



(a) Data texture of check-ins. (b) Zoomed in, terrain view.

Figure 4: Visualizations of the Brightkite dataset.

composed of a few simple elements: an upper menu for performing basic file operation; a left panel for tuning a loaded and currently displayed visualization; and a visualization view for rotating, panning and zooming.

**Graphical variables.** The user also defines the presentation model to be used and the visualization mapping (of aggregates to channels) to be employed, as discussed in §2. The system prototype supports three presentation models:

- a 2D heat-map;
- a  $2^{1/2}$ D heat-map by 3D barchart; and
- a  $2^{1/2}$ D terrain (by mesh and UV-mapping).

**Generating meshes and textures.** The data texture is generated by the D2I module by selecting the appropriate window out of the aggregate pyramid, and applying the visualization mapping. For the  $2^{1/2}$ D terrain model, a mesh is additionally computed by the D2I. The mesh is created based on a stratum of lower resolution, for better visual appeal, and for efficiency in the VC.

**User interface.** The user interface, as shown in Fig. 2b, allows the interactive visualization operations, as discussed above. The user can scale, translate, and rotate the currently displayed visualization in the Visualization Client (VC). The VC is implemented using JavaFX, which natively supports these functions. Scaling, translation, and rotation do not require to query the aggregate pyramid, hence are performed entirely within the VC, supported by the GPU.

Other *interactive* functions do require queries to be issued from VC to the DB module. For instance, if the user wants to focus more on a certain area of a visualization, then the system must request the data from the appropriate stratum; of higher resolution for zooming in, and lower for zooming out. Issuing such a request results in the loading and generating of the mesh and texture by the D2I. The VC then displays this using the GPU’s graphics pipeline.

## 5. DEMONSTRATION SCENARIO

**Datasets.** We test SKYDIVE using several datasets, two of which are described below.

*9-1-1 calls.* The Seattle Police Department 911 Incident Response dataset<sup>6</sup> contains over one million records. Each represents the police response to a 911 call within the city. Fig. 3a shows a *density* map of the calls. Color denotes the number of calls made within the area represented by a pixel. Based on the plotted heat-map, a user is not able to conclude anything more than that there are some areas of slightly higher density than their neighbors.

<sup>6</sup><https://data.seattle.gov/>

In Fig. 3b, we have switched to the terrain view, where elevation indicates the density, and color refers to a most frequent type of a call within the cell. In this view, we see more detail. For example, the red pixel indicated by the arrow represents unusual activity within the magnified area. *Brightkite check-ins.* The Brightkite Check-ins Dataset<sup>7</sup> consists of over four millions records of geographical positions reported by users of a geo-location social service. In Fig. 4a, the heat-map represents the dataset over one measure: color represents days of week for which user activity was highest within the areas represented by the pixels. In Fig. 4b, a terrain map is shown of a zoomed in portion with more in the mapping. The texture color again denotes day of week with highest activity. Elevation denotes number of check-ins. We can deduce that weekends were most active days for Brightkite users in the USA. We can additionally see the areas in which the most users were active.

**Richer mappings.** We will demonstrate the richer mappings offered by the  $2^{1/2}$ D model with normal, specular, and alpha channels. These are not easy to show in static pictures, but stand out in display in the demo. With these, additional aggregates can be conveyed to a viewer simultaneously. Roughness of the surface (a normal map) can be used to represent variance of a measure within cells. Shininess (a specular map) can be used to show spatial dispersion within the area represented by a point on the terrain.

## 6. REFERENCES

- [1] N. Elmqvist and J. Fekete. Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines. *IEEE Trans. Vis. Comput. Graph.*, 16(3):439–454, 2010.
- [2] P. Godfrey, J. Gryz, and P. Lasek. Interactive visualization of large data sets. Technical Report EECS-2015-03, York University, March 2015.
- [3] P. Godfrey, J. Gryz, P. Lasek, and N. Razvi. Skydive: An interactive data visualization engine. In *IEEE Symposium on Large Data Analytics and Visualization, Chicago, USA, October 25-26.*, 2015.
- [4] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343. IEEE, 1996.
- [5] E. Wu, L. Battle, and S. R. Madden. The case for data visualization management systems [vision paper]. *Proceedings of the VLDB Endowment*, 2014.

<sup>7</sup><https://snap.stanford.edu/data/>

# Keyword Search on microblog Data Streams: Finding Contextual Messages in Real Time

Manoj K Agarwal  
Search Technology Center  
Microsoft India  
agarwalm@microsoft.com

Divyam Bansal  
Bangalore, India  
Google Inc.  
divyamb@google.com

Mridul Garg, Krithi Ramamritham<sup>1</sup>  
Dept. of Computer Sc. and Eng.  
IIT-Bombay, India  
gmridul09@gmail.com, krithi@cse.iitb.ac.in

## ABSTRACT

Microblogging streams contain information pertaining to emerging real world events. Due to the rapid pace at which these data streams are generated, it is often difficult for users to discover the most relevant messages in the context of their keyword queries. Search over such data streams returns the most recent messages only; most recent messages may not be the most relevant messages. Hence users have to resort to the cumbersome task of sifting through a large amount of information to obtain the context of a live event.

We present a novel real time search system – *Contextual Event Search* – on dynamic message streams, to extract meaningful summaries for live events in real time. Our technique is unsupervised and automatically identifies different facets of the live events in a scalable and effective manner.

We demonstrate that for a given keyword search, users are presented with meaningful, compact and complete contextual event summaries for the most relevant events in a given time window, thus exposing the full context behind the messages.

## Categories and Subject Descriptors

H.3.3 [Information Search and Retrieval]: *Information filtering, Relevance feedback, Search Process.*

## General Terms

Algorithms, Experimentation.

## Keywords

Dynamic Graph, Indexing, Search, Summarization.

## 1. INTRODUCTION

Highly dynamic unstructured data streams – sequences of chronologically ordered messages posted by multiple users at a fast pace – occur in various social media and enterprise domains. In microblog streams (e.g., Twitter), messages are posted at a high rate due to their large user base. Twitter is often the first medium to report emerging events [1][2], ranging from globally important events to the events relevant only for a small community.

An *event* is a real world or an abstract activity, relevant for a group of people or a community. An *event* in a data stream is defined by “*messages, posted by multiple users, in the same context, within a bounded time window*”, for example, messages posted by the fans during the course of a football match. It is only natural that in a fast moving world, a large number of events occur concurrently.

Existing unsupervised approaches identify emerging events as clusters of keyword over dynamic message streams [1][2][3]. Each keyword cluster forms an ‘event-topic’. The technique described in [1], when used to discover events from the tweets posted during the Nairobi terrorist attack [7], discovered many event-topics including one containing the keywords:

- **A: UK, #kenya, #westgate, #nairobi**

Clearly, the context behind the keyword cluster is not available to the users – the keywords are insufficient to describe the underlying event. The same is true of another event-topic:

- **B: was, 69, kofi, among, #ghana, attacks, ghanaian, awoonor, killed, poet, prof., #kenya**

To better understand the event-topic, i.e., what the event is about, users are needed to search for the most relevant messages in the data stream by themselves. Besides burdening the users with the task of understanding the emerging events manually, for example, to determine if there is connection between **A** and **B**, this approach suffers from many shortcomings:

- (1) Message search is primitive, e.g., Twitter just returns the most recent tweets for a given search query [5]. It is not necessary that the recent tweets alone are the most relevant tweets for the event.
- (2) Simple keyword search results can produce an information overload for a fast moving data stream. Often a large number of tweets are returned by Twitter in response to a search query [6].
- (3) In such fast moving data streams, typically the rate at which messages are generated is high but messages are short. Therefore, it is often difficult for the users to understand the context of a standalone message even if the message is informative.
- (4) Events evolving in real time comprise different facets. Search results are continuously updated with recent messages and it becomes difficult for the user to keep pace with evolving events.

### 1.1 Contextual Search on Live Data Streams

We demonstrate our system for *Contextual Event Search*, to extract the complete contextual summaries for the events unraveling in a live message stream in real-time. The summaries are stored in an event thread as shown in Figure 1. Live real-world events are not just point events – they evolve continuously. The event summary must be updated every time there are significant changes in the event. When these changes are arranged temporally, an event thread results. The event thread captures the passage of time naturally. Challenges involved in discovering such event threads are many:

<sup>1</sup> The author list is in alphabetical order.



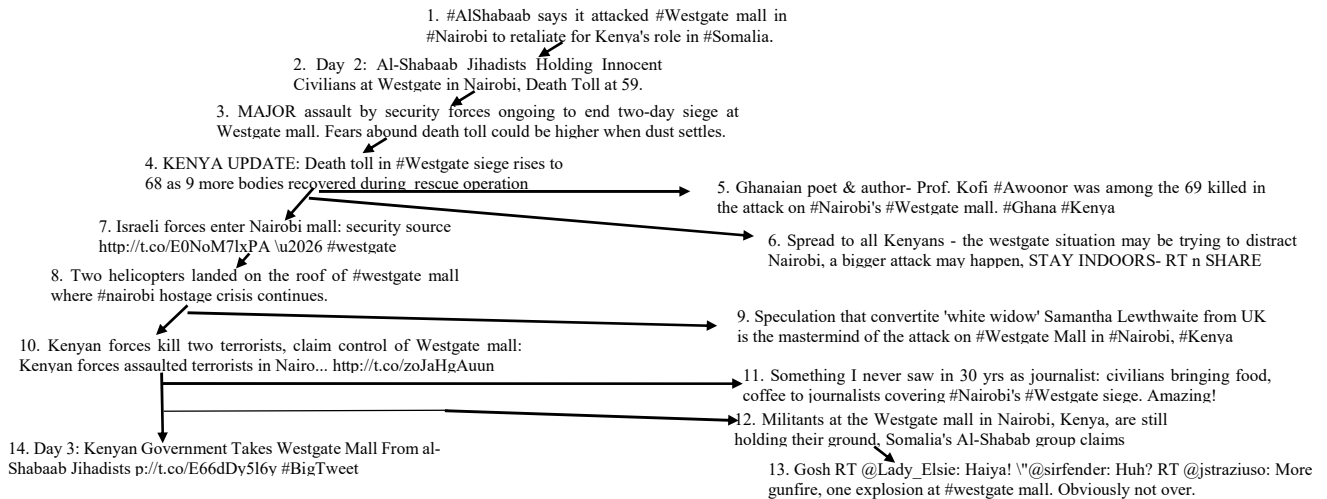


Figure 1. Contextual Event Summary Thread Discovered by our system for Nairobi Attack

- The first challenge is to identify and associate the relevant messages to the corresponding ‘event topic’.
- Secondly, most real world events do not evolve linearly and comprise several facets. Therefore, the summary for an event is represented as a Directed Acyclic Graph (DAG). It is non-trivial to discover such ‘contextual event threads’. Each unique path in the event thread is a different facet of the event.

The *contextual event summary* displayed in Figure 1 was constructed by our approach automatically for the event ‘Nairobi terrorist attack’. The event summary starts with a message that a mall has been attacked, followed by how the action against attackers was progressing, rumors, claims and counter claims by authorities and citizens, etc. were also discovered in real time. The important sub-events were discovered from approximately 164K tweets and arranged in a chronological sequence. For each of the sub-events in the event thread, our technique identified an appropriate summary as shown in Figure 1. Sub-event 9 corresponds to **A** and sub-event 5 corresponds to **B**.

We demonstrate our system that automatically constructs such summaries as shown in Figure 1, for a live data stream. Our system summarizes the event in a fast moving data stream in real time in an *unsupervised* manner. The event thread represent a compact, complete and meaningful event summary. A *minimal* set of related messages are identified that represent the *complete* event summary. The event summary discovered by our system is *stable*, i.e., it is updated only with additional information, which is appended to the summary discovered thus far. Our system also exposes the different *facets* of live events which are presented as a contextual event summary thread. The details of discovering the event threads are beyond the scope of this paper.

To the best of our knowledge, ours is the first system that discovers contextual event threads automatically for a fast moving data unfiltered data stream in real time in an unsupervised manner. Lin et al., explore the problem of generating storylines from microblog data [4]. Their system is only applicable to retrospective data analysis where on relevant tweets a storyline is generated via graph optimization. In [6], Shou et al. present a technique to summarize a twitter data stream, filtered in the context of a given user query. In [9], authors present a method to summarize a pre-specified event topic. Their methodology is applicable for structured and recurring events such as sports events and need the prior knowledge of similar events.

## 1.2 System Components

With the aid the event summaries, we enable the contextual search over data streams. Following are the components of our system:

**Discovery:** The event threads are discovered in a live data stream. They contain the most relevant messages in a chronologically ordered event threads representing its story line. Event thread is associated with a rank based on event popularity and its dynamicity. If an event is highly dynamic with fast updates, its rank increases.

**Indexing:** An index is maintained over the events threads. Since the index is updated in real time, we adopt a *lazy-update* strategy, i.e., index is updated only for the most popular events. For events, which are less popular, only a subset of these events are updated in the index, unless the underlying changes in the event result in significant increase in event rank.

**Search:** For a keyword query, a ranked list of most relevant event threads is returned. Hence, even the messages which may not contain the query keywords but are part of the event threads are returned. Thus, our system is able to find the most relevant contextual messages for a given keyword query.

The architecture of our system is shown in Figure 2. It contains three components; *Event Discovery and Summarization Engine* to discover event thread over a live data stream. The discovered events are pushed to the *Indexing Engine*, which maintains an index over them as well as keeps the index updated for live events. Finally, the *Search Engine* finds and returns the most relevant *topK* events, upon receiving a keyword query. *topK* is a tunable parameter. *Event Discovery and Summarization Engine* is based on the model in [1] but its details are beyond the scope of this paper. In next section, we present the details of other two components.

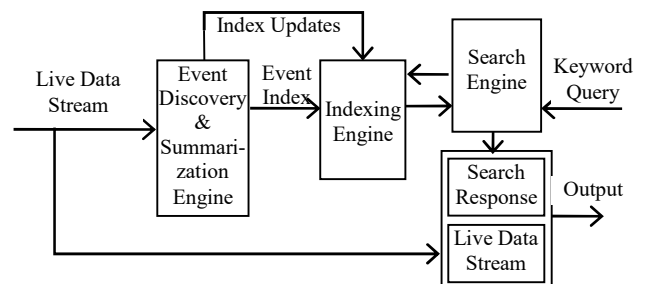


Figure 2: The system Architecture

## 2. REAL TIME CONTEXTUAL SEARCH

Each event is identified by a unique event ID  $e$ . As an event evolves, its summary is appended with the most recent updates. Each node in the event tree represents an ‘event topic’ and has a set of relevant tweets associated with it [1]. The algorithm to find meaningful summary of an ‘event topic’ and discovery of different event facets is beyond the scope of this paper. Each event topic, i.e., keyword cluster  $c_{ei}$  has a summary  $s_{ei}$  and a ranking score  $r_{ei}$ . For an event  $e$ , there is a list of clusters  $c_{ei} \mid_{i=1}^n \ n \geq 1$  associated with it.

Ranking score  $r_e$  for an event  $e$  is computed as  $r_e = \sum_{i=1}^n c_{ei}$ . Event summary  $s_e$  is defined by arranging the cluster summaries ( $s_{e1}, s_{e2}, \dots, s_{en}$ ) in a (multi-faceted) event thread.

### 2.1 Indexing Engine

The search engine maintains an inverted index. The inverted index consists of words mapped to a posting list of event IDs. For each word  $w$ , its posting list  $L_w$  contains event IDs which have  $w$  in its summary. In the posting lists, events are sorted in the decreasing order of their ranking scores. One of the challenges is to maintain the posting lists sorted, since inserting an event in  $L_w$  would take  $O(|L_w|)$  time;  $|L_w|$  is the number of events in  $L_w$ . To reduce the cost of insert operation, we adopt the following approach: List  $L_w$  is organized as a list  $L_w'$  containing sequence of buckets  $B_1, B_2, \dots, B_m$  where  $m = |L_w'|$ . Each bucket contains a max-heap and a min-heap. Both heaps contain same event IDs. We define the size of the bucket as the size of its max-heap. Each bucket  $B_i$  has an event with maximum score  $s_i^{max}$  and an event with minimum score  $s_i^{min}$ . The sequence of buckets is such that the following property is satisfied:

$$s_i^{min} \geq s_{i+1}^{max} \mid \forall i \quad \square$$

The size of buckets  $B_i$ s increases by a factor of 2 ( $|B_{i+1}|/|B_i|=2 \ \forall i; i < m$ ). Therefore,  $m = |L_w'| = O(\log(|L_w|))$ . The time complexity of the insert operation is equal to the time complexity of a) searching the bucket  $B$  in which the event should be inserted ( $O(m)$ ); b) inserting the event in  $B$  ( $O(m)$ ), since the size of each bucket is  $O(|L_w|)$  and the bucket is maintained as a heap; c) adjusting the size of  $B$  (its size increases by 1 on inserting an event). Step (c) takes  $O(m^2)$  time since the event with minimum ranking score is removed from  $B$  and inserted in next bucket and this procedure may continue till the first bucket in the sequence. If the size of the last bucket is larger than the maximum allowed, then the event with minimum ranking score is removed and is inserted in a new bucket appended at the end (number of buckets in list  $L_w$  increases by 1).

- The insert operation in each bucket is performed by inserting the event in max-heap and min-heap ( $O(m)$  time complexity).
- The remove operation in a bucket is performed by removing the minimum element in its heaps. This is  $O(1)$  for min-heap but for max-heap it takes  $O(|L_w|)$  if done naively. We store the pointer to the location of the minimum ranking score event in max-heap hence removal takes  $O(m)$  time.

Since there are  $m$  buckets and insertion and removal in each bucket takes  $O(m)$  time, step (c) takes  $O(m^2)$  time. Therefore, the overall insert operation takes  $O(m^2)$  time.

For each list, we maintain a mapping of event ID to the bucket which contains it. This is useful when an event has to be removed from a posting list or its ranking score has to be updated. With this map searching an event in a list takes  $O(1)$  time. Buckets are implemented as locator heap in which a map is maintained which contains the location of event IDs inside the bucket (i.e., heap) thus making the deletion of an event from the bucket  $O(m)$ .

**Lazy Update:** With more tweets flowing-in, events are updated which results in the update of its ranking score. We need to reflect these changes in the index. However, updating the index is costly as the event clusters get updated at a fast pace. Thus, we trade-off the minor drop in output accuracy for greater efficiency of the search engine. Whenever the score of an event changes, we check if the new score is greater than the score of the event at the  $2 \times topK^{th}$  position in the relevant posting list. If not, we assume that the event will not affect the final output for any query and hence the change is ignored. Otherwise, we update its score.

### 2.2 Search Engine

In this section, we describe our query processing system or search engine. The search engine takes a keyword query and returns  $topK$  most relevant event summaries. Suppose a user enters a query  $Q$  of length  $l : q_1, q_2, \dots, q_l$ , where  $q_i \mid 1 \leq i \leq l$  is a query word. To define  $topK$  relevant events, we compute  $maxScore(Q, e)$  [8] of an event  $e$  for a query  $Q$ . We define a function  $p(w, e)$  for a word  $w$  and an event  $e$ .  $p(w, e) = 1$ , if  $e$  is present in the posting list of word  $w$ , otherwise  $p(w, e) = 0$ .

$$maxScore = r_e \times \sum_{i=1}^l p(q_i, e)$$

where  $r_e$  is the event rank.  $topK$  events with highest  $maxScore$  form the output. We next define some important terms and data structures before describing the algorithm to find  $topK$  events:

**partial\_maxScore:** Initial *partial\_maxScore* of all events is set to 0. Maximum *partial\_maxScore* of an event  $e$  is the product of  $r_e$  and the words in the query for which  $e$  is present in their respective posting list.

**topEvents:** We maintain a min-priority queue of *topEvents* which stores candidate events and is initially empty. An event  $e_1$  is considered ‘less than’ event  $e_2$  if *partial\_maxScore* of  $e_1$  is less than that of  $e_2$ .

**min\_topEvents:** It is the event with minimum *partial\_maxScore* among all the events in *topEvents*.

**fcEvents:** It stores final candidate events. Any event evicted from *topEvents* is stored in *fcEvents*.

**getMax(B)** returns the event with maximum ranking score in the bucket  $B$ .

We search for the posting list for each word in the query. Words for which no posting list is found are discarded. For all the posting lists in consideration, an iterator is set to the first bucket. The highest ranked event from each bucket is inserted in *topEvents*. We take the event *min\_topEvents* and compare it to the event next to it, called  $e_n$ , from the posting list it belongs to. We check if  $e_n$  is a candidate event.  $e_n$  is a candidate event if either the number of events in *topEvents* and *fcEvents* is less than  $topK$  or  $r_{e_n} \times l \geq min\_topEvents.partial\_maxScore$ ;  $r_{e_n} \times l$  is  $maxScore(Q, e_n)$  which is the maximum possible value of *partial\_maxScore* for event  $e_n$ . Hence, if this value is less than *min\_topEvents.partial\_maxScore* and at least  $topK$  events are already present in *topEvents* and *fcEvents*, then  $e_n$  cannot occur in the final output. If it is a candidate event, it is inserted in the *topEvents*; otherwise the *min\_topEvents* may occur in the final output and hence moved from *topEvents* to *fcEvents*. If the event  $e_n$  is already present in *topEvents* or *fcEvents*, then we just update its *partial\_maxScore*. Once no more events can be inserted in *topEvents*, we move the remaining events from *topEvents* to *fcEvents*.  $topK$  events with highest *partial\_maxScore* in *fcEvents* form the Output. For faster retrieval, no posting list is traversed beyond  $2 \times topK^{th}$  event (which lies in  $\log(2 \times topK)^{th}$  bucket).

```

1 Create an array currB of the iterators;
  /* currB[i] stores the iterator of ith query word
  for which posting list exists */
  /* Iterators are pointing to the first bucket of
  corresponding posting list */
  /* Each event e in topEvents also stores the
  position of the bucket of its posting list in
  currB which is expressed as e.posInList. */
2 numPostingList := Size of currB;
3 while numPostingList ≠ 0 do
4   e := min_topEvents;
5   if The bucket B at currB[e.posInList] is empty
   then
6     if B is the last bucket in the posting list then
7       Remove e from topEvents and insert it in
       fcEvents;
8       numPostingList --;
9       continue;
10    end
11   else
12     Set currB[e.posInList] to the bucket next to
     B in the posting list;
13   end
14 end
15 en := getMax(currB[e.posInList])
16 Remove en from currB[e.posInList];
17 if en is present in topEvents or in fcEvents then
18   en.partial_maxScore :=
19   en.partial_maxScore + ren;
20 end
21 else if en is a candidate event then
22   en.posInList := e.posInList;
23   Insert en in topEvents;
24 end
25 else
26   Remove e from topEvents and insert it in
   fcEvents; numPostinList --;
27 end
28 end
29 Output summaries of topK events with highest
   partial_maxScore.

```

Our experiments have shown that typically in a posting list, initial top ranking events are followed by a long tail of low ranking events. Hence, it is unlikely that if any event beyond  $2 \times topK^{th}$  position in the any of the posting list under consideration, are in final *topK* list. This the reason behind this heuristic.

**Correctness of the algorithm:** The algorithm traverses all the posting lists of the query words present in the index. However, instead of traversing them completely, whenever an event *e* does not qualify to be a candidate event, *min\_topEvents* is removed from *topEvents*. This ensures that the corresponding posting list is not considered again as all the following events have lower ranking score. Hence, they can never become candidate events.

**Time Complexity:** In the worst case, each event can be a candidate event. So all the top  $2 \times topK$  events in the posting lists of query words are inserted in *topEvents*. The time complexity for all the insertions in *topEvents* for a query *Q* of length *l* is  $O(\log(l \times topK))$  as  $l \times 2 \times topK$  is the maximum number of events in *topEvents*. The time complexity of traversing the posting lists is  $O(l \times topK \times \log(topK))$  since each event is removed from a bucket on traversal. Hence, the time complexity of the algorithm is  $O(l \times topK + l \times topK \times \log(topK)) = O(l \times topK \times \log(topK))$ .

### 3. DEMONSTRATION

We demonstrate the ability of our system to find the most relevant messages in the context of a user keyword query. Specifically, for a given keyword query, we demonstrate;

- The ability of our system to find the most relevant messages in real time over live data streams.
- The discovery of contextual tweets in a live data stream, for a given user query, i.e., those tweets that do not even have the query keywords but are relevant.
- The ability of our system to create a story line for events unraveling in a live data stream in an unsupervised manner.

Our system returns a ranked list of the most relevant events for the user query. We will demonstrate the statistical summary of the live event including the number of tweets posted for that event and its ranking score. We will also demonstrate that the event summary discovered by our system is complete. At the demo, a user can see a fraction of randomly selected tweets from a live data stream and be able to compare our summary with the raw data. We will demonstrate our system on recorded as well as live Twitter stream.

The screenshot in Figure 3 shows the output of our search system for a given user query. For each event present in the result set, for the given keyword query, users can see a chronologically ordered DAG of most relevant tweets, representing contextual event threads, by clicking on the link.

#### Contextual Event Threads

Query:

- [Nairobi Westgate](#)
  - Nearer a full day after Kenyan mall attack began, gunmen and hostages still inside #westgate...
  - Fierce gunfire heard inside Nairobi Westgate shopping mall, 43 dead and 200 injured ... #Tweets = 459
- [LIVE UPDATES: THE WESTGATE MALL TERROR ATTACK](#)
  - Nairobi terror: Major malls shut down after attack in Westgate mall ...
  - UN Security Council condemns Kenya terror attack at Westgate shopping mall where over 46 people have been killed... #Tweets = 600
- [A senior figure of African letters, Ghana's Prof. Kofi Awoonor, born 1935, killed today in the attack on Westgate...](#)
  - Ghanaian poet Prof. Kofi Awoonor was among the 39 killed in the attack on Nairobi Westgate mall. #Gh ...
  - Prof. Kofi Awoonor of Ghana was killed today in Westgate Mall in Nairobi. His poem 'Song of Sorrow' ... #Tweets = 225
- [UPDATE #Kenya: Standoff betw. sec. forces and terrorists continue in Nairobi Westgate mall, as #alShabab...](#)
  - BBC News - Somalia's al-Shabab claims Nairobi Westgate Kenya attack http://t.co/7m09KOTza1 ... #Tweets = 240
- [If you can read this go out and donate blood. Blood drive at Kencom from 9am Nairobi to aid WestGate casualties...](#)
  - KenyaRedCross: Join #WestGate Blood Donation drive tomorrow at Kencom from 9am - 3pm Nairobi Please ... #Tweets = 130
- [Spread to all Kenyans - the westgate situation may be trying to distract Nairobi, a bigger attack may happen. STAY...](#)
  - #Tweets = 270

Figure 3: Real Time Search Engine over Live Data Streams

### 4. REFERENCES

- [1] M. K Agarwal, K. Ramamritham, M. Bhide "Real Time Discovery of Dense Clusters in Highly Dynamic Graphs: Identifying Real World Events in Highly Dynamic Environments", in VLDB 2012.
- [2] M. Mathioudakis, N. Koudas, "TwitterMonitor: Trend Detection over the Twitter Stream", in SIGMOD 2010.
- [3] N. Bansal, F. Chiang, N. Koudas, F. Tompa, "Seeking Stable Clusters in the Blogosphere", in VLDB 2007.
- [4] Chen Lin et al., "Generating Event Storylines from Microblogs", in CIKM 2012.
- [5] Chun Chen, et al., "T1: An Efficient Indexing Mechanism for RealTime Search on Tweets", in SIGMOD 2011.
- [6] L. Shou, Z. Wang, K. Chen, G. Chen, "Sumblr: Continuous Summarization of Evolving Tweet Streams", in SIGIR 2013.
- [7] [https://en.wikipedia.org/wiki/Westgate\\_shopping\\_mall\\_attack](https://en.wikipedia.org/wiki/Westgate_shopping_mall_attack)
- [8] Matthias Petri, J. Shane Culpepper, Alistair Moffat, "Exploring the magic of WAND", in 18th ADCS, 2013.
- [9] D. Chakrabarti, K. Punera, "Event Summarization using Tweets", in. ICSWM 2011.

# Answering Controlled Natural Language Questions on RDF Knowledge Bases

Giuseppe M. Mazzeo  
University of California, Los Angeles  
mazzeo@cs.ucla.edu

Carlo Zaniolo  
University of California, Los Angeles  
zaniolo@cs.ucla.edu

## ABSTRACT

The fast growth in number, size and availability of RDF knowledge bases (KB) is creating a pressing need for research advances that will let people consult them without having to learn structured query languages, such as SPARQL, and the internal organization of the KBs. In this demo, we present our Question Answering (QA) system that accepts questions posed in a Controlled Natural Language. The questions entered by the users are annotated on the fly, and an ontology driven autocompletion system displays suggested patterns computed in real time from the partially completed sentence the person is typing. By following these patterns, users can enter only semantically correct questions which are unambiguously interpreted by the system. This approach assures high levels of usability and generality, which will be demonstrated by (i) the superior performance of our system on well-known QA benchmarks, (ii) letting attendees suggest their own test questions, and (iii) accessing an assortment of RDF KBs that, besides the encyclopedic DBpedia from Wikipedia, will include others on specialized domains, such as music and biology.

## 1. INTRODUCTION

The last few years have seen major efforts toward organizing as RDF knowledge bases (KBs) both general and specialized knowledge. In the first group, we find DBpedia [1] that encodes the encyclopedic knowledge extracted from Wikipedia, and in the second group we have the thousands of projects that cover more specialized domains [2]. While these KBs can be effectively queried through their SPARQL [3] endpoints, the great majority of web users are neither familiar with SPARQL nor with the internals of the KBs. Thus, the design of user-friendly interfaces that will grant access to the riches of RDFKBs to a broad spectrum of web users has emerged as a challenging research objective of great social interest.

The importance of this topic has inspired a significant body of previous work, which includes the approaches de-

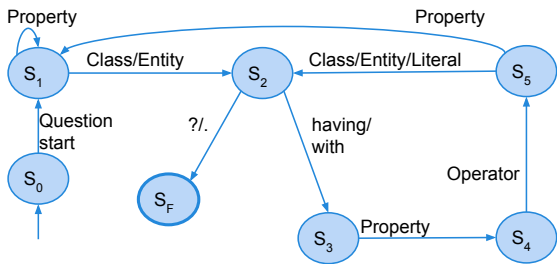
scribed in [6, 9, 10] and several others that rely on user-friendly graphical interfaces.

While some of these approaches [6] allow users to enter complex queries through a web browser, Natural Language (NL) interfaces remain the solution of choice when the used devices do not support well a full browser, or when voice recognition is used instead of typing. Translating NL questions into formal language queries represents an old, challenging, and widely studied problem [7, 8, 11], for which a general solution has not been found yet. This is, in fact, a very complex problem, combining several non-trivial sub-problems, such as parsing the syntactic structure of the question, mapping the phrases of the question to resources of the KB, and resolving ambiguities. The last problem is quite serious, because syntax often leaves much room for ambiguity, which cannot be resolved without much knowledge about the underlying application domain and understanding the context in which the question is asked.

In order to reduce the complexity of the problem, techniques replacing the ‘full’ natural language with a *controlled natural language* (CNL) have been proposed. A CNL system restricts the grammar that can be used to input questions, with the objective of making the language (i) ‘formal’ enough to be accurately interpreted by machines, but still (ii) ‘natural’ enough to be readily acquired by people as an idiomatic version of their NL. These systems are based on the idea that it is worth giving up the great flexibility and eloquence of the natural language in order to make the questions unambiguous to the machine that can thus produce answers of better accuracy and completeness.

In this demonstration session we will present our system for querying RDF data, called *CANaLI* (acronym for Context-Aware controlled Natural Language Interface). *CANaLI* has been applied to various QA testbeds [4], producing results of superior precision and recall. We will let *CANaLI* answer these testbed questions along with new questions suggested by the conference attendees, as needed to prove the usability and generality of the system. The attendees will thus be able to observe how *CANaLI* guides the users in typing questions, by allowing users to type only questions that are semantically correct w.r.t. the underlying KB, and syntactically correct w.r.t. the grammar of its CNL. Moreover, as soon as the user hesitates with typing, the system suggests correct completions she can select from. This allows people to self-learn *CANaLI* easily and quickly.

This short paper is organized as follows. Section 2 provides an overview of *CANaLI*, describing its basic operation, by means of some examples (Sec. 2.1 and 2.2), the index



**Figure 1: The main states and transitions of the automaton used by CANaLI**

used to suggest valid tokens (Sec. 2.3), and the system architecture (Sec. 2.4). Experimental results are presented in Section 3. Finally, we describe the demonstration scenario in Section 4.

## 2. OVERVIEW OF CANALI

CANaLI is a system that enables users to enter questions in a controlled and guided way, as a sequence of tokens, that define:

- KB resources: entities, properties, and classes,
- operators (e.g., equal to, greater than, etc.),
- literals: numbers, strings, and dates,
- NL phrases, such as “having”, that play a syntactic sugaring role.

Each token is represented by an NL phrase, consisting of one or more words from the application domain, since operators, variables or URLs used in SPARQL are not allowed. CANaLI operates on tokens in the style of finite state machines, with (currently) 12 states, including the initial and final state. Despite its simplicity, CANaLI is very general, since it can be used with arbitrary RDF KBs, and supports most of the common questions asked by users, including those contained in previous papers and various testbeds (see Section 3).

### 2.1 Answering Simple Questions

The operation of CANaLI can be explained with the help of the transition diagram in Figure 1, and a simple example<sup>1</sup>. For examples, say that the user wants to enter the question: “What is the capital of United States?”. When the user starts typing a new question, CANaLI’s automaton is in the initial state ( $S_0$ ), ready to accept tokens representing the question start. In this case, CANaLI sees “What is the” and it moves to the state  $S_1$ . At  $S_1$ , the system can accept a token representing an entity, a property, or a class. In our example, the user enters “capital”, that is a property recognized by CANaLI. Thus, the system loops back to  $S_1$ , ready to accept as next input token another property, entity, or class. In our simple example the user enters “United States”, that is an entity, and the system moves to  $S_2$ , after recognizing “United States” as an entity with “capital” as valid property. Thus, in order to be consistent with the semantics of the KB, our user must enter entities that have the property “capital”, and the system will stop her from progressing any further if that is not the case. Of course, to reach this ‘no progress’ point

<sup>1</sup>Here, the system response is based on the context provided by the question typed so far and the underlying KB, rather than just the current state and last token as a finite state automaton would.

the user must have ignored the suggestions that the system had previously generated as valid completions of the typed input. CANaLI shows completions under the input area: if the user selects any such completion its text is added to the input area. In  $S_2$  the question mark can be accepted, which marks the end of the question, whereby CANaLI moves to the final state  $S_F$  and launches the actual query execution. Alternatively, the user can enter conditions, using tokens such as “having”, which will be discussed later.

Let us now consider an example involving a chain of properties: “What is the population of the capital of United States?”. In this case, at  $S_1$ , user inputs the property “population”, whereby the system loops back to  $S_1$ . CANaLI now accepts “of capital” because the capitals have a population, and loops back to  $S_1$ , where “of United States” takes us to state  $S_2$  where the question mark completes the processing of the input and launches the query.

Thus, the four basic states  $S_0$ ,  $S_1$ ,  $S_2$ , and  $S_F$ , support a large set of very simple questions asked by everyday users<sup>2</sup>. More complicated but nevertheless common questions are those adding constraints, i.e., query conditions. For instance, assume that the user wants to ask<sup>3</sup>: “What is the capital of countries having population greater than 100 millions?” After the input “What is the capital”, has moved us to  $S_1$ , CANaLI accepts “countries”, as a class that has “capital” as a valid property, and moves to  $S_2$ . In  $S_2$ , CANaLI accepts “having”, and other uninterpreted connectives used as syntactic sugar, to move to  $S_3$ , where it will accept only a valid property. In this case, “population” can be accepted since countries have this property. However, this example illustrates the ambiguity that beset all NL interfaces, no matter how sophisticated their parser is. Indeed, this constraint is also applicable to “capital”, since capitals have population too. Clearly every NL system would suffer from the same problem, and only a person who knows that currently no city has more than 100 millions people, might be able to suggest that the question is probably about countries rather than capitals. However, CANaLI finesses this inherently ambiguous situation by displaying all alternative interpretations whereby the user has to make a choice. Once the property “population” is accepted, and its context clarified, the automaton moves to the state  $S_4$ , that accepts an operator. Thus, the user can input “greater than”. The automaton thus moves to state  $S_5$ , that accepts the right-hand side of the constraint. In general, the right hand side of a constraint can be an element of the KB or a literal. In our example, only a number can be accepted, since the right-hand side must be of the same type as the left-hand side, “population,” which is numerical. Thus, the user enters 100 millions and the automaton moves back to  $S_2$ . From this state, the user can specify more constraints, or input the question mark, ending the question.

Examples of constraints using resources of the KB as right-hand side are the following: “Give me the country having capital equal to Washington.”<sup>4</sup>, “Give me the movies having

<sup>2</sup>Indeed, the most frequent web questions are definition questions (e.g., What is Ebola?), that are even simpler.

<sup>3</sup>This provides a good example of the broken but effective English now supported by CANaLI.

<sup>4</sup>Indeed, the complete automaton of CANaLI has also a transition from  $S_4$  to  $S_2$  that allows to implicitly assume the equality operator. This allows to accept questions such as “Give me the country having capital Washington.”

director equal to a politician.” “Give me the cities having population greater than the population of Los Angeles.” In all the cases, the token accepted in  $S_5$  is a token whose type is semantically coherent with the property previously accepted in  $S_3$ . However, while accepting an entity or a class moves the automaton to  $S_2$ , accepting a property (e.g., *population*) moves the automaton to  $S_1$ , where the element possessing the property must be specified (e.g., *Los Angeles*).

## 2.2 More Complex Questions

For the sake of presentation we have shown in Fig. 1 only the states that are most commonly used in queries. In reality CANaLI has five more states which support the additional patterns which are discussed next via illustrative examples:

- “Give me the cities having population greater than that of Los Angeles.” The use of the pronoun *that* in place of the already used attribute *population*, makes the question more natural than the question where “population” is repeated. However, a special state for handling pronouns had to be added to CANaLI.
- “Give me the actors having birth place equal to their death place.” The use of the possessive determiner implies that the properties *birth place* and *death place* are related to the same variable. A new state is needed here too, since there is no simple way to the rephrase the question using the grammar accepted by the basic automaton in Fig. 1.
- “Give me the actors having birth date greater than that of their spouse.” This question combines the two situations described above.
- “Give me the country having the 2nd largest population”. Questions like this require to sort the results by the value of the attribute accepted in a specific state and to set the offset and number of returned results according to the token accepted in another state, i.e., a token such as *the nth greatest* or *one of the nth greatest*.
- “Give me the drugs without specified side effects”. This question requires negation. We remark that a token such as *without specified* can not be handled as the tokens like *having*, which defines a comparison between two operands.

## 2.3 How CANaLI suggests valid tokens

To achieve real-time response, CANaLI uses an index supported by Apache Lucene, which handles our tokens as if they were Lucene documents. Every acceptable token is associated with one or more phrases of the natural language. When the user types a string  $S$ , a query is performed on the Lucene index, to ensure that the returned tokens (i) have a phrase that matches  $S$ , (ii) have a type that is among the acceptable ones, according to the current automaton state and the previous token, and (iii) are semantically correct, according to the KB, as explained below.

To achieve (iii) above, besides indexing the elements of the KB by their label and type (i.e., entity, property, or class), we use two additional fields: *domain of*, and *range of*. The first is needed in cases such as “What is the population of”: a token can follow if it is domain of “population” (e.g., “capital”, “countries”, “United States”, etc.). The second is used in cases such as “...having capital equal to”: a token can

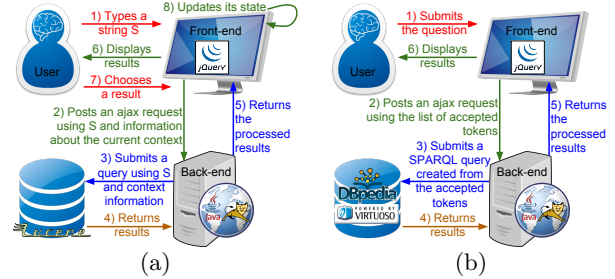


Figure 2: CANaLI’s architecture & work-flow guiding users in (a) typing questions, (b) retrieving answers.

follow if it is range of “capital” (e.g., “birth place”, “city”, “Washington”, etc.). In the case of properties, we also rely on the field *domain*, as needed for cases such as “What is the capital of countries having”, that can be followed by a property having as domain the property “capital”<sup>5</sup> or the class “country” (e.g., “population”, “language”, etc.).

We created the Lucene index using the elements of the 2014 DBpedia release, using all the entities, and all the properties and classes of the DBpedia KB (those in name space <http://dbpedia.org/ontology/>). Also some classes of the Yago KB and the 20k most frequent raw properties (those in name space <http://dbpedia.org/property/>) were indexed. Furthermore, for all the indexed properties having non-literal range, we created an inverted property, and indexed it. The time needed to create such index, by processing the ~100 millions triples of the English DBpedia, is ~25 minutes using a single machine with 32GB of RAM, starting from the raw files downloaded from the DBpedia website. The obtained Lucene index is ~1.1 GB large and can be easily stored in the main memory of a server, thus assuring a nearly instantaneous response to our search queries.

## 2.4 The Architecture of CANaLI

Figure 2 shows the architecture of CANaLI and its work-flow in suggesting and accepting tokens, and computing the answers to the submitted question (b). CANaLI provides a web client, that uses an autocompleter implemented in *JavaScript*, using *jQuery* libraries. The client keeps track of the input tokens and the current state of the automaton, and when the user types a string  $S$  in the auto-completer, an Ajax request is sent to a web server, implemented in *Java*. The server uses the string  $S$  and the status of the automaton to query a Lucene index, that enables to quickly extract the results matching the string  $S$  and coherent with the current status of the automaton. Specifically, the suggested tokens must be syntactically coherent, according to the grammar of the language, and semantically consistent, according to the semantics defined by the KB. Completions are returned to the user, and refined as she types more input. Alternatively the user can select one of the suggested completions, and this selection is used to update the question text entered so far and to select the next state of the automaton. When the final state is reached, a request is submitted from the client to server, that uses the sequence of accepted tokens to create a SPARQL query, that is submitted to DBpedia, or the corresponding endpoints for the other KBs, and the results are shown to the user in a user-friendly snippet format.

<sup>5</sup>Specifically, the range of the property “capital” must be domain for the property “population”

	Proc.	Right	Part.	F proc.	F glob.
CANaLI	46	44	1	0.98	0.92
Xser	42	26	7	0.73	0.63
QAnswer	37	9	4	0.40	0.30
APEQ	26	8	5	0.44	0.23
SemGraphQA	31	7	3	0.31	0.20
YodaQA	33	8	2	0.26	0.18

Figure 3: Results on QALD-5 benchmark - Total number of questions: 49.

### 3. EXPERIMENTAL EVALUATION

A popular set of benchmarks was used to measure the performance of QA systems, i.e., the QALD (Question Answering over Linked Data) benchmarks [4]. The benchmarks consist of sets of NL questions, each associated with a gold standard query in SPARQL, representing the translation of the question. The accuracy of the systems is measured by comparing their results with those obtained by the gold standard queries. We assessed the performances of CANaLI on these benchmarks and the result obtained on each query are presented in [5].

Figure 3 summarizes the results obtained on QALD-5, that consists of 49 questions (the results obtained on the previous benchmarks are equivalent), reporting the results obtained by CANaLI, together with the official results of the participating systems. The columns in the table represent the number of processed questions (“Proc.”), the number of questions answered with F-measure equal to 1 (“Right”), the number of questions answered with F-measure strictly between 0 and 1 (“Part.”), the average F-measure achieved over the processed questions (“F proc.”), and the F-measure over the whole set of questions (“F glob.”), assuming 0 as F measure for the non-processed questions.

CANaLI allows to process 46 questions. The 3 questions that could not be processed require two currently unsupported features: (i) sorting by an aggregate function (e.g., “Which musician wrote the most books?”), and (ii) using arithmetic (“What is the height difference between Mount Everest and K2?”). CANaLI provided a completely wrong answer to one question, namely, “Who is the heaviest player of the Chicago Bulls?”. The question was input in CANaLI by using the property *team*, while the gold standard query used the UNION of both *team* and *draftTeam*, and the correct result was a player associated to *Chicago Bulls* through the latter property. Therefore, CANaLI missed the correct answer. CANaLI provided a partially wrong answer to another question, “Which programming languages were influenced by Perl?”, whose gold standard query used the union of two properties in its constraints (*influence* and *influenced by*). Since CANaLI does not support the UNION operator, only one property was used (*influenced by*), thus missing some of the correct results. Finally, with 44 right answers and a higher F-measure on both the processed and the whole questions, CANaLI proved to be superior to the other systems. Clearly, restrictions imposed by a CNL make an interface like CANaLI a bit less user friendly than full NL interfaces. However, considering that, besides Xser [12], the accuracy of the other full NL systems is far from being acceptable, we believe that an accurate answer is worth a bit extra effort spent in rephrasing the question.

### 4. DEMONSTRATING CANALI

In this demonstration session, we will exhibit the power, usability and flexibility of CANaLI, by starting from simple questions and moving to more complex ones. The attendees will see how CANaLI guides users in typing questions by allowing to type questions that are only semantically correct w.r.t. the underlying KB: as soon as the typist hesitates or halts even momentarily, the system comes to the rescue by suggesting a list of correct completions the user can select from. This enables people to self-learn CANaLI easily and quickly. In fact, in our demo, after asking the attendee for new questions, we will invite them to enter their questions directly into CANaLI. We will then demonstrate the QA effectiveness of the system by testing its superior precision and recall on complex questions taken from published testbeds that have thwarted the efforts of other QA systems. Finally, we will explain briefly the working of the system, and how the SPARQL queries are generated. This will also allow us to clarify the reasons for the flexibility and generality of the approach, whereby we will show CANaLI in action on several KB, including MusicBrainz and biomedical KBs, and discuss our current work-in-progress to extend it to support temporal questions on the archived history of Wikipedia/DBpedia.

### Acknowledgements

The authors wish to thank Maurizio Atzori, Shi Gao, and Matteo Interlandi for very helpful discussions. This research was supported in part by a 2015 Google Faculty Research Award.

### 5. REFERENCES

- [1] <http://wiki.dbpedia.org/>.
- [2] <http://linkeddatacatalog.dws.informatik.uni-mannheim.de/>.
- [3] <http://www.w3.org/TR/sparql11-overview/>.
- [4] <http://greententacle.techfak.uni-bielefeld.de/~cunger/qald/>.
- [5] <http://yellowstone.cs.ucla.edu/canali/>.
- [6] M. Atzori and C. Zaniolo. Swipe: searching wikipedia by example. In *Proceedings of the 21st World Wide Web Conference*, 2012.
- [7] B. F. Green, Jr., A. K. Wolf, C. Chomsky, and K. Laughery. Baseball: An automatic question-answerer. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, 1961.
- [8] P. Gupta and V. Gupta. A survey of text question answering techniques. *International Journal of Computer Applications*, 53(4):1–8, 2012.
- [9] R. Hahn, C. Bizer, C. Sahnwaldt, C. Herta, S. Robinson, M. Bürgle, H. Düwiger, and U. Scheel. Faceted wikipedia search. In *Business Information Systems, 13th International Conference*, 2010.
- [10] L. Han, T. Finin, and A. Joshi. Schema-free structured querying of dbpedia data. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, 2012.
- [11] S. R. Petrick. On natural language based computer systems. *IBM J. Res. Dev.*, 20(4):314–325, July 1976.
- [12] K. Xu, Y. Feng, and D. Zhao. Answering natural language questions via phrasal semantic parsing. In *Working Notes for CLEF 2014 Conference*, 2014.

# tPredictor: A Micro-blog Based System for Teenagers' Stress Prediction

Jing Huang, Qi Li, Zhuonan Feng, Yiping Li, Ling Feng  
 Dept. of Computer Science and Technology  
 Centre for Computational Mental Healthcare Research, Institute of Data Science  
 Tsinghua University, Beijing 100084, China  
 {j-huang14,liqi13}@mails.tsinghua.edu.cn, fzn0302@163.com,  
 fengling@tsinghua.edu.cn

## ABSTRACT

Too much stress is easy to do harm to the physical and psychological health of teenagers, because most teenagers neither have the ability to cope with the stress by themselves nor like seeking adults' help initiatively. Social media has demonstrated its feasibility in detecting teenagers' stress with the micro-blog having become a popular channel for teenagers' self-expression. In this demonstration, we present a system called *tPredictor*, which can predict teenagers' future stress based on the social media micro-blog. Two questions are to be resolved: (1) what will the stress level of the teenager(s) be in the next time unit? (2) how will the stress level of the teenager(s) change (increase, decrease, remain unchanged) in the next time unit? *tPredictor* tackles the above prediction questions, taking into account the influence of future stressful events on teenagers' emotion. Considering the similarity of stock price movement and the stress level change, we define the stress candlestick charts and the stress reversal signals for stress change trend prediction. *tPredictor* can predict the stress of both individuals and a group of teenagers, which provides a platform for teenagers themselves, their parents or some institutions such as schools to know teenagers' future stress for taking measures timely to prevent the serious consequences.

## Keywords

Micro-blog, teenager, psychological pressure, prediction

## 1. INTRODUCTION

Nowadays, teenagers are inevitably suffering much stress from various aspects. A survey, conducted by the American Psychological Association in August 2013, showed that the teenagers were the most stressed-out age group in the U.S [1]. Due to the unsoundness of teenagers' psychological regulation mechanism, too much stress contributes to teenagers' bad consequences more easily such as sleepless-

ness, depression and even suicide. However, for most teenagers, they lack experience and can't realize the seriousness of their stress so that they seldom actively seek for adults' help. And for guardians, parents can't be around and focus on teenagers every minute and even abundant teachers are not sufficient to attend to each student. Therefore, it is very useful to find methods for teenagers to understand their own stress, for parents to timely know teenagers' stress situation when teenagers don't initiatively reveal their stress to them and for teachers to be able to get hold of all the students' stress status. Nowadays, micro-blog has become a major channel for teenagers' self-expression. Teenagers post so many tweets expressing their personal emotions everyday, which provides abundant available data to detect teenagers' stress and further predict their stress change. In the literature, many researchers have studied using micro-blog to analyze people's mental health. [8] found the difference between depressed and non-depressed people through analyzing their tweeting contents and behaviors. [9] proposed a depression detection model based on the sentiment analysis of the micro-blog. [10, 5] provided a machine learning method to detect teenagers' stress of study, affection, inter-personal and self-cognition. However, all the studies aimed to detect the existing psychological problems where the harm has done in fact. To prevent bad consequences, [3] investigated people's social media behaviors and built a statistical classifier to predict the depression. Our previous work focused on the teenagers group and predicted their future stress using a stress level time series detected from micro-blog. It integrated the stressful event to predict future stress level and defined the stress candlestick chart to predict future stress general change trend [7, 6].

In this demonstration, we present a system called *tPredictor* to predict teenagers' future stress value and change trend. It is a system implementation of our previous work [7, 6] which can visualize the prediction results and enable users to easily understand the stress prediction results. For the functionalities, we further extend the system to the group stress prediction. It aggregates all the prediction results of teenagers and presents some statistical results of the group stress prediction. It also picks up the teenagers needing help based on out predicted future stress status from a group of teenagers. *tPredictor* provides a straightforward way for users to obtain the most important information, as well as get hold of the overall stress situation and meanwhile know well each teenager's stress status. Therefore, our system can be applied to both institutional users, such as schools and



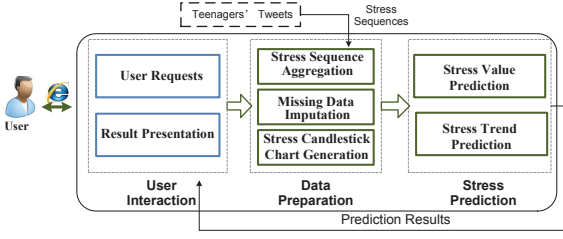


Figure 1: System Architecture

individual users such as teenagers themselves and parents.

## 2. SYSTEM ARCHITECTURE

Fig.1 shows the system framework including the *User Interaction*, the *Data preparation* and the *Stress Prediction*.

### 2.1 User Interaction

**User Requests** receives users' requests including adding new teenagers to their concerning list, choosing teenagers to be predicted, and setting prediction parameters like the time granularity and the event information.

**Result Presentation** aims to visualize the prediction results for users' easy understanding. It analyzes the group prediction results and picks up the most important information to users. In details, it aggregates teenagers by different predicted stress values and change trends, and presents their distributions through lucid charts, which helps users get hold of the overall stress situation. Besides, it demonstrates each teenager's predicted stress together, which enables users to do the comparative analysis and quickly find the teenagers whose future stress will be severe.

### 2.2 Data Preparation

The *Data preparation* preprocesses the original stress sequences which detected from teenagers' micro-blog [10].

**Stress Sequence Aggregation** aggregates the tweets' stress sequence with different time granularity (day, month, etc.) to obtain six stress-related indexes. The stress sequence is the  $Stress(T): (t_1, l_1), (t_2, l_2) \dots (t_n, l_n)$ , where  $t_i$  represents the time of  $i_{th}$  tweet, and the  $l_i$  represents the stress level of the  $i_{th}$  tweet. The six stress-related indexes include the max stress level  $L_{max}$ , the min stress level  $L_{min}$ , the average stress level  $L_{avg}$ , the sum stress level  $L_{sum}$ , the number of stress tweets  $L_{scount}$ , the total number of tweets  $L_{count}$ , and the proportion of stress tweets  $L_{proportion}$  in the aggregated time interval.

**Missing Data Imputation** aims to solve the missing data problem which is because of the casualness of users' tweeting time and frequency. We apply the Gaussian Process Regression to do the data imputation. The adjacent data, including the stress value before and after the missing data, will be used to inference the missing value. The  $(\Delta t, L)$ , where  $\Delta t$  and  $L$  denote the time distance and the stress level of the adjacent time unit, is used as the input feature vector form. We use non-missing data as the training data and then get the estimated value of the missing data through the trained model. Our experiments showed that the Gaussian Process Regression performed better than other imputation methods including the nearest mean approach, the linear interpolation and exponential smoothing.

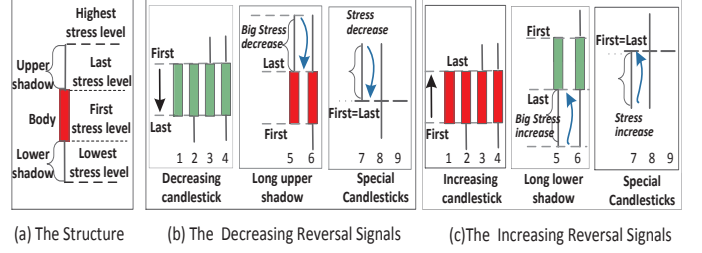


Figure 2: The Stress Candlestick Charts

**Stress Candlestick Chart Generation** generates the stress candlestick chart  $SC$  and its corresponding feature vector  $SCF$ . The stress candlestick derives from the candlestick chart of stock price due to some similarities between stock price movement and stress level change. For instance, the stock price is influenced by the game between sellers and buyers and the related events of companies while the stress level is influenced by the personal self-regulation mechanism and the stressful events. The stress candlestick chart  $SC$  is defined as  $(L_{first}, L_{last}, L_{high}, L_{low})$ , namely the first, last, highest and lowest stress level in the time unit respectively. The stress candlestick chart feature vector  $SCF$  is defined as a five tuples  $(shape, bodylen, upperlen, lowerlen, changeslope)$  which represents the shape, body length, upper shadow length, lower shadow length of the  $SC$  as Fig. 2 (a) shows and the stress change rate between two  $SC$ s.

### 2.3 Stress Prediction

#### 2.3.1 Stress Value Prediction

*Stress Value Prediction* aims to predict the max, min and average stress level of the next time unit. The three stress values can comprehensively represent the teenagers' stress status and are also the major concerns to their followers.

**Feature-Aware Time Series Prediction.** According to Granger Causality analysis, the  $L_{max}$ ,  $L_{min}$  and  $L_{avg}$  are related to not only their past values but also the other stress-related indexes such as the  $L_{sum}$ ,  $L_{scount}$ ,  $L_{count}$  and  $L_{proportion}$ . Based on the confidence of 95%, we find that the  $L_{max}$  is correlated to  $L_{sum}$ ,  $L_{min}$  is correlated to  $(L_{proportion}, L_{count})$ , and  $L_{avg}$  is correlated to  $(L_{sum}, L_{proportion})$ . We apply the seasonal Autoregressive Integrated Moving Average (SVARIMA)[2] approach to our problem. The key function of the stress value prediction is:

$$L_{n+1} = C + \sum_{i=0}^{k-1} A_i L_{n-i} + \sum_{i=0}^{k-1} B_i \bar{X}_{n-i} + \sum_{i=0}^{r-1} \theta_i \varepsilon_{n-i} + \varepsilon_{n+1}$$

$L_{n+1}$  is the stress value we want to predict, where  $L$  is the past values of the corresponding predicted stress indexes (max, min and average). The  $\bar{X}$  represents different features of the other correlated stress value sequences such as the  $(L_{sum}, L_{proportion})$  to  $L_{avg}$ . The order  $k$  and  $r$  is determined by the Akaike's Information Criterion (AIC) and other parameters are estimated through damped least squares method.  $\varepsilon_i$  is the white noise error terms satisfying  $E(\varepsilon_i) = 0$ , and  $C$  is a constant.

**Stressful Event Influence.** The SVARIMA model predicts stress values based on the historical stress change pattern. However, some happened or forthcoming stressful events may add extra influence which can't be neglected. Hence, we use the Moving Average Convergence/Divergence(MACD) to depict the extra stressful event influence. We first find

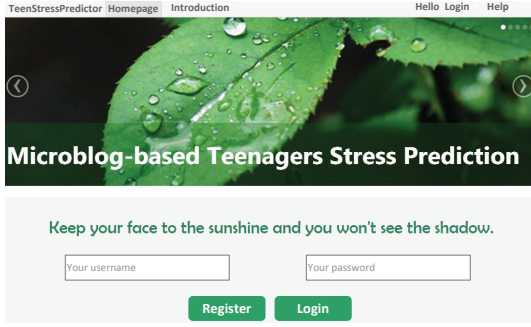


Figure 3: The System Homepage

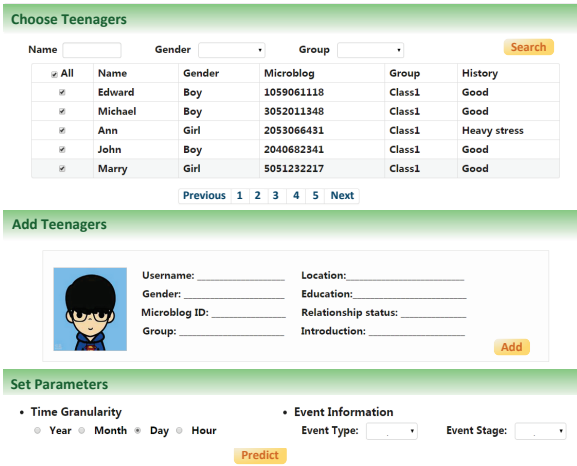


Figure 4: User Requests

the stressful events and divide them into study and emotion event sets according to the users' tweets. Then we determine the start and end of the event influence cycle along with the stress value: 0. In the event influence cycle, we compute the MACD of the stress values, which can get a sequence representing the extra event influence. For the two event influence sequence sets, we use the Generalized Sequential Pattern(GSP) mining algorithm to mine the frequent sequence with 50% confidence to represent the extra influence of the two kinds of events. Finally, we divide equally the mined sequence into early, middle and later stage, and use the average MACD value of each stage as the adjustment value, which will be added to the original value predicted by SVARIMA model, if the predicted time is in the corresponding stage of different kinds of stressful events.

### 2.3.2 Stress Trend Prediction

*Stress Trend Prediction* aims to predict the future stress change trend including increase, decrease and remaining unchanged. We explore the candlestick chart to predict the stress trend. It outperforms the trend prediction method using the predicted stress values to subtract the last one, which might be a small fluctuation within the future trend.

**Stress Reversal Signals.** Fig. 2 (b) shows the decreasing reversal signals in a increasing process and Fig. 2 (c) shows the increasing reversal signals in a decreasing process. We take the decreasing reversal signals as examples for interpretation. For the 1-4 decreasing reversal signals, their

last stress level is lower than the first stress level, which represents a decrease in the time unit of the stress candle stick and indicates a continual decrease trend in the future. For the 5-6 decreasing reversal signals, their max stress level is much higher than the last stress level. It indicates the heavy stress is gradually released through teenagers' stress regulation mechanism in the time unit of the stress candle stick, where the release mechanism is likely to remain effective in the future. For the 8-9 decreasing reversal signals, their first stress level is the same as the last stress level which means the power of stress release balances the stress accumulation. In the past increasing process, the power of stress accumulation is stronger than the stress release and now they are balanced. Therefore, the power of releasing stress may become stronger than accumulating stress next, which signifies the stress will decrease in the future.

**Trend Decision Making.** Let  $n$  be the current time, we trace back to the nearest local highest or lowest stress level and form a stress pattern  $P_{current}$ :  $(SCF_{n-k+1}, \dots, SCF_n)$  sequence, where  $k$  is the length of  $P_{current}$ .

[Case 1] If  $SCF_n$  is not the reversal signal, the stress change trend will continue.

[Case 2] If  $SCF_n$  is the reversal signal, we decide whether the stress change trend will reverse according to the past experience. Firstly, we define the distance  $D(SCF_i, SCF'_i)$  between two  $SCF$ s to find similar past stress pattern  $P_{past}$  to the current stress pattern  $P_{current}$ .

$$D(SCF_i, SCF_j) = \sum_{k=1}^5 w_k D(f_{ik}, f_{jk}), \quad \sum_{k=1}^5 w_k = 1.$$

Here,  $f_{ik}$  and  $f_{jk}$  denote feature values of  $SCF_i$  and  $SCF_j$ , and  $w_k$  is the parameter determined by the analytic hierarchy process (AHP). For the nominal feature *Shape* of  $SCF$ ,

$$D(f_{i1}, f_{j1}) = \begin{cases} 1, & \text{if } f_{i1} \neq f_{j1} \\ 0, & \text{if } f_{i1} = f_{j1}. \end{cases}$$

For other four numeric features,  $D(f_{ik}, f_{jk}) = |f'_{ik} - f'_{jk}|$ ,  $j = 2, \dots, 5$ , where  $f'_{ik}$  and  $f'_{jk}$  are the normalized  $f_{ik}$  and  $f_{jk}$  between 0 and 1. We set a distance threshold to judge whether two  $SCF$ s match successfully. After matching the last  $SCF_{n-k+1}$ , we get a past pattern  $P_{past}$  with length of  $t$ . If the *matchrate* =  $k/t$  is higher than a threshold, we consider the two patterns match successfully. Finally, we obtain a set of matched patterns  $P_{matched}$  and observe the future trend after the reversal signals in  $P_{matched}$ , where the stress level time series have been segmented [4] to eliminate the influence of fluctuation to get the general change trend. We choose the majority change trends of  $P_{matched}$  as the predicted result. If there is no matched sequence, the stress change trend will be predicted to reverse.

## 3. EVALUATION

We collect stress sequences of 91 teenagers obtained by the stress detection method whose precision is proved to be 82.6% [10], for the prediction results evaluation. For the stress value prediction, integrating stressful events and SVARIMA model reduces the MAPE (Mean Absolute Percentage Error) of the predicted max, min and average stress level by 18%, 43%, and 3% separately, compared to the single SVARIMA model. For the stress trend prediction, the precision of our method achieves 83.79% which outperforms the time series (74.82%), MACD (63.59%) and KDJ (Stochastic Oscillator) (66.73%) based prediction approaches.

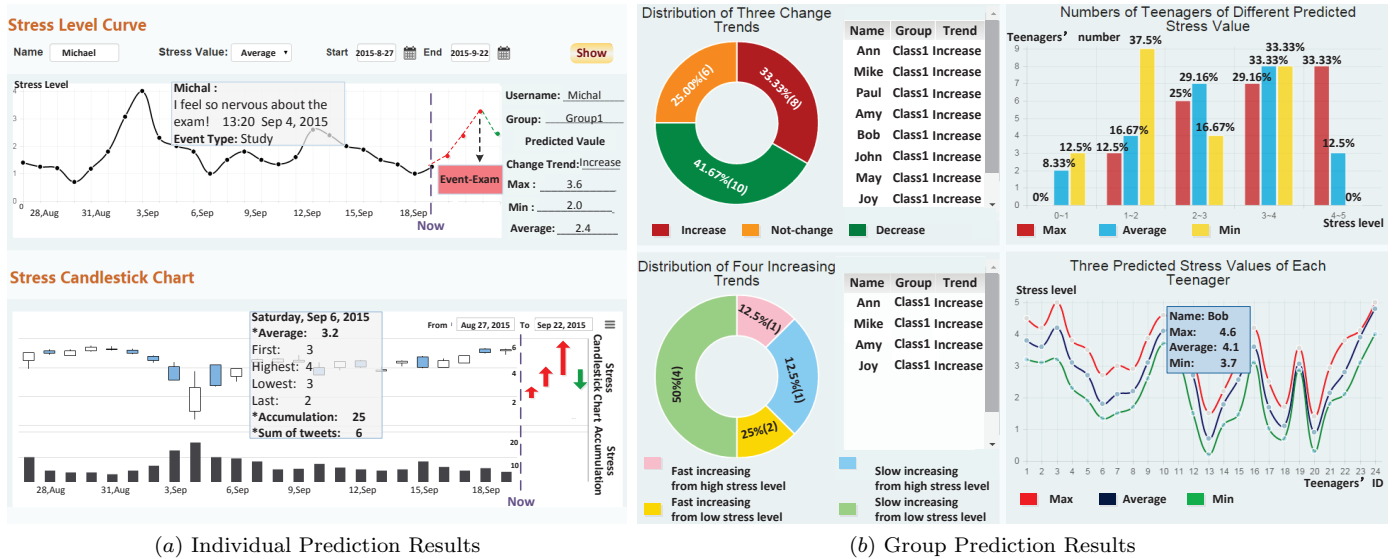


Figure 5: Prediction Results

#### 4. SYSTEM DEMONSTRATION

During the demonstration, attendees can access *tPredictor* through the browser and experience our friendly interaction.

First, the user enters the home page of *tPredictor* and logs in with his/her account as Fig. 3 shows.

After logging in, the user enters the second page as Fig. 4 shows. In this page, the user can view the profile of the teenagers listed based on the user's search. The user can click teenagers in the list to see more detailed information such as the photo and self-introduction. Besides, a new teenager can be added to his/her teenagers list. For the prediction, the user chooses one or a group of teenagers to be predicted and sets the parameters including the time granularity and the stressful event information. Here the user set the time granularity to be day, event type to be study and the event stage to be early about the upcoming exam.

When the user chooses one teenager to be predicted, the prediction results are presented through two charts as Fig. 5 (a) shows. The stress curve exhibits the past and predicted stress values of the teenager, where the original tweets content and event information can also be presented in this chart when the user clicks the corresponding point. The stress candlestick chart is to demonstrate the future stress change trend of the teenager and the detailed stress-related indexes are also shown in this chart.

When the user chooses a group of teenagers to be predicted, he/she can see both individual prediction results of two charts in Fig. 5 (a) and the statistical group prediction results as Fig. 5 (b) shows. The two doughnut charts tell the user about the teenagers proportion of different future stress change trends and the corresponding teenagers will be listed directly when the user clicks one part of the doughnut chart. It helps the user easily know whose stress will increase. The histogram shows the corresponding number of teenagers whose future stress values are in different stress level range. Through the histogram, the user can understand the overall stress situation of the group in the future. For example, the stress situation of the group is severe when most teenagers' future stress values are between 2-5. The line

chart shows the predicted stress value of each teenager of the group, through which the user can easily find teenagers who have much higher stress level

#### Acknowledgement

The work is supported by National Natural Science Foundation of China (61373022, 61532015, 71473146) and Chinese Major State Basic Research Development 973 Program (2015CB352301).

#### 5. REFERENCES

- [1] APA's 2013 Stress In America survey. <http://www.apa.org/news/press/releases/stress/2013>, 2013.
- [2] G. Box and G. Jenkins. *Time series analysis: Forecasting and Control*. Holden-Day, San Francisco, 1970.
- [3] M. De Choudhury, M. Gamon, S. Counts, and E. Horvitz. Predicting depression via social media. In *ICWSM*, 2013.
- [4] J. Jiang, Z. Zhang, and H. Wang. A new segmentation algorithm to stock time series based on pip approach. In *Wireless Communications, Networking and Mobile Computing, 2007. WiCom 2007. International Conference on*, pages 5609–5612. IEEE, 2007.
- [5] Q. Li, Y. Xue, J. Jia, and L. Feng. Helping teenagers relieve psychological pressures: A micro-blog based system. In *EDBT*, pages 660–663, 2014.
- [6] Y. Li, Z. Feng, and L. Feng. Using candlestick charts to predict adolescent stress trend on micro-blog. *Procedia Computer Science*, 63:221–228, 2015.
- [7] Y. Li, J. Huang, H. Wang, and L. Feng. Predicting teenager's future stress level from micro-blog. In *Proc. of CBMS*, 2015.
- [8] M. Park, D. W. McDonald, and M. Cha. Perception differences between the depressed and non-depressed users in twitter. In *Proc. of ICWSM*, 2013.
- [9] X. Wang, C. Zhang, Y. Ji, L. Sun, L. Wu, and Z. Bao. A depression detection model based on sentiment analysis in micro-blog social network. In *Trends and Applications in Knowledge Discovery and Data Mining*, pages 201–213. Springer, 2013.
- [10] Y. Xue, Q. Li, L. Jin, L. Feng, D. A. Clifton, and G. D. Clifford. Detecting adolescent psychological pressures from micro-blog. In *Health Information Science*, pages 83–94. Springer, 2014.

# OSNI: Searching for Needles in a Haystack of Social Network Data

Shiwen Cheng, James Fang, Vagelis Hristidis, Harsha V. Madhyastha<sup>†</sup>,  
 Niluthpol Chowdhury Mithun, Dorian Perkins, Amit K. Roy-Chowdhury,  
 Moloud Shahbazi, and Vassilis J. Tsotras

University of California, Riverside, Riverside, CA, USA

<sup>†</sup>University of Michigan, Ann Arbor, MI, USA

schen064@cs.ucr.edu, amitrc@ece.ucr.edu, jfang003@cs.ucr.edu, vagelis@cs.ucr.edu, harshavm@umich.edu,  
 nmithun@ece.ucr.edu, dperkins@cs.ucr.edu, mshah008@cs.ucr.edu, tsotras@cs.ucr.edu

## ABSTRACT

This paper presents the Online Social Network Investigator (OSNI), a scalable distributed system to search social network data, based on a spatiotemporal window and a list of keywords. Given that only 2% of tweets are geolocated, we have implemented and compared various state-of-art location estimation techniques. Further, to enrich the context of posts, associations of images to terms are estimated through various classification techniques. The accuracies of these estimations are evaluated on large real datasets. OSNI's query interface is available on the Web.

## 1. INTRODUCTION

The amount of user generated data increases every year, as more social interaction tools like Twitter, Instagram and Facebook are being created and more users use them to share their everyday experiences. Most research on analyzing social data has focused on detecting trends and patterns such as bursty topics [11] and popular spatiotemporal paths [10], event extraction [7], studying how information is spread [12], or analyzing properties of the social graph.

In contrast, in this paper, we study how to search social network data items, posts and images, based on spatiotemporal keyword queries. That is, we created methods to find the right needles (social data items) in the haystack (social networks), which we refer to as *investigative search*, to contrast it to the trending queries studied by previous work. Investigative search can also be viewed as exploring the currently untapped long tail of the distribution of topics in social networks. We use law enforcement as our focus application. The system capabilities and user interface were created in consultation with the University of California Police Department.

Figure 1 shows the user interface of the developed OSNI, available at <http://db1lab-rack30.cs.ucr.edu/IARPA/>, where a user may select a spatial area on the map, a time range, and specify keywords. OSNI returns a list of posts (tweets)

ranked by their relevance to the query. The relevance is computed using a combination of the relevance of the text (based on an Information Retrieval function) and of the image (based on the confidence of the relevance of the query to an image) to the query. Only posts that belong to the specified spatiotemporal window are returned.

A key challenge is that only about 2% of the tweets are geolocated (have GPS location). Another challenge is how to associate images with terms. For example, if a tweet's image shows a "bike" we would like to return this tweet for the query "bike" even if it does not contain this word in its text. And a third key challenge is how to scale OSNI to a throughput of millions of posts per day, and how to facilitate interactive query response times.

This demo paper has the following contributions:

- We have adapted and implemented several social posts location estimation methods, and we have evaluated them on a dataset of millions of posts.
- We implemented a method to classify images based on the terms they are relevant to. We evaluated this classifier for various datasets.
- We built a scalable overall architecture, by combining several leading big data technologies. We experimentally evaluate the throughput and distributed performance of the system. We make the system publicly available on the Web.

## 2. ARCHITECTURE

The architecture of OSNI is shown in Figure 2. OSNI was written in approximately 8K lines of Java code. The whole OSNI, including all modules in the figure, is deployed on a cluster of two machines, which host instances of several systems (Cassandra, ElasticSearch, Spark). The Web server runs on one of the two servers.

The OSNI uses the Twitter Streaming API to collect tweets from Twitter. We specify a keyword-based filter on the Streaming API to only retrieve tweets that contain at least one of a collection of 64 keywords (provided by the law enforcement agency). This makes our data more focused to our domain, given that a single machine can only receive up to about 1% of Twitter's traffic. Matching records are stored in a Cassandra [5] database cluster of two nodes with replication factor 2 (each tweet is copied in both machines).

The preprocessing module continuously queries Cassandra for unprocessed records and runs a distributed job on a

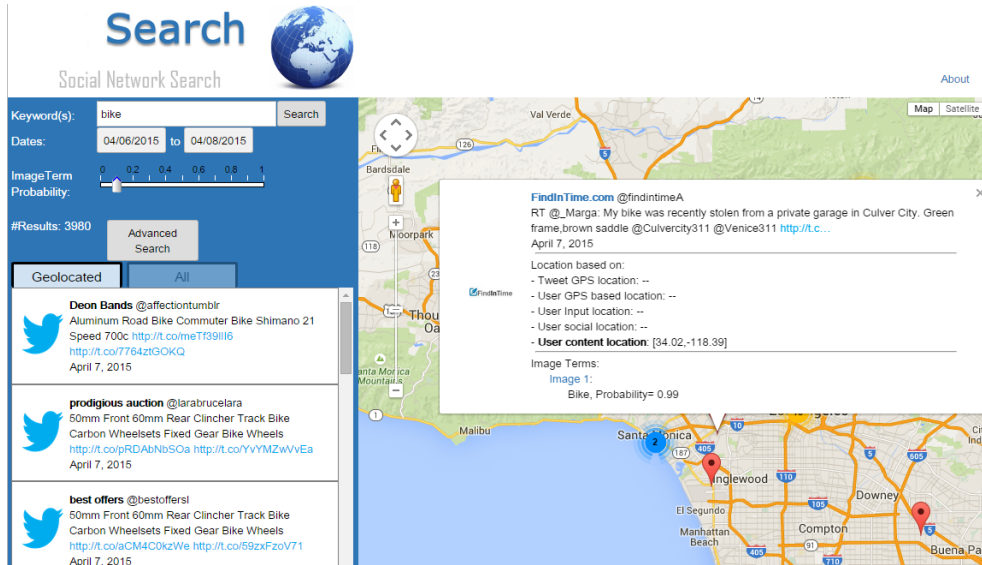


Figure 1: User Interface

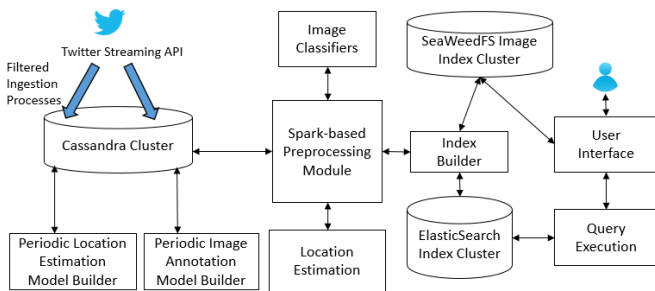


Figure 2: Architecture

Spark [1] cluster to process these records. For each record, we perform location estimation to determine the approximate location of the user at the time the tweet was posted (Location Estimation module in Figure 2, details in Section 3). Further, if the tweet contains an image, we also apply an image classifier, which downloads the image(s) from the Internet (only a URL is stored in the tweet record), and then runs the classification algorithm (Image Classifiers module in Figure 2, details in Section 4). If desired, images can be stored locally in SeaweedFS [4], a distributed storage system tuned towards storing images.

The enriched posts (enriched by location and image terms) are inserted into Elasticsearch [3], a distributed document search system. The Web-based query user interface, shown in Figure 1, is built using Google’s Web Toolkit (GWT), which is a Java-based Web toolkit. It is hosted on Apache Tomcat.

### 3. LOCATION ESTIMATION

We found that only about 2% of tweets are geolocated (e.g., using the GPS of a mobile phone). Hence, we need effective ways to estimate the location of the majority of tweets. We consider several methods to assign location:

*User Location String.* Here we use the user’s location as

an estimate of the tweet’s location. Twitter allows users to enter a location on a text field. Many users specify their city, e.g., “Riverside, CA”, but others specify imaginary addresses, e.g., “On the moon.” We use Google’s Map API to map a location string to latitude and longitude.

*Places.* Users may check-in or otherwise specify a place in the tweets (there is a *place* field in the JSON of a tweet), which can be something like “UCR Campus, Riverside, CA.” Again we use Google’s Map API to get coordinates.

*User GPS.* As in *User Location String*, we use the user’s location as an estimate of the tweet’s location, but here we compute the user’s location as the median of that user’s GPS locations in the last 30 days. Only users with at least 3 geotagged tweets are assigned a location using this method.

*Social Network Approach.* This approach is based on Compton et al. [9], where the assumption is that users are often located close to their friends. Users A and B are considered “friends” if User A has tweeted at least 3 times to User B and User B has tweeted at least 3 times to User A. We first calculate GPS medians for the users that have at least three geotagged tweets in the past and store the information in the user table. Afterwards, we build a social graph and estimate the locations that are still unknown using an error minimization technique. Our approach differs from the original approach [9] in a few ways. First, the number of tweets needed to be a “friend” is decreased from 3 to 2 to increase the size of the graph because our graph was too sparse. The graph size increased by 21% and the accuracy went down only by 2%. Second, we are making use of the places field used in the user’s tweets to assign more locations in the graph.

*Content-based.* This method looks for “local” words – e.g., “howdy” in Texas or “White House” around the White House in Washington, DC – in the body of the tweets to assign location to them. We build upon the method proposed in Cheng et al. [8], where instead of using cities, we use zipcodes (over 31,000 zipcodes in the US), that is, we assign terms to zipcodes. Further, we use a much larger number of tweets to increase the accuracy. Specifically, we use over

100 million geotagged tweets to calculate the focus and dispersion for each word to decide if it is a local word. Then, we build an index that maps terms to zipcodes with a probability, and assign to a tweet the zipcode with the highest probability among its words.

**Implementation considerations.** The *social network approach* has two parts. The preprocessing and the graph processing. The first step of the preprocessing is to create a unidirectional graph based on who tweets to whom so we can later build the friendship graph. The second is to collect tweets that have GPS locations and places location. In the graph processing, we first calculate the GPS medians for users with more than 3 geotagged tweets. Next, we build the graph and estimate the locations of their friends. The graph processing takes more than one day to run on a single machine, and hence we execute it once a week. The preprocessing is performed once per day. Similarly, for the *content-based approach* the local words are calculated once per day using gathered data from the last week.

**Dataset and Evaluation.** To measure the *accuracy* of our approaches, we used as ground truth 100 million geotagged tweets and compared the estimations of the various methods to these GPS locations. The *coverage* is the percentage of tweets that are assigned a location using a method; note that a tweet may be assigned a location based on multiple methods.

Type	10 miles	30 mile	50miles	coverage
Tweet GPS	100%	100%	100%	1.97%
User String	71%	73%	74%	27.6%
Places	81%	82%	83%	2.3%
User GPS	92%	95%	96%	1%
Social	82%	83%	85%	3.31%*
Content-based	24.7%	25.9%	27.3%	76%

Table 1: Tweet Location Estimation Accuracy and Coverage Evaluation. In Social Network approach, we only consider posts that have not already been assigned a location through Tweet GPS, User String, Places or User GPS.

## 4. IMAGE ANNOTATION

In this section we study how to extract keywords from the images of tweets to enhance the accuracy and effectiveness of our query interface. That is, a tweet that shows a picture of a bike, should be returned for query ‘bike’ even if it does not contain the word ‘bike’ in its body. We found that about 25% of Tweets have images.

Specifically, the image annotation module inputs an image and outputs a set of (term, confidence) pairs, e.g., (bike, 0.72), where the confidence denotes how probable it is that the image is related to the term. We use a vocabulary of terms that we want to detect in images and for each term we build a classifier.

After experimenting with various classifiers and image feature extraction methods, we chose Support Vector Machine (SVM) as classifier and SURF Detector [6] based Bag-of Words Model [13]. We found that single-class classifiers are more accurate than multi-class ones.

The method for training the classifier works in two major steps. At the first step, SURF features are extracted from the training set (described below) that contains all images of all classes. A visual vocabulary of features (Bag of Words)

is created by reducing the number of features through quantization of feature space using K-means clustering. The resulting space has 5000 features. Then, in second step, occurrences of all visual features in the vocabulary are calculated from each of the images. A histogram of features is created per image, which is a 5000-entries long feature vector, which is a reduced representation of the image. Using these feature vectors and corresponding known labels for images in the training set, the SVM classifier is trained (offline, once per week). In the online system, when a new image comes, SURF features for the image are calculated and the image is represented as a feature vector. The label of the image and the confidences are estimated by feeding the feature vector into the SVM models created during the training phase.



Table 2: Examples of a few top ranked and relevant images collected from Google and Twitter based on textual query ‘bike’.

**Data Collection and Evaluation.** We consider 12 terms, and we build a training set as follows. We query Google Images and Twitter Images interfaces for each of the terms. Then, the retrieved images are checked manually to remove false matches. In total, we keep 200 images from Google and 200 from Twitter per term, that is, we have a total of 4800 images. The query words are: bike, gun, robbery, crowd, car, concert, murder, drunk, fire, helmet, family and friends. These words were chosen based on the law enforcement focus of the project.

The reason behind collecting training images from Google and Twitter is twofold. First, there is no existing image dataset, that contains images corresponding to all possible query words. Second, carefully chosen images from web search and social media search will make the training set both diverse and relevant. Table 2 shows examples of diverse images collected from Google and Twitter for ‘bike.’ We see that the Google images are generally more “clean”, whereas the Twitter images offer more diversity. As we will see later the combination of the two leads to better accuracy.

K-fold ( $k = 10$ ) cross-validation is used to test the classification performance. Table 3 shows the performance of the image classifier significantly varies, when trained with different training sets. The results demonstrate that incorporating images from both Google and Twitter for training improves classifier accuracy, as more diverse and informative set of images are included.

Our classifier has high accuracy (more than 80%) on categories like bike, gun, crowd etc. On the other hand, the accuracy on complex categories like drunk and murder was lower (around 50%). Some interesting examples of correctly identified and incorrectly identified images for category ‘bike’,

‘friends’ and ‘gun’ are shown in Table 4.

	Testing			
Training \		Google	Twitter	Google & Twitter
Google		63%	36%	49.5%
Twitter		43%	31%	37%
Google & Twitter		79%	55%	67%

Table 3: Accuracy of Image Classifier with different sets of training and test images

JavaCV (Java interface to open computer vision library) is used to extract the image features. When running as a stand-alone application, the image classifier takes 5.98 millisecond on average to classify an image for a term on a system of quad-core Intel Core i5-4200M 2.50GHz CPU, 8 GB DRAM, 500 GB 7.2K RPM HDD.

True Positive						
	Bike(0.652)	Bike(0.879)	Friends(0.437)	Friends(0.673)	Gun(0.828)	Gun(0.99)
False Positive						
	Gun(0.805)	Gun(0.89)	Friends(0.374)	Friends(0.332)	Bike(0.722)	Bike(0.247)

Table 4: A few True-Positive and False-Positive examples from Image Classifier. The detected class is given below the images; confidence is in parenthesis.

## 5. SCALABILITY

Our OSNI is currently deployed on a cluster with two servers, each with two 6-core Intel Xeon E5-2630 2.30GHz CPUs, 96 GB DRAM, 4x 4 TB 7.2K RPM SATA HDDs, and connected via gigabit Ethernet.

Total records	3.46M	100%
Records w/ image URL	863.62K	24.96%
Records w/ non-image URL	1.32M	38.15%
Records w/ no URL	1.59M	45.95%

Table 5: Tweet record statistics for July 2015.

**Latency.** Based on our current set of 65 keywords used by the Twitter Streaming API to scrape Twitter’s data, we store approximately 3.5 million tweet records per day. On average, it takes 18 hours to process 1-day worth of tweet records, or 18.5 ms per record. Table 5 shows the average daily composition of these records over a one-month period (July 1, 2015 to July 31, 2015). We see that about half of the posts have no URL, which support our intuition that there are many non-news related posts, which can be used for investigative exploration (viewing a user as a witness).

P	Location	Image	Index	Total
1	17.42	10.32	534.45	6.68
8	168.0	67.79	1622.67	38.44

Table 6: Throughput, in records per second, of the OSNI indexer for different levels of parallelism (P).

**Throughput.** We evaluate the throughput of our OSNI indexer on a sample set of 50K records to understand the

performance of the location and image classifiers, and inserting into the Elastic Search index. We compare performance using parallelism level 1 and 8. In Spark, parallelism determines the number of shards the data is split into before being distributed for processing. We show the throughput of each component and the full indexer in Table 6. We show clear speedup improvement when increasing the parallelism, and expect further improvements on a larger-scale cluster.

## 6. CONCLUSIONS AND FUTURE WORK

We presented OSNI, an interdisciplinary system to facilitate searching in social networks. The key contributions are the location estimation, the terms extraction from images, and the scalable architecture. To scale to more terms in the image annotation phase, we will study how multi-class classifiers can be effectively applied – single-class classifiers performed better in our experiments. Further, we are working on building a system that can collect informative training example images without human effort in filtering out irrelevant images. Finally, instead of utilizing different modules (eg. Cassandra, Spark), we are examining how to implement OSNI over a unified framework like AsterixDB[2].

## 7. ACKNOWLEDGMENTS

This research was supported by the Intelligence Advanced Research Projects Activity contract number 2014-14071000011. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA or the U.S. Government.

## 8. REFERENCES

- [1] Apache Spark. <http://spark.apache.org/>.
- [2] AsterixDB. <https://asterixdb.incubator.apache.org/>.
- [3] Elasticsearch. <https://www.elastic.co/products/elasticsearch>.
- [4] SeaweedFS. <https://github.com/chrislusf/seaweedfs>.
- [5] The Apache Cassandra Project. <http://cassandra.apache.org/>.
- [6] H. Bay, T. Tuytelaars, and L. V. Gool. Surf: Speeded up robust features. In *ECCV*, pages 404–417. Springer Berlin Heidelberg, 2006.
- [7] H. Becker, D. Iyer, M. Naaman, and L. Gravano. Identifying content for planned events across social media sites. In *ACM WSDM*, pages 533–542, 2012.
- [8] Z. Cheng, J. Caverlee, and K. Lee. You are where you tweet: a content-based approach to geo-locating twitter users. In *ACM CIKM*, pages 759–768, 2010.
- [9] R. Compton, D. Jurgens, and D. Allen. Geotagging one hundred million twitter accounts with total variation minimization. In *Big Data*, pages 393–401. IEEE, 2014.
- [10] M. De Choudhury, M. Feldman, S. Amer-Yahia, N. Golbandi, R. Lempel, and C. Yu. Automatic construction of travel itineraries using social breadcrumbs. In *ACM Conference on Hypertext and hypermedia*, pages 35–44, 2010.
- [11] T. Lappas, M. R. Vieira, D. Gunopulos, and V. J. Tsotras. On the spatiotemporal burstiness of terms. *Proceedings of the VLDB Endowment*, 5(9):836–847, 2012.
- [12] K. Lerman and R. Ghosh. Information contagion: An empirical study of the spread of news on digg and twitter social networks. *ICWSM*, 10:90–97, 2010.
- [13] J. Yang, Y.-G. Jiang, A. G. Hauptmann, and C.-W. Ngo. Evaluating bag-of-visual-words representations in scene classification. In *Workshop on multimedia information retrieval*, pages 197–206. ACM, 2007.

# PROX: Approximated Summarization of Data Provenance

Eleanor Ainy  
Tel Aviv University  
eleanora@mail.tau.ac.il

Pierre Bourhis  
CNRS CRISTAL UMR 9189  
pierre.bourhis@univ-  
lille1.fr

Susan B. Davidson  
University of Pennsylvania  
susan@cis.upenn.edu

Daniel Deutch  
Tel Aviv University  
danielde@post.tau.ac.il

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

## ABSTRACT

Many modern applications involve collecting large amounts of data from multiple sources, and then aggregating and manipulating it in intricate ways. The complexity of such applications, combined with the size of the collected data, makes it difficult to understand the application logic and how information was derived. *Data provenance* has been proven helpful in this respect in different contexts; however, maintaining and presenting the full and exact provenance may be infeasible, due to its size and complex structure. For that reason, we introduce the notion of approximated summarized provenance, where we seek a compact representation of the provenance at the possible cost of information loss. Based on this notion, we have developed PROX, a system for the management, presentation and use of data provenance for complex applications. We propose to demonstrate PROX in the context of a movies rating crowd-sourcing system, letting participants view provenance summarization and use it to gain insights on the application and its underlying data.

## 1. INTRODUCTION

Complex applications that collect, store and aggregate large-scale data, and interact with a large number of users, are commonly found in a wide range of domains. Notable examples are *crowd-sourcing applications* such as Wikipedia, social tagging systems for images, traffic information aggregators such as Waze, or recommendation sites such as TripAdvisor and IMDb. In the context of such applications, several questions arise relating to *how data was derived*. As a user, what is the basis for trusting the presented information? How do crowd contributions vary among the crowd members, based on their characteristics? If some contribution seems wrong, how does the information change if we discard it? These questions are of fundamental importance for better understanding the application and its results.

At its core, the answer to these questions is based on the *provenance* of the collected data and resulting information, that is, *who* provided the information, in *what* context and *how* the information was manipulated. As shown in e.g. [10, 4], provenance is much more powerful than simply a log of the application execution. In particular, the algebraic model of provenance (based on semirings) has been shown to allow to correlate input data with output data; to track important details of the computational process that took place; and to further ([8]) *provision* the computation result with respect to hypothetical scenarios, namely to observe changes to the result based on changes to the input (without actually re-running the process). Detailed tracking of provenance was thus proved to be a suitable (and necessary) vehicle for the applications that we have mentioned above.

Consider a crowd-sourcing application for movie reviews. The number of movies may be quite large and so is the number of reviewers for every movie; the *aggregated score* for the movie is computed by combining the scores of multiple users, possibly accounting for their previous reviews and for their preferences. These features and the way in which they are used in the computation should all be reflected in the provenance. In turn, provenance may be presented to *explain* results (computed ranking of movies), or to *provision* them (e.g. “how would the average movie rating change if we ignore ratings by some (group of) users?”).

However, the complexity of processing and the large scale of data also mean that detailed semiring provenance information is extremely intricate; and so presenting it in full, as an explanation to the computation, would be extremely difficult to understand. To this end, we present PROX, a system that provides approximated summarization of provenance information for complex applications. The summarization is based in part on the semantics of the underlying data (such as gender, age or occupation of users), where annotations of “similar” data items are intuitively more amenable to be grouped together. But importantly, it is also geared towards the intended use of provenance (namely explanation and/or provisioning): we define a distance function between provenance expressions that is based on the intended use, and optimizing this distance (while obtaining small expressions) is an important consideration guiding the summarization.

**Demonstration.** We will demonstrate the system in the context of a movies recommendation website called *MovieLens* [1]. We will show that while full provenance is too large



to present, PROX allows for a summarized representation of the provenance that provides a concise explanation of the reviews, and further allows for approximate provisioning.

We next provide details on the technical background underlying PROX (Sec. 2), on the system implementation (Sec. 3), and on the demonstration scenario (Sec. 4).

## 2. TECHNICAL BACKGROUND

We (informally) introduce the main technical notions involved in the development of PROX, through examples. The full details can be found in [3].

**Semiring provenance model.** We first explain in general the provenance model described in [10, 5, 4]. We start by fixing a finite set  $X$  of *provenance annotations*, corresponding to the basic units of data manipulated by the application, and which can be thought of as abstract variables identifying the data. Depending on the application, these annotations may correspond to different tuples in a database, to different users, to different questions, etc. Given our set  $X$  of basic provenance annotations, the *provenance semiring* is the semiring of polynomials with natural coefficients, with indeterminates from the set  $X$ . It was shown in [10] to capture provenance for positive relational queries. Intuitively, the  $+$  operation corresponds to the *alternative use* of data (as in union and projection) and  $\cdot$  to the *joint use* of data (as in join); 1 annotates data that is present, and 0 annotates data that is absent. To capture aggregate queries, in [5], relations were further generalized by extending their data domain with *aggregated values*. In this extended framework, relations have provenance also as part of their values, rather than just in the tuple annotations. Such a value is a *formal sum*  $\sum_i t_i \otimes v_i$ , where  $v_i$  is the value of the aggregated attribute in the  $i^{\text{th}}$  tuple, while  $t_i$  is the provenance of that tuple. We can think of  $\otimes$  as an operation that pairs values (from a monoid  $M$ ) with provenance annotations. Each such pair is called a *tensor*. The formal sum, presented by the  $\oplus$  operation is used to capture the aggregation function.

**EXAMPLE 2.1.** Consider a crowdsourcing application, similar to IMDB, that allows users to rate different movies and aggregates their ratings. A possible provenance expression for the movie “Pretty Woman”, may e.g. be  $\mathbf{P}_1 = \text{UID}_1 \otimes (5, 1)$  where  $\text{UID}_1$  is a user id, and as aggregation we use a monoid of pairs to capture the aggregated rating (MAX rating with value 5 here) and how many users contributed to this aggregated value (1 here). Multiple reviews (say, for “Free Willy”) can be combined using the formal sum operation, e.g.  $\mathbf{P}_2 = \text{UID}_2 \otimes (1, 1) \oplus \text{UID}_3 \otimes (3, 1) \oplus \text{UID}_4 \otimes (5, 1)$ . The  $\oplus$  operation is given a “concrete semantics” depending on the aggregation function used to aggregate the ratings (e.g. SUM, MAX or AVG<sup>1</sup>).

**Valuations and provisioning.** An important use of semiring provenance is for provisioning, i.e. examining changes to the application’s execution that are the result of some hypothetical modifications to the data. Examples include “how would the movie ratings change if we ignore some reviews (suspected as spam)?” Provenance expressions enable this using *truth valuations* applied to annotations. Intuitively,

<sup>1</sup>These correspond formally to a choice of operation for the aggregation monoid

specifying that  $\text{UID}_1$  is a spammer corresponds to mapping it to *false* (and that  $\text{UID}_1$  is reliable to mapping it to *true*), and recomputing the derived value w.r.t this valuation. Such valuation can again be extended in the standard way to a valuation  $V : \mathbb{N}[X] \mapsto \{\text{true}, \text{false}\}$ .

**Summarization through mappings.** Full description of the provenance may be extremely long and convoluted, and so instead we would like to summarize the provenance expression. We formalize such summarization through the notion of mappings. Let  $X$  be a domain of annotations (for the  $\mathbb{N}[X]$  semiring) and  $X'$  be a domain of annotation “summaries”. Typically, we expect that  $|X'| \ll |X|$ . We then define a mapping  $h : X \mapsto X'$  which maps each annotation to a corresponding “summary”. Abusing notation, this extends naturally to a homomorphism  $h : \mathbb{N}[X] \mapsto \mathbb{N}[X]'$  (i.e. define  $h(x + y) = h(x) + h(y)$ ,  $h(x \cdot y) = h(x) \cdot h(y)$ ) and further extends to  $\mathbb{N}[X]' \otimes M$  by the standard construction  $h(k \otimes m) = h(k) \otimes m$ . Essentially, to apply  $h$  to a provenance expression  $p$  (we denote the result by  $h(p)$ ), each occurrence of  $x \in X$  in  $p$  is replaced by  $h(x)$ . The mapped expression is a “summary” of the real provenance, in the sense that we lose track of some exact annotations and summarize the provenance using the “abstract” annotations in  $X'$ .

**EXAMPLE 2.2.** We may map user annotations to annotation summaries that intuitively reflect values of attributes of the corresponding users. Then if we map  $\text{UID}_3$  and  $\text{UID}_4$  to an “annotation summary” called “Female”<sup>2</sup>, we obtain (by applying congruences in the tensor structure and the use of max as aggregate function), an expression describing a maximum female score of 5 collected from two users):

$$\mathbf{P}'_2 = \text{UID}_2 \otimes (1, 1) \oplus \text{Female} \otimes (5, 2)$$

Another possible summary may e.g. be the result of mapping annotations  $\text{UID}_2$  and  $\text{UID}_3$  to the annotation “Student”:

$$\mathbf{P}''_2 = \text{Student} \otimes (3, 2) \oplus \text{UID}_4 \otimes (5, 1)$$

Both of these mappings do not concern the provenance expression  $\mathbf{P}_1$  which stays intact.

In the example, we used two possible mappings  $h$  that combine reviews based on gender or occupation. In general there may be many possible mappings and the challenge is, given a provenance expression  $p$ , to (a) define what a good mapping  $h$  is (correspondingly, what is a good summary  $h(p)$ ), and (b) find such good  $h$ .

**Quantifying Summary Quality.** Several, possibly competing, considerations need to be combined.

**Provenance size.** Since the goal of summarization is to reduce the provenance size, it is natural to use the size of the summary, the number of annotations it consists of after simplifications, as a measure of its quality.

**Semantic Constraints.** The obtained summary may be of little use if it is constructed by identifying multiple unrelated annotations; it is thus natural to impose constraints on which annotations may be grouped together. One simple example of such a constraint is to allow two annotations  $x, x' \in X$  to be mapped to the same annotation in  $X'$  (or to 0 or 1) only if they annotate tuples in the *same input table*, meaning that they belong to the same domain. We further allow user-defined constraints based on equality of values of

<sup>2</sup>We later describe which mappings are possible and which are preferable to ours.

these annotated tuples in user-selected attributes, such as occupation or gender in the above examples.

**Distance.** Depending on the *intended use* of the provenance expression, we may *quantify the distance* between the original and summarized expression. As an example, consider a distance function designed to use provenance for *provisioning in presence of spammers*. For that we use again the notion of valuations, and consider as input to the problem a subset  $\mathcal{V}_X$  of all possible valuations w.r.t. the original provenance. Intuitively  $\mathcal{V}_X$  reflects possible scenarios that are of interest to the user. A central issue is how we transform a valuation in  $\mathcal{V}_X$ , on the original annotations to one in  $\mathcal{V}_{X'}$ , on the annotation summaries. We propose that this will be given by a combiner function  $\phi$ , that sets a boolean value to  $x' \in X'$  based on the truth values assigned to  $x$  annotations that were mapped to it. E.g.  $\phi$  may be a disjunction of these values, then intuitively an annotation summary is cancelled only if all of the annotations it summarizes are cancelled.

We next define the distance between a provenance expression  $p$  and its summary  $h(p)$  as an average over all truth valuations, of some property of  $p$ ,  $h(p)$ , and the valuation. This property is based on yet another function we call VAL-FUNC, whose choice depends on the intended provenance use. For provisioning, we may e.g. use the absolute difference between the two expressions values under the valuation or, alternatively, a function whose value is 0 if the two expressions agree under the valuation, and 1 otherwise (so the overall distance is the fraction of disagreeing valuations). Similarly, when dealing with multiple expressions (such as one for each movie) we need a function to combine the VAL-FUNC values; here a natural choice is Euclidean distance.

**EXAMPLE 2.3.** *To simplify the example we assume that the scenarios include at most a single spammer. So the class of valuations consists of those assigning 0 to some single user annotation, and 1 to all others. Observe that  $P_2''$  is at distance 0 from  $P_2$  with respect to this class of valuations: all these valuations yield the same value with respect to the two provenance expressions (if  $UID_4$  is mapped to true then the aggregated MAX value is 5 regardless of other truth values, and otherwise both  $UID_2$  and  $UID_3$  are mapped to true and so is Student). In contrast,  $P_2'$  differs from  $P_2$  for the valuation that maps  $UID_4$  to false and the rest to true.*

**Computing Summarizations.** We can show that computing an optimal summarization is  $\#P$ -hard, since even computing the distance (even under highly limiting restrictions) is already  $\#P$ -hard. On the other hand, we have implemented an *absolute approximation* algorithm for computing the distance between two such provenance expressions, based on sampling the possible valuations. This leads to a simple greedy algorithm. The details of the algorithm are omitted for lack of space and can be found in [3].

**Related Work.** Provenance models have been extensively studied in multiple lines of research such as provenance for database transformations (see [6]), for workflows (see [7]), for the web [2], for data mining applications [9], and many others, but typically full and exact provenance is presented. Provenance views have been proposed in context of workflows (see e.g. [7]), but the summarization obtained through these views is based on a notion of granularity levels, and is lossless rather than approximate. A notion of approximate

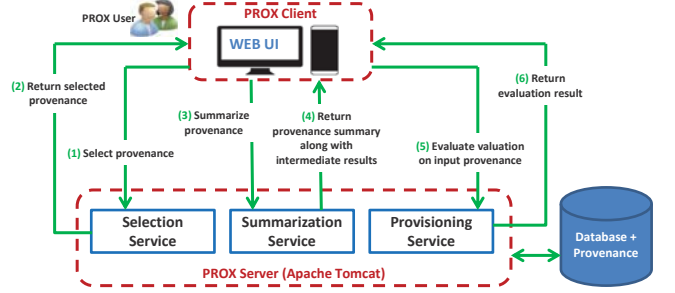


Figure 1: System Architecture

provenance was proposed in [11], and somewhat resembles ours, but is limited to UCQs (and in particular allows no aggregates), geared towards probabilistic computation, and does not account for semantic constraints. Our notion of mapping to summarized annotations is also reminiscent of clustering, however the function that we optimize is one that depends on the provenance expression itself and its intended uses, which leads to different design choices and results.

### 3. SYSTEM IMPLEMENTATION

**PROX Architecture.** PROX server-side is implemented in Java and its client-side is implemented in Angular JS. This web application is deployed to Apache Tomcat server on a Windows 7 machine. The system architecture is depicted in Figure 1. The server is comprised of three major services: a selection service that allows simple restriction of the provenance according to user-defined selection criteria, the summarization service that summarizes the selected provenance; and a provisioning service that allows to use the summarized provenance for exploration of hypothetical scenarios.

**PROX Web UI.** We developed a web UI which contains three views. The selection view allows the user to choose movies, whose provenance she would like to observe, according to title or genre and year (as shown in Figures 2a and 2b respectively). The summarization view presented in Figure 2c shows the selected provenance and allows the user to configure parameters for the summarization algorithm. The third view presents the summary in two views shown in Figures 2d and 2e: the expression view that shows the summary in its polynomial form, as exemplified throughout this paper and the groups view that shows the groups of users that the algorithm chose to map together. For instance, for the Female group in the figure, we can see the group size (9), its aggregated (MAX) rating (AGG:4), its users, the movies they rated and their aggregated ratings. Also, on hover on group user or movie their meta data is displayed. Using this last view, the user can choose a valuation to evaluate on the current provenance by selecting annotations or attributes to cancel (assign to *false*), as shown in Figure 2f. Using the left and right arrows, the user can also view and provision the algorithm’s intermediate results.

### 4. DEMONSTRATION SCENARIO

We will demonstrate the usefulness of PROX in the context of a movies review system. We will use a real-life movies data set taken from [1]. The first example will demonstrate how PROX can be used for provisioning. We will first select provenance by title, e.g. the movie “Free Willy”. Before

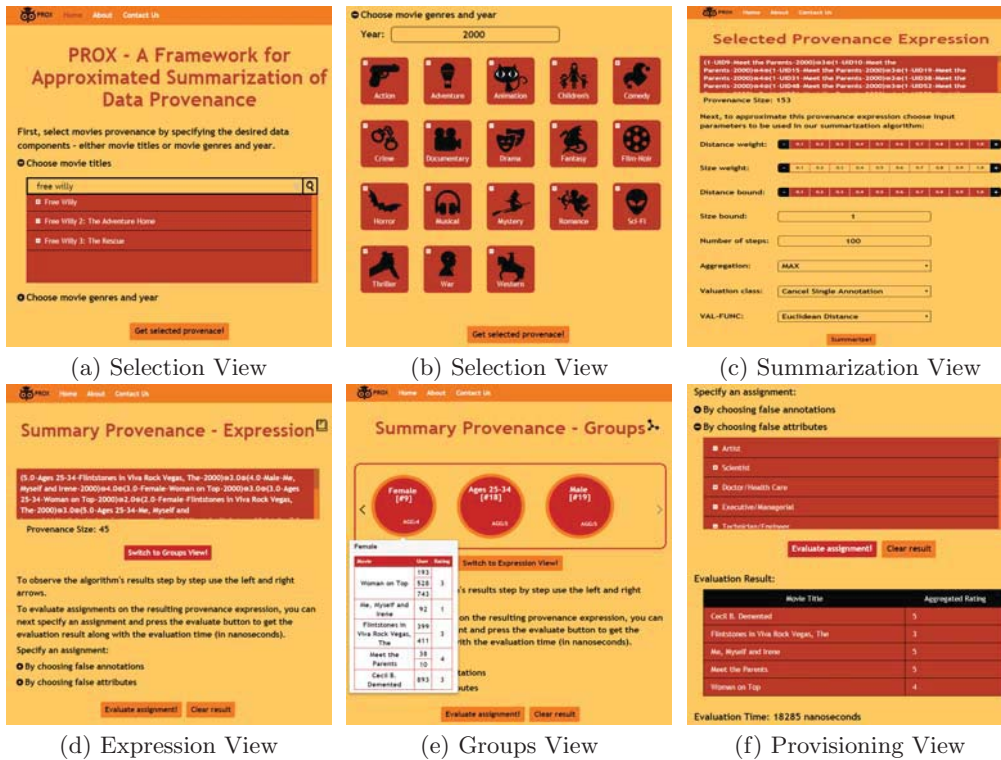


Figure 2: PROX Web UI

we compute the summary, we will show the different parameters for the summarization algorithm. We will use the default values e.g. MAX for aggregation and will limit the number of steps to 1 for this example. Using the groups view, we will show the user the two annotations that the algorithm chose to map to an annotation summary. For the same reasons discussed in Example 2.3, we expect that the algorithm would prefer to first map the annotations of users that did not give the movie the MAX rating and we will show that this is indeed the case. To end this example, we will provision the result, by choosing a valuation that cancels the two annotations. We expect that the result would be the same as if we applied the valuation on the original expression. To prove this, using the left arrow for navigating back, we will evaluate the valuation on the original expression as well. By summarizing the original provenance, we are able to provision the result by evaluating valuations on the summary, which is more efficient.

The second example will demonstrate another important provenance use which is presentation. For this example, we will choose a large provenance expression, e.g. provenance of “Comedy” movies released in the year 2000. We will summarize it using Average aggregation and a large number of steps. We will next show the groups of annotations along with their meta data and then switch to the expression view and compare its size to the original expression size which is much greater. Finally, we will let a volunteer user select her own provenance, summarize and provision the result.

## 5. ACKNOWLEDGMENTS

This work has been partially funded by the European Research Council under the FP7, ERC grant MoDaS, agreement 291071, by the Broadcom Foundation and Tel Aviv University Authentication Initiative, by the Israeli Science

Foundation (Grant No. 1636/13), by the National Science Foundation Institute (NSF), Information and Intelligent Systems (IIS) Division (Grant No. 1302212) and by the French National Research Agency (ANR), Aggreg project.

## 6. REFERENCES

- [1] MovieLens site. <https://movielens.org/>.
- [2] Provenance working group. <http://www.w3.org/2011/prov/>.
- [3] E. Ainy, P. Bourhis, S. B. Davidson, D. Deutch, and T. Milo. Approximated summarization of data provenance. In *CIKM*, 2015.
- [4] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *PVLDB*, 2012.
- [5] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *PODS*, 2011.
- [6] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.
- [7] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, pages 1345–1350, 2008.
- [8] D. Deutch, Y. Moskovitch, and V. Tannen. A provenance framework for data-dependent process analysis. *PVLDB*, 7(6):457–468, 2014.
- [9] B. Glavic, J. Siddique, P. Andritsos, and R. J. Miller. Provenance for Data Mining. In *Theory and Practice of Provenance (TAPP)*, 2013.
- [10] T. J. Green, G. Karvourarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.
- [11] C. Ré and D. Suciu. Approximate lineage for probabilistic databases. *PVLDB*, 1(1):797–808, 2008.

# PAW: A Platform for Analytics Workflows

Maxim Filatov  
 University of Geneva  
 maxim.filatov@unige.ch

Verena Kantere  
 University of Geneva  
 verena.kantere@unige.ch

## ABSTRACT

Big Data analytics in science and industry are performed on a range of heterogeneous data stores, both traditional and modern, and on a diversity of query engines. Workflows are difficult to design and implement since they span a variety of systems. To reduce development time and processing costs, automation is needed. We present PAW, a platform to manage analytics workflows. PAW enables workflow design, execution, analysis and optimization with respect to time efficiency, over multiple execution engines, namely a DBMS, a MapReduce engine, and an orchestration engine. This configuration is emerging as a common paradigm used to combine analysis of unstructured data with analysis of structured data (e.g., NoSQL plus SQL). The demonstration of PAW focuses on the usability of the platform by users with various expertise, the automation of the analysis and optimization of execution, as well as the effect of optimization on workflow execution. The demonstration scenarios are based on synthetic and real workflows on real data.

## 1. INTRODUCTION

Enterprises today employ a variety of data repositories and processing engines to meet their needs for analytics. Analytics workflows are becoming longer and more complex. Currently, analytics workflows are designed and implemented manually. This is time-consuming and labor-intensive. To address this, we demonstrate a platform to automate part of this process.

Workflow management is not a new topic [17]. However workflow optimisation is a relatively new field of research, but there are already some promising results.

Commercial Extract-Transform-Load (ETL) tools (e.g. [5], [10]) provide little support for automatic optimization. They provide hooks for the ETL designer to specify for example which flows may run in parallel or where to partition flows for pipeline parallelism. Some ETL engines such as PowerCenter [5] support PushDown optimization, which pushes operators that can be expressed in SQL from the ETL flow

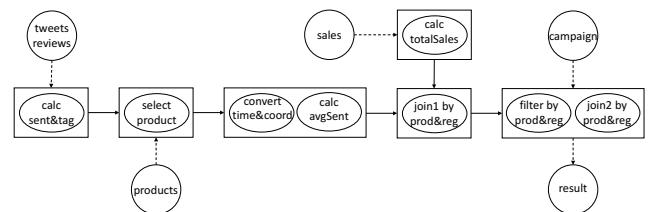


Figure 1: Workflow for a product marketing campaign

down to the source or target database engine. The rest of the transformations are executed in the data integration server. The challenge of optimizing the entire workflow remains.

Towards this direction, HFMS [13] performs optimization and execution across multiple engines. Work related to HFMS [14] focuses on optimizing flows for several objectives: performance, fault-tolerance and freshness over multiple execution engines. HFMS uses many optimization strategies, such as parallelization, recovery points, function shipping, data shipping, decomposition, etc. Complementary to the above, our work focuses on the automation of the total process of workflow manipulation, from the creation till the execution of a workflow. Furthermore, our work focuses on the challenge of enabling users with various levels of data management expertise to create a workflow for the same application logic.

In this paper, we demonstrate our work through PAW, a platform for the design, analysis and execution of analytics workflows. PAW implements a novel workflow language [7, 8] that allows the design of a workflow that spans multiple engines and data stores by either giving specific details on execution semantics of tasks and data stores or leaving the platform to determine the execution semantics and data stores, through an automated workflow analysis phase. Then, the workflow goes through an automated optimization phase, before being sent for execution. PAW is part of the ASAP project [1], which develops scalable solutions for complex analytical tasks over multi-engine environments.

In the following, Sections 2 and 3 give an overview of the workflow model and the platform architecture, respectively. Section 4 summarizes the functionalities of the platform and Section 5 describes the proposed demonstration.

## 2. WORKFLOW MODEL

PAW implements a novel workflow model [7, 8]. The workflows are directed, acyclic graphs (DAGs). The vertices represent data processing and the edges represent the flow of data. Data processing is computation or modification

©2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

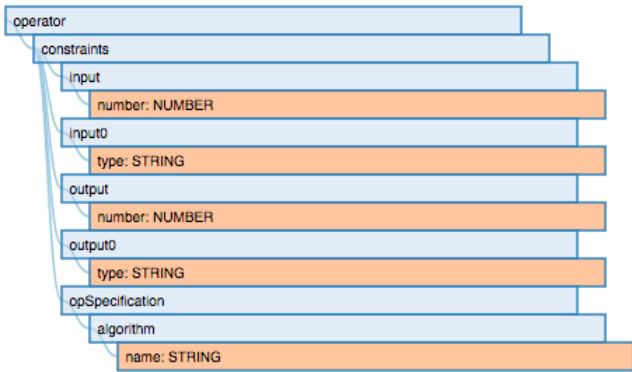


Figure 2: The generic metadata tree for operator

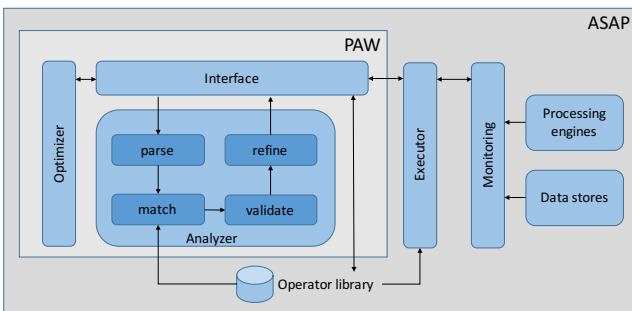


Figure 3: The architecture of PAW

of data. Each vertex in a workflow represents *one or more* tasks of data processing. Each task is a set of *inputs*, *outputs* and an *operator*. Tasks may share or not inputs, but they do not share operators and outputs. The inputs and outputs of the tasks of a vertex can be related to incoming and outgoing edges of this vertex, but they do not identify with edges: inputs and outputs represent consumption and production of data, respectively, and edges represent the flow of data. Figure 1 displays a workflow about a product marketing campaign. It combines sales data with sentiments about the product, gleaned from tweets crawled from the Web.

Data and operators can be either abstract or materialized. Abstract are the operators and datasets that are described partially or at a high level by the user, when composing the workflow, whereas materialized are the actual operator implementations and existing datasets, either provided by the user or residing in a repository. Both data and operators need to be accompanied by a set of metadata, i.e., properties that describe them. Such properties include input data types and parameters of operators, location of data objects or operator invocation scripts, data schemas, implementation details, engines etc. These metadata are used to:

- Match abstract operators to materialized ones.
- Check the usage of a dataset as input for an operator. If the dataset does not match the operator's input, its metadata can be also used to check for appropriate transform/move operators that can be applied.
- Provide optimization parameters, e.g. profiling of input/output.
- Provide execution parameters like a file path or arguments for the execution of the operator script.

The internal representation of a workflow is in the Tree-

metadata language, which captures structural information, operator properties (e.g., type, data schemas, statistics, engine and implementation details, physical characteristics like memory budget), and so on. The metadata tree is user extensible. Figure 2 shows the generic metadata tree for an operator. To allow for extensibility, the first levels of the metadata tree are predefined. Users can add their ad-hoc subtrees to define their custom data or operators. Moreover, some fields (like the ones related operators and data) are compulsory, while the rest are optional and user-defined. Materialized data and operators need to have all their compulsory fields filled in with information. Abstract data and operators do not adhere to this rule.

### 3. ARCHITECTURE & IMPLEMENTATION

PAW is part of a larger system, called Adaptable Scalable Analytics Platform (ASAP) [1], but it can also stand as an independent tool for workflow management and optimization. Other ASAP components include execution, monitoring, visualization of results, online adaptation, etc. PAW presents a unified interface for users to create, modify, analyze, optimize and execute analytics workflows over a diverse collection of data stores and processing engines. Figure 3 depicts the architecture of PAW, as well as its interaction with the rest of ASAP. The components of PAW communicate using the internal workflow representation and are:

- **Operator library.** This library shows operator implementations imported from the ASAP library. The operators are classified as, either analytics operators, which perform the core analytics jobs over the data, or the associative operators, which serve as 'glue' between different engines and perform move and transformation operations.
- **Interface.** The interface enables users to interactively create and/or modify a workflow.
- **Analyzer.** The analyzer parses the workflow, identifies operators and data stores and maps them to the library of operators, generates metadata of edges, finds edges where the data conversion should be applied and adds the appropriate conversions.
- **Optimizer.** The optimizer generates a functionally equivalent workflow, optimized for performance objective.
- **Executor.** The executor receives workflows from the optimizer and schedules them for execution. When a workflow is ready for execution, it dispatches the workflow to the engines and monitors its execution.

Code generation and the executor is implemented in Java. The interface is a web application in Jade [6] and CoffeeScript [2], and Grunt [3] compiles it in HTML and JavaScript, respectively. The interface communicates with other modules using Nginx web server [9] and PHP-FPM [11]. The analysis and optimisation modules are implemented in Python.

### 4. FUNCTIONALITY OF PAW

This section describes the PAW functionalities and discusses relevant aspects of the components.

#### 4.1 Management of operators

Each operator can have an abstract definition and several implementations, i.e. one or more implementations per engine. For example, a 'join' of two inputs, has an abstract definition, and can be implemented for a relational DBMS

Operator	Blocking	Non-blocking	Restrictive
Filter		x	x
Calc		x	
Filter Join	x		
groupBy Sort	x		
PeakDetection	x		
TF-idf	x		
k-Means	x		

Table 1: Categorization of operators

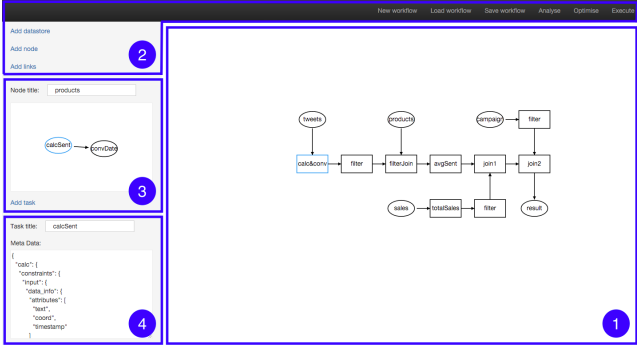


Figure 4: Interface of PAW

and a NoSQL database. An operator that performs a simple operation or a complex algorithm computation needs to have a tailored implementation for every engine on which it is going to be executed. An operator definition includes restrictions on the type and number of inputs and specifies the number and type of outputs.

Operators are categorized as:

- **Blocking operators** require knowledge of the whole data, e.g., a grouping operator or an operator *join* or *sort*.
- **Non-blocking operators** process each tuple separately, e.g., operators *filter* or *calc*<sup>1</sup>.
- **Restrictive operators** output a smaller data volume than the incoming data volume, e.g. *filter*.

Defined and implemented operators form a library from which a user can select operators to describe tasks. Table 1 shows operators from the library and their categorization.

Users can register their own operators, provide respective implementations and define optional attributes of a new operator. These attributes include functions to compute cardinality and processing cost, and characteristics of the operator. In most cases, the operator developer or provider does not disclose a cost formula for the operator. Then, PAW can use the ASAP profiling process for operators using micro-benchmarks. As an optional step, PAW allows users to run their workflows with a data sample and uses the obtained statistics to fine-tune our cost models before workflow optimization.

## 4.2 Design of a workflow

A workflow is created in the PAW interface, which consists of several areas (Figure 4) that perform the following:

- Display the workflow (Area 1).
- Design a new workflow adding vertices and edges, save and load it (Area 2).
- Perform workflow analysis or optimization (Area 2).
- Add tasks from a library or create new ones (Area 3). A task from the library is accompanied by a set of metadata.

<sup>1</sup>*calc* is a generic operator that can be customized for a specific numeric calculation

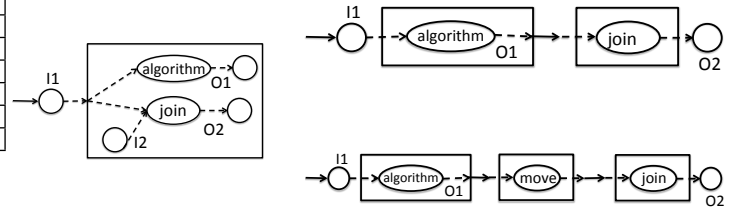


Figure 5: Analysed workflows for a multi-task vertex

A task created from scratch has a metadata tree with predefined first levels; users can add their ad-hoc subtrees to define their custom data or operators.

- Display metadata of the selected task (Area 4).

## 4.3 Analysis of a workflow

The workflow model alleviates from the user the burden of determining any or some execution semantics of the application logic. The execution semantics of the workflow includes the execution of tasks in vertices and the execution of input-output dependencies of edges. The determination of the execution semantics of vertices and edges leads to an execution plan of the workflow. We refer to this plan as the *analysed* workflow. The latter is actually an enhancement of the initial workflow with more vertices and substitution of vertices and/or edges in the initial workflow with others.

PAW analyses a workflow in several steps:

1. Parses the workflow.
2. Categorizes operators (see Section 4.1).
3. Validates consistency. A workflow is checked for cycles and correspondence of metadata of adjacent vertices. Cycles cannot be resolved, the analysis stops and returns a list of errors. If possible, metadata mismatches are solved by adding associative tasks in Step 6.
4. Generates metadata of edges, as a join of input and output metadata of source and target vertices, respectively.
5. Splits multi-task vertices to several single-task vertices: A vertex that corresponds to multiple tasks is replaced with an *associative subgraph* that contains a set of new vertices that correspond to these tasks. New vertices may correspond 1-1 to tasks, but it can be the case that two or more vertices correspond to the same task (task replication).
6. Augments the workflow with associative tasks that are converting data flow: buffers and format conversions.

Users may describe the same application logic by creating workflows with different levels of detail concerning the execution planning of this logic. The analysis phase determines missing execution semantics. Figure 5 shows an example: A user defines a vertex with two tasks, algorithmic processing on some data, and a join of these with some other data. The user is not interested or does not know the execution details of this complex task. This representation depicts that the two tasks should be executed together, after the tasks of the vertices on which this vertex depends, and before the tasks of vertices that depend on this vertex. Another user, represents the same tasks with two connected vertices, dictating that the join should be executed on the data processed first by the algorithm. A third user dictates even more detail in the execution plan, by adding one more vertex that includes a task that moves the data, e.g. from one disk to another.

## 4.4 Optimization of a workflow

After the analysis phase, a workflow is optimized for performance. The optimization uses the following operations:

- **Swap.** The *swap* operation applies to a pair of vertices,  $v_1$  and  $v_2$ , which occur in adjacent positions in an workflow graph  $G$ , and produces a new graph  $G'$  in which the positions of  $v_1$  and  $v_2$  have been interchanged. The goal of *swap* is to change the execution order of tasks.
- **Merge.** The *merge* operation takes as input two vertices and produces one new vertex that includes the tasks of both initial vertices. The latter may either be connected with an edge, i.e. there is some task dependency(ies), or not. The goal of *merge* is to allow for a united optimisation of the tasks included in the two vertices, e.g. joint micro-optimization on an execution engine.
- **Split.** The *split* operation takes as input one vertex and produces two new vertices that, together, include all the tasks included in the initial vertex. The new vertices may or may not be connected. The goal of *split* is to lead to separate optimisation of subgroups of the tasks.

The Optimizer applies to the analysed workflow a series of the above operations, each producing a functionally equivalent workflow with possibly different cost. The goal is to find an optimal workflow in the state space of equivalent workflows (for the optimization algorithm see [16]). To improve search, the space is pruned employing heuristics:

- **H1:** Move restrictive operators to the root of the workflow to reduce the data volume, e.g., rather than *extract*  $\rightarrow$  *function*  $\rightarrow$  *filter* do *extract*  $\rightarrow$  *filter*  $\rightarrow$  *function*.
- **H2:** Place non-blocking operators together and separately from blocking operators, require knowledge of the whole dataset, e.g., rather than *filter*  $\rightarrow$  *sort*  $\rightarrow$  *function*  $\rightarrow$  *group* do *filter*  $\rightarrow$  *function*  $\rightarrow$  *sort*  $\rightarrow$  *group*.
- **H3:** Parallelize adjacent non-blocking operators so that they can be executed concurrently on separate processors, e.g., split *filter1*  $\rightarrow$  *filter2* to two new parallel paths. Parallelized workflow parts should be chosen such that their latency is approximately equal.

## 5. DEMONSTRATION

In the following, we describe the demonstration of PAW.

**System setup.** PAW is demonstrated on a cluster, with the following configuration: The cluster consists of 4 server-grade physical nodes. Each one of those is equipped with a 3rd generation i5 CPU (@ 2.90 GHz) and 16GB of physical memory and an array of two HDDs on RAID-0. The operating system is Debian 6 (squeeze) Linux. For the time being, three software platforms are running: Hadoop [4], Spark [15] and Weka [18]. The distribution of Hadoop is CDH 4.6.0 and the version of Spark is 1.4.1.

**Workloads.** The demonstration uses synthetic and real workflows on real data. The synthetic workflows are constructed based on ETL benchmarking [12]. Real workflows and data come from the two use cases of ASAP [1] and belong to the domains of telecommunications and web analytics. The telecommunication use case involves processing anonymised Call Detail Records (CDR) data for Rome, from 01/06/2015 until 30/06/2015 and stored in CSV format. For the computation on graph-structured data workflows are implemented in Apache Spark.

The web analytics use case involves anonymization of web

content (WARC files) stored in Elasticsearch. The workflows are implemented in Spark and run over varying data set sizes ranging from 1 million to 1 billion rows. There are two types of workflows: one models entity recognition/disambiguation and k-means, and another models continuous processing of incoming data, e.g., subscription/notification at scale.

**Demonstration scenarios.** The demonstration aims to exhibit the functionalities of PAW, focusing on the following aspects: (a) the usability of the platform for workflow creation by users that have different expertise, (b) the effectiveness of the automated analysis of the execution of the workflow, and (c) the effectiveness of the automated optimization in workflow execution. The demonstration includes interesting scenarios for all (a,b,c) to be shown to the audience, but also interactive scenarios, especially for (a) and (b), which allow the audience to experience the functionalities of PAW. The interactive scenarios enable the participant to create workflows from scratch or change existing ones, watch the automated management of the workflow as well as review the internals of the platform, e.g. internal workflow representation. Concerning (a) and (b), the scenarios exhibit how the same application logic can be expressed via workflow versions that have different level of detail of execution semantics, and how the analysis phase specifies missing execution semantics through already executed workflows and/or by giving alternative choices to the user. Concerning (c), the scenarios show how new operators are added, including cost functions, and how the latter may be tuned by running a workflow with sampling for statistics collection; finally, the scenarios show actual workflow execution and the optimization benefit and tradeoffs on different engines.

## Acknowledgments

This work has received funding from the European Union's 7th Framework Programme under grant agreement n<sup>o</sup> 619706.

## 6. REFERENCES

- [1] Asap. <http://www.asap-fp7.eu/>.
- [2] Coffeescript. <http://coffeescript.org/>.
- [3] Grunt - the javascript task runner. <http://gruntjs.com/>.
- [4] Apache hadoop. <http://hadoop.apache.org/>.
- [5] Informatica 'powercenter'. <http://www.informatica.com/products/powercenter/>.
- [6] Jade - template engine. <http://jade-lang.com/>.
- [7] V. Kantere and M. Filatov. A framework for big data analytics. In *C3S2E*, 2015.
- [8] V. Kantere and F. Maxim. Modelling processes of big data analytics. In *WISE (To Appear)*, 2015.
- [9] Nginx. <http://nginx.org/>.
- [10] Oracle warehouse builder 10g. <http://www.oracle.com/technology/products/warehouse/>.
- [11] Php-fpm (fastcgi process manager). <http://php-fpm.org/>.
- [12] A. Simitis, P. Vassiliadis, U. Dayal, A. Karagiannis, and V. Tziouva. Benchmarking ETL workflows. TPCTC, 2009.
- [13] A. Simitis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing analytic data flows for multiple execution engines. In *ACM SIGMOD*, 2012.
- [14] A. Simitis, K. Wilkinson, U. Dayal, and M. Hsu. Hfms: Managing the lifecycle and complexity of hybrid analytic data flows. In *ICDE*, 2013.
- [15] Apache spark. <https://spark.apache.org/>.
- [16] Technical report. [https://github.com/project-asap/workflow/blob/master/tech\\_report/report.pdf](https://github.com/project-asap/workflow/blob/master/tech_report/report.pdf).
- [17] W. M. P. Van Der Aalst and A. H. M. T. Hofstede. Yawl: Yet another workflow language. *Inf. Syst.*, 30(4), June 2005.
- [18] Weka. <http://weka.pentaho.com/>.

# StreamLoader: An Event-Driven ETL System for the On-line Processing of Heterogeneous Sensor Data

M. Mesiti, L. Ferrari, S. Valtolina,  
G. Licari, G. Galliani  
Dip. di Informatica, Università di Milano, Italy  
{mesiti,lferrari,valtolina}@di.unimi.it

M. S. Dao, K. Zetsu  
Universal Communication Research Institute  
NICT, Kyoto, Japan  
{dao.minhson,zetsu}@nict.go.jp

## ABSTRACT

ETL (Extraction-Transform-Load) tools, traditionally developed to operate offline on historical data for feeding Data-warehouses, need to be enhanced to deal with big and fresh data and be executed at network level during data streams acquisition. In this paper, we present StreamLoader, a Web application for the specification of conceptual ETL dataflows on heterogeneous sensor data that leverages the peculiarities of network configuration, data stream management, and specification and deployment of ETL operations in a programmable network. It can be used for feeding traditional/real-time data-warehouses or visual analytic tools.

## 1. MOTIVATION

Nowadays we are witnesses of the proliferations of different sensor devices able to produce heterogeneous types of data that can be profitably used for detecting, handling and advising people of the verification of events such as natural disasters (like flooding, storming, extreme temperatures etc.), traffic congestions (due to accidents, strikes, football matches), and social web interactions. Beside the physical sensors, able to detect data about physical phenomena (like, temperature, humidity, wind, rain, pressure, level of sea water), there is a proliferation of social sensors able to collect data from people (like, twitter data, traffic information, train or flight schedule). These data are characterized both from the temporal, spatial and thematic dimensions that can be exploited from for the identification of meaningful events in a given context.

Several challenges should be faced for handling sensors and their data especially in emergency situations. First, sensors (both physical and social) are located in different networks and made available by different institutes, agencies and NPOs. In this context, network configuration, sensor detection and discovery are difficult issues to be solved. Moreover, data produced by sensors are heterogeneous in structures (different types), in spatial and/or temporal granularities (e.g. temperature in a room versus temperatures in

a geographical area), in thematic (data about traffic jams vs data about pollutions). Therefore, there is the need of ETL (Extract-Transform-Load) operations that can be applied on data streams for their reconciliation. These operations should be applied during data acquisition and bound with reactive capabilities in order to properly identify the relevant streams when abnormal events occur and undertake the proper actions. Finally, the specification and actuation of the ETL operations should be efficiently performed on-line and on fresh and timely data in order to properly handling big real-time data streams. All these technical requirements should be addressed in graphical, user-friendly environments supporting the user in the design and execution of the operations.

Many systems have been proposed for configuring programmable networks ([4, 2, 9]), for data stream management (e.g. Niagara [14], TelegraphCQ [5], Borealis [1]), for the specification and actuation of ETL operations (e.g. [16, 10, 13]) and dataflow (e.g. Talend [www.talend.com](http://www.talend.com), Pentaho-kettle [www.pentaho.com](http://www.pentaho.com), CoverETL [www.cloveretl.com](http://www.cloveretl.com)), and for complex event processing (e.g. Apache S4 [15], Storm [12], StreamBase [www.tibco.com](http://www.tibco.com)). However, all of them are quite complex to use, seldom provide web GUIs for designing and monitoring dataflows and are not integrated in a single tool. This limits their use in the management of emergency situations.

In this paper we propose StreamLoader, a Web application for the specification of conceptual ETL dataflows on heterogeneous sensors to be applied during data stream acquisition on a programmable network. It exploits different technologies (AngularJS, Cytoscape, SparkJava) for providing the graphical environment. StreamLoader is equipped with an interactive environment that supports the user in charge of handling events to discover the sensors useful in a given situation, specify the adequate dataflow for extracting, filtering, integrating, (eventually) storing, and analyze the data coming from the identified sensors, optimize the schedule for the execution of the dataflow and visualize the results. By exploiting samples produced by the involved sensors, the user can easily debug the developed dataflow. Once the dataflow is consistent (i.e. it can be soundly activated at network level), the translation is automatically invoked. Then, the underlying network is configured and the processes associated with the ETL operators executed. At this point, logs of the execution of the dataflow components are directly shown in the interactive environment to provide statistics on the dataflow execution.

In the remainder, Section 2 presents the requirements we



started from. The main characteristics of the system and of the interactive environment are discussed in Section 3. Section 4 presents the features of the system we wish to demonstrate and discusses the system significance.

## 2. StreamLoader REQUIREMENTS

Starting from the idea to develop a tool that is easy to use for people without a computer science background, the following requirements have been posed at the basis of the development of our tool.

*Dynamic (and automatic) configuration of ETL dataflow on events.* Starting from several data streams, the user interface should offer the possibility to identify relevant sources of information depending on the verification of events. For example, suppose we have several sensors for detecting the humidity and temperature of a given area; apparent temperature represents the temperature that is perceived by humans and depends on both temperature and humidity. The computation and acquisition of the apparent temperature in a given area can be triggered when the temperature is greater than 24° C. Events can be used both for triggering or stopping the acquisition and elaboration of streams. Moreover, ETL dataflows should be generated on the fly depending on the needs, immediately actuated and its execution monitored in the same tool. The dataflow should become “live” and give execution feedbacks to the user.

*ETL operations for integrating heterogeneous streams.* The heterogeneity of the data flows requires the application of common operations developed in the context of data integration and data fusion in order to identify different representations of the same real world entities. For this purpose, the system should be equipped with transformation operations: (1) for changing the unit of measure (e.g. from yards to meters) or geographical coordinates (from one standard to another one); (2) for introducing virtual properties relying on the values assumed by other attributes (e.g. the apparent temperature discussed above); (3) for checking that data conform to given validation rules (e.g. dates conforming to given patterns). Moreover, operation for filtering data relying on different conditions should be included and for culling data belonging to a temporal interval or a geographical area according to a reducing factor. Finally, operations for aggregating and joining streams should be included for combining information coming from different streams.

*Discovery of sensor data sources.* The large amount of sensors with different levels of availability that can be monitored by the developed system imposes the adoption of different solutions for their discovery and management. First, sensors should be handled by means of a publish-subscribe system in order to handle the dynamicity with which they can join and leave the network. Then, sources of dataflows should be specified by means of the sensor and location characteristics. Finally, sensors can be organized according to different criteria (temporal/spatial, type/location) in order to facilitate the specification of dataflows.

*Isolation of data traffic based on the ETL dataflow.* Hard-coded configurations of network architectures and paths where data traffics are routed are not an easy task and prevent the possibility to adapt to new user requirements. Approaches based on declarative networking [4, 2, 9] have been recently proposed for the application of database query-

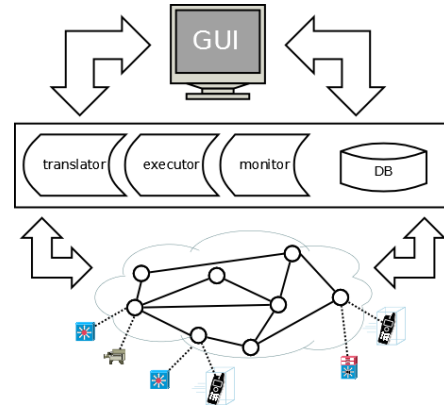


Figure 1: StreamLoader Architecture

language and processing techniques to the domain of networking. In [8] an extension of the declarative networking [4] approach has been proposed that consists of two components: declarative service networking (DSN) and network control protocol stacks (SCN). DSN provides a method to model and describe a high-level network of information services for an application, which includes service discovery, service monitoring, execution control, and service message exchanges. SCN aims at capturing application requirements and requesting appropriate configuration to the network platform more directly and effectively. The network control protocol stack interprets the DSN description and dynamically coordinates the network configurations, such as data flows, segmentations, and QoS parameters. The dataflow graphically described by the user should be then translated in DSN/SCN language to be actuated in the network and monitored.

## 3. StreamLoader OVERVIEW

*Architecture.* Figure 1 reports the architecture of the StreamLoader system. At the bottom there is a network. Each node of the network is in charge of managing a bunch of sensors and can execute the proposed ETL stream processing operations. Sensors are handled through a distributed publish-subscribe system [3]. Each time a sensor is published, its type, schema, and frequency of data generation are made available to subscribers.

Our Web environment is made available for the design and monitoring of dataflows. When a conceptual dataflow is realized, the **translator** module is in charge to translate it in DSN/SCN and execute it at network level. Processes are generated for each operation of the dataflow and executed on a network. The **executor** module coordinates their execution. For the execution, the sources are bound to specific sensors handled by the network nodes, and operations located on the machines that, depending on workload, apply the logic specified in the conceptual dataflow.

Logs of the activities are then collected by the **monitor** module and made available to the Web Interface to show statistics on the dataflow execution. Specifically, we are able to report on the Web Interface the number of tuples that each operation handle per second, the node that suffers because of high workload, which node is in charge of executing an operation and when the assignment changes.

Operation	Symbol	Meaning
Aggregation	$@_{op}^{t, \{s_1, \dots, s_n\}}(s)$	Every $t$ time intervals, aggregate $s$ on the attributes $\{a_1, \dots, a_n\}$ and apply the aggregation function $op \in \{\text{COUNT, AVG, SUM, MIN, MAX}\}$
Cull Time	$\gamma_r(s, \langle t_1, t_2 \rangle)$	Culling the tuples in the temporal interval $[t_1, t_2]$ by a reducing rate $r$
Cull Space	$\gamma_r(s, \langle coord_1, coord_2 \rangle)$	Culling the tuples that fall in the area delimited by $coord_1, coord_2$ by a reducing rate $r$
Filter	$\sigma(s, cond)$	Filter out tuples in $s$ that do not adhere to the condition $cond$
Join	$s_1 \bowtie_{pred}^t s_2$	Every $t$ time intervals, $s_1$ and $s_2$ are joined according to the join predicate $cond$
Transform	$\diamond_{trans} s$	The transformation function $trans$ is applied on the tuples in $s$
Trigger On	$\oplus^{ON, t}(s, \{s_1, \dots, s_n\}, cond)$	Every $t$ time intervals the condition $cond$ is checked on the tuples collected from $s$ . If the condition is verified, the streams of the sensors $\{s_1, \dots, s_n\}$ are activated
Trigger Off	$\oplus^{OFF, t}(s, \{s_1, \dots, s_n\}, cond)$	Every $t$ time intervals the condition $cond$ is checked on the tuples collected from $s$ . If the condition is verified, the streams of the sensors $\{s_1, \dots, s_n\}$ are de-activated
Virtual property	$\uplus_s(p, spec)$	A new attribute $p$ is added to the schema of $s$ according to the specification $spec$

Table 1: Stream Processing Operations

**Stream Processing Operations.** Sensors produce streams of tuples according to the multigranular space, time and thematic (STT) data model [7]. Relying on the concepts of temporal and spatial granularities, we exploit the concept of *event*, that is a value associated with a spatial object at a given time according to given thematics. Therefore, an event is a value represented at a given spatio-temporal granularity for which thematic information is added. Granularities are used for identifying correlations among data produced by different sensors and for imposing consistency constraints in the composition of sensor data produced by heterogeneous devices. We remark that whenever a sensor is not able to produce the spatio-temporal information of the produced data, this information is added by the Publish-Subscribe system that we adopt in our architecture.

Several operations have been developed for processing and combining the streams produced by the sensors in accordance with the requirements previously discussed at network level. Table 1 reports the principal operations concerning the application of filters, transformation, aggregation and composition, and event detection. Among the operations we point out those that are non-blocking (filter, cull-time/space, transform, virtual property) from those that are blocking (aggregation, trigger, join). The former are directly applied on each tuple when they are processed, whereas the others require the maintenance of a cache of tuples that are processed every  $t$  time intervals (e.g. 1 second, 2 minutes).

**Visual Interactive Environment.** By using a visual interface in Figure 2, users can drag-and-drop data-sources and apply the proposed operations on streams. In a window placed at the bottom of the canvas used for designing the data-flow, the user can see the schema of data that are processed by the operation, specify the conditions of each operation and visualize a data sample coming from each source. The user interface provides different checks in order to draw only dataflows that can be soundly translated in the DSN/SCN specification. We remark that data schema are not fixed but depend on the sensors. Finally, at the use phase, the dataflow developed at design time will be annotated with information coming from the SCN about the execution of the dataflow. In this way, the dataflow becomes “live” and the domain expert can monitor its execution.

**Scenario.** There are different sensors in the area of Osaka that produce data about the temperatures and levels of rains monitored in the current year. Moreover, tweets and traffic information from the same area in the current year

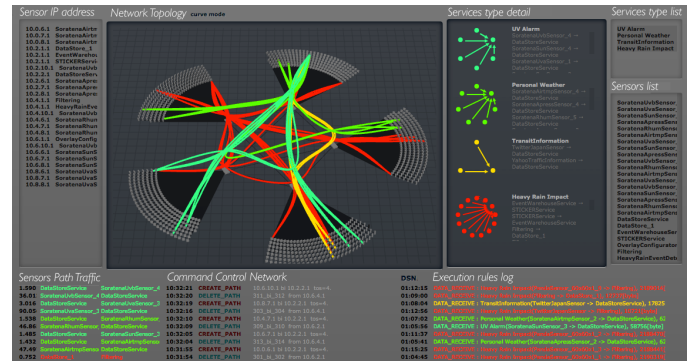


Figure 3: Monitoring the execution of the dataflow

can be acquired. Suppose, that there is interest in acquiring the data about torrential rain, tweets and traffic only when the temperature identified in the last hour is above 25° C. The dataflow reported in the canvas of Figure 2 realizes the proposed behavior. The acquired data can be stored in a data-warehouse or sent to a visualization tool in order to perform further analysis. Moreover, Figure 3 shows the flows of data that are monitored for this and other dataflows that are under control.

## 4. DEMO WALKTROUGH

In the demonstration of StreamLoader we will considering a network (established at NICT in Japan) in which different physical and social sensors related to the described scenario are connected and produce continuous data streams to be processed. By using this setup, we will show the capabilities of our tool in an interactive demo for the automatic configuration (and re-configuration) of a programmable network by means of our visual tool for *more efficient and interactive* analysis when transmitting data from sensors (sources) to the Event Data Warehouse (destination) [6]. The demo will consist of the following parts:

- P1 By exploiting the Web interface of our tool, users can create their own dataflows. Specifically, they will be able to identify the different sensors that are currently available in the network and select those on which they wish to specify ETL operations. Moreover, they will be able to apply different processing operations on such sources and check, step-by-step, their results on samples made available from the source.

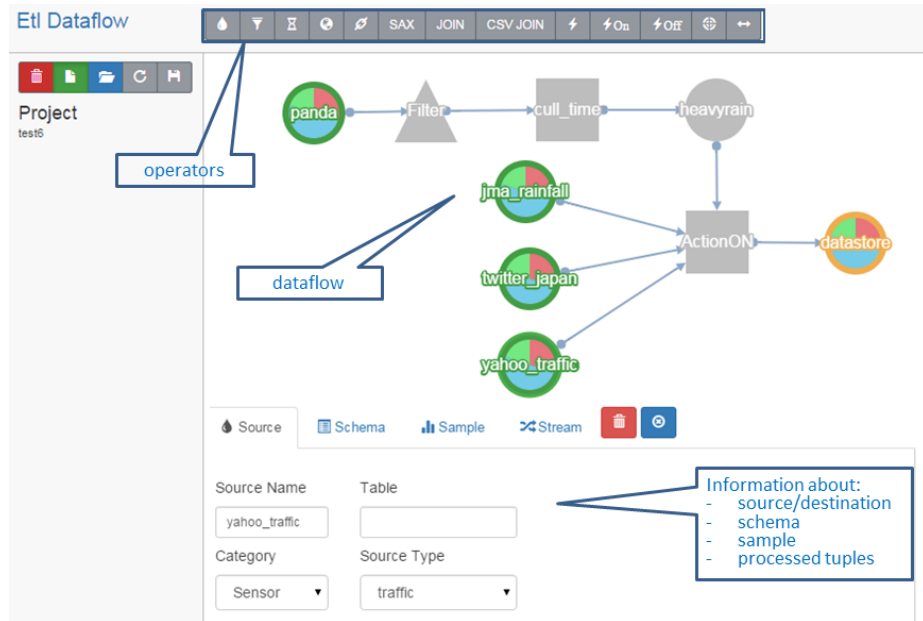


Figure 2: Main screen of the Web Application

P2 Once the dataflow is consistent, we will show its translation in the DSN/SCN language and deployment at network level. Then, we will monitor its execution by means of the tools presented in this paper. Finally, we will show how the data processed by means of the dataflow can be stored in the Event Data Warehouse [6] or visualized in the Sticker visualization tool [11]. Both tools have been developed by NICT.

P3 In the last part of the demo, we will show how it is easy to plug-and-play new sensors to the network and make them directly available to StreamLoader. We will also show how the system react when sensors or operators in the dataflow are modified on the fly. Finally, we will show statistics on the execution of the dataflow and on the performances of the network.

The demonstration will thus prove the flexibility of the developed system in the specification of dataflows to be executed at network level and actuated on the fly. All the ETL operations that have been considered can be applied on-line on fresh data arriving from sensors of different types. Different controls have been included in the dataflow specification in order to guarantee the sound translation and execution of the corresponding DSN/SCN specification.

The code of SCN and ETL dataflow editors are open source and can be freely downloaded from GitHub (<https://github.com/nict-isp>).

## 5. REFERENCES

- [1] D. J. Abadi, et al. The design of the Borealis Stream Processing Engine. In *CIDR*, 277–289, 2005.
- [2] P. Alvaro et al. BloomUnit: Declarative Testing for Distributed Programs. In *DBTest*, 2012.
- [3] R. Baldoni, et al. Distributed Event Routing in Publish/Subscribe Communication Systems: a Survey. *Middleware for Network Eccentric and Mobile Applications*, 219-244. Springer, 2009
- [4] B. T. Loo, W. Zhou. Declarative Networking. *Synthesis Lectures on Data Management*, Morgan & Claypool, 2012
- [5] S. Chandrasekaran, et al. TelegraphCQ: Continuous Dataflow Processing. In *SIGMOD*, page 668, 2003.
- [6] M.S. Dao, et al. A Real-Time Complex Event Discovery Platform for Cyber-Physical-Social Systems. In *Multimedia Retrieval Conf.*, 201-208, 2014.
- [7] M.S. Dao, et al. EventShop. From Heterogeneous Web Streams to Personalized Situation Detection and Control. In *ACM Web Science Conf.*, 105-108, 2012.
- [8] M. Dong, T. Kimata, and K. Zettsu. Service-Controlled Networking: Dynamic in-Network Data Fusion for Heterogeneous Sensor Networks. In *Reliable Distributed Systems Workshops*, 94–99, 2014.
- [9] D. Gay et al. The nesC language: A Holistic Approach to Networked Embedded Systems In *SIGPLAN*, 2003.
- [10] M. Gorawski and A. Gorawska. Research on the stream ETL process. In *BDAS*, 61–71, 2014.
- [11] K.S Kim, R. Lee and K. Zettsu. mTrend: Discovery of Topic Movements on Geo-Microblogging Messages. In *ACM SIGSPATIAL*, 529–532, 2011.
- [12] N. Marz. Storm: Distributed and Fault-Tolerant Realtime Computation, 2012.
- [13] M. Mesiti and S. Valtolina. Towards a User-Friendly Loading System for the Analysis of Big Data in the Internet of Things. In *COMPSACW*, 312–317, 2014.
- [14] J. F. Naughton, et al. The NIAGARA Internet Query System. *IEEE Data Eng. Bull.*, 24(2):27–33, 2001.
- [15] L. Neumeyer, et al. S4: Distributed Stream Computing Platform. In *ICDMW*, 170–177, 2010.
- [16] P. Vassiliadis, A. Simitis, and S. Skiadopoulos. Conceptual Modeling for ETL Processes. In *DOLAP*, 14–21, 2002.

# TINTIN: a Tool for INcremental INtegrity checking of Assertions in SQL Server

Xavier Oriol  
 Universitat Politècnica de Catalunya  
 xoriol@essi.upc.edu

Ernest Teniente  
 Universitat Politècnica de Catalunya  
 teniente@essi.upc.edu

Guillem Rull  
 Universitat de Barcelona  
 Barcelona, Spain  
 grull@ceipac.ub.edu

## ABSTRACT

We present TINTIN, a tool to perform efficient integrity checking of SQL assertions in SQL Server. TINTIN rewrites each assertion into a set of standard SQL queries that, given a set of insertions and deletions of tuples, allow to incrementally compute whether this update violates the assertion or not. If one of such queries returns a non empty answer, then the assertion is violated. Efficiency is achieved by evaluating only those data and those assertions that can actually be violated according to the update. TINTIN is aimed at two different purposes. First, to show the feasibility of our approach by implementing it on a commercial relational DBMS. Second, to illustrate that the efficiency we achieve is good enough for making assertions to be used in practice.

## Keywords

Integrity checking, SQL, Assertions

## 1. INTRODUCTION

In standard SQL, users can specify general constraints using the `CREATE ASSERTION` statement. The basic technique for writing assertions is to specify a query that selects those tuples that violate the desired condition. By including this query inside a `NOT EXISTS` clause, the assertion will specify that the query result must be empty. Thus, the assertion is violated if and only if the query result is not empty [2].

Assertions were initially defined in SQL-92 [1] and they serve as a means for expressing global integrity constraints not tied to a particular table, but ranging over several ones. They are sufficient for expressing most constraints since almost the full expressiveness of SQL can be used to define the query inside the `NOT EXISTS` clause. It is also well known that many integrity constraints can only be expressed via assertions since the other constructs provided by SQL are not powerful enough. Thus, assertions provide an elegant way to define general constraints in SQL.

However, assertions are still not supported by any of the most well-used commercial RDBMS (Oracle, MySQL, SQL

Server, PostgreSQL, DB2). It might be argued that assertions can be emulated via manually writing a set of triggers, which is a widely supported feature of RDBMS. However, its manual definition is error prone and the whole set of necessary triggers to write might not be evident when given a complex constraint, thus, compromising the integrity of the data if just one trigger is missing or ill-defined. Hence, it is better to delegate this complex checking code to RDBMS capabilities [8], as we do in TINTIN<sup>1</sup>.

TINTIN is a tool that provides incremental integrity checking of assertions in SQL Server. Given an SQL Server DB, and a set of SQL assertions written on its schema, TINTIN automatically builds all the necessary procedures/queries to efficiently check whether any update satisfies the assertions.

As an example, consider the schema of the well-known TPC-H benchmark [7] shown in Figure 1, a benchmark for illustrating decision support systems that examine large volumes of data, execute complex queries, and give answers to critical business questions.

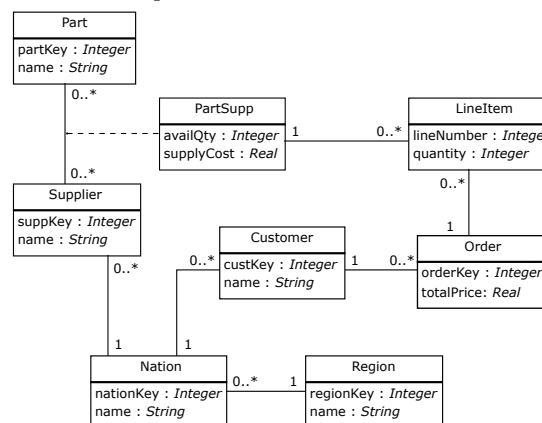


Figure 1: The TPC-H Schema.

Now, assume that we want to define a general constraint over the previous schema stating that all orders have at least one line item. This constraint could be specified by means of the SQL assertion shown below:

```
CREATE ASSERTION atLeastOneLineItem CHECK(
  NOT EXISTS(
    SELECT * FROM ORDERS AS o
    WHERE NOT EXISTS (
      SELECT * FROM LINEITEM AS l
      WHERE l.L_ORDERKEY = o.O_ORDERKEY));
```

<sup>1</sup><http://www.essi.upc.edu/~xoriol/tintin/>

TINTIN allows checking the assertion *atLeastOneLineItem* efficiently in data sets consisting of 1GB to 5GB of data and with 1MB to 5MB of tuple insertions/deletions, with times ranging from 0.01 to 0.04 seconds depending on the scenario. These results are much better than the time required for directly executing the query inside the assertions on the database, ranging from x89 to x2662 times faster.

The approach we follow in TINTIN is to automatically generate, for each assertion, several standard SQL queries which incrementally determine whether the update violates the original assertion or not. If the queries return a non-empty answer, then, the assertion is violated, otherwise, it is satisfied. The queries are incremental in the sense that they are stated in terms of the current database tables and also some automatically generated auxiliary tables containing the insertions/deletions of tuples requested by the user.

This is the crucial point for achieving efficiency of integrity checking. When a user requests an update, the tuples s/he wants to insert/delete are put in some auxiliary tables. Then, the generated queries join these tuples being inserted/deleted with the current data, and return those that violate the assertions. For each assertion, the generated queries only join those insertions/deletions of tuples that might cause its violation. Therefore, an update not including any of those insertions/deletions trivially makes the query result to be empty. For this reason, no current data of the database is accessed unless some update may cause the violation of an assertion.

The join between the current data and the tuples being inserted/deleted ensures also that only the tuples affected by the update are checked. Thus, the rest of the data, potentially the major part of the database, is skipped.

These incremental SQL queries are generated outside the database and then stored in it as views. Since we use standard SQL to define them, they could be used for checking assertions on any relational DBMS. However, and for the purpose of checking the feasibility and the efficiency of our approach, we have chosen SQL Server to implement TINTIN because of our previous expertise on this system.

Once the incremental SQL queries are defined, TINTIN builds a stored procedure called `safeCommit` to allow SQL Server to check the assertions. This procedure must be invoked at the end of each transaction, so that the procedure can check whether it violates any of the assertions. More specifically, the procedure checks whether the incremental SQL queries are empty or not. If they are, the update does not violate any assertion, so, the procedure commits the update stored in the auxiliary tables. Otherwise, the procedure shows the tuples answering the queries, i.e., the tuples violating the assertions.

## 2. PROBLEM AND SOLUTION

*Integrity checking* is the problem of efficiently determining whether a given update satisfies a set of integrity constraints, SQL assertions in our case. This is an important problem in data management since any violation of an integrity constraint would indicate an invalid state of the database. One possible way to achieve efficiency relies on, first, just checking the assertions which may actually be violated by the update and, second, considering only the relevant updated data for computing whether the assertions are violated.

We assume in this work that the update consists of a (possibly large) set of insertions and/or deletions of database

tuples and that the queries defining the assertions are specified by means of the fragment of SQL that is equivalent to relational algebra. In particular, TINTIN accepts assertions to be defined through selection, projection, join, subselect (`exists, in`), negation (`not exists, not in`) and union but it does not allow functions (e.g. aggregates, arithmetic functions) for the moment.

TINTIN is aimed at providing to an SQL Server database with the capability of performing integrity checking of SQL assertions efficiently. For this purpose, TINTIN allows a user to specify assertions according to the fragment of SQL stated above, and the tool automatically builds a stored procedure in the database, called `safeCommit`, that the user will have to call at the end of each transaction. Whenever called, `safeCommit` checks whether the updates in the transaction violate any of the assertions. If no violation is found, the update is committed to the database. Otherwise, it provides the answers to the queries that detected the violation of the assertions.

The `safeCommit` procedure works by executing several SQL queries, stored as views in the database, for each one of the assertions that need to be checked. Each query captures a different situation in which some updates may lead to the violation of the assertion. These updates are explicitly stated in the query definition itself and provide the key for efficiency of integrity checking.

In the rest of this section we explain the approach we follow to obtain the SQL queries that allow checking incrementally an assertion; which is based on our previous work for handling integrity checking of OCL constraints in conceptual models [4, 5]. We only require the users to define their desired assertions. From there, all the following steps are automatically performed.

The first step consists in rewriting each SQL assertion into a logic *denial* in the same way as we did in [6]. A denial is a formula stating a condition that must not be true in any state of the database. These denials are the basis for obtaining the incremental SQL queries.

For instance, the assertion `atLeastOneLineItem` of our running example would be rewritten as:

$$order(o) \wedge \neg lineIt(l, o) \rightarrow \perp \quad (1)$$

Clearly, the previous denial states that if there is an order  $o$  without any line item  $l$ , an inconsistent state will be reached, which is exactly the condition to be avoided by `atLeastOneLineItem`.

Then, for each denial, TINTIN obtains its corresponding *Event Dependency Constraints* (EDCs, for short). Each EDC is a logic rule identifying a particular situation where some update applied to a certain state of the database  $D$  causes the violation of the denial, i.e., of the corresponding assertion. The main idea for obtaining EDCs is to replace each literal in the logic rule obtained from the assertion by the expression that evaluates this literal in the new state of the database  $D^n$ , i.e., the state obtained after applying the update. Positive and negative literals in the denial are handled in a different way according to the following formulas:

$$\forall \bar{x}. p^n(\bar{x}) \leftrightarrow (up(\bar{x})) \vee (\neg \delta p(\bar{x}) \wedge p(\bar{x})) \quad (2)$$

$$\forall \bar{x}. \neg p^n(\bar{x}) \leftrightarrow (\delta p(\bar{x})) \vee (\neg up(\bar{x}) \wedge \neg p(\bar{x})) \quad (3)$$

Rule 2 states that a literal  $p(\bar{x})$  will be true in the new state of the database  $D^n$  if it has been inserted or if it was already true in the initial state  $D$  and it has not been deleted.

In an analogous way, rule 3 states that  $p(\bar{x})$  will not hold in  $D^n$  if it has been deleted or if it was already false and it has not been inserted.

By applying the substitutions above to all logic denials, we get a set of EDCs which states all possible ways to violate the assertions by means of insertions and/or deletions of tuples.

In particular, we get the following EDCs for the denial 1 of our running example:

$$\iota order(o) \wedge \neg lineIt(l, o) \wedge \neg \iota lineIt(l, o) \rightarrow \perp \quad (4)$$

$$\iota order(o) \wedge \delta lineIt(l, o) \wedge \neg aux(o) \rightarrow \perp \quad (5)$$

$$order(o) \wedge \neg \delta order(o) \wedge \delta lineIt(l, o) \wedge \neg aux(o) \rightarrow \perp \quad (6)$$

$$aux(o) \leftarrow \iota lineIt(l, o)$$

$$aux(o) \leftarrow lineIt(l, o) \wedge \neg \delta lineIt(l, o)$$

Intuitively, EDC 4 states that *atLeastOneLineItem* will be violated if a new order  $o$  is inserted and there was no line item for  $o$  in the initial state of the database and no line item for  $o$  has been inserted by the transaction. EDC 5 behaves in a similar way, while EDC 6 determines that the assertion will be violated if a line item for an existing order  $o$  has been deleted and neither a new line item has been inserted for  $o$  nor the database contains any other line item for  $o$  (given by the rules defining  $aux(o)$ ).

Note that, in this example, EDC 5 can be safely discarded assuming that the foreign key constraint from *lineitem* to *order* is satisfied in the current state of the data. TINTIN incorporates some semantic optimizations like this one that allow obtaining a reduced and simplified number of EDCs which allow performing integrity checking more efficiently.

The idea of obtaining EDCs to identify the different situations that may lead to the violation of a constraint is grounded on the concept of *event rules* [3], which were aimed at performing integrity checking in deductive databases.

Finally, each EDC is translated into an SQL query as proposed in [4]. Roughly, each positive literal in the EDC is translated into a table reference placed in the **FROM** clause of the query, possibly with a **JOIN** condition with some previously translated literal that shares a common variable with it. Built-in literals and constant bindings are directly translated to the **WHERE** clause, and negated base and derived literals are translated as correlated subqueries.

In our running example, we would translate EDC 4 as:

```
CREATE VIEW atLeastOneLineItem1 AS
SELECT *
FROM ins_orders AS T0
WHERE NOT EXISTS(SELECT *
  FROM lineitem AS T1
  WHERE T1.l_orderkey = T0.o_orderkey)
AND NOT EXISTS(SELECT *
  FROM ins_lineitem AS T1
  WHERE T1.l_orderkey = T0.o_orderkey)
```

We have defined the query as a view to store it into the database. It is worth noting the usage of the auxiliary tables storing the insertions and the deletions of tuples for each table of the database, as it happens with *ins\_orders* and *ins\_lineitem* in the previous view. TINTIN automatically builds them, together the necessary triggers to capture the insertions/deletions of tuples to place them into such auxiliary tables. Thus, the existence and maintenance of these auxiliary tables is fully transparent to the database users.

Note also that the key for incrementality is not based on batching updates for delaying the assertions checking, but on the join in the SQL queries between the update and the current data. First of all, any SQL query joining an insertion/deletion which is not being applied (i.e., whose corresponding SQL table is empty) is immediately discarded since it trivially returns the empty set. Therefore, we only check those constraints that can be violated according to the ongoing update. Second, the data considered by an SQL query during its execution is necessarily the data joining the update applied, thus, avoiding to look through all the database.

### 3. DEMO DESCRIPTION

The demo that we will present is intended to show the usage and efficiency of our prototype tool TINTIN by means of applying it to the checking of some assertions in the TPC-H benchmark SQL schema.

We will first request TINTIN to build the necessary auxiliary tables and triggers to capture any insertion and deletion applied to the TPC database. As a result, we will see a newly generated database event\_TPC with an ins/del table for each TPC SQL table. At this point, whenever we apply an insertion/deletion of any tuple in TPC, the tuple will be captured and inserted in the corresponding ins/del table of event\_TPC. In this way, the contents of TPC remains unchanged, and event\_TPC contains the requested update.

Next, we will introduce in TINTIN some SQL assertions of different complexity. Consequently, TINTIN will create a procedure called **safeCommit** in TPC. This procedure, when called, will check whether applying the updates contained in event\_TPC raises the violation of any assertion. If no violation is found, the procedure will commit the events into TPC; otherwise, it will report the violations. Lastly, the procedure will truncate all the events stored in event\_TPC, so that a new set of events can be proposed.

At this stage, TINTIN will have created all the necessary elements to automatically check the satisfaction of the assertions when updating TPC, and will have persistently stored them in the database. Thus, TINTIN can be disconnected from SQL Server, and users might operate with the database normally with the unique consideration of invoking **safeCommit** at the end of each transaction.

To make the demonstration, we will apply some updates mixing both: updates that violate some assertion and updates that do not violate any of them. After each update, we will call the **safeCommit** procedure to see its effects, that is, we will see that it rejects the update if some violation occurs, or that it commits them if they satisfy the assertions.

With this demonstration, we will show that TINTIN enjoys the following features: 1. *Portability*: it can be easily installed in any SQL Server database—no need for special plugins nor additional technologies—. 2. *Clean installation*: all the necessary logics of the method is installed in another database—without modifying the original one—, except the **safeCommit** procedure and the triggers to capture the events, which are necessarily placed in the target database. 3. *Easy of use*: users can update the database without modifying their SQL statements/procedures. The unique mandatory requirement is to call the automatically generated **safeCommit** procedure at the end of each transaction. 4. *Efficiency*: the incremental nature of our method provides better execution times than executing non-incremental queries to perform the integrity checks.

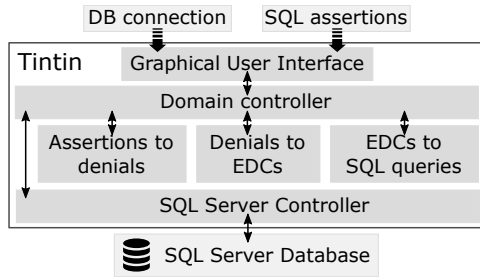


Figure 2: Tintin architecture

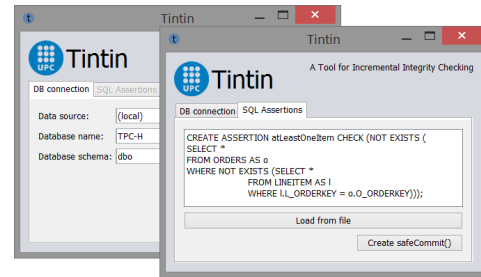


Figure 3: Graphical User Interface Design

## 4. PROTOTYPE

The architecture of TINTIN is shown in Figure 2, while its GUI is depicted in Figure 3. Basically, the GUI asks the user for a database (DB) connection, and the assertions that s/he wants to check in that database.

When the user introduces the DB connection, the SQL Server Controller creates a new auxiliary database event\_DB to store the different events applied to it; that is, for each table  $T$  in DB, the SQL Server Controller builds two new tables ( $ins\_T$  and  $del\_T$ ) to store the different tuples being inserted and deleted in  $T$ . In order to capture these tuples, the SQL Server Controller creates two different **INSTEAD OF** triggers, which capture the tuple insertions/deletions and place them in the corresponding  $ins\_T$  or  $del\_T$  table.

Afterwards, when the user introduces the SQL assertions, they are firstly mapped into logic denials. Then, these denials are translated into EDCs and, finally, EDCs are rewritten as SQL queries. Each of these steps is implemented in a different module following the previously presented method.

The resulting SQL queries are stored as views in event\_DB. Then, the SQL Server Controller builds the **safeCommit** procedure. This procedure, when called, performs the following: 1. queries the previous views. 2. if all queries are empty, it disables the triggers, applies the update (insert in the DB the tuples contained in the  $ins$  tables, and remove from the DB the tuples contained in the  $del$  tables), and enables again the triggers. 3. truncates the  $ins/del$  tables.

The prototype has been developed in Java, with the exception of the *Assertions to denials* translator component, for which we have reused a previously existing C# software.

We made some experiments to evaluate the efficiency of our tool. We have checked some assertions of different complexity with TINTIN (like the one of our running example), in data sets consisting of 1GB to 5GB of data and with 1MB to 5MB of updates, and compared its efficiency with that of a non incremental method consisting of directly querying the assertions to the database. The time TINTIN required for checking the assertions ranges from 0.01 to 1.29 seconds and it is always better than in the non incremental approach, with a benefit of orders of magnitude when considering 5MB of updates (up to x2662 times faster).

## 5. CONCLUSIONS

TINTIN is a tool for checking assertions in SQL Server databases. The tool takes as input a set of assertions and it automatically builds a procedure called **safeCommit** which efficiently checks whether an update violates any of the assertions and, afterwards, commits the update if no violation is found or shows the tuples causing the violation otherwise.

TINTIN works almost transparently for the database users since it only requires the users to invoke **safeCommit** at the end of each transaction. Internally, the tool builds several triggers to capture the update requested by the user and to place it in some SQL auxiliary tables. These auxiliary tables are queried with the current database tables to check whether applying the update in the current data may cause any assertion violation. This join between the update and current data is the key for efficiency.

The fundamentals of TINTIN are the Event Dependency Constraints (EDCs), which we previously used to handle integrity checking of OCL constraints in conceptual models. More details about these rules, and also on their translation to SQL queries, can be found in [4, 5].

As further work, we plan to extend TINTIN to handle aggregate functions in assertions. We also expect to make it available for other DBMSs apart from SQL Server and to exploit other DBMS capabilities such as temporary tables.

**Acknowledgements** This work has been partially supported by the Ministerio de Economía y Competitividad, under project TIN2014-52938-C2-2-R and by the Secretaria d'Universitats i Recerca de la Generalitat de Catalunya under 2014 SGR 1534 and a FI grant.

## 6. REFERENCES

- [1] ANSI Standard. *The SQL 92 Standard*, 1992.
- [2] R. Elmasri and S. B. Navathe. *Fundamentals of database systems*. Pearson, 2014.
- [3] A. Olivé. Integrity constraints checking in deductive databases. In *Proceedings of the 17th Int. Conference on Very Large Data Bases (VLDB)*, pages 513–523, 1991.
- [4] X. Oriol and E. Teniente. Incremental checking of OCL constraints through SQL queries. In *Proc. of the 14th Int. Workshop on OCL and Textual Modelling*, pages 23–32, 2014.
- [5] X. Oriol and E. Teniente. Incremental checking of OCL constraints with aggregates through SQL. In *Conceptual Modeling*, volume 9381 of *LNCS*, pages 199–213. Springer, 2015.
- [6] E. Teniente, C. Farré, T. Urpí, C. Beltrán, and D. Gañán. SVT: schema validation tool for microsoft sql-server. In *Proc. of the 30th International Conference on Very Large Data Bases*, pages 1349–1352, 2004.
- [7] Transaction Processing Performance Council. *TPC-H benchmark specification 2.17.1*, 2014. <http://www.tpc.org>.
- [8] V. Tropashko and D. Bursleson. *SQL Design Patterns: Expert Guide to SQL Programming*. Rampant Techpress, 2007.

# Efficient regular path query evaluation using path indexes

George H. L. Fletcher  
Eindhoven University of  
Technology  
The Netherlands  
g.h.l.fletcher@tue.nl

Jeroen Peters  
Eindhoven University of  
Technology  
The Netherlands  
j.peters.1@student.tue.nl

Alexandra Poulouvasilis  
London Knowledge Lab  
Birkbeck, University of  
London, UK  
ap@dcs.bbk.ac.uk

## ABSTRACT

We demonstrate the use of localized path indexes in generating efficient execution plans for regular path queries. This study is motivated by both the practicality of this class of queries and by the current dearth of scalable solutions for their evaluation. Our proposed solution leverages widely available relational database technology and is often orders of magnitude faster than currently known approaches. We aim in this hands-on demonstration to both highlight the promise of our approach and to stimulate further discussion and study of engineering solutions for this practical yet challenging class of graph queries.

## 1. INTRODUCTION

Massive graph-structured data collections are ubiquitous in contemporary data management scenarios such as social networks, linked open data, and chemical compound databases. A fundamental paradigm in graph query languages are the so-called *regular path queries (RPQs)* [16]. RPQs specify a regular expression over the edge labels in a graph, and the query answer consists of every path in the graph such that the sequence of edge labels along the path forms a word in the language recognized by the regular expression. Variations and extensions of RPQs are supported in recent query languages such as SPARQL 1.1 [8] and the Cypher language of the Neo4j graph database.<sup>1</sup>

*State of the art.* Indexing of paths occurring in data has been shown to be effective for query processing in the context of object-oriented and semistructured databases [1, 15]. To our knowledge, however, there has been no investigation of using path indexing for the evaluation of RPQs over graph databases. In particular, three general approaches to RPQ evaluation have been proposed in the literature:

1. Automata- and search-based processing (e.g., [5, 10, 13]), where queries are evaluated by strategies such

<sup>1</sup><http://neo4j.com>

as breadth-first-search pattern matching of the query graph on the data graph;

2. Datalog-based processing (e.g., [3, 17]), where the Kleene star operator is translated into recursive Datalog programs or recursive SQL views;
3. Reachability-index-based processing (e.g., [6]), where restricted uses of Kleene star are translated into reachability queries, which are then evaluated using off-the-shelf reachability indexes.

*Contributions.* We present an overview of our ongoing study of the use of path indexes for generating efficient execution plans for RPQs. Our approach supports the evaluation of arbitrary RPQs (unlike approach (3) above), and exhibits significant improvement (often by several orders of magnitude) in query processing times over approaches (1) and (2).

In the next section we briefly define the problem. In Section 3 we introduce the main data structures used in our solution. We then discuss query plan generation and execution in Section 4. We conclude in Section 6 with an overview of our system demonstration.

## 2. PRELIMINARIES

### 2.1 Data Model

We consider finite, directed, edge-labeled graphs. A *graph vocabulary* is a finite non-empty set  $\mathcal{L}$  of *edge labels* drawn from some universe of labels. Let  $N$  be an infinite universe of atomic data objects. An *edge relation* is a finite subset of  $N \times N$ . A *graph* over vocabulary  $\mathcal{L}$  is an assignment  $G$  of an edge relation  $\ell^G$  to each  $\ell \in \mathcal{L}$ , i.e., there is an edge labeled  $\ell$  from  $m$  to  $n$  if  $(m, n) \in \ell^G$ . In the sequel, we will sometimes denote this by  $\ell(m, n)$  or  $m \xrightarrow{\ell} n$ . As an example, a graph  $G_{ex}$  over the vocabulary  $\{\text{supervisor, knows, worksFor}\}$  is shown in Figure 1.

The *node set* of  $G$  is the collection  $nodes(G) = \{n \mid \exists m \in N, \ell \in \mathcal{L} : (n, m) \in \ell^G \text{ or } (m, n) \in \ell^G\}$ . For example, the node set of the example graph contains nine elements:  $nodes(G_{ex}) = \{\text{ada, jan, \dots, zoe}\}$ .

Let  $k$  be a natural number. If  $k = 0$ , we say there is a *k-path* from  $s$  to  $s$ , for every  $s \in nodes(G)$ . If  $k > 0$ , for  $s, t \in nodes(G)$ , we say there is a *k-path* from  $s$  to  $t$  if there exist  $n_0, \dots, n_k \in nodes(G)$  and edge labels  $\ell_1, \dots, \ell_k \in \mathcal{L}$  such that  $n_0 = s$ ,  $n_k = t$ , and, for  $0 < i \leq k$ ,  $(n_{i-1}, n_i) \in \ell_i^G$  or  $(n_i, n_{i-1}) \in \ell_i^G$ .



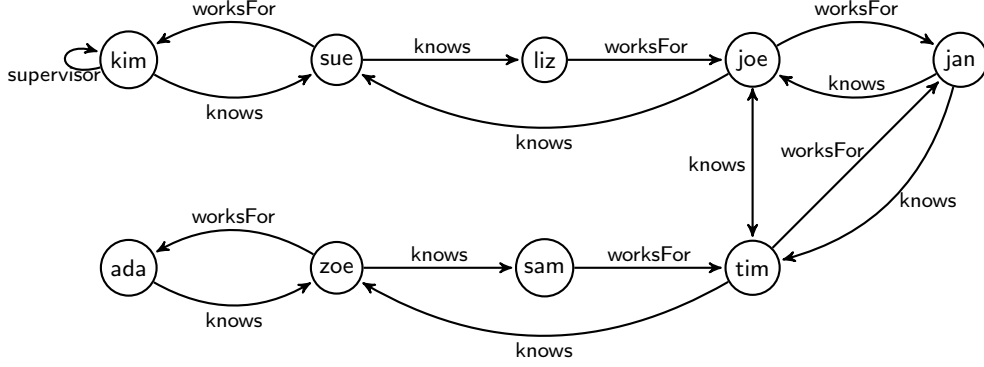


Figure 1: A graph  $G_{ex}$  over vocabulary  $\mathcal{L} = \{\text{supervisor, knows, worksFor}\}$ .

We denote by  $paths_k(G)$  the set of all pairs of nodes  $(s, t) \in nodes(G) \times nodes(G)$  such that there is an  $i$ -path from  $s$  to  $t$ , for some  $i \leq k$ .

As an example, in the graph  $G_{ex}$  we have that  $(\text{sam}, \text{ada}) \in paths_2(G_{ex})$  via the paths  $\text{sam} \xleftarrow{\text{knows}} \text{zoe} \xrightarrow{\text{worksFor}} \text{ada}$  and  $\text{sam} \xleftarrow{\text{knows}} \text{zoe} \xleftarrow{\text{knows}} \text{ada}$ , but  $(\text{sam}, \text{ada}) \notin paths_1(G_{ex})$ .

## 2.2 Regular Path Queries

Fix a vocabulary  $\mathcal{L}$ . A *regular path query* (RPQ) is a regular expression  $R$  over the alphabet  $\{\ell, \ell^- \mid \ell \in \mathcal{L}\}$ , i.e.,  $R$  is generated by the grammar

$$R ::= \epsilon \mid \ell \mid \ell^- \mid R \circ R \mid R \cup R \mid R^{i,j}$$

for  $\ell \in \mathcal{L}$  and natural numbers  $i$  and  $j$  where  $i \leq j$ . Intuitively, “ $\epsilon$ ” is the identity transition, “ $\ell$ ” is forward navigation along an edge with that label, “ $\ell^-$ ” is backwards navigation, “ $\circ$ ” is path composition, “ $\cup$ ” is path disjunction (i.e., union of paths), and “ $R^{i,j}$ ” is bounded path recursion.

Given graph  $G$  over  $\mathcal{L}$ , the semantics of evaluating an RPQ  $R$  on  $G$  is a set  $R(G)$  consisting of all pairs  $(a, b)$  in  $paths(G)$  such that there exists a path from node  $a$  to node  $b$  in  $G$  whose label sequence  $\ell_1 \cdots \ell_n$  defines a word in the regular language specified by  $R$ .

As examples, we have in the graph  $G_{ex}$  of Figure 1 that:

$$\text{supervisor} \circ \text{worksFor}^-(G_{ex}) = \{(\text{kim}, \text{sue})\}$$

and

$$\begin{aligned} (\text{supervisor} \cup \text{worksFor} \cup \text{worksFor}^-)^{4,5}(G_{ex}) = \\ \{(\text{kim}, \text{kim}), (\text{kim}, \text{sue}), (\text{sue}, \text{kim}), (\text{sue}, \text{sue}), \\ (\text{ada}, \text{zoe}), (\text{ada}, \text{ada}), (\text{zoe}, \text{ada})\}. \end{aligned}$$

Note that we deviate from the traditional syntax of regular expressions by replacing Kleene star “ $*$ ” with bounded recursion. This is motivated by the following two observations. First, bounded recursion is supported (and encouraged) in practical graph query languages such as Neo4j’s Cypher.<sup>2</sup> Second, since we focus in this work on index construction and use, we are interested in query evaluation on a given graph. It is easy to establish that for any graph  $G$  there exists a natural number  $n(G)$  such that for every RPQ  $R$  it is the case that  $R^*(G) = R^{0, n(G)}(G)$ .

<sup>2</sup><http://neo4j.com/docs/stable/>, Section 8.8

## 3. INDEXES AND HISTOGRAMS

Given a graph  $G$  and a fixed  $k > 0$ , we now present our indexing and selectivity estimation approaches for paths in  $G$  localized to neighborhoods of size  $k$ .

### 3.1 $k$ -path indexing

A *label path* is a sequence  $\mathbf{p} = \ell_1 \cdots \ell_n$ , where  $n > 0$  is the *length* of  $\mathbf{p}$ , and  $\ell_i \in \{\ell, \ell^- \mid \ell \in \mathcal{L}\}$  for each  $1 \leq i \leq n$ .

Our index on  $G$  is based on an ordered dictionary (which can be implemented, for example, as a B+tree). In particular, we index  $paths_k(G)$  using an ordered  $k$ -path index  $I_{G,k}$  having search key  $\langle \text{label path}, \text{sourceID}, \text{targetID} \rangle$ . Specifically, for each label path  $\mathbf{p}$  of length at most  $k$ , and for each pair of nodes  $(a, b) \in \mathbf{p}(G)$ , we insert  $(\mathbf{p}, a, b)$  into  $I_{G,k}$ . Given a non-empty prefix  $p$  of a search key,  $I_{G,k}$  returns an ordered list  $I_{G,k}(p)$  of all matching entries.

EXAMPLE 3.1. In the graph of Figure 1, we have

$$\begin{aligned} I_{G,k}(\langle \text{knows} \cdot \text{knows} \cdot \text{worksFor} \rangle) = \\ \langle (\text{ada}, \text{tim}), (\text{jan}, \text{ada}), (\text{jan}, \text{jan}), (\text{jan}, \text{kim}), \\ (\text{joe}, \text{ada}), (\text{joe}, \text{jan}), (\text{joe}, \text{joe}), (\text{kim}, \text{joe}), \\ (\text{tim}, \text{jan}), (\text{tim}, \text{kim}), (\text{tim}, \text{tim}) \rangle, \end{aligned}$$

$$\begin{aligned} I_{G,k}(\langle \text{knows} \cdot \text{knows} \cdot \text{worksFor}, \text{jan} \rangle) = \\ \langle (\text{ada}), (\text{jan}), (\text{kim}) \rangle, \end{aligned}$$

$$I_{G,k}(\langle \text{knows} \cdot \text{knows} \cdot \text{worksFor}, \text{jan}, \text{ada} \rangle) = \langle \rangle,$$

$$I_{G,k}(\langle \text{knows} \cdot \text{knows} \cdot \text{worksFor}, \text{jan}, \text{joe} \rangle) = \langle \rangle.$$

We have developed a prototype  $k$ -path index implementation that leverages the B+ tree index support of PostgreSQL<sup>3</sup>. We translate RPQs into equivalent SQL statements over  $I_{G,k}$  implemented as a relational table and backed by a B+tree (see [12] for full implementation details). In building on mature relational technologies, we are following an emerging trend in this direction [4, 7, 9] with practical benefits such as simplicity, ease of integration with and deployment within existing IT ecosystems, and leveraging field-proven technologies.

Notwithstanding the fact that we have described here an implementation of our proposed  $k$ -path indexing technique

<sup>3</sup><http://www.postgresql.org>

using existing RDBMS technologies, other recent work [14] describes an implementation of a B+tree-based  $k$ -path index “from scratch”, focusing on issues such as index size, compression and performance, and undertaking a comparative performance study with the Neo4j graph DBMS over several real and synthetic datasets and query workloads. That evaluation too demonstrates the potential of our  $k$ -path indexing approach (showing speed-ups in query evaluation times ranging from 2 times to 8,000 times faster compared with Neo4j). Detailed performance comparison between these approaches to path indexing is an area of future work.

### 3.2 $k$ -path histogram

For query plan generation over  $I_{G,k}$ , it is useful to have a data structure  $sel_{G,k}$  which, given a label path  $\mathbf{p}$  of length at most  $k$ , returns an estimate of the *selectivity* of  $\mathbf{p}$  in  $G$ , i.e., the fraction of paths in  $paths_k(G)$  which satisfy  $\mathbf{p}$ . As an example, we have in the graph  $G_{ex}$  of Figure 1 that  $sel_{G_{ex},2}(\text{supervisor} \circ \text{knows}) \approx 0.02$ , since only one of the 53 paths in  $paths_2(G_{ex})$  is in  $\text{supervisor} \circ \text{knows}(G_{ex})$ .

There is a rich literature on statistics for query optimization [2]. Here, we adopt the well-established histogram data structure, since it is easy to deploy and extremely successful in practice. In particular, we implement  $sel_{G,k}$  as an *equi-depth histogram*. The basic idea here is, given space to store cardinality information about  $B$  label path ranges, we keep track of the cardinality of label paths in the graph falling into  $B$  contiguous ranges, as induced by their lexicographic order (i.e., in the order maintained in  $I_{G,k}$ ); the ranges are selected such that they each have roughly the same cardinality. We then estimate the cardinality of any given label path by dividing the cardinality of the range in which it occurs by the number of label paths in that range. As a simple example, suppose  $k = 1$ ,  $B = 2$ , and we have edge labels  $\mathbf{a}$ ,  $\mathbf{b}$ , and  $\mathbf{c}$ , with cardinalities 2, 4, and 6, resp. Then the first range covers  $\mathbf{a}$  and  $\mathbf{b}$  and the second range covers  $\mathbf{c}$ , with cardinalities 6 and 6, resp. Using this histogram, the estimated cardinality of  $\mathbf{a}$  is  $\frac{6}{2} = 3$ .

As with the path indexes, in our prototype implementation we store and access our histogram as a PostgreSQL table; see [12] for full implementation details.

## 4. QUERY EVALUATION WITH PATH INDEXES

The processing of a RPK  $R$  proceeds in three steps: The **first step** is to replace each occurrence of bounded recursion in  $R$  as a union over its expansion. The result is a semantically equivalent query  $R'$  involving only edge labels or their inverses, compositions, and unions. In the **second step**, all unions in  $R'$  are “pulled up” to the top level of the query, resulting in a semantically equivalent query  $R''$  consisting of a union of expressions each free of unions and bounded recursion, i.e.,  $R'' = R_1 \cup \dots \cup R_n$  where each  $R_i$  is a label path. In the **third step**, each disjunct  $R_i$  is processed in turn, with the aim of generating a physical execution plan for each  $R_i$  in which a merge-join is used whenever possible (to make the best use of the physical sort order of the index) and a hash-join is used otherwise.

As an illustrative example, consider the query  $R = \mathbf{k} \circ (\mathbf{k} \circ \mathbf{w})^{2,4} \circ \mathbf{w}$ , where  $\mathbf{k}$  and  $\mathbf{w}$  abbreviate  $\text{knows}$  and  $\text{worksFor}$ , resp. Query plan generation proceeds as follows, where for clarity of presentation we drop explicit use of the concatenate

operation  $\circ$ :

1.  $(\mathbf{k}\mathbf{w})^{2,4}$  is expanded, giving

$$R' = \mathbf{k}(\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w} \cup \mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w} \cup \mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w})\mathbf{w}.$$

2. Nested unions are pulled up to the top level, giving

$$R'' = \mathbf{k}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{w} \cup \mathbf{k}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{w} \cup \mathbf{k}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{w}.$$

3. Finally, physical execution plans are generated for each of the disjuncts of  $R''$ . In particular, suppose that  $k = 3$ ; then processing each of the disjuncts proceeds as follows:

- $\mathbf{k}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{w}$  is processed, from left to right, generating the physical plan

$$I_{G,k}(\mathbf{w}^- \mathbf{k}^- \mathbf{k}^-) \bowtie I_{G,k}(\mathbf{k}\mathbf{w}\mathbf{w})$$

in which  $\bowtie$  is implemented as a merge join. Note the subexpression  $\mathbf{k}\mathbf{k}\mathbf{w}$  has been inverted to obtain the correct sort order to perform a merge join.

- $\mathbf{k}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{w}$  is processed from left to right, generating the physical plan

$$[I_{G,k}(\mathbf{w}^- \mathbf{k}^- \mathbf{k}^-) \bowtie_1 I_{G,k}(\mathbf{k}\mathbf{w}\mathbf{k})] \bowtie_2 I_{G,k}(\mathbf{w}\mathbf{w})$$

in which  $\bowtie_1$  is implemented as a merge join and  $\bowtie_2$  as a hash join.

- $\mathbf{k}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{k}\mathbf{w}\mathbf{w}$  is processed from left to right, generating the physical plan

$$[[I_{G,k}(\mathbf{w}^- \mathbf{k}^- \mathbf{k}^-) \bowtie_1 I_{G,k}(\mathbf{k}\mathbf{w}\mathbf{k})] \bowtie_2 I_{G,k}(\mathbf{w}\mathbf{k}\mathbf{w})] \bowtie_3 I_{G,k}(\mathbf{w})$$

in which  $\bowtie_1$  is implemented as a merge join and  $\bowtie_2$  and  $\bowtie_3$  as hash joins.

We term this evaluation strategy *semi-naive*. The complete physical plan is formed as a union of these three sub-plans.

The third step can be optimized by using the histogram  $sel_{G,k}$ , as follows. For each disjunct  $D$  of  $R''$

1. if  $|D| \leq k$ , return  $I_{G,k}(D)$ .
2. Find the most selective  $k$ -path subquery  $D'$  of  $D$  (i.e., the  $k$ -path with smallest  $sel_{G,k}$  value). There are  $|D| - (k - 1)$  such subqueries to consider.
3. Let  $D = D_{\text{left}} \circ D' \circ D_{\text{right}}$ , and recur on  $D_{\text{left}}$  and  $D_{\text{right}}$ , to generate query plans for respective output streams  $LEFT$  and  $RIGHT$ .
4. Determine the cost of each of the following alternative query plans, and return the cheapest plan:
  - $[LEFT \bowtie I_{G,k}(D')] \bowtie RIGHT$ ,
  - $LEFT \bowtie [I_{G,k}(D') \bowtie RIGHT]$ ,
  - $[LEFT \bowtie I_{G,k}(D'^-)] \bowtie RIGHT$ , or
  - $LEFT \bowtie [I_{G,k}(D'^-) \bowtie RIGHT]$ .

We term this evaluation strategy *minSupport*.

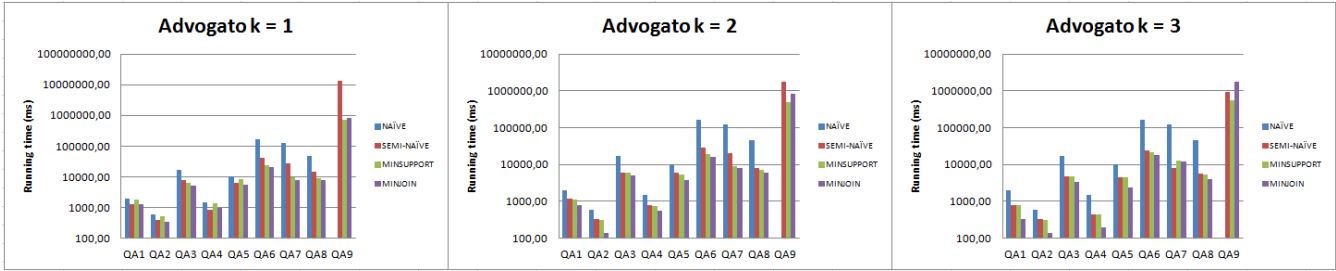


Figure 2: Advogato query execution times (ms)

## 5. EMPIRICAL EVALUATION

We refer the reader to [12] for full details of an empirical evaluation of our system with respect to a broad spectrum of RPQs over four different real and synthetic datasets. In addition to the semi-naive and minSupport evaluation methods described above, two other methods are investigated: *naive*, in which  $k$  is fixed at 1 (so indexing is on edge labels, not path labels), which corresponds to automaton-based evaluation (approach 1, discussed in the Introduction); and *minJoin*, which is similar to minSupport but also aims to minimize the number of joins.

As an indicative subset of the empirical results obtained, the three graphs in Figure 2 show the run-times of 8 queries over the Advogato data set, for each of the four evaluation methods, with values of  $k$  ranging from 1 to 3. Advogato is a real-world social network having 6,541 nodes and 51,127 edges with  $|\mathcal{L}| = 3$ , where edges indicate varying degrees of trust between users in the network [11].<sup>4</sup>

We observe that the naive method always performs worst, that the semi-naive method is generally outperformed by minSupport and minJoin, and that the latter two perform similarly. This demonstrates the value of the lightweight histogram data structure for selectivity estimation. We also see that increasing the value of  $k$  generally improves the run-times for all methods (apart from naive, where  $k$  is fixed at 1 throughout).

## 6. DEMONSTRATION OVERVIEW

We give participants a hands-on overview of the life of a regular path query, from its submission to our system, through parsing and optimization, to execution. We further demonstrate the speed-ups achieved by our approach compared with Datalog-based evaluation (approach (2), discussed in the Introduction), where our solution is on average 1200x faster on the Advogato queries [12].

Through these interactive activities, we hope to both demonstrate the promise of our approach to RPQ evaluation and to stimulate further broader discussion and study in the research community of engineering strategies for this challenging practical class of graph queries. A system demonstration is an excellent setting in which to accomplish these goals.

## 7. REFERENCES

- [1] E. Bertino et al. Object-oriented databases. In E. Bertino et al, editor, *Indexing Techniques for Advanced Database Systems*, pages 1–38. Kluwer, 1997.
- [2] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.
- [3] S. C. Dey et al. On implementing provenance-aware regular path queries with relational query engines. In *GraphQ*, pages 214–223, Genoa, 2013.
- [4] J. Fan, G. S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR*, Asilomar, California, 2015.
- [5] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. *Frontiers of Comp. Sci.*, 6(3):313–338, 2012.
- [6] A. Gubichev, S. J. Bedathur, and S. Seufert. Sparql kleene: fast property paths in RDF-3X. In *GRADES*, New York, NY, 2013.
- [7] A. Gubichev and M. Then. Graph pattern matching – do we have to reinvent the wheel? In *GRADES*, Snowbird, Utah, 2014.
- [8] S. Harris and A. Seaborne, editors. *SPARQL 1.1 Query Language*, W3C Recomm., 2013.
- [9] A. Jindal and S. Madden. GRAPHiQL: A graph intuitive query language for relational databases. In *Big Data*, pages 441–450, Washington, DC, 2014.
- [10] A. Koschmieder and U. Leser. Regular path queries on large graphs. In *SSDBM*, pages 177–194, Chania, Crete, Greece, 2012.
- [11] P. Massa, M. Salvetti, and D. Tomasoni. Bowling alone and trust decline in social network sites. In *DASC*, pages 658–663, Chengdu, China, 2009.
- [12] J. Peters. Regular path query evaluation using path indexes. Master’s thesis, Eindhoven University of Technology, 2015.
- [13] P. Selmer, A. Poulouvasilis, and P. T. Wood. Implementing flexible operators for regular path queries. In *GraphQ*, Brussels, 2015.
- [14] J. Sumrall, G. H. L. Fletcher, and A. Poulouvasilis et al. Investigations on path indexing for neo4j, 2015. Under review.
- [15] K.-F. Wong, J. X. Yu, and N. Tang. Answering XML queries using path-based indexes: A survey. *World Wide Web*, 9(3):277–299, 2006.
- [16] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.
- [17] N. Yakovets, P. Godfrey, and J. Gryz. WAVEGUIDE: evaluating SPARQL property path queries. In *EDBT*, pages 525–528, Brussels, 2015.

<sup>4</sup>The data set is publicly available at <http://konect.uni-koblenz.de/networks/advogato>

# Galaxy: A Platform for Explorative Analysis of Open Data Sources

Seyed-Mehdi-Reza Beheshti #, Boualem Benatallah #, Hamid Reza Motahari-Nezhad #,\*

# *University of New South Wales, Sydney, Australia*  
{sbeheshti,boualem,hamidm}@cse.unsw.edu.au

\* *IBM, Almaden Research Center, San Jose, USA*  
motahari@us.ibm.com

## ABSTRACT

A large volume of Open Data is being generated on a continuous basis. Examples of this are the case of social, natural, and information systems such as World Wide Web and social networks. Most entities and objects in the Open Data are interconnected, forming a complex, semi-structured, and information-rich networks. In this sense, Linked Open Data has the potential to be similar to a federated database. Since Linked Open Data is based on W3C standards, it is possible to implement a federation infrastructure, however, the current SPARQL standard makes it challenging to analyze the Open Data in an explorative manner. Consequently, it will be hard to discover the hidden knowledge in the relationships among entities in Open Data sources. In this paper, we present Galaxy, a platform for explorative analysis of Open Data Sources. Galaxy facilitates the analysis of Open Data graphs based on simple abstractions, i.e. folders and paths, which enable an analyst to group related entities in the graph or find paths among entities. Galaxy uses Hadoop data processing platforms to store and retrieve large numbers of RDF triples and to support cost-effective and Web-scale processing of Semantic Web data through a Folder-Path enabled extension of SPARQL.

## Keywords

Linked Data, Open Data Analytics, Querying Graphs

## 1. INTRODUCTION

Open Data sources may include any information that can be obtained without a privileged position. Examples include electronic and print media (e.g. RSS feeds from newspapers), social media (Twitter, Facebook, Instagram, YouTube), and blog sites (e.g. Tumblr, Wordpress). The production of knowledge from Open Data is seen by many organizations as an increasingly important capability that can complement

the traditional intelligence sources. In particular, most entities and objects in the Open Data are interconnected, forming a complex, semi-structured, and information-rich networks which can be modeled using graphs. In this sense, Linked Open Data has the potential to be similar to a federated database: combining these data sources offer a rich information resource for enterprise analysis.

Since Linked Open Data is based on W3C standards (e.g. RDF format and the SPARQL query language), it is possible to implement a federation infrastructure, however, the current SPARQL standard makes it challenging to analyze the Open Data in an explorative manner. Consequently, it will be hard to discover the hidden knowledge in the relationships among entities in Open Data sources. For example it is important to quickly form an intelligence picture from the Open Data sources around a topic of interest (such as country, person, organization or event), group related entities around that topic, find paths among entities, and use all these information for the follow-on analysis. There is a need for graph representation models and efficient approaches for expressing and executing these types of queries. In particular, manipulating, querying, and analyzing Linked Open Data graphs to discover new knowledge is of high interest.

In this paper, we present Galaxy, a platform for explorative analysis of Open Data Sources. Galaxy helps in facilitating the analysis of Open Data graphs based on simple abstractions, i.e. folders and paths (introduced in our earlier work [4]), which enables an analyst to group related entities in the graph or find paths among entities. A folder node contains a set of entities that are related to each other, i.e., the set of entities in a folder node is the result of a given query that requires grouping graph entities in a certain way. We define a path node for each query that results in a set of paths (i.e. transitive relationship between two entities which can be codified using regular expressions). Folder and Path nodes, can represent a network snapshot, i.e. a subgraph, from multiple perspectives and granularities. Folder and Path nodes can be timed [3]: Timed folder/path nodes can show their evolution for the time period that they represent. Galaxy uses Hadoop data processing platforms to store and retrieve large numbers of RDF triples and to support cost-effective and Web-scale processing of Semantic Web data through a Folder-Path enabled extension of SPARQL.

The rest of the paper is organized as follows. In Section 2, we present some key components of our system, while in Section 3 we describe our demonstration scenario.

## 2. SYSTEM OVERVIEW

Figure 1 represents the architecture of the Galaxy. The main components of the system include the Extracted Data Folder and the Graph Query Engine.

**Extracted Data Folders.** Open data are complex, unstructured and generated at a high rate, resulting in many challenges to ingest, store, index, and analyze such data efficiently. The notion of *extracted data folders* serves to enable the ingestion of data (from open data sources), and the persistence of this data in accordance with a particular defined schema. Machine learning techniques can be used to construct the schema for an open data source [5]. We assume that the expert analysts will construct the schema for each folder. Figure 4 illustrates the Twitter schema where the main entities include users, tweets, links, domains, and hashTags. Folders provide a federated data access infrastructure upon which the federated analysis will operate. We also envisage folders to support multiple layers of granularity (e.g., split or merge existing folders). Folders can also be combined to create higher-level virtual folders, called federated folders, using filter, project and join operators.

**Graph Query Engine (SPARQL extension).** Due to space restrictions, in this paper we highlight the main component of the query engine. However, we refer to our paper [2] for algorithmic and other details. Figure 2 presents the graph processing architecture which consists of the following components: graph loader, data mapping layer, query mapping layer, regular expression processor, time-aware controller, and OLAP (on-line analytical processing) controller.

**Graph Loader.** Input graph (e.g. Twitter extracted folder) can be in the form of RDF, N3, or XML. We developed a workload physical design by developing a loader algorithm. This algorithm is responsible for: (i) validating the input graph; and (ii) generating the triples, where two types of triples are recognized: attribute-edges (e.g., “Bob @age 35”) and relationship-edges (e.g., “Bob knows Fred”). We use the ‘@’ symbol for representing attribute edges and distinguishing them from the relationship edges.

**Data Mapping Layer.** This layer is responsible for creating: (i) object-store, which contains all objects in the input graph uniquely identified by an identifier. Each object contains an arbitrary list of attribute-edges describing its features; (ii) link-store, which contains all directed links between pairs of objects represented as relationship-edges; and (iii) data element mappings between semantic web technology (i.e. Resource Description Framework) and Hadoop file system. As a result, object-store and link-store will be stored in Hadoop cluster.

**Query Mapping Layer.** This layer is consist of a parser for parsing SPARQL like queries (based upon the syntax of Folder-Path extension [2, 3, 4] of SPARQL) and a SPARQL-to-PigLatin translation algorithm. In order to translate the SPARQL queries into Pig-Latin we follow a specific format in which data is read from the HDFS, a number of Pig-Latin operations (e.g., LOAD, SPLIT, JOIN, FILTER, GROUP, and STORE) are performed on the data, and then the resulting relation is written back to the file system. In particular, SPARQL graph pattern matching is dominated by join operations, and is unlikely to be efficiently processed. We use existing query optimization techniques [7, 8, 9] to generate the optimal query plan by reinterpreting certain join tree structures as grouping operations, i.e., to enable a greater degree of parallelism in join processing. In the

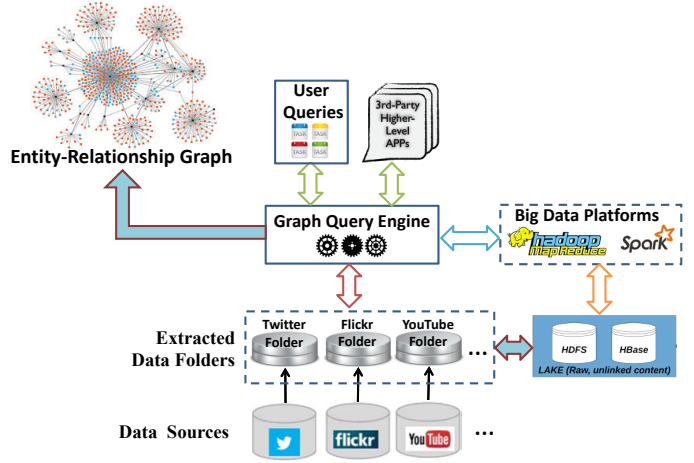


Figure 1: The Galaxy Architecture.

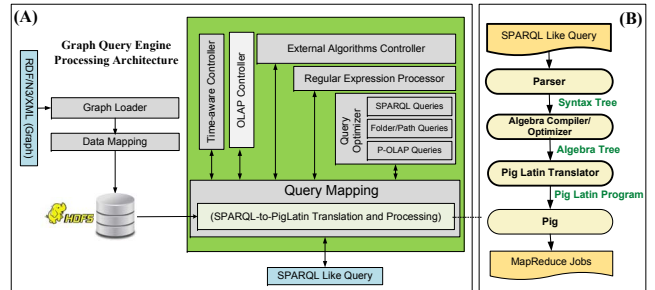


Figure 2: The Query Engine Architecture.

following, we illustrate an example for a sample mapping between SPARQL and Pig-Latin.

*Example 1.* DBLP<sup>1</sup> is an open source data for computer science bibliographical network. Adam, an OLAP analyst, is interested in partitioning the DBLP graph into a set of authors having same interests. Then he plans to apply a set of OLAP style operations (e.g., calculating authors ranking and contribution degree) on constructed partitions. Details about this example, including the SPARQL query can be found in [2]. Processing this query using Pig Latin’s query algebra, results in the query plan shown in Figure 3. The logical plan can be described as follows: (1) load the input dataset using the LOAD operator in Pig-Latin; (2) split the dataset, based on the partitioning condition, and create triple tables for related predicates. Next step is to filter the dataset into related authors, where the ‘interest’ triplestore will be needed for the partitioning phase and ‘publications’ and ‘citations’ triplestores will be needed to apply OLAP style operations on partitions; (3) filter the graph using the result of previous step, i.e., to support the triple syntax and weave the predicated to related partitions. Notice that, in the case of using JOIN operator in this step, the triple syntax will be no longer available; (4) group by the interest table on the object column to remove redundant values, e.g., cases where two or more authors, different subjects, having same interests; (5) evaluation of OLAP operations on graphs independently for each partition, providing a natu-

<sup>1</sup><http://dblp.uni-trier.de/db/>

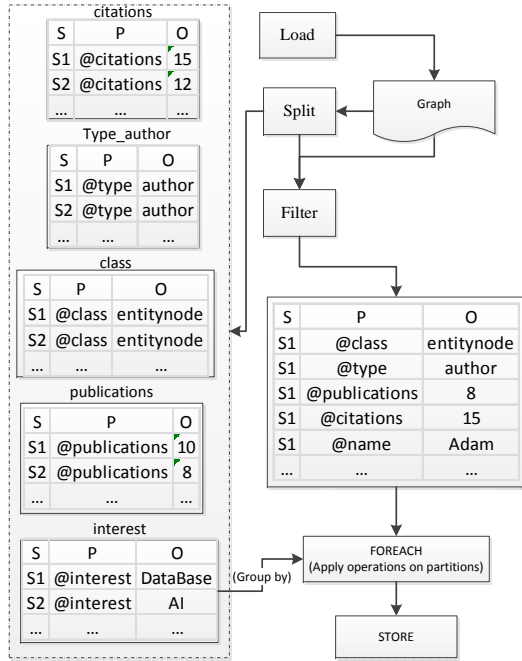


Figure 3: Query plan for the Example 1.

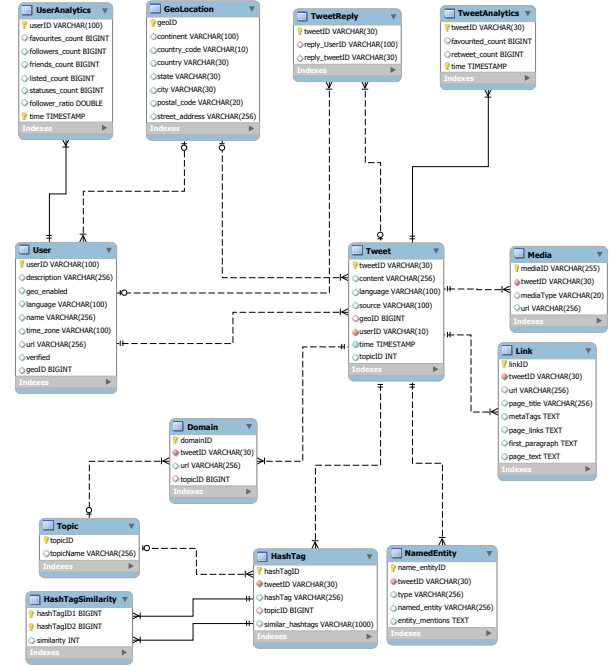


Figure 4: Twitter Extraction Folder Schema.

ral parallelization of execution; and (6) store the final result on Hadoop cluster using the STORE operator in Pig-Latin.

**Regular Expression Processor.** This component is responsible for parsing graph patterns. In particular, graph analysts can codify their knowledge into regular expressions that describe paths through the nodes and edges in the graph. The regular expression processor supports optional elements (?), loops (+, \*), alternation (|), and grouping (...).

**Time-aware Controller.** RDF databases are not static and changes may apply to graph entities (i.e. nodes, edges, and folder/path nodes) over time. Time-aware controller is responsible for data changes and incremental graph loading. Moreover, it creates a monitoring code snippet and allocate it to a folder/path node in order to monitor its evolution and update its content over time.

**GOLAP controller.** This component is responsible for supporting on-line analytical processing on graphs, through partitioning graphs (using folder and path nodes) and allows evaluation of OLAP operations on graphs independently for each partition, providing a natural parallelization of execution, details can be found in [2].

**External Algorithms Controller.** This component is responsible to support applying existing graph mining algorithms (e.g. graph reachability and shortest path) to the open data graph, and store the result in a folder/path node for the follow on analysis. For example we developed interfaces to support various graph mining algorithms [1] such as Transitive Closure, GRIPP, Tree Cover, Chain Cover, Path-Tree Cover, and Shortest-Paths.

### 3. DEMONSTRATION SCENARIO

The demonstration scenario consists of three parts. First, we would like that the attendee appreciates the difficulties that one can encounter when dealing with open data sources.

We start with a Twitter dump<sup>2</sup> and illustrate how we generate the Twitter extraction folder according to the generated Twitter schema (Figure 4). Next, we illustrate how we use the query language to construct content-based relationships among related entities. When talking about content we mainly deal with entity attributes, where we consider content-based relationships as correlation condition-based relationships. For example, a correlation condition in Twitter may enable grouping entities in different ways, e.g. Tweets coming from the same location or users from the same timezone, and store them in folder nodes. Figure 5 presents the set of related tweets whose location country is the same as Australia.

Next, we present to the attendee an interactive scenario where she would be able to generate the ‘influence graph’ among users in the Twitter open data through the following steps: (i) Using Folder Nodes, to form an intelligence picture from the Twitter data around a topic of interest (i.e. Twitter User) by grouping related entities around that topic and store them in folder nodes. Examples are, folders for users who: (a) belong to the same location, (b) tweet the same topics (we assume that we have a topic discovery algorithm), (c) use similar hashTags/links in their tweets; and (d) retweet similar tweets; (ii) Using Path Nodes to construct relationships among twitter constructed folders. For example a reachability algorithm will be used (in path node queries) to see if two different twitter users are reachable through a shard friend or a retweet path. As the result set of related patterns can be stored in path nodes for further analysis; (iii) Constructing relationships among open/federated data sources. We will provide the attendee with a set of folders constructed from other open data sources such as Wiki-data and DBpedia. The attendee will use query templates

<sup>2</sup><https://archive.org/details/archiveteam-twitter-stream-2012-02>

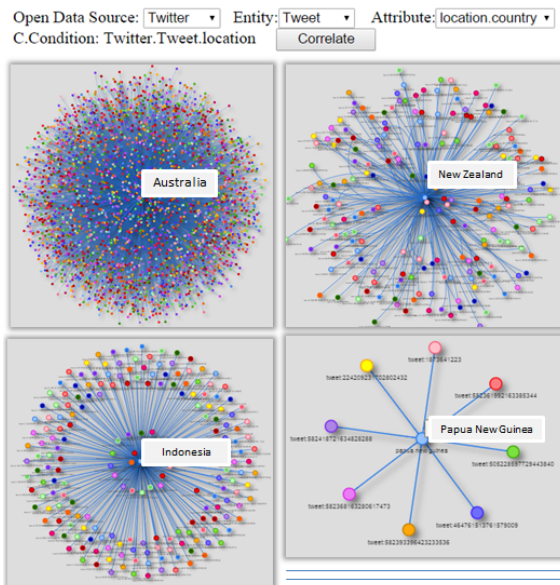


Figure 5: An example of partitioning the graph for the follow on analysis.

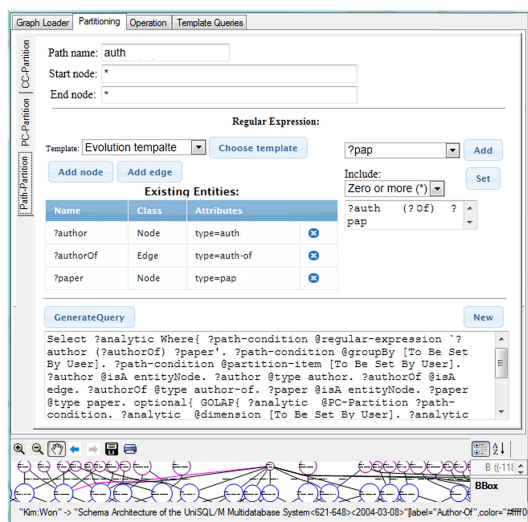


Figure 6: Screenshots of the front-end tool: assisting user to generate regular expressions.

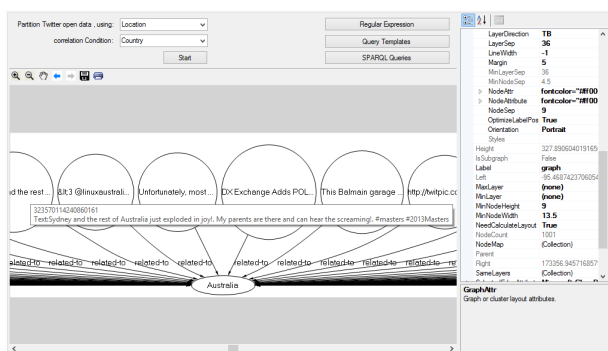


Figure 7: Screenshots of the front-end tool: assisting user to generate correlation conditions.

to discover similarity among different folders, e.g. a tweet in Twitter folder can be related to a topic in Wikidata folder; and (iv) Generating the ‘influence graph’ among users in Twitter. The attendee will use SPARQL like queries to further analyze the folder and path nodes and use the front-end tool to visualize the influence graph. In order to facilitate creating SPARQL queries, we provide a front-end tool for assisting users to create SPARQL queries in an easy way. Figures 6 and 7 illustrates screenshots of the front-end tool,

#### 4. CONCLUSION AND FUTURE WORK

In this paper, we presented Galaxy, a platform for explorative analysis of Open Data Sources. Galaxy assists the analysts to quickly form an intelligence picture from the Open Data sources around a topic of interest, group related entities (path nodes), find paths among entities (path nodes), and use all these information for the follow on analysis. Galaxy uses Hadoop data processing platforms to store and retrieve large numbers of RDF triples in Hadoop file system. As future work, we plan to make use of interactive graph exploration and visualization techniques which can help users to quickly identify the interesting parts of a graph.

#### 5. ACKNOWLEDGEMENTS

We Acknowledge the Data to Decisions CRC (D2D CRC) and the Cooperative Research Centres Programme for funding part of this research.

#### 6. REFERENCES

- [1] C. C. Aggarwal and H. Wang. *Managing and Mining Graph Data*. Springer Publishing Company, Incorporated, 2010.
- [2] S.-M.-R. Beheshti, B. Benatallah, and H. Motahari-Nezhad. Scalable graph-based olap analytics over process execution data. *Distributed and Parallel Databases*, pages 1–45, 2015.
- [3] S.-M.-R. Beheshti, B. Benatallah, and H. R. M. Nezhad. Enabling the analysis of cross-cutting aspects in ad-hoc processes. In *CAiSE*, pages 51–67, 2013.
- [4] S.-M.-R. Beheshti, B. Benatallah, H. R. M. Nezhad, and S. Sakr. A query language for analyzing business processes execution. In *BPM*, pages 281–297, 2011.
- [5] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *ACM Sigmod Record*, volume 30, pages 509–520. ACM, 2001.
- [6] M. Fayzullin, V. Subrahmanian, M. Albanese, C. Cesarano, and A. Picariello. Story creation from heterogeneous data sources. *Multimedia Tools and Applications*, 33(3):351–377, 2007.
- [7] M. F. Husain, L. Khan, M. Kantarcioglu, and B. M. Thuraisingham. Data intensive query processing for large rdf graphs using cloud computing tools. In *IEEE CLOUD*, pages 1–10, 2010.
- [8] H. Kim, P. Ravindra, and K. Anyanwu. From sparql to mapreduce: The journey using a nested triplegroup algebra. *PVLDB*, 4(12):1426–1429, 2011.
- [9] P. Ravindra, H. Kim, and K. Anyanwu. An intermediate algebra for optimizing rdf graph pattern matching on mapreduce. In *ESWC*, pages 46–61, 2011.

# OAPT: A Tool for Ontology Analysis and Partitioning

Alsayed Algergawy<sup>1\*</sup>, Samira Babalou<sup>2</sup>, Friederike Klan<sup>1</sup>, Birgitta König-Ries<sup>1</sup>

<sup>1</sup>Institute of Computer Science  
Friedrich Schiller University of Jena, Germany  
firstname.lastname@uni-jena.de

<sup>2</sup> Department of Computer Engineering  
University of Science and Culture, Iran  
s.Babaloo@son.ir

## ABSTRACT

Ontologies are the backbone of the Semantic Web and facilitate sharing, integration, and discovery of data. However, the number of existing ontologies is vastly growing, which makes it is problematic for software developers to decide which ontology is suitable for their application. Furthermore, often, only a small part of the ontology will be relevant for a certain application. In other cases, ontologies are so large, that they have to be split up in more manageable chunks to work with them. To this end, in this demo, we present *OAPT*, an ontology analysis and partitioning tool. First, before a candidate input ontology is partitioned, *OAPT* analyzes it to determine, if this ontology is worth to be considered using a predefined set of criteria that quantify the semantic richness of the ontology. Once the ontology is investigated, we apply a seeding-based partitioning algorithm to partition it into a set of modules. Through the demonstration of *OAPT* we introduce the tool's capabilities and highlight its effectiveness and usability.

## Categories and Subject Descriptors

H.4 [Information Systems]: WWW, web applications;  
H.4 [Information Systems Applications]: Data mining

## Keywords

Semantic Web, ontology, modularization, analysis

## 1. INTRODUCTION

Ontologies are the backbone of the Semantic Web. By making information understandable for machines [7] they enable integrating, searching, and sharing of information on the Web. The growing value of ontologies has resulted in the development of a large number of these. According to [3], at least 7000 ontologies exist on the Semantic Web, providing

\*Department of Computer Engineering, Tanta University, Egypt

an unprecedented set of resources for developers of semantic applications. On the other hand, this large number of available ontologies makes it hard for software engineers to decide which ontology(ies) is (are) suitable for their needs. Even, if a developer settled on an ontology (or a set of ontologies), she is most often interested in a subset of concepts of the entire ontology, only. For example, the CHEBI ontology<sup>1</sup>, contains 46,477 fully annotated concepts describing chemical entities of which not all will be relevant to a specific application. Also, it might be necessary to split up large ontologies like CHEBI in more manageable chunks before feeding them to ontology matching tools or other applications.

To cope with these challenges, in this demo paper, we present *OAPT*, a tool for analyzing and partitioning ontologies. The tool allows the user to interactively investigate a candidate input ontology based on a predefined set of quality criteria. This will help to build trust for sharing and reusing ontologies. Once an ontology has been analyzed, the partitioning algorithm can be applied to partition the ontology into a set of disjoint modules. Our method to examine the ontology quality is based on the consistency and richness of the input ontology. First, a suitable reasoner is applied to the ontology to validate its consistency. It is clear that the way an ontology is engineered is largely based on the domain for which it is designed and modeled. Therefore, a measure for the semantic richness of an ontology should consider different aspects and its potential for knowledge representation [9]. To this end, we then consider a set of structural, semantic, and syntactic metrics. The structural and syntactic criteria can be used to quantify the ontology design and its potential for knowledge representation, while the semantic-based criterion can be used to evaluate how instances are placed within the ontology.

To partition the analyzed ontology into a set of disjoint modules, we introduce a seeding-based clustering approach, called *SeeCOnt*. In particular, input ontologies are parsed and represented as concept graphs. A *Ranker* function is then used to rank ontology concepts exploiting the concept graph features. The highest ranked concepts are finally selected as cluster seeds (cluster heads). Each of these constitutes the initial concept of a resulting module. To assign the remaining concepts to their proper modules, we introduce a membership function. This reduces the complexity of the comparisons by comparing concepts with only seeds instead of all other concepts. Please note that this partitioning method is independent of the concrete application or a concrete subset of concepts a user is interested in. Rather, it

<sup>1</sup><https://www.ebi.ac.uk/chebi/>



relies on intrinsic ontology characteristics only. This allows, e.g., precomputation of the modules.

The rest of the paper is organized as follows. In Section 2 we present an overview of the proposed system, while in Section 3 we describe our demonstration scenario. Due to space restrictions, in this paper we provide only a glimpse of the techniques employed by *OAPT*. However, we refer to our full research paper [1] for algorithmic details and for more elements of related work.

## 2. THE TOOL OVERVIEW

First, input ontologies are parsed using the Apache Jena<sup>2</sup> framework. Then, a concept graph is extracted. We define the *concept graph*  $\mathcal{G} = (\mathcal{C}, \mathcal{R}, \mathcal{L})$  as a labeled directed graph.  $\mathcal{C} = \{c_1, c_2, \dots, c_n\}$  is a finite set of nodes representing the concepts of the ontology.  $\mathcal{R} = \{r_1, r_2, \dots, r_m\}$  denotes a finite set of directed edges representing various relationships between concepts in an ontology  $\mathcal{O}$ , such that  $r_k \in \mathcal{R}$  represents a directed relation between two adjacent concepts  $c_i, c_j \in \mathcal{C}$ .  $\mathcal{L}$  is a finite set of labels of graph nodes and edges defining the properties of each entity, such as the names of concepts.  $n(= |\mathcal{C}|)$  and  $m$  are the number of nodes (concepts) and edges (relationships) in  $\mathcal{G}$ , respectively.

After an input ontology has been parsed and represented as a concept graph, the *OAPT* tool analyzes the ontology based on a predefined set of criteria. If the user is satisfied with the result of the analysis, the ontology is then partitioned into a set of modules, as shown in Fig. 1.

### 2.1 Ontology analysis

In order to build trust for an ontology as a prerequisite for reuse and sharing, we evaluate and analyze the quality of ontologies. We start the analysis process by applying an OWL reasoner to make sure that the input ontology is consistent. As it is known that the way an ontology is engineered is specific to the domain for which it has been designed and modeled, the ontology design and its potential to represent knowledge should be examined. To this end, we then use the ontology richness as a metric for its quality. The richness of the ontology considers different aspects of ontologies and their potential for knowledge representation [9]. We categorize measures for the richness of an ontology into three categories: *structural*, *semantic*, and *syntactic*. We exemplarily present some implemented metrics for each type in detail:

- **Structural richness.** This dimension describes the topology of the concept hierarchy of an ontology. It includes several criteria, such as relationship, attribute, depth, and connection richness. The **Relationship richness, RR**, reflects the variability in the types of relations and the placement of these relations within the ontology. An ontology that contains numerous relation types other than class-subclass relations is richer than a taxonomy with just class-subclass relations. The relation richness (RR) of an ontology  $\mathcal{O}$  can be defined as:  $RR(\mathcal{O}) = \frac{|R| \cdot |SC|}{|R|}$ , where  $|R|$  is the number of relationships in the ontology, and  $|SC|$  is the number of sub-class relations. The value of the relation richness criterion is normalized between 0 and 1, where the value of 0 means that the ontology contains only sub-class relationships. Another criterion

that can be used to evaluate the structural richness of an ontology is the **connection richness, ConnR**. It indicates the number of connected components of the concept graph, i.e., the number of subgraphs linked to its root element. So, for calculating *ConnR* we determine the number of root classes, i.e., the children of the root node.  $ConnR(\mathcal{O}) = \frac{1}{No\_root\_classes}$ . This metric can help to avoid "islands" forming in a knowledge base as a result of extracting data from separate sources that do not have common knowledge.

- **Semantic & Syntactic richness.** These two dimensions describe the semantics and the descriptive information of the ontology. In this context, we make use of several metrics, such as class richness and readability [9]. The **Class Richness, CR**, is an instance-based criterion used to reflect how instances in an ontology are distributed across classes. The class richness (CR) criterion can be defined as follows:  $CR(\mathcal{O}) = \frac{|C^I|}{|C|}$  where  $|C^I|$  is the number of classes that have instances. Another criterion that is important during the evaluation of the semantic richness of an ontology is the **descriptivity richness, DR**. This measure indicates availability of human-readable knowledge provided by an ontology. The descriptivity of an ontology can be defined as the number of concepts that have comments and/or labels:  $DR(\mathcal{O}) = \frac{|C'|}{|C|}$  where  $|C'|$  is the number of concepts having comments and/or labels.

We combine these three dimensions to compute the total richness of an ontology using a simple weighted-sum approach. Therefore, the ontology richness (OR) criterion is defined as follows:

$$OR(\mathcal{O}) = w_1 \times StrR(\mathcal{O}) + w_2 \times SemR(\mathcal{O}) + w_3 \times SynR(\mathcal{O}) \quad (1)$$

where  $StrR(\mathcal{O})$ ,  $SemR(\mathcal{O})$ , and  $SynR(\mathcal{O})$  are the total structural, semantic, and syntactic richness of the ontology ( $\mathcal{O}$ ), respectively.  $w_1, w_2, w_3$  are weights that reflect the importance of each of the richness metrics, such that  $\sum w_i = 1$ . The normalized score is then listed for the user to decide whether to partition the ontology or to look for another one.

### 2.2 Ontology partitioning

Once the input ontology is inspected, the next step is to partition the concepts  $\mathcal{C}$  of the concept graph  $\mathcal{G}$  into a set of separate (disjoint) modules  $\mathcal{M}_1, \mathcal{M}_2, \dots, \mathcal{M}_k$  such that the *cohesion* of concepts within modules is high, while the *coupling* across modules is low. To this end, we develop a seeding-based clustering algorithm. The steps of the algorithms are described in the following:

#### 2.2.1 Determining the proper number of modules

Typically, the number of modules a given ontology should be split up in is determined by trial and error without using objective criteria [6]. In contrast, the *OAPT* tool provides two options to determine the number of modules. First, if the user has enough experience with the ontology to be partitioned, she can directly input the number of modules. Otherwise, the user asks the tool to suggest an "optimal" number of modules.

#### 2.2.2 Ranking the concepts

<sup>2</sup><https://jena.apache.org/>

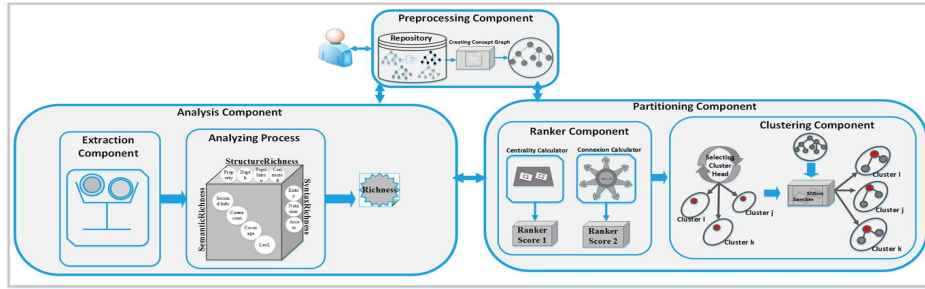


Figure 1: Tool overview

The seeding-based algorithm starts by selecting a set of nodes distinguished as *important nodes*. These nodes are then selected to be cluster heads,  $\mathcal{CH}$ . To quantify the role a node has within the concept graph, we introduce a new function, called *Ranker*. This function should be as simple as possible but effective. I.e. computing the *Ranker* function should not consume much time, however, correctly rank the concepts inside an ontology.

### Ranker Function.

The importance of a node in a concept graph is determined by the properties of the node itself and its surroundings [5, 10]. This leads us to use graph-theoretic measures based on graph connections in the *Ranker* function. In the current software, we included two different implementations of the *Ranker* function. The first is based on the *centrality* measure of a concept, while the second depends on the context of the concept. To consider the effect of the concept itself through its edges, we use a set of centrality measures [4]. During the employment of the first *Ranker* function, we observed that it is an effective measure but requires a lot of time to rank concepts. This makes it unsuitable for partitioning large ontologies. Therefore, we propose another ranking function, which is based on the *connexion set* concept. The *connexion set*  $\Psi(c_i, d)$  of a concept  $c_i \in \mathcal{C}$  is defined as:  $\Psi(c_i, d) = \{SubClass(c_i, d) \cup SuperClass(c_i, d)\}$ , where  $\Psi(c_i, d)$  is the set of all concepts within  $d$  levels that effect on  $c_i$ .  $SubClass(c_i, d)$  is the set of children of  $c_i$  within  $d$  hierarchical levels, and  $SuperClass(c_i, d)$  is the set of parents of  $c_i$  within  $d$  hierarchical levels. It is evident that the importance of a concept increases as it has a larger number of surrounding nodes.

### 2.2.3 Determining cluster heads.

Once having computed the importance of the concepts of a concept graph, the next step is to select which concepts represent cluster heads,  $\mathcal{CH}$ . If simply the nodes with the highest score are selected as the cluster heads, the distribution of cluster heads across the concept graph would be disregarded. To avoid this problem, the distance between two cluster heads is measured, and among the highest scored nodes, those with a minimum distance  $\mathcal{D}$  from each other are selected as the cluster heads.

### 2.2.4 Finalizing Clustering

The seeding-based algorithm creates one module per cluster head. Then, it places direct children in the corresponding cluster and finally, for the remaining nodes, a membership function is used to determine the cluster each node shall

be assigned to. The direct placement of children reduces the time complexity, since it reduces the number of comparisons by avoiding to compute the membership function for all concepts.

### Membership Function.

Once having determined the cluster heads, ( $\mathcal{CH}$ ), and having assigned direct children to their proper heads, the next step is to place the remaining concepts into the fitting cluster. To this end, we developed a membership function, *MemFun*. First, each concept is associated with a flag,  $\mathcal{F}$ , such that if  $\mathcal{F}$  of the concept  $c$  is false, it means  $c$  is not assigned to any cluster yet and thus, the membership function has to be called for the concept  $c$ . In addition, the  $\mathcal{F}$  flag can be set only once, i.e. each node can be placed in only one cluster so that there is no overlap between clusters. The membership function determines for each concept  $c_i \in \mathcal{C}$  to which module  $\mathcal{M}_i, i < \mathcal{K}$  it shall be assigned. For this, the similarity of  $c_i$  with all  $\mathcal{CH}_i$  is calculated and then  $c_i$  is placed in a cluster with the maximum similarity value. Using the proposed membership function, each concept is compared with the Cluster Heads only, instead of comparing it to all concepts as usually done (see e.g. [2, 8]). This reduces the complexity of the algorithm.

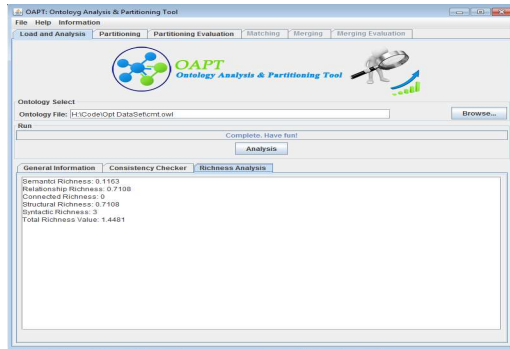
In order to compute the membership of a concept to a cluster head, a linear combination of structural and semantic similarity measures is calculated as follows:

$$MemFun(c_i, \mathcal{CH}_k) = \alpha \times SNSim(c_i, \mathcal{CH}_k) + (1 - \alpha) \times SemSim(c_i, \mathcal{CH}_k) \quad (2)$$

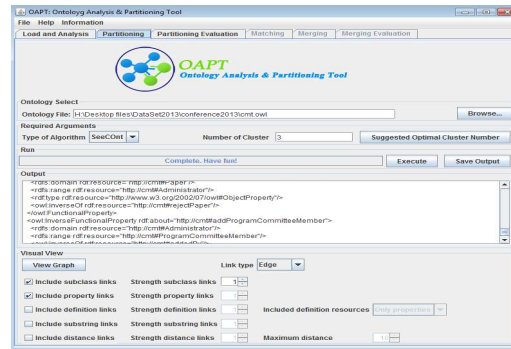
where  $\alpha$  is a constant between 0 and 1 that reflects the importance of each similarity measure, *ShareNeighbors*( $SNSim$ ) and semantic similarity *SemSim* are two similarity measures that quantify the structural properties of the concept  $c_i$ , respectively. The *SNSim* measure considers the number of shared neighbors of  $c_i$  and  $\mathcal{CH}_k$ . The neighbors of a concept are the concept's direct children, the concept's parents, the concept's siblings and the concept itself. While the *SemSim* measure considers the semantic connection between a concept and a cluster head, which is based on the concept hierarchy.

### 2.2.5 Partitioning analysis

After having partitioned an ontology into a set of modules, the *OAPT* tool analyzes the quality of the partitioning output. This step aims at monitoring the quality of the partitioning process. We implemented a set of quality metrics such as size, cohesion, coupling, and connectivity. The module size is used to check the number of classes and number



(a) Ontology analysis



(b) Ontology partitioning

Figure 2: OAPT Screenshots

of relations within the module to validate if it is adequate.

### 3. DEMONSTRATION SCENARIO

In this demonstration, we will start by presenting the different features of the *OAPT* tool<sup>3</sup> such as the process of analyzing an ontology (Figure 2a), partitioning the ontology, or sharing the partitioning quality (Figure 2b). The demonstration will consist of two main parts. First, we would like the user to appreciate the importance of the analysis phase. Second, we present the core of the *OAPT* tool.

#### Ontology analysis.

In this part, we aim at demonstrating the importance of the ontology analysis phase and how it largely affects the following steps. The user can select an ontology from the given repository and then start to study the effect of different evaluation criteria. The user starts with applying a single evaluation criterion and studies its effect on the semantic richness of an ontology. Then, the user attempts to combine different sets of the evaluation criteria to see why this set of metrics is needed.

#### Ontology partitioning.

In this part, we demonstrate the various steps of the ontology partitioning component. First, we allow the user to validate the importance of determining an optimal value for the number of modules. She starts with guessing a value for the number of modules an ontology should be partitioned into, and then she asks the tool to suggest such a number. Once the number of modules that an ontology should be partitioned into has been determined, the user can apply the *SeeCont* algorithm to get these modules. The set of the output modules as well as a set of evaluation metrics will be shown to the user to validate the quality of the partitioning algorithm.

### 4. CONCLUSIONS

In this demo, we show how *OAPT* can be used to investigate ontologies in a way that enables knowledge engineers to determine the quality of an ontology. Once an ontology is investigated, the tool can partition it into a set of disjoint modules. We developed and implemented a new seeding-based clustering approach. The tool has been evaluated and validated with ontologies from different domains which demonstrates the effectiveness and the usability of *OAPT*.

<sup>3</sup><http://fusion.cs.uni-jena.de/fusion/activity/oapt/>

This will be demonstrated in the sample workload that we prepare for the demo. In our future work, we will complete the tool in order to support the developers in selecting which module(s) fulfill(s) his requirements. Furthermore, we plan to improve the ontology analysis phase by considering more measures and criteria and to improve also the partitioning phase by taking into account other partitioning techniques. Furthermore, we plan to visualize all these processes and steps to be more user-interactive.

### 5. ACKNOWLEDGMENTS

This work is partly funded by DFG in the INFRA1 project of CRC 1067 AquaDiva.

### 6. REFERENCES

- [1] A. Algergawy, S. Babalou, M. J. Kargar, and S. H. Davarpanah. SeeCont: A new seeding-based clustering approach for ontology matching. In *ADBIS*, pages 245–258, 2015.
- [2] A. Algergawy, S. Massmann, and E. Rahm. A clustering-based approach for large-scale ontology matching. In *ADBIS*, pages 415–428. 2011.
- [3] M. d’Aquin, C. Baldassarre, L. Gridinoc, S. Angeletou, M. Sabou, and E. Motta. Characterizing knowledge on the semantic web with watson. In *5th International Workshop on Evaluation of Ontologies and Ontology-based Tools*, pages 1–10, 2007.
- [4] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 1997.
- [5] A. Graves, S. Adali, and J. Hendler. A method to rank nodes in an rdf graph. In *ISWC*, 2008.
- [6] G. Hamerly and C. Elkan. Learning the k in k-means. In *Advances in Neural Information Processing Systems*, pages 281–288, 2003.
- [7] J. Hendler. Agents and the semantic web. *IEEE Intelligent Systems Journal*, 16:30–37, 2001.
- [8] W. Hu, Y. Qu, and G. Cheng. Matching large ontologies: A divide-and-conquer approach. *Data Knowl. Eng.*, 67:140–160, 2008.
- [9] S. Tartir and I. B. Arpinar. Ontology evaluation and ranking using OntoQA. In *ICSC*, pages 185–192, 2007.
- [10] G. Troullinou, H. Kondylakis, E. Daskalaki, and D. Plexousakis. RDF digest: Efficient summarization of RDF/S kbs. In *ESWC*, pages 119–134, 2015.

# ShapeExplorer: Querying and Exploring Shapes using Visual Knowledge

Tong Ge<sup>1</sup>, Yafang Wang<sup>1\*</sup>, Gerard de Melo<sup>2</sup>, Zengguang Hao<sup>1</sup>, Andrei Sharf<sup>3</sup>, Baoquan Chen<sup>1</sup>  
<sup>1</sup>Shandong University, China; <sup>2</sup>Tsinghua University, China; <sup>3</sup>Ben-Gurion University, Israel

## ABSTRACT

With unprecedented amounts of multimodal data on the Internet, there is an increasing demand for systems with a more fine-grained understanding of visual data. ShapeExplorer is an interactive software tool based on a detailed analysis of images in terms of object shapes and parts. For instance, given an image of a donkey, the system may rely on previously acquired knowledge about zebras and dogs to automatically locate and label the head, legs, tail, and so on. Based on such semantic models, ShapeExplorer can then generate morphing animations, synthesize new shape contours, and support object part-based queries as well as clipart-based image retrieval.

## Keywords

Shape Knowledge Harvesting, Shape Matching, Shape Segmentation, Shape Synthesis

## 1. INTRODUCTION

In recent years, we have seen an explosion in the availability of multimodal data on the Internet, driven mostly by the ubiquity of mobile devices and online sharing platforms. Despite great advances in tasks such as object detection and tracking and multimedia retrieval, we still lack systems that provide more fine-grained semantic analyses of visual data.

In their widely noted work, Deng et al. [4] introduced ImageNet, a hierarchical organization of visual knowledge in raw images, according to semantic categories and relations. We take a further step in this direction and utilize the semantics of individual parts, subparts, and their shapes to facilitate their interpretation and manipulation. We present ShapeExplorer, an interactive software tool that analyzes images of objects and locates and labels specific object parts. For instance, given an image of a donkey, it can draw on previously analyzed images of related objects, e.g. of zebras or even just of dogs, to infer the location and labels of likely parts such as the head, legs, tail, and so on. An analysis in terms of parts is motivated by extant evidence from cognitive research on human vision showing that shape parts play an

important role in the lower stages of object recognition [9]. Seeing a small part of an object often suffices for a human to be able to recognize the object, provided that the part is sufficiently unique [3, 2]. Still, fine-grained shape understanding remains a challenging problem in computer vision. It appears that richer data is necessary so that systems can be equipped with the required background knowledge.

Independently from the developments in computer vision, there has been considerable progress on automatically constructing knowledge bases (KB), utilizing textual information to extract relational facts and attributes. Examples include YAGO [10, 12], DBpedia [1], Freebase (www.freebase.com), ConceptNet [7], and WebChild [11]. Often, the backbone of such KBs is a taxonomy of entity types or of part-whole relationships (e.g., Head *isPartOf* Horse).

In our work, we have constructed a visual knowledge base called PartNet, for object parts and their shapes. PartNet semantically describes objects in terms of their classes, parts, and visual appearance. Unlike regular KBs, it gathers examples of the shape contours of objects and object parts.

Based on this, ShapeExplorer provides several higher-level operations, including (partial) shape querying, semantic morphing, shape synthesis, and part-based image retrieval using cliparts.

## 2. FRAMEWORK

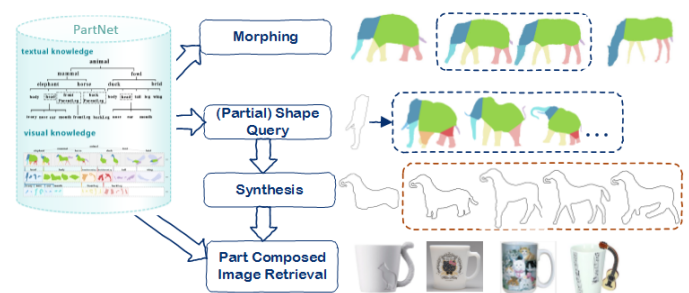


Figure 1: Flow diagram

**Hierarchical Part Exploration.** Figure 1 provides an overview of ShapeExplorer’s operational flow. The system is based on the PartNet knowledge repository, which users can explore hierarchically. This knowledge is also used in several applications such as morphing and querying.

PartNet is organized according to taxonomic categories (animals, dinosaurs, home appliances, etc.), sub-categories

\*Corresponding author: yafang.wang@sdu.edu.cn

(mammals, brontosaurus, chairs, etc.), and their part decompositions. At each level, the system presents corresponding shapes that the user may select and analyze. Internally, these are stored as subject-predicate-object triples, similar to regular knowledge bases, but including multimodal items.

There are two main user interfaces. Figure 3 presents a screenshot of the primary control center for part knowledge exploration. On the left side, the user can explore the knowledge in a convenient hierarchical tree based on categories (animals, home appliances, etc.), sub-categories (mammals, chairs, etc.), and their part decompositions. The right side of the screen serves as a working area. The users drags shapes into the bottom part of that area and can then select from several operations. These include an image analysis to infer the segmentation and labeling, which we describe below, as well as higher-level applications such as querying, morphing, and automated synthesis and completion (described in Section 3). Another user interface, shown in Figure 4, is used for part-based image retrieval. Users may compose a clipart-style query based on object parts and the system retrieves matching images from the database (see Section 3).

**Image Analysis.** ShapeExplorer’s image analysis is based on a joint inference procedure for joint classification, segmentation, and labeling, leveraging visual knowledge from previously analyzed images. In order to bootstrap this process in a particular domain, a small number of manually annotated seed images need to have been fed to the system initially. For those initial seeds, the user manually provides an image label, a segmentation, and part labels for the segments. The labels are chosen from the WordNet taxonomy [5]. For instance, the user could mark the image as portraying an *elephant*, and then the individual segments can be annotated with part labels such as *head*, *tail*, etc.

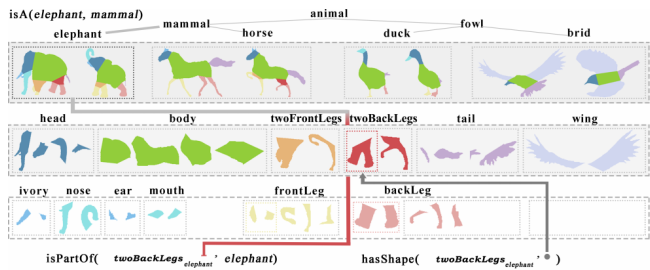


Figure 2: ShapeExplorer’s hierarchical organization

After that, one can progressively augment ShapeExplorer’s collected knowledge by adding new images, which the system analyzes using a transfer learning strategy. The system first generates a raw set of segmentation candidates, considering merely the image geometry using the short cut strategy [8]. It then matches the image with similar, previously seen ones using the inner-distance method [6], described in more detail later on in Section 3. From the top-3 matching images, we transfer additional candidate segmentations based on the contour alignments. The resulting set of candidate segmentations is pruned using semantic constraints.

Next, we determine label hypotheses for the image and for the parts based on the top-5 matching images in terms of the inner-distance method. For each cut/label candidate, we compute a confidence score based on the probability of the label within the top-5 matches and based on the cost

of the contour point alignment. Finally, we perform a joint optimization step, relying on an Integer Linear Program to maximize the sum of confidence scores for the chosen candidates subject to compatibility and cardinality constraints.

In Figure 2, we illustrate the kind of knowledge that ShapeExplorer collects. The general taxonomy comes from WordNet, while image representations are derived from the seeds and the subsequent joint inference procedure for new images.

**Implementation Details.** ShapeExplorer is implemented as a web application with a JavaScript-driven browser interface that users can access via their web browsers. The back-end is implemented in Java and includes the powerful PartNet shape part knowledge base as well as images indexed using Lire<sup>1</sup>.

### 3. APPLICATIONS

Based on the core framework, ShapeExplorer implements algorithms for several higher-level applications.

#### 3.1 (Partial) Shape Queries

Users may provide an input image with an unknown shape and then ShapeExplorer attempts to match it against known shapes, retrieving top-k matches from the repository. The input image may be partial (i.e., with occlusions or missing parts). Thus, Alg. 1 considers subsets of the parts of every shape class as possible candidates. This is restricted to parts that are adjacent and sufficiently large, in order to avoid a combinatorial explosion. We use the inner-distance method [6] for similarity computation, which we found to be efficient, rotation-invariant, and robust. Given two shapes  $A$  and  $B$ , described by their contour point sequences  $p_1, p_2, \dots, p_n$  and  $q_1, q_2, \dots, q_m$ , respectively, we use  $\chi^2$  statistics to compare point histograms, resulting in a cost value  $c(p_i, q_j)$ . Then we solve for the optimal matching between  $A$  and  $B$ , denoted as  $\pi : (p_i, q_{\pi(i)})$  using dynamic programming. We compute the minimum cost value as  $C(\pi) = \sum_{i=1}^n c(i, \pi(i))$  and the number of matching points as  $M(\pi) = \sum_{i=1}^n \delta(i)$ , where  $\delta(i) = 1$  if  $\pi(i) \neq \emptyset$ , and 0 otherwise. Finally, given a best matching shape, we define a segment cut, denoted as  $\text{cut}_A(p_i, p_j)$ , as the 2D line connecting contour points  $p_i, p_j$  in  $A$ . We use the computed shape matching  $\pi$  to map  $\text{cut}_A(p_i, p_j)$  onto the input shape  $B$  as  $\text{cut}_B(q_{\pi(i)}, q_{\pi(j)})$ . Thus knowledge from existing images is transferred onto new ones to classify them and annotate their parts.

---

#### Algorithm 1 (Partial) Shape Querying

---

**Input:** connected input part and shape database, candidate object classes  $\mathcal{C} = \{c_0, c_1, \dots, c_{n_c}\}$ , number of results  $k$   
**Output:** top-k matching shapes

- 1: **for** each *class*  $c_i \in \mathcal{C}$  **do**
- 2:   **for** each *part*  $p_j$  of  $c_i$  **do**
- 3:      $\text{partialShape}[] \leftarrow$  set of relevant combinations of parts of  $p_j$
- 4:     **for** each *part*  $ps_z \in \text{partialShape}[]$  **do**
- 5:        $\text{cost}[ps_z] \leftarrow$  matching cost  $C(\pi(\text{part}, ps_z))$
- 6:      $\text{shape}[] \leftarrow$  ranking of shapes in  $\text{cost}[]$  according to cost values
- 7: **return** top-k entries in  $\text{shape}[]$

---

In Figure 3, the user selects an elephant head (blue) and a horse body (green) and synthesizes them together in the bottom right input region. This new shape is converted into a simple contour and given as input to ShapeExplorer.

<sup>1</sup><http://www.lire-project.net/>

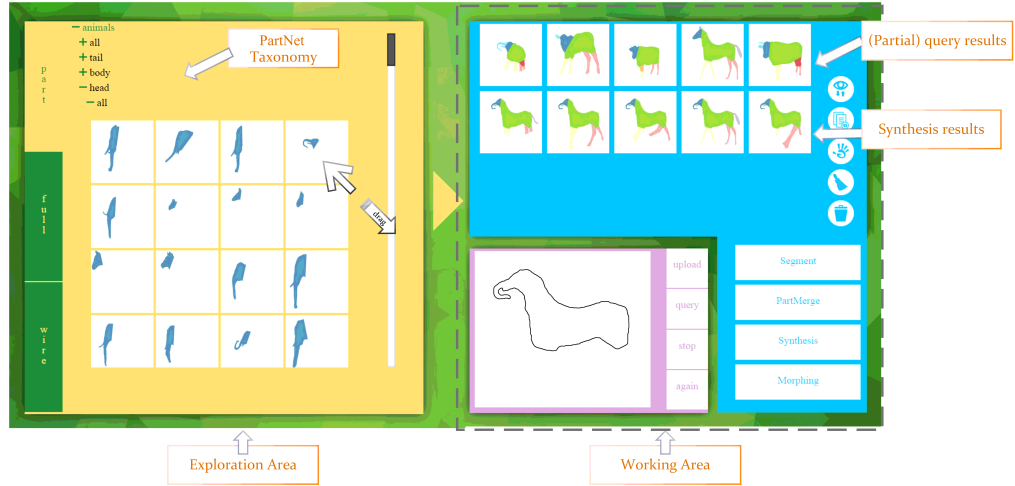


Figure 3: Screenshot for part knowledge exploration



Figure 4: Screenshot for part-based clipart image retrieval

In partial query mode, the top-5 most similar results with respect to their partial similarity are shown in the top row on the right. These can be used to retrieve images with objects showing similar shapes.

### 3.2 Semantic Morphing

Given two images, ShapeExplorer can automatically perform a semantic form of morphing by classifying and performing a conformal joint segmentation of the two images. Alg. 2 first produces a segmentation of the two images into the same meaningful parts. This task is accomplished by searching for the lowest common cuts in the part hierarchies (e.g., a joint segmentation of a horse and elephant will return heads without the unique trunk of the elephant).

---

#### Algorithm 2 Part Based Morphing

---

**Input:**  $shape_1$  and  $shape_2$ , part labels  $\mathcal{L} = \{l_0, l_1, \dots, l_n\}$

**Output:** morphing animation sequence

- 1:  $part_1[] \leftarrow shapeSegmentation(shape_1)$
  - 2:  $part_2[] \leftarrow shapeSegmentation(shape_2)$
  - 3: align  $part_2[]$  to  $part_1[]$  using common parts
  - 4: **for** each label  $l_i \in \mathcal{L}$  **do**
  - 5:      $point_1[l_i][] \leftarrow samplePoints(part_1[l_i])$
  - 6:      $point_2[l_i][] \leftarrow samplePoints(part_2[l_i])$
  - 7: **return**  $morphing(shape_1, shape_2, point_1[], point_2[])$
- 

Having a full part correspondence between the two shapes, our system generates a morphing sequence which gradually interpolates from one to the other. See the morphing sequence in Figure 1 for an example of morphing from elephant to horse. This is accomplished by performing a per-part morphing while maintaining connectivity between adjacent parts during the deformation.

The morphing algorithm generates animations by smoothly interpolating transitions between corresponding parts in each shape (see morphing sequence in Figure 1). During the animation, users may pause and resume it to review intermediate frames, which can also serve as new inputs for querying and synthesis operations. Additionally, the user may select the intermediate frames as new inputs, which may further be queried, synthesized, and used to retrieve images.

### 3.3 Shape Synthesis and Completion

In shape synthesis mode, ShapeExplorer starts with a new user-provided partial shape and then uses best matching shapes from the repository to synthesize the missing parts so as to obtain a complete image.

Given an unknown shape, Alg. 3 first finds the top-1 best matching shape in the repository using the (Partial) Shape Query method. The segmentation and labels of the matching

shape are transferred to the input image. Then, missing parts in the input shape with respect to the matched shape are detected. We synthesize the missing parts by transferring them from the matched shape onto the input image. Specifically, we subtract from the retrieved shape the parts in common with the input one and gracefully translate, scale, and rotate the shape and its parts so that they fit to their adjacent ones in the unknown one. Please note that *body* might have many part-cut labels, such as head, leg and tail. Therefore, line 6 means whether  $p_i$  and  $p_j$  have matching part-cut labels. For example, head can match body.

---

### Algorithm 3 Shape Synthesis

---

**Input:** query *shape*, shape database  
**Output:** new shape

```

1:  $rParts[] \leftarrow$  parts of top-1 partial query result for shape (Alg. 1)
2:  $parts[] \leftarrow$  labeled segmentation of shape via Alg. 1
3:  $mParts[] \leftarrow rParts[] - parts[]$ 
4: for each part  $p_i \in mParts[]$  do
5:   for each part  $p_j \in parts[]$  do
6:     if  $label(p_i)$  matches  $label(p_j)$  then
7:       transfer from  $p_i$  to  $p_j$  in shape
8: return shape

```

---

In Figure 3, the user selects an elephant head (blue) and a horse body (green) and synthesizes them together in the bottom right input region. Synthesis results are displayed in the second row of the results region in the work area. These results can be used for retrieving similar images.

### 3.4 Clipart-based Image Retrieval

Another option is to perform image retrieval from a large image repository by composing a clipart-like query using parts or sketches. Our multimodal retrieval interface is composed of four components (see Figure 4): a control panel on the left, a text query field on the top, the working canvas in the middle, and the retrieval results on the right. Figure 5 illustrates the workflow of the part-based clipart image retrieval system. Users can issue textual queries to retrieve parts of interest from the PartNet knowledge repository. They can explore the results and drag parts of interest into the working area so as to craft a query image. This query image may be composed of parts stemming from different objects in the repository. Users also are able to modify the composition by drawing sketches using a pencil tool in conjunction with a color selection interface. This can be used to add additional items to the image, or to modify the original colors and textures of the parts coming from PartNet. An eraser tool is also provided for cleaning.

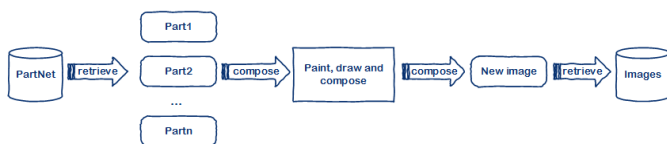


Figure 5: Part-based clipart image retrieval

Finally, the query image is used to retrieve similar images from the database. In the example in Figure 4, the user seeks to find images of cups with cat tails as handles. With standard image retrieval tools, it is hard to find such images unless they have sufficient textual metadata. With ShapeExplorer, the user can first issue a query for the word “cup” to find the cup body in the PartNet repository. This is then

dragged from the results area onto the canvas. Similarly, the user finds the body and legs of a cat in PartNet and incorporates them into the query canvas. In order to ensure that the handle looks like a cat tail, the user can pick the color brown via the color selection interface and use the pencil tool to sketch a brown handle. This results in a sort of clipart image that can be used to retrieve matching real images from the database. For image matching, ShapeExplorer relies on a set of image features, combining JCD (Joint Composite Descriptor) and edge histograms. Thus, we can find cups with cat-like handles. Figure 4 shows the top-3 results, and the user may scroll to obtain further images.

## 4. CONCLUSION

In this paper, we have presented ShapeExplorer, a system aimed at fine-grained analyses of images in terms of object parts that captures multimodal knowledge in more detail than previous work. We see that explicit semantic representations of the parts enable several novel applications, including novel forms of querying, semantics-driven morphing, and synthesis.

## Acknowledgments

We thank the anonymous reviewers for their valuable comments, and Kang Feng and Wei Wu et al. for their hard work preparing the dataset. This project was sponsored by National Natural Science Foundation of China (No. 61503217), Shandong Provincial Natural Science Foundation of China (No. ZR2014FP002), and The Fundamental Research Funds of Shandong University (No. 2014TB005, 2014JC001).

## 5. REFERENCES

- [1] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, and Z. Ives. DBpedia: A nucleus for a web of open data. In *Proc. ISWC*, 2007.
- [2] I. Biederman. Recognition-by-components: A theory of human image understanding. *Psychological Review*, 94:115–147, 1987.
- [3] T. O. Binford. In *Proc. IEEE Conf. Systems & Control*.
- [4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Proc. CVPR.*, 2009.
- [5] C. Fellbaum. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.
- [6] H. Ling and D. Jacobs. Shape classification using the inner-distance. *IEEE PAMI*, 2007.
- [7] H. Liu and P. Singh. ConceptNet: A practical commonsense reasoning toolkit, 2004.
- [8] L. Luo, C. Shen, X. Liu, and C. Zhang. A computational model of the short-cut rule for 2D shape decomposition. *CoRR*, abs/1409.2104, 2014.
- [9] D. Marr. Early processing of visual information. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, 275(942):483–519, 1976.
- [10] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A Core of Semantic Knowledge. In *Proc. WWW*, 2007.
- [11] N. Tandon, G. de Melo, F. Suchanek, and G. Weikum. WebChild: Harvesting and organizing commonsense knowledge from the web. In *Proc. ACM WSDM*, 2014.
- [12] Y. Wang, B. Yang, L. Qu, M. Spaniol, and G. Weikum. Harvesting facts from textual web sources by constrained label propagation. In *Proc. CIKM*, 2011.

# Distributed Secure Search in the Personal Cloud

Thu T.B. Le<sup>1,3</sup>, Nicolas Anciaux<sup>1,2</sup>, Sébastien Gilloton<sup>1</sup>, Saliha Lallali<sup>1,2</sup>,  
Philippe Pucheral<sup>1,2</sup>, Iulian Sandu Popa<sup>1,2</sup>, Chao Chen<sup>1,2</sup>

<sup>1</sup>INRIA, France  
FirstName.LastName@inria.fr

<sup>2</sup>U. Versailles St-Q. en Y., France  
FirstName.LastName@uvsq.fr

<sup>3</sup>HCMC U. of Technology, Vietnam  
thule@hcmut.edu.vn

## 1. INTRODUCTION

The Personal Cloud paradigm emerges as a decentralized and privacy preserving solution to manage personal documents under users' control. It can be seen as an alternative to the current Web model, which centralizes the complete digital life of millions of individuals in data silos, and increases frustration generated by the weak control of the individuals on the way their personal data are shared, used and disseminated. The home cloud is the most emblematic form of Personal Cloud. It can be thought of as a dedicated box connected to the user's internet gateway, equipped with storage, computing and communication facilities [11], running a personal server and acquiring data from multiple sources [3]. This personal server is in charge of organizing the personal dataspace in a document database style to ease its management and to protect it against loss, theft and abusive use. Many startups (e.g., OwnCloud, CozyCloud, etc.) and research projects (e.g., PlugDB at Inria or OpenPDS at MIT) investigate this direction.

To make the vision of the Personal Cloud reality, two important challenges related to data management have to be considered. First, leaving the data management control into the user's hands pushes the security issues to the user's computing platform as well. Hence, besides the management of collections of personal documents of any type, the personal cloud takes the security and privacy in charge. This requires protecting personal documents and their metadata by means of encryption and evaluation of access control rules. This challenge is paramount considering that the Personal Cloud paradigm puts a significant part of the digital life of the individual in the user's hands.

The second challenge is rooted in the high level of decentralization of the Personal Cloud, i.e., each user owns her personal cloud (see Figure 1). Indeed, this user-centric architecture must not hinder the development of global data services of great interest for the individuals, the companies and the community, which is also required to meet a viable business model. Hence, as in a centralized setting, certain applications require crossing data from multiple individual Personal Clouds. This is the case of any application developed for communities of users sharing a similar interest. For example, within a community of patients suffering from the same pathologies, each participating user may provide her own set of information such that distributed searches may help to identify within the community the most relevant documents related to current treatments or symptoms, or more generally, help users to share and benefit from each other experiences. Clearly, this must be performed without exposing the privacy of the participating users.

This demonstration tackles precisely these two challenges. Our approach relies on a secure hardware based co-server, called *secure token* hereafter, which provides a search engine interface to users and applications to manage the documents with high security and privacy guarantees. This search engine, described in

detail in [5], manages the encryption/decryption of documents, answers local searches and enforces access control rules on the fly with good performance. In this demonstration, we extend this previous work to provide a secure distributed search engine, such that applications can query the documents stored in a large number of Personal Clouds. Any global search has to be accomplished while preserving the participants' privacy, i.e., no further information beside the computation result can be learned by any participant or a third party. This property is key to encourage users to participate in global computations.

The implementation of a secure distributed search engine is however challenging. Considering a top-k search where the score of each document is evaluated by a *tf-idf* metric and answering global searches over a (large) community of users require (1) to compute some global values (i.e., the total number of shared documents and the inverse frequency of the query terms in all these documents), (2) to evaluate the score of each document accessible for the query according to these global values, and then (3) to identify the *k* documents with the highest scores among the set of participants.

Although users may accept to contribute to a given community of interest by granting a right to search over a subset of their own personal documents, the risk that any compromised participant gains access to the complete collection of documents must be avoided. This can be achieved by minimizing the amount of information exposed during query evaluation. While the final result of each query (i.e., the *k* most relevant documents) can be published to the community, neither the intermediate computations nor the complete set of local documents eligible for a given query should be revealed.

Computing a result without revealing input data can be done (1) by outsourcing the data on a trusted party, but we consider this option as not satisfactory in personal cloud context where no trusted entity clearly appears in the scenario, (2) by using Secure-Multi-Party (SMC) cryptographic techniques, but these techniques cannot currently meet both query generality and scalability objectives [10] or (3) by relying on privacy preserving distributed query computation techniques (see Section 3).

Our approach capitalizes on the tamper resistant hardware available on each personal cloud to form a global secure decentralized data platform. No plaintext data will be exposed outside of the secure elements except the final result to be published. The risk of data disclosure thus only depends on the possibility of the secure elements of certain participants to be compromised, i.e., the secure token has been tampered with and the decryption keys it contains may become accessible to malicious participants. Although these attacks are highly difficult and costly to conduct, they cannot be totally ignored. It is therefore mandatory to quantify the impact of such an attack and to propose computation strategies minimizing it. We then introduce privacy metrics linked to the amount of intermediate personal data made accessible to the secure infrastructure during the computation and show how it can be minimized.

The aim of this demonstration is to show that practical and efficient solutions can be devised to evaluate distributed searches within large communities of Personal Cloud users with a very limited privacy risk for the participants. More precisely, we



demonstrate that “gossip” based computations [8] (1) can provide accurate search results with good performance and scalability (i.e., large communities of users, with thousands to millions of personal clouds) and (2) can drastically minimize the risk of privacy violation even if compromised participants are involved in the computation.

## 2. ARCHITECTURE AND SCENARIOS

Let us consider the Secure Personal Cloud Platform of Alice as pictured in Figure 1. The main component is a home cloud data system gathering personal data from multiple sources (employer, banks, hospitals, commercial web sites, etc.) and devices (smart meters, quantified-self devices, smartphones, cameras, etc.). The Personal Cloud can be implemented by any type of computing platform with storage facilities such as a set-top box or a plug computer. This data system is complemented by a secure co-server which can be hosted by any type of tamper-resistant devices flourishing today, e.g., Mobile Security Card (produced by Giesecke & Devrient), Personal Portable Security Device (produced by Gemalto and Lexar), Multimedia SIM card [4] or Secure Portable Token [5]. Whatever its commercial name and form factor, a tamper-resistant device embeds a secure microcontroller (e.g., a smart card chip) connected to a large NAND Flash memory (e.g., an SD card) and can communicate with a host through a USB, Bluetooth or Ethernet port. Open hardware secure tokens are also provided (e.g., by the Versailles Science Lab, <http://tinyurl.com/UVSQ-Lab>), and can be built by any electronic manufacturer.

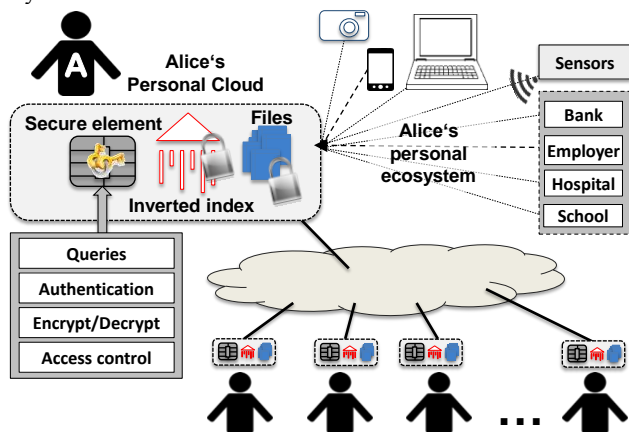


Figure 1: Secure Personal Cloud architecture

Alice participates here to a community of patients. Each secure token acts as a participant in a secure distributed search allowing users and applications to express full-text search queries over document collections stored within the community. The token being the root of security, it is in charge of data access control, encryption/decryption and metadata maintenance, that is the insertion, update and deletion of documents in the full-text search index. Each secure token locally stores an index of the user’s documents build on the documents content and including a set of access control terms associated with these documents. It also stores cryptographic keys used to protect the documents, stored encrypted locally or remotely on external cloud storage. Each user grants access to the documents by specifying access control rules (conjunction and disjunction of access control terms). The secure tokens collaboratively compute the result of top- $k$  information retrieval queries issued by users or applications by only considering the documents matching the access control rules on each personal cloud.

The documents can be any form of files (pictures, text files, pdf files, mails, data streams produced by sensors, etc.) associated to a set of terms. The terms are extracted from the file content and

from metadata describing it (e.g., name, type, date, creator, words, visterms, tags set by the user herself). A query issued by a user or an application can be any form of term search expression, with a ranking function (e.g., *tf-idf* [15]) identifying the top- $k$  most relevant documents (see Section 3). Only the documents granted to the user/application must appear in the query result. Hence, such a search engine can be used locally to query any documents entering a user dataspace, in the same spirit as a Google desktop or Spotlight augmented with access control capabilities.

A distributed search can be initiated by any member of the community to retrieve the most relevant documents for the query within the community. Each member within the community can contribute to the query by granting access to her own documents.

The question is how to pool these dataspace without the assistance of a central server. The objective indeed is to avoid centralizing sensitive information that may be at risk in case the server is compromised. We consider the existence of a network infrastructure enabling direct information exchange between any pair of personal clouds taking part in a distributed search query. We also assume that each participant owns a public/private key pair. Within a community, all participants exchange their public key at the time of registration. This could be achieved using traditional PKI or GPG techniques.

## 3. TECHNICAL CHALLENGES

**Search engine requirements.** To identify the top- $k$  most relevant documents in a given collection for a certain query, a ranking function is used to score each document. For this demonstration, we use the classical *tf-idf* function:

$$tf.idf(d) = \sum_{t \in Q} f_{d,t} \cdot \log \left( \frac{N}{F_t} \right)$$

where  $f_{d,t}$  is the occurrence number of term  $t$  in the document  $d$ ,  $N$  is the total number of indexed documents and  $F_t$  is the number of documents that contain  $t$ . For a local query (which searches into single personal cloud)  $N$  and  $F_t$  are local values (i.e. the local number of indexed documents and the local number of documents that contain the term  $t$ ). To evaluate a global search, the scores computed in the different personal clouds must be comparable. This requires computing beforehand the global values of  $N$  and  $F_t$  to be used in the previous formula. Then, the local top- $k$  scores have to be exchanged and compared to find the global result. To this end each participant has to transmit their data to others.

**Security constraints.** The secure tokens are the unique source of trust in the architecture. They are endowed with a tamper resistant element (secure microcontroller) which prevents physical attacks and also its owner from having access to the secret data it contains and manipulates. Hence, even the holder of the secure token cannot spy intermediate data manipulated by his token during a computation (similar with a banking card holder that has no access to the cryptographic secret stored in his card and cannot spy its data processing). However, despite its high level of security, we cannot exclude the possibility of having a small percentage of hacked tokens (e.g., as a result of a sophisticated attack from the token owner). Such an attack would lead the personal cloud owner to have access to the cryptographic material stored inside her secure token. From this, she can potentially decrypt any encrypted information sent during the computation to her personal cloud. One objective of this demonstration is to evaluate the risk taken by the participants of a community to have their personal data unexpectedly exposed in this case. This risk analysis is essential since the hardware is left into users hands. Breaking a set of secure tokens should not put the personal documents of the whole community at risk. To evaluate that risk, we measure (1) for a given set of (potentially compromised) secure tokens, the amount of intermediate results and documents exposed during the evaluation of a distributed search query, and (2) for a given set of (not compromised) participants, the amount

of her own information transmitted to remote secure tokens and the number of secure tokens to which this information is disseminated. The first metric gives an estimate of the benefit of an attack for the attackers. The security being evaluated as a ratio between the cost of the attack and its benefit, the lower the value is, the better the security is. The second metric estimates the impact of the information leak for an honest participant. Our objective is to keep both metrics as low as possible, and never favor a solution which puts the complete dataset at risk in case of successful attacks.

**State of the art solutions.** Distributed query processing and the top-k queries are well investigated topics. Although, to our knowledge, decentralized secure computation of top-k queries on a population of secure elements has not been investigated yet, several previous proposals are related to our work. A first approach to solve the problem is to rely on a super peer (i.e., a central manager or a designated peer used as a coordinator) as proposed in [6, 12, 13, 14]. However, these solutions do not comply with our security requirement since the complete dataset becomes at risk if the super peer is compromised. Other existing approaches [1, 7] propose to organize the peers as a tree to process the queries. However, tree architectures are very sensitive to peers failures. In addition, the peers participating in the tree potentially gather a lot of branches and can thus be the transit point of a large amount of data, leading to large privacy breaches if compromised. Other solutions found in the literature use gossip protocols, which are highly suitable for fully decentralized architectures. In [6], a gossip protocol is used to broadcast top-k queries only to the peers who have similar interests as the querier, which is very interesting in our context if transposed to "trusted peers" and is part of our future works. However, the solution proposed in [6] assumes that the querier can see all the intermediate results coming from the participants, which would not be acceptable in our context. In Chiaroscuro [2], participating devices collaborate using gossip style computations to achieve privacy thanks to encryption and differential privacy. But this solution is dedicated to perform clustering operations on time series. A recent proposal [13] addresses the problem of computing SQL aggregate queries over an asymmetric architecture composed of potentially large populations of secure tokens and a central server. However, the focus is to prevent data inferences while delegating operations to the central server. Also, the secure tokens share the same secret key, which incurs the risk of exposing the complete dataset if one secure token is hacked.

## 4. DESIGN OF THE SOLUTION

Our solution to perform the distributed search relies on three main phases described as follows. In Phases 1 and 3, gossip computations algorithms [8] are used to respectively compute a sum and a top-k using of gossip computation algorithms.

**Phase 1: computation of global  $N$  and  $F_t$ .** The query is broadcasted to the participants. The secure token of each participant computes locally its own contribution to the global  $N$  and  $F_t$ , considering only the documents compliant with its active access control rules. The local contributions are then aggregated in order to compute the global values for  $N$  and  $F_t$  according to the push-sum algorithm proposed in [8]. The precision on the approximate values obtained on each participant for  $N$  and  $F_t$  can be controlled by the number of gossip exchange rounds which remains reasonable even for a large number of participants:

$$nb\_round = O(\log(n) - \log(\epsilon) - \log(\delta))$$

where  $nb\_round$  is the number of rounds necessary in a network of  $n$  participants to obtained with a probability at least  $1 - \delta$  a result with an error under  $\epsilon$ . At the end of this step, each participant has an approximate value of the global  $N$  and  $F_t$  values.

**Phase 2: local computations of the top-k.** Based on these global values, each secure token computes locally the global scores for

the documents compliant with its active access control rules and produces a local top-k.

**Phase 3: computation of the global top-k.** A new phase of gossip communication starts during which secure tokens exchange their top-k. At each round, each secure token receives  $k$  tuples (cloud\_id, doc\_id, score) from another peer and selects the  $k$  highest scores within the union of the received tuples and the local tuples. Then, it chooses randomly the next peer to which it sends its current local result. In this way, the tokens refine their local results at each round of the protocol. After a given number of rounds [8], all the tokens share a set of local results that are close to the exact global result. Due to the random characteristic of gossip protocols, the final result is only an approximation of the exact result (i.e., the one which would have been obtained on a centralized union of all the authorized documents in all participating personal clouds). The present demonstration will show that the results are good and that the improvement of the result accuracy is fast in number of rounds.

To enforce the security of the protocol and meet the privacy requirement, an asymmetric encryption/decryption system is used. Each token owns a private/public key pair. We assume in the demonstration that every participant has at disposal the complete list of public keys of all the participants. During the gossip communication phases tokens randomly choose a public key in the list, encrypt their message to transmit it to the corresponding secure token which decrypts the message with its private key. A token can thus be sure that its message can only be read by the chosen recipient, which limits the data exposure risk.

## 5. DEMONSTRATION

In this section, we present our prototype platform and describe the demonstration scenario covering the security and the performance of the proposed solution for distributed search in the secure Personal Cloud architecture.

### 5.1. Platform

**Hardware Platform.** The demonstration platform is an instance of the architecture depicted in Figure 1. A laptop is used as the communication infrastructure, and 20 secure tokens (see Figure 2) are used as participants. Each token is running the distributed search algorithm based on gossip computations as presented in the previous section, and evaluates the local access control rules. The local searches are performed using a previous prototype [11]. The secure tokens are equipped with a 32 bit RISC MCU clocked at 120 MHz with 128 KB of static RAM and 1MB of NOR Flash (to store the code of the distributed search engine). The MCU is connected to a smartcard chip hosting the cryptographic material and to a  $\mu$ SD card which stores the inverted index use by the embedded search engine. The PC which connects to all tokens via a USB port plays the role of the network infrastructure, controls the communications between tokens, and shows exchanged data and results it receives from tokens.

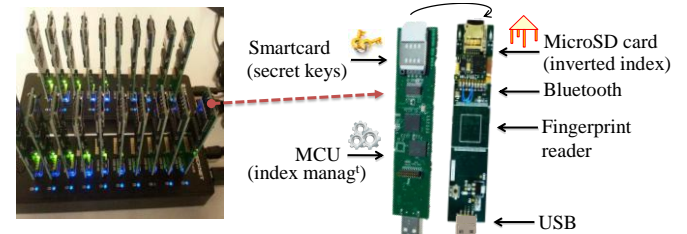


Figure 2: Secure Tokens used in the demonstration.

**Graphical User Interface.** The GUI is used to control the system and evaluate our privacy metrics. Any participant can issue a global search query and retrieve the relevant files. The demonstration interface compares the efficiency, performance and privacy offered by our distributed search algorithm with a secure

solution where all the secure tokens would share the same encryption key (in the spirit of [13]) and with tree based approaches inspired by [1]. The accuracy of the search is compared with a central search computed over the union of the document databases stored in the participating personal clouds (using traditional precision and recall metrics). To evaluate the privacy level of our computation technique, the GUI also shows the probability of data disclosure to the attackers and the number of external documents that an attacker could read. To this end, the interface enables choosing certain participants to be considered as being corrupted.

**Dataset.** We use the Pseudo-desktop collection [9] as the baseline dataset of the demonstration. It includes more than 27000 documents (representing emails, photos, pdf, docs, and ppts). We randomly spread the documents among the tokens. We also use a synthetic dataset to show the scalability of the approach with more documents in each personal cloud.

## 5.2. Scenario

The first goal of this demonstration is to present the performance of the distributed search. Each participant has set her own access control rules. One of them issues a query (set of terms) and chooses the expected error rate. The number of iterations in the gossip phases is then fixed according to this error rate. The result (an approximate top- $k$  obtained by each participant) is compared to the exact one, which is computed by the PC on the union of the authorized documents. Query times, average score error, precision and recall are presented and compared. The demonstration shows that good performance can be achieved even for low error rates. The average score error is given by the difference between the scores obtained in the approximate results for each token and the exact score obtained with a centralized search. The recall (respectively, precision) is given by the ratio between the number of documents present in both the approximate result and the exact result, and the number of results (respectively, the number of documents in the exact result with a score greater than the minimum in the approximate result).

The second goal of this demonstration is to focus on the privacy properties of gossip computations. The attendees choose some personal clouds as being corrupted before running the query. First, the interface will show the ratio between the number of distinct couples (cloud\_id, doc\_id, score) computed by each token which are exposed to the attackers, and the total number of couples (cloud\_id, doc\_id, score) in the local top- $k$  computed during the query. Second, for each participant considered as honest, the interface plots the ratio between the number of couples which do not appear in the result and have been transmitted (directly or transitively) to one of the attackers during the execution, and the number couples which have been computed locally (typically  $k$ ).

These two metrics obtained with our search algorithm will be compared to those obtained using alternatives representative of state of the art solutions: (1) a tree based evaluation, where the data flow between the participants is modeled as a tree structure (inspired by [1]) and (2) a secure query execution where all secure tokens would share a same secret key to evaluate the query (inspired by [13]).

## 5.3. Demonstration results

In terms of performance, the execution query time with our proposal is larger compared to a centralized database. This is obvious since our system requires a number of gossip iterations. However, the execution time is reasonable even on large databases and with large numbers of participants (obtained by simulation in the demonstration). In terms of precision, the average score error is less than 10% and the precision lies between 0.6 and 0.9 with a relatively low number of iterations. These values show that our proposal can be used in practice. In terms of privacy and security, our proposal shows much better results than existing approaches. Typically, honest users involved in the

computation disclose few couples (cloud\_id, doc\_id, score) to the attackers. Considering 10% of attackers, the probability to expose a couple to an attacker is around 20% in our experiments. Our technique could be improved by choosing in the first gossip steps only users considered as trusted by the participant. This would decrease this number drastically (this is part of our future works). And for an attacker, the benefit of an attack is very small since only a very tiny proportion of the intermediate results can be obtained.

## 6. CONCLUSION

The emerging Personal Cloud paradigm holds the promise of a Privacy-by-Design storage and computing platform where personal data remains under the individual's control while being shared by valuable applications. In this demonstration, we present a distributed secure search engine with the objective to provide a high level of security founded on the introduction of low cost secure tokens in the architecture. This architecture minimizes the loss of privacy risks even if some participants are compromised, i.e., could bypass the tamper resistance of the token. While many personal cloud platforms are flourishing, riding the wave of repeated scandals blemishing the typical centralized management of personal data, none of them provides such a tangible source of trust to the individuals. We hope that the platform demonstrated here, which enable both local application and distributed ones, emphasizes the interest of studying new database techniques based on secure hardware for the database community. .

## 7. ACKNOWLEDGMENTS

This study is funded by the ANR KISS project grant n°ANR-11-INSE-0005 and by the [Inria International Project Lab CityLab](#).

## 8. REFERENCES

- [1] Akbarinia, R., Pacitti, E., and Valduriez, P. 2006. Reducing network traffic in unstructured p2p systems using top-k queries. *Distributed and Parallel Databases*, 19.
- [2] Allard, T., Hébrail, G., Masegla, F., Pacitti, E. Chiaroscuro: Transparency and Privacy for Massive Personal Time-Series Clustering. In *ACM SIGMOD 2015*.
- [3] Anciaux, N., Bonnet, P., Bouganim, L., Nguyen, B., Sandu Popa, I., and Pucheral, P. Trusted cells: A sea change for personal data services. In *CIDR*, 2013.
- [4] Anciaux, N., Bouganim, L., Guo, Y., Pucheral, P., Vandewalle, J.-J., & Yin, S. Pluggable personal data servers. In *ACM SIGMOD*, demo. paper, 2010.
- [5] Anciaux, N., Lallali, S., Sandu Popa, I. and Pucheral, P. A Scalable Search Engine for Mass Storage Smart Objects. *PVLDB*, 8(9), 2015.
- [6] Bai, X., Guerraoui, R., Kermarrec, A.-M. and Leroy, V. 2011. Collaborative personalized top-k processing. *ACM TODS*, 36.
- [7] Cao, P. and Wang, Z. Efficient top-k query calculation in distributed networks. In *PODC*, 2004.
- [8] Kempe, D., Dobra, A. and Gehrke, J. Gossip-based computation of aggregate information. In *FOCS*, 2003.
- [9] Kim, J. Y. and Croft, W. B. Retrieval Experiments using Pseudo-Desktop Collections. In *CIKM*, 2009.
- [10] Kissner, L., Song, D. X. Privacy-Preserving Set Operations. In *CRYPTO*, 2005.
- [11] Lallali, S., Anciaux, N., Sandu Popa, I., Pucheral, P. A secure search engine for the personal cloud. In *ACM SIGMOD*, demo. paper, 2015.
- [12] Michel, S., Triantafillou, P. and Weikum, G. Klee: a framework for distributed top-k query algorithms. In *VLDB*, 2005.
- [13] To, Q.-C., Nguyen, B., and Pucheral, P. Privacy-Preserving Query Execution using a Decentralized Architecture and Tamper Resistant Hardware. In *EDBT*, 2014.
- [14] Wang, H., Tan, C. C., Li, Q. 2010. Snoogle: A search engine for pervasive environments. In *Trans. on Par. & D. Sys.*, 21(8).
- [15] Zobel, J. and Moffat, A. 2006. Inverted files for text search engines. In *ACM Computing Surveys*, 38(2).

# Type-aware Web-search

Michael Gubanov  
University of Texas at San Antonio  
mikhail.gubanov@utsa.edu

Anna Pyayt  
University of South Florida  
pyayt@usf.edu

## ABSTRACT

Keyword-search engines (e.g. Web-search) usually can be outperformed by a specialized system optimized for a specific domain, type of data, or queries [8, 2, 12, 5, 11, 9]. For example, Halevy et. al. in [13] demonstrate how a specialized Google Fusion Tables spatial search can outperform the general-purpose Google Web-search on *bike trails search* in San Francisco Bay Area. At the same time, Web content providers usually exhibit a specific focus for their postings. For example, information at <http://www.csail.mit.edu> is devoted to Computer Science research and education, *Hannah Montana* is mostly tweeting about music, and the same is true for most sources.

This paper describes the work in progress on a new Type-aware Web-search system that uses topical focus of information sources to process a large class of queries better than a regular Web search-engine. It leverages semantic profiles similar to [10, 6, 7] and a new Type-aware Locality-Sensitive Hashing (TLSH) scheme to accomplish it.

## 1. INTRODUCTION

Figure 1 illustrates search-results from one of the Web search engines for query *Frozen in Phoenix* where a user is trying to find a theater to watch a movie. You can see the search-results are not the best (about ice cream and frozen yogurt). It happened, because the generic Web-search engine employs simple term matching of the query with the Web pages, and did not take into account type information, which can be done to get more relevant results. Table 1 illustrates the Web-search results of a *type-aware* search-engine described here for queries *Careers of People with Ph.D.* You can see, it returns precisely what the user has been asking for in these queries. A regular Web-search engine would return career-pages of companies and recruiting agencies, resulting from term matching to *careers*.

## Categories and Subject Descriptors

H.2 [Database Management]: Heterogeneous Databases

©2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

### Ice Cream & Frozen Yogurt Phoenix, AZ - Yelp

[www.yelp.com/search?cft=icecream&find\\_loc=Phoenix%2C...](http://www.yelp.com/search?cft=icecream&find_loc=Phoenix%2C...) Yelp  
Top Ice Cream & Frozen Yogurt in Phoenix, AZ Melt., Churn, Twirl, Sweet Republic, Frost Gelato, Kamana' Wana Hawaiian Treats, Yogurtology, Zoyo ...

### Self serve frozen yogurt Phoenix, AZ - Yelp

[www.yelp.com/search?find\\_desc=Self..Frozen..Phoenix%2C...](http://www.yelp.com/search?find_desc=Self..Frozen..Phoenix%2C...) Yelp  
Reviews on Self serve frozen yogurt in Phoenix, AZ Twirl, Zoyo Neighborhood Yogurt, A Touch of Yogurt, Yogurt Plus, Yogurtology, Orange Leaf Frozen Yogurt, ...

### Twirl Froyo |

[twirlfroyo.com/](http://twirlfroyo.com/)  
We source the best, freshest frozen yogurt available. ... Twirl Frozen Yogurt was created with the idea you could bring amazing healthy ... phoenix • az • 85012 ...

Figure 1: Search-results for *Frozen in Phoenix*

Query: Careers of People with Ph.D.

Results: - Romania News Watch:

...Ponta obtained his PhD from the University of Bucharest while acting as Secretary of State in the government of an earlier prime minister..

Table 1: Type-aware Search Results

## 2. ARCHITECTURE

The crawled Web pages are processed by a Natural Language Processing domain-dependent parser, which emits the entity data along with the text fragments they came from and saves the result into a large-scale storage (see Figure 2 for a schematic). Both a large-scale semi-structured sharded storage engine as well as a parallel relational engine are used.

The earlier work in [10] introduces *semantic profiles* intended to capture the semantics of an information source and store it in a compact and reusable manner. It summarizes and accumulates all *types* of entities from the source. For example, the newspaper *New York Times* often publishes about *companies*, *products*, and *organizations*; *The Finance* usually tweets about *dividends* and *products*; *The Oregonian* publishes about *sports*, *holidays*, *music*, and hence their profiles are comprised of these types. These profiles are calculated and saved for each source. Due to space limitations, interested readers are referred to [10] for more details on profile construction.

Next, the hashing routines treat each profile as a vector and assign it to one of the hash tables. Similarly, the incoming query is represented a vector, the query processing module computes the set of relevant hash tables for a query, the relevance score of the documents from these hash tables

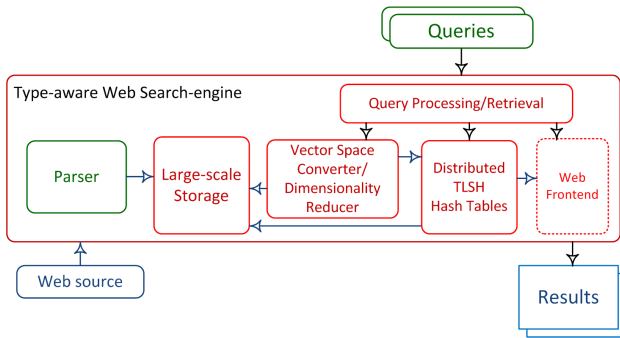


Figure 2: Architecture

and the query is computed, and finally the documents are ranked by this relevance score and output to the user.

### 3. TYPE-AWARE WEB-SEARCH

**Type-aware Locality Sensitive Hashing:** Locality-Sensitive Hashing (LSH) [14] is an algorithm that enables searching for near neighbors in a high-dimensional vector space  $S^n$  with dimensionality  $n$ . Formally, given a query  $q \in S^n$ , return the nearest neighbors of  $q$  within certain radius  $R$ . LSH performance crucially depends on a family of hash functions  $F$  that it uses to map the input vectors to its internal data structures. In order for the algorithm to perform well,  $F$  usually has to reduce the dimensionality of the original vector space still satisfying the *locality-sensitive* requirements on the reduced vector space.  $F$  is considered to be *locality-sensitive* if collision of two vectors  $v_1$  and  $v_2$  under a random choice of a hash function from  $F$  depends only on the distance between  $v_1$  and  $v_2$ . Refer to [3] for an overview of locality-sensitive hash-function families.

Here, a new two-tier family of hash functions  $\Psi$  is described and used. First, it maps the original vector space  $V$  of *terms* into a vector space of *types* -  $T$ , hence reduces dimensionality (there are much less types than terms). Second,  $k$  random unit vectors  $u \in T$  are generated, which defines a family of hash-functions  $h \in \Psi$  as follows  $h(v) = \text{sign}(u \cdot v / \|v\|) : u, v \in T$ . Refer to [4] for a proof of its locality-sensitivity. Angular distance measure is used here for this vector space.

**Query Processing:** The queries and Web documents are represented as vectors in a high-dimensional vector space  $S^n$  with dimensionality  $n$  (number of types). To return vectors (Web documents) within radius  $R$  of the query  $q$  the algorithm concatenates  $k$  hash functions  $h_i \in \Psi$  described above into a composite hash function  $h_c(v) = h_1(v), \dots, h_k(v)$ , hence creating a family of hash functions  $h_c \in \Phi$ .

Next, for query  $q$  it computes all functions from  $h_c$  and considers the documents only from the corresponding hash tables. It returns all vectors  $v$  from those hash tables that are within angular distance  $R$  from  $q$ . The evaluation below justifies that using this semantic hashing/retrieval algorithm outperforms a generic Web-search engine by relevance of search-results.

**Relevance Evaluation:** Here, relevance gain of TLSH hashing/retrieval scheme compared to a general purpose Web-search engine for “type-containing” queries (i.e. containing a Named-entity) is quantitatively evaluated. An experiment

was conducted to calculate NDCG (Normalized Discounted Cumulative Gain) [1] on a static set of queries with respect to a general purpose Web-search engine, which provides quantitative insight into their performance difference. NDCG is one of the standard widely used search relevance measures, which is employed by major search engines and, similarly to F-measure, measures both precision and recall of retrieval. NDCG is well suited for search evaluation, because it rewards relevant results in the top positions more than those ranked lower. Due to space limitations, interested readers are referred to [1] for details about NDCG computation. Total NDCG gain over all queries turned out to be very large  $> 6\%$ . Usually for two industrial Web-search engines NDCG difference more than 4% is considered to be significant.

### 4. REFERENCES

- [1] E. Agichtein, E. Brill, and S. Dumais. Improving web search ranking by incorporating user behavior information. In *SIGIR*, 2006.
- [2] B. Alexe, M. Gubanov, M. Hernandez, H. Ho, J.-W. Huang, Y. Katsis, L. Popa, B. Saha, and I. Stanoi. Simplifying information integration: Object-based flow-of-mappings framework for integration. In *BIRTE*, volume 27 of *Lecture Notes in Business Information Processing*, pages 108–121. Springer, 2009.
- [3] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, Jan. 2008.
- [4] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*. ACM, 2002.
- [5] S. Cheemalapati, M. Gubanov, M. Del Vale, and A. Pyayt. A real-time classification algorithm for emotion detection using portable eeg. In *IRI*. IEEE, 2013.
- [6] M. Gubanov and A. Pyayt. Readfast: High-relevance search-engine for big text. In *ACM CIKM*, 2013.
- [7] M. Gubanov and A. Pyayt. Readfast: Optimizing structural search relevance for big biomedical text. In *IRI*, 2013.
- [8] M. Gubanov and L. Shapiro. Using unified famous objects (ufo) to automate alzheimer’s disease diagnostics. In *BIBM*, 2012.
- [9] M. Gubanov, L. Shapiro, and A. Pyayt. Readfast: Structural information retrieval from biomedical big text by natural language processing. In *Information Reuse and Integration in Academia and Industry*, pages 187–200. Springer, 2013.
- [10] M. Gubanov and M. Stonebraker. Large-scale semantic profile extraction. In *EDBT*, 2014.
- [11] M. Gubanov, M. Stonebraker, and D. Bruckner. Text and structured data fusion in data tamer at scale. In *ICDE*, 2014.
- [12] M. N. Gubanov, P. A. Bernstein, and A. Moshchuk. Model management engine for data integration with reverse-engineering support. In *ICDE*, 2008.
- [13] A. Halevy. Data publishing and sharing using fusion tables. In *CIDR*, 2013.
- [14] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. *STOC*. ACM, 1998.

# Indexing and Querying A Large Database of Typed Intervals

Jianqiu Xu<sup>1</sup>, Hua Lu<sup>2</sup>, Bin Yao<sup>3</sup>

<sup>1</sup>Nanjing University of Aeronautics and Astronautics, China

<sup>2</sup>Aalborg University, Denmark

<sup>3</sup>Shanghai Jiao Tong University, China

jianqiu@nuaa.edu.cn, luhua@cs.aau.dk, yaobin@cs.sjtu.edu.cn

## ABSTRACT

Assume that a database stores a set of intervals associated with types and weights. Typed intervals enrich the data representation and support applications involving different kinds of intervals. Given a query time and type, the system reports  $k$  intervals that intersect the time, contain the type and have the largest weight. We develop a new structure to manage typed intervals based on the standard interval tree and propose efficient query algorithms. Experiments with synthetic datasets are conducted to verify the performance advantage of our solution over alternative methods.

## 1. INTRODUCTION

In this paper, we study top- $k$  queries on typed intervals. Assume that a database stores a set of tuples, each of which defines three attributes: an interval with start and end points, a type and a weight. Given a query time and type, the system reports  $k$  tuples fulfilling the conditions: (i) intersect the query time; (ii) contain the type; and (iii) have the maximum weight, i.e., return  $k$  intervals with maximum weights among all fulfilling conditions (i) and (ii).

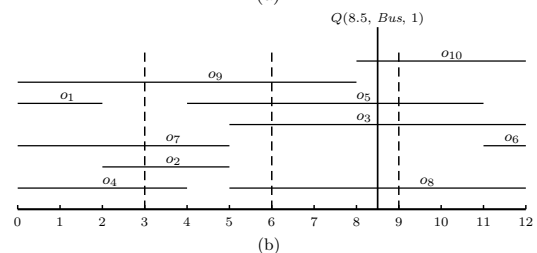
To help understand the problem, Figure 1 shows a running example. In traffic monitoring systems, the database stores the number of vehicles appearing in a district over time. There are different kinds of vehicles: {*Taxi*, *Bus*, *Truck*, *Private Car*}. Each tuple records a time interval, the vehicle type and the count. A top- $k$  query is “return the district with the largest number of buses at the time 8.5”, and the system returns  $o_3$ . The following objects  $\{o_3, o_5, o_8, o_{10}\}$  intersect the query time, but only  $o_3$  and  $o_5$  fulfill the type condition.

In the literature, queries on interval data have been studied with operators such as intersecting [3], stabbing [1] and top- $k$  on keyword intervals [5]. However, they do not consider intervals associated with types and therefore do not support applications involving different types of intervals, e.g., various genome intervals in genomic datasets and different versions of data items.

©2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.

Id	Time	Type	Count
$o_1$	[0, 2]	<i>Taxi</i>	40
$o_2$	[2, 5]	<i>Car</i>	15
$o_3$	[6, 12]	<i>Bus</i>	60
$o_4$	[0, 4]	<i>Car</i>	45
$o_5$	[4, 11]	<i>Bus</i>	20
$o_6$	[11, 12]	<i>Truck</i>	45
$o_7$	[0, 6]	<i>Bus</i>	30
$o_8$	[6, 12]	<i>Taxi</i>	23
$o_9$	[0, 9]	<i>Truck</i>	23
$o_{10}$	[9, 12]	<i>Car</i>	83

(a)



(b)

Figure 1: Typed Intervals

Edelsbrunner’s interval tree [2] is a popular structure for reporting intervals intersecting a given query. In principle, an interval tree is a binary tree that serves as the *primary* structure. Each node in the tree maintains two lists (*secondary* structure) of sorted intervals. One can directly employ the two-list structure to manage typed intervals, but the method is not optimal. Since the standard interval tree does not support intervals with types, a linear scanning is performed in each accessed node to find intervals that intersect the time and contain the type, even some of them are not equal to the query type. Another problem is, too many intervals are visited. In fact, the query only needs  $k$  intervals.

We propose a new structure to replace the sorted lists in each node to maintain typed intervals. Given a node storing a set of intervals, the new method is able to determine part (even all) of the intervals intersecting the query time without accessing the data. An index is built on managing types, leading to quickly finding intervals with a particular type. Employing the new structure, much less intervals are accessed to report  $k$  results. We carry out the experimental evaluation to demonstrate the performance of our method by using synthetic datasets.

## 2. THE SOLUTION

In each node, we replace two lists by a new structure for interval management. The idea is, we partition the interval space into a set of equal-length slots, each of which has a unique id and defines a subspace. We use two tables in which one maintains intervals *containing* the relevant slots and the other maintains intervals *intersecting* the slots, named as *full* and *partial* tables, respectively. Each row in the tables corresponds to a slot and stores a list of interval ids.

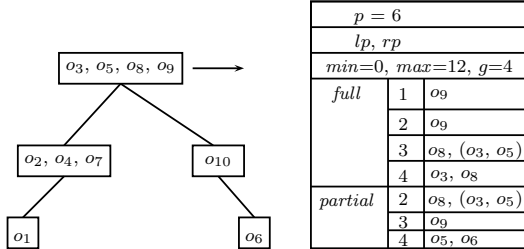


Figure 2: Slot representation based on interval tree

Figure 2 depicts the binary tree built on intervals and the slot representation for the root node. The new secondary structure includes the center point  $p$  and two child pointers  $lp, rp$ . These items are the same as the traditional structure [2]. Next,  $min$  and  $max$  are lower and upper endpoints of all intervals at this node. To perform the partition, the number of slots is defined, denoted by  $g$ . In the example, four slots are created for the root node. Intervals located in each slot are first sorted on type and then on weight. The type index is a list of items and each item records the type and the start position of intervals with that type. For example, the index for the third slot in the full table will be  $\{(Taxi, 0), (Bus, 1)\}$  because  $o_8$  is the first interval in the slot and  $o_3$  is the second interval.

To answer the query, we perform a binary search on the tree. Given a node, we first determine the corresponding slot and then access the full and/or partial tables. Intervals in the full table do not have to be tested on the intersection condition. We use the type index to find intervals having the query type and return the first  $k$  intervals. For intervals in the partial table, we find those fulfilling the type condition and then iteratively test each on the intersection condition. Intervals from the two tables are inserted into a min-heap with the size  $k$ . We keep updating the min-heap until the searching procedure is terminated. Apparently, the better the performance is, the more intervals are in the full table. This depends on the slot number defined to partition the space.

### 3. EXPERIMENTAL EVALUATION

We use synthetic datasets in the preliminary evaluation:  $\{S1(1M), S2(5M), S3(10M), S4(20M), S5(50M)\}$ . The start point of an interval is randomly chosen within the domain  $[1, 100000]$ , and the length is a random value between 1 and 1000. Let  $T$  be the number of types and we set  $T = 100$  in the experiment. The weight is randomly selected as an integer from  $[1, 500]$ .

Three competitive algorithms are developed in the evaluation. One extends the standard interval tree by integrating a boolean bit string representing whether there are intervals

with certain types in the node. The secondary structure in each node is defined to be  $2 \cdot T'$  ( $T' \leq T$ ) lists. Each list stores intervals with the same type. The second algorithm uses a relational interval tree [4] in which the bit string is also integrated. The last method employs a 2D R-tree. The three algorithms are named by *Ext-I-tree*, *RI-tree*, and *R-tree*, respectively, and our method is named *Slot*.

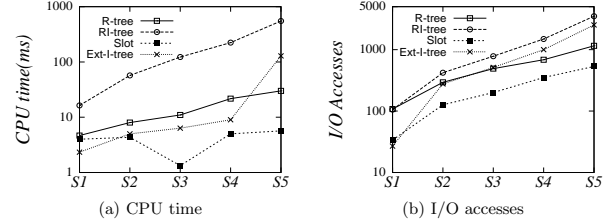


Figure 3: scaling the data size

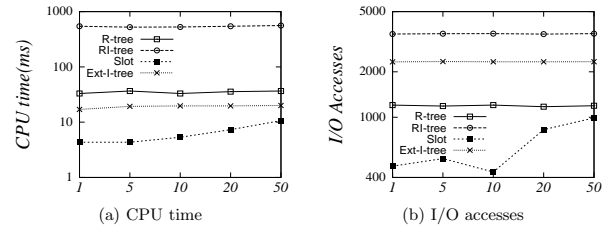


Figure 4: synthetic dataset,  $S5$

We perform the evaluation by scaling the number of data intervals and the number of returned intervals  $k$ . The CPU time and I/O accesses are reported in Figure 3 and Figure 4. The results demonstrate that our method significantly outperforms other methods, e.g., 3-6 times faster than *R-tree*, 2-10 times faster than *Ext-I-tree*. Since the CPU time is only several milliseconds, a small deviation may lead to a sharp slope of the curve, e.g., in Figure 3(a). The I/O variation in Figure 4(b) is attributed to the randomness of the generated queries.

### Acknowledgment

This work is supported by NSFC under grants 61300052 and NSFC of Jiangsu Province under grants BK20130810.

### 4. REFERENCES

- [1] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM J. Comput.*, 32(6):1488–1508, 2003.
- [2] H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Technical report, Tech. Univ. Graz, Graz, Austria, 1980.
- [3] J. Enderle, N. Schneider, and T. Seidl. Efficiently processing queries on interval-and-value tuples in relational databases. In *VLDB*, pages 385–396, 2005.
- [4] H.P. Kriegel, M. Pötke, and T. Seidl. Managing intervals efficiently in object-relational databases. In *VLDB*, pages 407–418, 2000.
- [5] R. Li, X. Zhang, X. Zhou, and S. Wang. INK: A cloud-based system for efficient top-k interval keyword search. In *CIKM*, pages 2003–2005, 2014.

# Quantifying Likelihood of Change through Update Propagation across Top-k Rankings\*

Evica Milchevski  
 TU Kaiserslautern  
 Kaiserslautern, Germany  
 milchevski@cs.uni-kl.de

Sebastian Michel  
 TU Kaiserslautern  
 Kaiserslautern, Germany  
 smichel@cs.uni-kl.de

## ABSTRACT

Rankings are a widely used techniques to condense a potentially large amount of information into a concise form. However, rankings are dynamic and undergo changes, thus need to be maintained, which can be a tedious and expensive task. Given a ranking  $\tau$  that got updated to  $\tau'$ , we aim at identifying those rankings  $\sigma$  that are very likely to have changed as well, as they are close in distance to the original ranking  $\tau$ . We do so by modeling the expected change in form of a hypothetical ranking  $\sigma'$  and mark  $\sigma$  to require a refresh if the expected change is above a threshold. We do this for the Footrule distance and demonstrate through a preliminary evaluation the potential of our approach.

## 1. INTRODUCTION

We focus on the task of maintaining a set of crowdsourced entity rankings. One important characteristic of crowdsourced rankings is that although they share the same entities, they conform to different constraints, thus, a change in one ranking does not imply the same change in another ranking. We propose a framework that uses the similarity between the rankings, to reason about the degree of change in a set of rankings due to an update in one ranking. Since the distance between two rankings resembles not only structural but semantic similarity as well, it is reasonable to assume that once a ranking changes, it is more likely that similar rankings change, rather than dissimilar ones. Specifically for top-k rankings that only report on a (usually short) subset of items, if two rankings are similar, they need to share also a fraction of items. If ranking  $\tau$  changes, this means that items that are present in  $\tau$  changed, respectively their features. Such changes might or might not propagate to a ranking  $\sigma$  that is in distance  $\lambda$  to  $\tau$ —the likelihood of such a propagation is what we aim at quantifying in this work.

When considering rankings created over some objective (measurable) scoring function, like wealth in USD, the update of the rankings can be done by maintaining one global ranking, and directly updating all rankings affected by an update. However, keeping a global ranking in the case of

\*This work has been supported by the German Research Foundation (DFG) under grant MI 1794/1-1.

crowdsourced rankings, where there is no measurable scoring function, but instead entities are ranked by some user perceived quality, like popularity, is not only expensive, but also unintuitive, as there is no ground truth.

## 1.1 Problem Statement

As **input** we are given a set  $\mathcal{T}$  of top-k rankings  $\tau$ . Each ranking  $\tau$  has a domain  $\mathcal{D}_\tau$ —the items it ranks. We further know the Footrule distance between each pair of rankings in  $\mathcal{T}$ . We define an **update**  $u_\tau(i, j)$  **to a ranking**  $\tau$  **as a swap of two items**  $i, j \in \mathcal{D}_\tau$ . A size of an update  $u_\tau(i, j)$ , for brevity denoted simply as  $|u_\tau|$ , is the difference between the positions of the swapped items,  $|\tau(i) - \tau(j)|$ . We denote with  $\tau'$  the ranking  $\tau$  after applying an update  $u_\tau(i, j)$ .

For a given update  $u$  over a ranking  $\tau$ , the task is to compute the likelihood that  $u$  affects other rankings  $\sigma \in \mathcal{T}, \sigma \neq \tau$  such that  $F(\sigma, \sigma')$  is larger than some user defined threshold  $\theta$ . In that case,  $\sigma$  is marked to be refreshed (e.g., crowdsourced) to bring it up to date. If we believe a ranking is not affected by a change but in fact it is, we suffer loss in **recall**, the ranking is not refreshed and our database  $\mathcal{T}$  is getting stale. If we, however, believe it is affected and it is not, we suffer loss in **precision**, which leads to wasting cost to refresh a ranking when it is not required to do so. For an overview to methods for comparing top-k rankings, see [1].

## 2. APPROACH

Algorithm 1 shows the procedure for determining (allegedly) affected rankings. As input the algorithm takes a specific update, a set of rankings, where the Footrule distance between all pairs is known, and a threshold  $\theta$ . First, we compute the maximum distance,  $d_{max}$  that would likely result in the updated items  $i, j$  being present in both rankings  $\tau$  and  $\sigma$ , i.e.,  $i, j \in \mathcal{D}_\tau \cap \mathcal{D}_\sigma \Rightarrow F(\tau, \sigma) \leq d_{max}$  with some probability  $p$ . We explain how this bound can be computed below. The next step is computing the expected change according to the distance. For this purpose, in a pre-processing step for each possible distance, for a given  $k$ , we compute an average difference between the positions of the items in two rankings  $\tau$  and  $\sigma$ , such that  $F(\tau, \sigma) = \lambda$ . We call this average displacement (see Section 2.1) and it does not depend on the actual rankings in  $\mathcal{T}$ . Using the average displacements, we compute the expected change, according to the actual update. If the change is larger than a user specified threshold  $\theta$ , we retrieve all distances and output all rankings within the retrieved distance to the changed ranking.

When we have a sequence of updates, we can simply accumulate the change. Note that an update over several items can also be considered as a sequence of updates of two items.

### 2.1 Computing the Expected Change

Since we do not know the positions, if any, of the affected items in the rankings, the first step toward quantifying the expected change in a ranking is reasoning about the most



```

input:  $u_\tau, \mathcal{T}, \theta$ 
1  $d_{max} = \text{get\_distance\_limit}()$ 
2 for each  $E[\mu]$  in  $\text{get\_expectations}(k)$  do
3    $e_{change} = \text{get\_expected\_change}(u_\tau, \tau, E[\mu])$ 
4   if  $e_{change} \geq \theta$  then
5     for each  $d \leq d_{max}$  in  $\text{get\_distances}(e_{change})$  do
6        $R \leftarrow \text{get\_all\_rankings}(d)$ 
7     return  $R$ 

```

**Algorithm 1:** Algorithm for determining all the rankings in  $\mathcal{T}$  affected by a change  $u_\tau$  according to their distance.

likely position of the affected items, when the distance  $\lambda$  between the rankings is known. To do this we first define a displacement of an item  $i$  in two rankings  $\tau$  and  $\sigma$ :

**DEFINITION 1. Displacement of an item:** For two rankings  $\tau$  and  $\sigma$ ,  $\tau \neq \sigma$ , we define a displacement of an item  $i$ , denoted with  $\mu_i$ ,  $i \in D_\tau \cap D_\sigma$ , as the difference of the position of the item in the two rankings, i.e.,  $\mu_i = |\sigma(i) - \tau(i)|$ . In the case when  $i \in D_\tau \setminus D_\sigma$ ,  $\mu_i = k + 1 - \tau(i)$  or vice versa.

The Footrule distance between two rankings  $\tau$  and  $\sigma$  is in fact the sum over the displacements of the items in  $D_\tau \cup D_\sigma$ . We can compute the most likely position of the affected items,  $i, j$  in  $\sigma$  as  $E[\sigma(i)] = \tau(i) + E[\mu]$  and  $E[\sigma(j)] = \tau(j) + E[\mu]$ , where  $E[\mu]$  is the average displacement between the items in two rankings within distance  $\lambda$  of each other.

To compute the average displacement  $E[\mu]$  for a given Footrule distance  $\lambda$  we need to compute all the displacements that contribute to that distance. The naive way to do this is to generate all top- $k$  rankings for a given  $k$ , compute the distances between all combinations of rankings, and compute the average item displacements for every distance. However, this is computationally very expensive, as generating all the rankings of size  $k$  has a complexity in  $O(k!)$ .

One key observation that allows us to more efficiently compute the sample space is the fact that the Footrule distance is a sum over non-negative integers, where each integer is a displacement of an item. In number theory and combinatorics, an unordered collection of positive integers whose sum is  $n$  is called a partition of  $n$ . Several efficient algorithms for generating all the partitions for a number, working in constant amortized time, have been proposed. Thus, we could use one of those algorithms to generate all the partitions for the distance  $\lambda$ . The resulting set of partitions could be used to compute the average displacement  $E[\mu]$ . Note that not all partitions of  $\lambda$  should be used in computing  $E[\mu]$ . The details of how exactly  $E[\mu]$  can be computed we leave out of this paper due to lack of space. Considering an update  $u_\tau(i, j)$  of  $\tau$ , one can compute the expected change  $E[F(\sigma, \sigma')]$  as:

$$E[F(\sigma, \sigma')] = (1 - \lambda) \times (2 \times |E[\sigma(i)] - E[\sigma(j)]|)$$

where  $\lambda$  is the distance between  $\tau$  and  $\sigma$ . The reasoning is that since these are the only two items that changed in  $\sigma'$  with respect to  $\sigma$ , the change can be computed as  $(2 \times |E[\sigma(i)] - E[\sigma(j)]|)$ . However, since  $\sigma$  is only similar to  $\tau$ , we do not want to fully propagate the change. Thus, we multiply the change by  $1 - \lambda$  (we normalize  $\lambda$ , thus  $0 \leq \lambda \leq 1$ ).

The above formula only covers the case when we assume that the affected items belong to the domains of both rankings. However, since we are working with top- $k$  rankings it can happen that the updated items in  $\tau$  cannot be found in  $\sigma$  at all. To eliminate the rankings that are so dissimilar to the updated ranking, and thus it is very unlikely that they changed, we define a maximum distance bound  $d_{max}$ . This bound is computed using the probability of not finding both updated items  $i$  and  $j$  in  $\sigma$ ,  $P(i, j \notin D_\tau \cap D_\sigma) = \frac{\binom{2 * (k - w)}{2}}{\binom{2 * k - w}{2}}$ , where  $w$  is the overlap between the rank-

	$\theta = 0.1$		$\theta = 0.15$	
	Precision	Recall	Precision	Recall
Our approach	0.58	0.6	0.59	0.41
Baseline	0.12	1	0.08	1

Table 1: Experimental results: Precision and Recall

ings. Since we only know the distance between the rankings, we can compute the maximum overlap that two rankings can have within a distance  $\lambda$  as  $w_{max} = \lceil \frac{1}{2} \times (-1 + \sqrt{1 - 4 \times \lambda + 4 \times k + 4 \times k^2}) \rceil + 1$  and then plug this value into the above probability estimation formula.

### 3. EXPERIMENTS

We have implemented the described algorithm in Java 8. We created one synthetic dataset by first creating a small set of base rankings, by randomly choosing at least  $\rho$  from  $k$  items, where  $k$  is the size of the rankings, and then randomly choosing the remaining  $k - \rho$  items. All the other rankings are created by swapping a random number of items from the base rankings. The dataset contains 500 rankings with size  $k = 10$ . We compared our approach with the baseline approach—retrieving all rankings that have at least one item in common with the affected ranking. For the experiments, we randomly selected one ranking from the dataset, randomly selected a pair of items from this ranking, and then swapped their places. We then used our method and the baseline to find the affected rankings in the dataset.

Table 1 reports the average precision and recall for the two approaches over 100 trials. We report on results for two values of the threshold  $\theta$ , 0.1 and 0.15. To compute  $d_{max}$ , we set  $P(i, j \notin D_\tau \cap D_\sigma)$  to 0.9. Note that the recall of the baseline is always 1 since the relevant rankings must have at least one item in common with the changed ranking. We can see that with our approach we can achieve high precisions (much higher than the baseline) while still maintaining relatively high recall.

### 4. RELATED WORK

Research around crowdsourcing information usually addresses the problem of reducing the cost, while still retaining high quality results. Guo et al. [3] address the problem of finding the highest ranked object using the least number of questions, from a set of objects, in a crowdsourcing database system. Wang et al. [5] use transitive relations to reduce the number of questions asked to the crowd for the case of crowdsourced joins. Gruenheid and Kossmann [2] investigate the cost and quality trade-offs of different algorithms in a crowdsourcing environments. Polychronopoulos et al. [4] propose an algorithm for creating top- $k$  lists using the crowd. The idea behind the algorithm is to create a high agreement top- $k$  list for a low latency and monetary cost, by adaptively choosing the number of tasks posed to the crowd. To the best of our knowledge, there has not been any work that focuses on maintaining a set of crowdsourced rankings.

### 5. REFERENCES

- [1] R. Fagin et al. Comparing Top  $k$  Lists. SIAM J. Discrete Math., 2003
- [2] A. Gruenheid and D. Kossmann. Cost and quality trade-offs in crowdsourcing. *DBCrowd*, 2013.
- [3] S. Guo et al. So who won?: dynamic max discovery with the crowd. *SIGMOD*, 2012.
- [4] V. Polychronopoulos et al. Human-powered top- $k$  lists. *WebDB*, 2013.
- [5] J. Wang et al. Leveraging transitive relations for crowdsourced joins. *SIGMOD*, 2013.

# Optimizing B+-Tree for PCM-Based Hybrid Memory

Lu Li<sup>1,2</sup>, Peiquan Jin<sup>1,2</sup>, Chengcheng Yang<sup>1,2</sup>, Zhanglin Wu<sup>1,2</sup>, and Lihua Yue<sup>1,2</sup>

<sup>1</sup> University of Science and Technology of China, Hefei, China, 230027

<sup>2</sup> Key Laboratory of Electromagnetic Space Information, Chinese Academy of Sciences, Hefei, China, 230027

jpg@ustc.edu.cn

## ABSTRACT

Phase change memory (PCM) as a newly developed storage medium has many attractive properties such as non-volatility, byte addressability, high density and low energy consumption. Thus, PCM can be used to build non-volatile main memory databases. However, PCM's long write latency and high write energy bring challenges to PCM-based memory systems. In this paper, we propose an improvement over the B+-tree for PCM. Particularly, we consider the read/write tendency of leaf nodes. For write-intensive leaf nodes, we use an overflow-node technique to reduce PCM writes, while for read-intensive ones, we adjust the tree structure to remove overflow nodes to improve read performance. Our experimental results suggest that our proposal outperforms the traditional B+-tree and the overflow B+-tree.

## CCS Concepts

- Information systems → Information storage systems
- Information storage technologies → Storage class memory.

## Keywords

B+-tree; Index; PCM

## 1. INTRODUCTION

The increasing needs for large energy-efficient main memory call for new types of memories, such as phase change memory (PCM). PCM is byte-addressable and supports random access. Compared with DRAM, PCM is non-volatile and is expected to have higher storage density in the future [1, 2]. However, two problems make it difficult to replace DRAM in current computer systems. First, the write latency of PCM is about 6 to 10 times slower than that of DRAM. Second, PCM cells have limited write endurance [3].

Therefore, a more practical way to utilize PCM in memory architecture is to use both PCM and DRAM to construct hybrid memory architecture [2]. In such hybrid memory architecture, the high density and non-volatility of PCM makes it possible to build non-volatile main-memory databases. However, due to the special properties of PCM, many existing database algorithms such as indexing have to be revised to take advantage of PCM.

In this paper, we focus on the indexing issue in PCM/DRAM-based hybrid memory systems. We aim to improve the traditional B+-tree to make use of PCM and DRAM efficiently. Specifically, we improve the traditional B+-tree in two aspects. First, we use an overflowing mechanism [4, 5] to reduce write operations to PCM, where each leaf node in the B+-tree is allowed to have several overflow nodes to keep newly inserted records when the leaf node is full. Thus, we can reduce the split operations on the index and consequently reduce writes

© 2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

and lengthen the lifetime of PCM. Second, we propose to predict the read/write tendency of index requests, based on which we use different ways to process index requests. Particularly, we use the overflowing scheme for write-intensive requests, but adopt a tree-adjusting operation to remove overflow nodes so as to improve read performance. We conduct experiments to evaluate our proposal and make comparison with the traditional B+-tree and the overflow B+-tree. The results suggest the efficiency of our proposal.

## 2. CB+-TREE

The structure of the CB+-tree is similar with the B+-tree, except that each leaf node can have a few overflow nodes. A leaf node without overflow nodes is called a *tra*, while a leaf node containing overflow nodes is called an *ovf*.

All the data stored in leaf nodes of CB+-tree is the same as that in the B+-tree. Nodes of the CB+-tree are ordered, thus binary search is allowed. Every node of the CB+-tree consists of a set of  $\langle key, value \rangle$  pairs and some auxiliary information. The auxiliary information includes *num\_keys*, *is\_leaf*, *parent*, and *brother*. Here, *num\_keys* denotes the number of  $\langle key, value \rangle$  pairs, *is\_leaf* indicates whether the node is a leaf node, and *parent* and *brother* denote the parent and the brother of the node, respectively.

In the CB+-tree, when an *ovf* leaf node has a read tendency, we remove it from the overflow chain and make it be a new *tra*. With this mechanism, we can reduce the number of overflow nodes and therefore improve read performance. As shown in Fig. 1, the leaf nodes LN1, LN2 are *tra* and LN3 is an *ovf*. If LN3 is required to be changed into a *tra*, we remove LN3 from its overflow chain and change it into a *tra*. Other *ovf* in the original overflow chain remain unchanged. Consequently, the overflow chain is changed into two segments, as shown in the right part of Fig. 1.

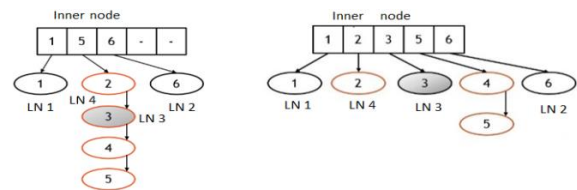


Figure 1. Structure adjustment of the CB+-tree.

The key issue in the CB+-tree is how to predict the read/write tendency ( $\mathcal{T}$ ) of a leaf node. Formally,  $\mathcal{T}$  is calculated by (1). Here,  $ratio_t$  and  $ratio_c$  are total and recent proportion of writes to all accesses on a leaf node, respectively.

$$\mathcal{T} = ratio_t * \delta + ratio_c * (1 - \delta), s.t. \delta \in [0, 1]. \quad (1)$$

We make use of a sliding window to record  $k$  latest read/write requests for a leaf node, and further get the value of  $ratio_c$ . Generally, a node is considered to be write-intensive when  $\mathcal{T}$  is higher than 0.7, and be read-intensive when  $\mathcal{T}$  is lower than 0.3. The read/write tendency prediction scheme works when  $\mathcal{T}$  is in 0.3-0.5 for an *ovf* and in 0.5-0.7 for a *tra*.

We perform the prediction by using the polynomial fit technique according to the recent access information in the sliding window. In addition, we do not trigger the prediction when the  $\mathcal{I}$  of an *ovf* drops a little below 0.7 immediately, but wait for some time to collect more information about the access pattern. This strategy is also applied to the change of the parameter *tra*.

### 3. EVALUATION

To observe the performance benefits of the proposed CB+-tree, we implemented three algorithms: the traditional B+-tree, the OB+-tree[4] and the CB+-tree, and we run them on a computer with Ubuntu 14.04, a CPU of AMD Athlon II X2, 4GB RAM, and 1 TB Seagate hard disk. In addition, we used DRAM to simulate PCM by artificially increasing write latency.

We used the TPC-C<sup>1</sup> workload to generate the traces in the experiments. When running the TPC-C workload, 10 warehouses and 100 clients are configured. The TPC-C workload contains eight index files built on eight tables, and the size of the tables is approximately 1 GB. We used BenchmarkSQL<sup>2</sup> to generate the TPC-C workload running on the open-source PostgreSQL, and collect page requests on the eight tables at the same time. We first performed insertions on the tables to get index insertions. Consequently, we performed 5.4 million insertions requests and the original B+-tree index file is about 135 MB. Then, we prepared a trace containing about 3.8 million index requests including 74.2 % searches, 23.8 % insertions, and 1% deletions.

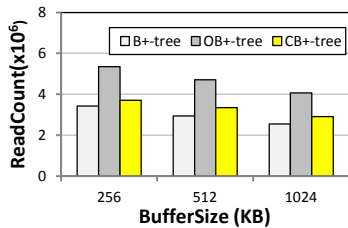


Figure 2. PCM read counts with different buffer sizes.

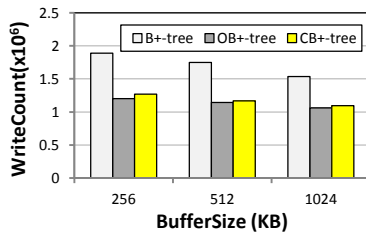


Figure 3. PCM write counts with different buffer sizes.

We first measure the read and write count of cache lines. Figures 2 and 3 show the read and write counts when varying the buffer size of each index. As shown in Fig. 2, the B+-tree has the least read count under all settings, because it does not involve any overflow nodes. However, as shown in Fig. 3, the B+-tree introduces more PCM write operations, which will shorten the lifetime of PCM and worsen the overall time performance. The OB+-tree has the highest read count because each leaf node in the OB+-tree is likely to contain a long chain of overflow nodes, which introduces more read operations. On the other side, the proposed CB+-tree has much less read operations compared with the OB+-tree, because it uses read/write tendency to dynamically remove overflow nodes.

Note that our index has a little more PCM writes than the OB+-tree. This is due to the read/write-tendency-based adjustment of the index structure. Figure 4 shows the run time of each index, which is normalized according to the run time of the B+-tree, i.e., the run time of the B+-tree is always set to 1. It indicates that the CB+-tree outperforms the B+-tree and the OB+-tree in terms of overall run time. As a result, the CB+-tree is able to balance the read and write costs, yielding a better indexing mechanism for PCM-based memory systems.

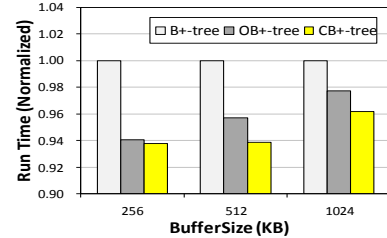


Figure 4. Run Time (normalized w.r.t. B+-tree).

### 4. CONCLUSIONS

PCM has been regarded as a new kind of future memories. In this paper, we proposed an efficient tree indexing approach called CB+-tree that is an improved version of the traditional B+-tree. We used the overflow-node design to reduce writes to PCM, and thus the endurance of PCM can be improved. We also proposed to predict the read/write tendency of index requests, based on which we performed necessary adjustment on the tree index to reduce additional read operations caused by overflow nodes. The comparative experimental results including read/write count and run time, show the efficiency and superiority of our proposal.

### 5. ACKNOWLEDGMENTS

This work is partially supported by the National Science Foundation of China (61379037 and 61472376).

### 6. REFERENCES

- [1] Qureshi, M. K., Srinivasan, V., Rivers, J. A. 2009. Scalable high performance main memory system using phase-change memory technology. In *ACM SIGARCH Computer Architecture News*, 37(3), 24-33.
- [2] Dhiman G, Ayoub R, Rosing T. 2009. PDRAM: a hybrid PRAM and DRAM main memory system. In *Proceedings of the 46th Design Automation Conference (San Francisco, CA, USA, July 26 - 31, 2009)*. ACM/IEEE, 2009: 664-669.
- [3] Wu, Z., Jin, P., Yue, L. 2015. Efficient space management and wear leveling for PCM-based storage systems. In *Proceedings of the 15th International Conference on Algorithms and Architectures for Parallel Processing (Zhangjiajie, China, November 18 - 20, 2015)*. LNCS 9531, 784-798.
- [4] Chi, P., Lee, W., Xie, Y. 2014. Making B+-tree efficient in PCM-based main memory. In *Proceedings of International Symposium on Low Power Electronics and Design (La Jolla, CA USA, August 11 - 13, 2014)*, ACM, 69-74.
- [5] Jin, P., Yang, C., Jensen, C. S., Yang, P., Yue, L., 2015. Read/write-optimized tree indexing for solid-state drives, *The VLDB Journal*, online: 1-23. DOI = <http://dx.doi.org/10.1007/s00778-015-0406-1>

<sup>1</sup> <http://www.tpc.org/tpcc/>

<sup>2</sup> <http://sourceforge.net/projects/benchmarksql/>

# A Data Mining Approach to Choosing Categorical Attributes for Ranked Lists<sup>\*</sup>

Koninika Pal  
 University of Kaiserslautern  
 Kaiserslautern, Germany  
 pal@cs.uni-kl.de

Sebastian Michel  
 University of Kaiserslautern  
 Kaiserslautern, Germany  
 smichel@cs.uni-kl.de

## ABSTRACT

This work proposes and evaluates a novel approach to determine interesting category for ranked lists using  $\nu$ -SVM. We identify three characteristics (features), entropy, unlikability, and peculiarity and show how to train a classifier on these features using a set of Wikipedia tables. The learned model is evaluated by relevance assessments obtained through a user study, reflecting the correctness of our approach.

## 1. INTRODUCTION

Understanding and exploring information is becoming more and more complex due to the dramatic growth of data. Scale, dynamics, and (schema) heterogeneity advocate for solely automated means, by which users can sit back and explore data already put in meaningful and interesting categories. In this work, we specifically look at ranked lists, a concise form of data summarization, that can be found in nearly all domains as virtually everything can be ranked, if not by measurable means then by crowdsourcing rankings. Given a ranked list, for instance, the list of tallest building in the world, we consider the task to decide which dimension is worth being used as a constraint to specialize the query. In database terminology, we are interested in determining OLAP-cube dimensions for the drill-down operation, but consider the case of data beyond a well understood database schema. Specifically, our approach is using statistical measures that can be computed from any table, no knowledge about the semantics of the schema or human input is required. Getting back to the above example, a list of tallest buildings by continent or country appears interesting, while a list of the tallest buildings that are 31 stories tall might be less important to be investigated, if at all. The key idea behind this work is to assume that it is feasible to train a classifier based on training data obtained from Web tables, such as tables in Wikipedia, assuming that the presence or absence of a table can act as an indicator of general interest or disinterest of humans in such a table.

### 1.1 Problem Statement, Setup, and Key Idea

Consider a set of rankings-style tables  $\mathcal{R}$ , where  $r \in \mathcal{R}$  is a ranking table with its attributes  $\mathcal{A}$ . A subset of  $\mathcal{A}$  is of

<sup>\*</sup>This work has been supported by the German Research Foundation (DFG) under grant MI 1794/1-1.

numeric type, denoted as  $\mathcal{N}$ . We divide  $\mathcal{A}$  in three kinds of attributes: (i) the subject of ranking denoted as  $a_s$  which represents the set of entities, (ii) the criterion of the ranking, denoted as  $a_{cr}$ , based on which the ranking order of subject entities are made and (iii) the remaining attribute considered as categorical attribute denoted as  $a_c$ . Hence, the ranking table  $r$  is written as  $r = (a_s, a_{cr}, a_{c1}, a_{c2} \dots)$ .

Each attribute  $a \in \mathcal{A}$  is associated with a set  $\mathcal{V}_a$  of possible values. Following the previous example, assigning a constraint on an attribute, for instance,  $a_{c1} = \text{'Canada'}$  where  $\text{'Canada'} \in \mathcal{V}_{a_{c1}}$  and  $c_1$  denotes the country a building is placed in, specifies the ranking of the tallest buildings in Canada. That is,  $(a_{c1}, \text{'Canada'})$  becomes the category for  $r_{new}$ .

Let  $\mathcal{I}$  be the complete set of interesting categorical attribute. Our objective is to create a classifier  $\mathcal{C}$  that can tell whether using a non-numeric categorical attribute as a constraint to a table would lead to an interesting “new” table or not, i.e.,

$$\mathcal{C}(a_c) = \begin{cases} \text{interesting,} & \text{if } a_c \in \mathcal{I} \text{ and } a_c \notin \mathcal{N} \\ \text{not interesting,} & \text{otherwise} \end{cases} \quad (1)$$

In this work, we opt for a classification-based approach using  $\nu$ -SVM as our classifier. Hence, we first need to determine training data  $\mathcal{T}$  of interesting categorical attributes, and we do so by harnessing a set of Wikipedia tables  $\mathcal{R}$ . **Based on our assumption, categorical attributes for a specific subject are considered interesting iff we find at least one ranking in Wikipedia that is created by imposing a constraints over that categorical attributes.** Formally,  $(a_s, a_c) \in \mathcal{T}$  iff  $\exists r, r_{new} \in \mathcal{R}$ . Note that the interestingness of a category is bound to the entity type (i.e., class of the subjects), which satisfied by the condition:  $r.a_s = r_{new}.a_s$ . Then, we learn the model using SVM on the training data and verify how accurate we can predict interesting category for ranking by evaluating it on held-out test data and by relevance assessments obtained from a user study.

## 2. WORKING MODEL

**Creation of Training Data:** Algorithm 1 retrieves interesting and non-interesting categorical attributes for a specific ranking subject (i.e.,  $(a_s, a_c)$ ). The constraints of a ranking table is parsed from the title/caption of the table or the title of the Wikipedia page by using propositions from the English dictionary, presented by the function in line 6 in Algorithm 1.

**Learning Interesting Categories:** In this work, we use the *soft-margin classifier*  $\nu$ -SVM [4] to learn the interesting characteristics of categorical attributes.  $\nu$ -SVM suits best for our purpose as it can detect outliers while learning. According to our intuition, conciseness and diversity of a categorical value of ranking entity is important to capture

---

**Algorithm 1** Generating Training Samples
 

---

```

1: procedure GENERATESAMPLES(wikitables)
2:   constraintsmap  $\leftarrow$  empty map(constraints, subjectList)
3:   interesting  $\leftarrow$  empty list(subject, attribute)
4:   nonInteresting  $\leftarrow$  empty list(subject, attribute)
5:   for  $\mathcal{T} \in$  wikitables do
6:      $\mathcal{T}.a_s, \mathcal{T}.cons \leftarrow$  parse_cons( $\mathcal{T}.title$ )
7:     constraintsmap[ $(\mathcal{T}.cons)$ ]  $\leftarrow$  ( $\mathcal{T}.cons, \mathcal{T}.a_s$ )
8:   for  $\mathcal{T} \in$  wikitables do
9:     [ $\mathcal{T}.a_c, \mathcal{V}_{A_c}$ ]  $\leftarrow$  parse( $\mathcal{T}.table$ )
10:    for  $a_c \in \mathcal{T}.A_c \setminus \mathcal{N}$  do
11:      for  $x \in \mathcal{V}_{A_c}$  do
12:        subjectList  $\leftarrow$  constraintsmap.contains[ $x$ ]
13:        if  $\mathcal{T}.a_s \in$  subjectList then
14:          interesting  $\leftarrow$  ( $\mathcal{T}.a_s, a_c$ )
15:          break;
16:        else
17:          noninteresting  $\leftarrow$  ( $\mathcal{T}.a_s, a_c$ )
18:  return interesting, noninteresting

```

---

human interests. Hence, we use the following three measures as features to feed our learning model.

**Shannon Entropy** reflects the uncertainty of information content of a discrete random variable. Here, we treat a categorical attribute  $a_c \in \mathcal{A} \setminus \mathcal{N}$  as a random variable, where  $\mathcal{V}_{a_c}$  is the set of possible values that  $a_c$  can hold. Shannon entropy is calculated by  $H(a_c) = -\sum_{x \in \mathcal{V}_{a_c}} P(x) \log_2 P(x)$ , where  $P(x) = \text{count}(x)/|\mathcal{T}|$ ,  $|\mathcal{T}|$  is the size of the ranking table. The normalized entropy is calculated as  $\hat{H}(a_c) = \frac{-\sum_x P(x) \log_2 P(x)}{\log_2 |\mathcal{T}|}$ ,  $x \in \mathcal{V}_{a_c}$ .

**Unlikeability** is a diversity measures for categorical attribute that measures how often the observation of random variables differs from one another [2], calculated as  $U(X) = 1 - \sum_{x \in \mathcal{V}_{a_c}} P(x)^2$ .

**Peculiarity** is another diversity measure for categorical value used here to measure the peculiarity which is defined by the probability that a randomly chosen categorical value has not been seen previously [2]. It is defined by  $D(X) = 1 - \sum_{x \in \mathcal{V}_{a_c}} \frac{\text{count}(x)(\text{count}(x)-1)}{|\mathcal{T}|(|\mathcal{T}|-1)}$

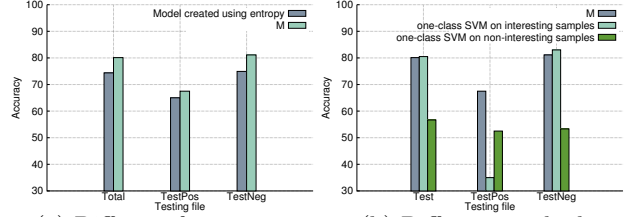
All three measures are normalized to  $[0, 1]$ , the values towards 1 indicate few or only one distinct categorical value of the ranking entity and a value towards 0 represents a large (maximum) diversity.

Thus, it is clear that a categorical attribute of an interesting ranking should have more tendency toward having a feature-value near the mid-range of  $[0, 1]$ , i.e., around 0.5. In contrast, uninteresting ranking would have a tendency toward having a feature value closer to 0 or close to 1.

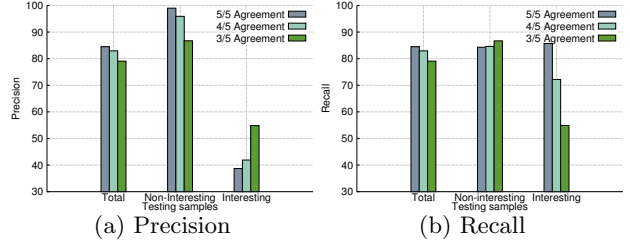
Clearly, the range of values is responsible for the classification of interestingness measures. Hence, we use the Radial Basis Function (RBF) kernel to map our data space to the dot product space needed for SVM .

### 3. EXPERIMENTS

**Setup:** We use the LIBSVM [1] tool to learn the models. 2,744 non-interesting and 158 interesting samples are extracted using Algorithm 1 from 2,045 ranking tables out of Wikipedia. 25% of samples are chosen randomly from each class, and also merged for testing purpose denotes as **TestPos** and **TestNeg**, and **Total**. The remaining non-interesting samples are divided into 10 smaller chunks and merged with the remaining interesting samples to create 10 training files, each containing 323 samples. We learn the respective model for all these files and chose the best performing one, denoted as **M** with accuracy of 80.11% for **Total**. According to our testing samples, the accuracy is a fraction of the correctly classified samples. For our training data, a feasible solution for SVM is found where  $0 \leq \nu \leq 0.73$  and



(a) Different feature set (b) Different method  
**Figure 1: Comparison among learning models**



(a) Precision (b) Recall  
**Figure 2: User study**

the optimal  $\nu = 0.51$ . Here, we also present the evaluation of a user study on test set of 130 randomly selected samples of  $(a_s, a_c)$ , i.e., classification tasks. We gathered five user relevance assessments per classification task. The accuracy of model **M** is evaluated by using ground truth that is built for different agreement levels of users.

**Validation of Models on Test Data:** In Figure 1(a), we can see that the accuracy increases at least 5% for all testing files except **TestPos** (2.5%) while using all three features together in learning. As our training data is unbalanced, a common option is to use the simpler one-class SVM model. However, Figure 1(b) shows that the model **M** is a better classifier than the model possible to create from a one-class SVM. **M** is also more robust as it classifies samples from each class better than the other two models.

**Results Based on User Study:** Figure 2 is showing precision and recall achieved by **M** with varying agreement level of user. We see that for a user agreement of 5/5 the highest accuracy is reached. That is, considering the tasks with 5/5 user agreement as ground truth of the test data, our model correctly identify more than 80% of the test cases (Figure 2). We use Fleiss Kappa to determine the reliability of user agreements. The 5/5 user agreement has a Kappa score 0.28, which denotes fair agreement according to a commonly cited interpretation of Kappa values [3]. Also, the 95% confidence interval for Kappa has range between 0.24–0.32 for the collected user data. These values significantly differ from 0 and, thus, prove the statistical significance of 5/5 agreement level for our user-study.

### 4. CONCLUSION

From the experimental evaluation, we can conclude that our model of classifying category to capture the interesting ranking performs very well. We also saw that all three features together create a better classification model than the commonly used entropy measure.

### 5. REFERENCES

- [1] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27, 2011.
- [2] G. D. Kader and M. Perry. Variability for categorical variables. *Journal of Statistics Education*, 15(2).
- [3] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33, 1977.
- [4] B. Schölkopf et al. New Support Vector Algorithms. *Neural Computation*, 12(5), 2000.

# Efficient Implementation of Joins over Cassandra DBs

Haridimos Kondylakis

FORTH-Inst. of Computer Science  
N. Plastira 100, 700 13 Heraklion,  
Crete, Greece  
kondylak@ics.forth.gr

Antonis Fountouris

Computer Science Department  
University of Crete,  
700 13 Heraklion, Greece  
afountour@gmail.com

Dimitris Plexousakis

FORTH-Inst. of Computer Science  
N. Plastira 100, 700 13 Heraklion,  
Crete, Greece  
dp@ics.forth.gr

## ABSTRACT

NoSQL databases provide new opportunities by enabling elastic scaling, fault tolerance, high availability and schema flexibility. Despite these benefits, their limitations in the flexibility of query mechanisms impose a real barrier for any application that has not predetermined access use-cases. One of the main reasons for this bottleneck is that NoSQL databases do not support joins. In this poster we present a solution that efficiently supports joins over such databases. More specifically, we present a query optimization and execution module placed on top of Cassandra clusters that is able to efficiently combine information stored in different column-families. Our preliminary evaluation demonstrates the feasibility of our solution and the advantages gained when compared to a recent commercial solution by DataStax. To the best of our knowledge our approach is the first and the only available open source solution allowing joins over NoSQL Cassandra databases.

## 1. INTRODUCTION

During the latest years, the explosive growth of data and the emerging requirements for big data management solutions led to the development of NoSQL databases. Among the reasons for the rapid adoption of NoSQL databases is that they scale across a large number of servers by horizontal partitioning of data items, they are fault tolerant and achieve high write throughput, low read latencies and schema flexibility. To achieve all these benefits, the main idea is that you have to denormalize your data model and avoid costly operations in order to speed up the database engine. As such, the NoSQL databases were initially designed to support only single-table queries and explicitly excluded the support for join operations allowing applications to implement such tasks. However, modern applications increasingly require the efficient combination of information from multiple tables and column-families.

To this direction the first approaches are starting to emerge for operators similar to join, based on Map-Reduce such as rank-join queries [1] and set-similarity joins [2]. Rank-join queries try to find the most relevant documents for two or more keywords whereas set-similarity joins are those that try to find similar pairs of records instead of exact ones. However, both these approaches execute joins at the application level using Map-Reduce implementations and the joins implemented do not focus on an exact matching of the joined tuples. This emerging need has also been recently recognized by DataStax, the biggest vendor of Cassandra NoSQL commercial products which recently introduced a commercial join-capable ODBC driver. The company claims that Cassandra can

This work was partially support by the iManageCancer (H2020-643529) and the MyHealthAvatar (FP7-600929) EU projects.

© 2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

now perform joins just as well as relational database management systems. However, no results were presented nor the specific join implementation algorithms and optimization techniques..

To fill these gaps, in this poster we present a naïve, yet efficient query optimization and execution module enabling joins over Cassandra NoSQL databases surpassing DataStax's commercial solution and highlighting the differences between NoSQL and relational solutions.

## 2. PRELIMINARIES

Cassandra is a NoSQL database developed by the Apache Software Foundation. It uses a hybrid model between key-value and column-oriented database. The structure of the database is defined by super-columns and column-families. In this paper the term column-family and table will be used interchangeably although they are not exactly the same.

All stored data can be easily manipulated using the Cassandra Query Language (CQL) which is based on the widely used SQL. CQL can be thought of as an SQL fragment with the following restrictions over the classical SQL:

- *R1.* Joins are now allowed.
- *R2.* You cannot project the value of a column without selecting first the key of the column. Every select query requires that you restrict all partition keys. Select queries restricting a clustering key have to restrict all the previous clustering keys in order. Queries that don't restrict all partition keys and any possibly required clustering keys, can run only if they can query secondary indices. To be allowed to run a query including more than two secondary indices, Cassandra requires that "*allow filtering*" is used in the query to show that you really want to do it. All Cassandra queries that require this run extremely slow and Cassandra's recommendation is to avoid running them. Tables can be stored sorted by clustering keys. This is the only case in which you are allowed to run range queries and *order by* clauses.
- *R3.* Unlike the projection in a CQL SELECT, there is no guarantee that the results will contain all of the columns specified because Cassandra is schema-optional. An error does not occur if you request non-existent columns.
- *R4.* Nested queries are not allowed, there is no "OR" operator and queries that select all rows of a table are extremely slow.

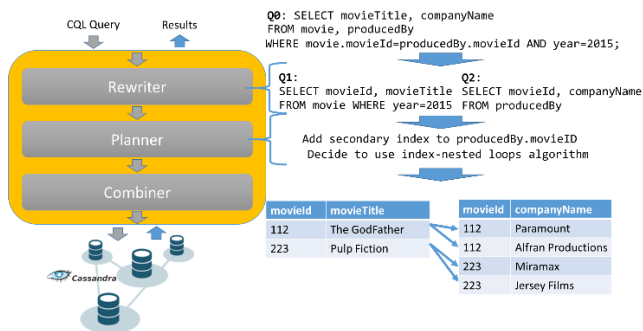
CQL statements change data, look up data, store data or change the way data is stored. A select CQL expression selects one or more records from Cassandra column family and returns a result-set of rows. Similarly to SQL each row consists of a row key and a collection of columns corresponding to the query.

## 3. QUERY OPTIMIZER & EXECUTION

Our query optimization and execution module can be placed on top of any Cassandra cluster and is composed of the following components, shown in Fig. 1:

a) *Rewriter*: The rewriter accepts the CQL query containing joins and creates the queries for accessing each individual column-family/tables. For example assuming that  $Q0$  is issued by the user, this module produces as output  $Q1$  and  $Q2$  as shown in Fig. 1.

b) *Planner*: This component plans the execution of the individual queries as constructed by the rewriter. First it identifies the available indices on the queried column-families and tries to comply with  $R2$ . For example, if the queries don't restrict all partition keys they can only run if there are available secondary indices on these keys. To satisfy this restriction the planner automatically generates secondary indices on the required fields. In our running example, a secondary index will be automatically generated to the *producedBy.movieID* column.



**Figure 1. Components of the optimization & execution module**

Besides trying to comply with all Cassandra restrictions the planner identifies which join algorithms should be used for executing the various joins by comparing the cost of left-deep trees. Currently two join algorithms have been implemented: a) a variation of Index-Nested Loops taking advantage of the existing indexes and additionally allowing joins over collection sets – indexed collections of elements (maps, sets and lists) supported after the Cassandra version 2.1; b) the sort-merge join allowing the join to be implemented in one pass over the data when the joined relations are indexed. When joining two column-families, if only one of them has an index on the joined field, the optimizer reads all rows from the non-indexed one and then uses the index for searching the indexed column-family. On the other hand when both column-families are sorted on the join column, the Sort-Merge join algorithm would be faster and is preferred by the optimizer.

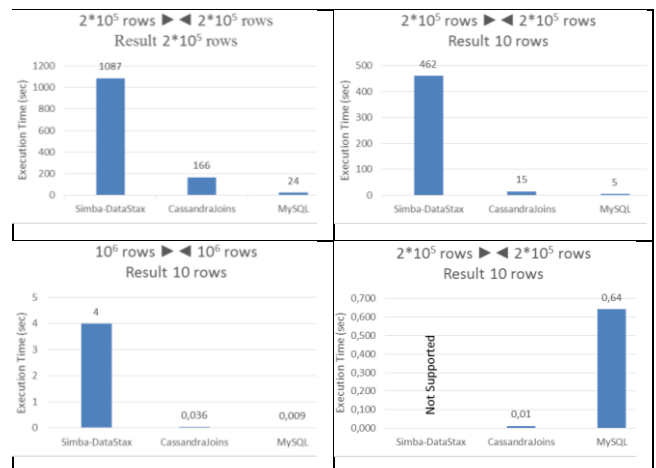
c) *Combiner*: This component executes the queries, calculates the join using the selected algorithm and returns the results to the users.

## 4. EVALUATION & CONCLUSION

All algorithms reported in this paper were implemented as a Java API named *CassandraJoins*. The API is going to be released soon under an open-source license. To perform a preliminary evaluation of our implementation we used a single Cassandra DataStax Community Server 2.1.5 x64 node running on a system with an I5 Intel Processor, 8GB of RAM on a Samsung SSD 850 EVO. We compared our approach with the Simba-DataStax ODBC 0.7 driver and with a MySQL Server CE 5.6.24. The execution time reported in each case is the average of 50 runs of each query execution.

The first series of experiments we performed tries to join two tables with a join on the indexed field. When we have indices on the joined field the *CassandraJoins* optimizer is using the Index-Nested Loops join algorithm whereas, when the input relations are sorted, the optimizer uses sort-merge join. We cannot identify the specific algorithm used by Simba-DataStax - the source code is not publicly available. The results are shown in Fig. 2 for different input and

output sizes. We can observe that *CassandraJoins* is by far more efficient than the Simba-DataStax implementation in all cases. For example, when joining column-families with  $2 \cdot 10^5$  rows each and the result is of the same size our approach needs 166 secs whereas Simba-DataStax ODBC driver needs 1087 secs. Obviously, when the selectivity of the query is increased the execution time is decreased. This is reasonable since Cassandra is known to be extremely slow when a query needs to retrieve all rows of a table, whereas it is extremely fast when only a small subset of the rows is selected. In addition, in all cases the implementation of Index-Nested Loops in a relational database (MySQL) is more efficient as shown in the third column of the graphs, whereas when the selectivity of the queries is high, our results are similar. However, we have to note that Cassandra scales linearly in a multi-node environment and we expect that our implementation will have even better results than MySQL when more nodes are used. Finally, to demonstrate the advantages of our implementation compared to a MySQL Database, we performed another experiment trying to join two column-families using collection indices. Since MySQL does not support collection indices the dataset has to be modelled using an additional indexed table. On the other hand Simba-DataStax does not support joins on collections. The results depicted in the last graph show that using *CassandraJoins* we need 0,01 sec whereas using MySQL we need 0,64 sec.



**Figure 2. Results of preliminary evaluation on a single node**

To the best of our knowledge our implementation is the only available non-commercial solution implementing joins over Cassandra databases. Our experiments demonstrate the advantages of our solution and confirm that our algorithms run efficiently and effectively. In all cases, we achieved better execution times than the commercial Simba-DataStax Driver currently available and our results are comparable to the execution times achieved in the relational database world. We have to note that our experiments were performed in an environment that favors relational databases (single node cluster). Surprisingly, our implementation is more efficient than relational databases when collection indices are used. The next step is to evaluate our implementation in a multi-node cluster with more data, to integrate our algorithms directly in the CQL language and to implement additional join methods.

## 5. REFERENCES

- [1] Ntarmos, N., Patlakas, I. Triantafyllou, P. 2014, Rank Join Queries in NoSQL Databases, PVLDB, 7(7), 493-504.
- [2] Kim, C., Shim, K. 2015. Supporting set-valued joins in NoSQL using MapReduce, IS Journal, 49, 52-64.

# Double Chain-Star: an RDF indexing scheme for fast processing of SPARQL joins

Marios Meimaris  
<sup>1</sup>University of Thessaly,  
<sup>2</sup>ATHENA Research Center,  
 Greece  
 m.meimaris@imis.athena-innovation.gr

George Papastefanatos  
 ATHENA Research Center,  
 Greece  
 gpapas@imis.athena-innovation.gr

## ABSTRACT

State of the art RDF stores often rely on exhaustive indexing and sequential (self-)joins for SPARQL query processing. However, query execution is dependent on, and often limited by the underlying storage and indexing schemes. Even though RDF can give birth to datasets with loosely defined schemas, it is common for an emerging structure to be present in the data. In this paper we introduce a novel indexing scheme, called Double Chain Star (DCS), that takes advantage of the inherent structure that is often found in RDF datasets by extending the notion of Characteristic Sets to cater for chain-star joins. DCS essentially reduces pairs of chain-star patterns that typically involve multiple self-joins, to mere index scans. We perform preliminary experiments and show promising results in comparison with Jena TDB and RDF-3X.

## Categories and Subject Descriptors

H.2.8 [Information Systems Applications]: Database Management—*Database Applications*

## Keywords

RDF, SPARQL, Query Processing, Query Optimization, Performance

## 1. INTRODUCTION

RDF and SPARQL are W3C recommendations for representing and querying graph data in the Data Web. There exists a rich body of literature for storing and querying RDF, however, many of these do not take advantage of the data's inherent structure in order to accelerate query processing. SPARQL optimizers depend on traditional methods for providing good query plans, including the use of data statistics and cardinality estimation for ordering triple patterns. The assumption of data independence imposes a risk of propagating errors in the planning process, especially in joins that reside deeper in the query plan. This can result in the creation of plans with large intermediate results between joins.

In this paper, we discuss a novel indexing scheme that aims to decrease the effects of bad estimates by quickly filtering triples that collectively participate in multiple joins, in one single scan. We extend the notion of Characteristic Sets, that typically represents the inherent structure of nodes in an RDF dataset, in order to characterize subject-object joins instead of single subject nodes. We call this *Extended Characteristic Sets* (ECS), and we discuss how ECS can be used for the construction of an inverted index, named *Double Chain-star*, that maps ECS's to collections of triples.

## 2. RELATED WORK

RDF stores often rely on the mapping of triples to relational settings, such as a single table with three columns representing subjects, predicates and objects (SPO) [2], or sets of *property tables* that are used for grouping instances of the same classes [7]. Other approaches include indexing of various SPO permutations. For example, RDF-3X [5] and Hexastore [6] make use of exhaustive indexing which includes all six permutations of subject-predicate-object, while exclusively relying on the indexes for the actual storage. Virtuoso [3] uses a large table for triples (quads), and a combination of full and partial indexes.

Characteristic Sets have been introduced as a way to provide better estimations for join cardinalities [4], and implemented in the RDF-3X high performance triple store. Brodt et al [1] discuss how an SPO index can be used to identify Characteristic Sets (CS) and use them for fast retrieval of star-shaped queries. We propose an extension of CS as a means to store and index triples, and enable scan-based answering of chain-star queries.

## 3. EXTENDED CHARACTERISTIC SETS

By definition, a Characteristic Set of a subject node  $s$  contains all properties  $p_i$  that appear in triples with  $s$  as subject. More formally, given a collection of triples  $D$ , and a node  $s$ , the Characteristic Set  $S_c(s)$  of  $s$ , as given by [4], is:

$$S_c(s) = \{p \mid \exists o : (s, p, o) \in D\}$$

and the set of all  $S_c$  for a dataset  $D$  is:

$$S_c(D) = \{S_c(s) \mid \exists p, o : (s, p, o) \in D\}$$

The upper bound for  $|S_c(D)|$  is  $|D|$ , but the existence of an inherent structure in RDF data makes the distinct set of Characteristic Sets that appear in real-world data small [4]. We introduce the *Extended Characteristic Set* ECS, as the



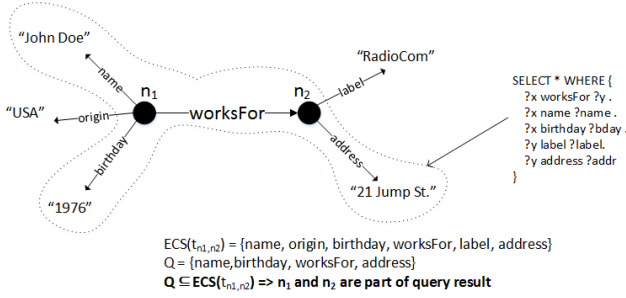


Figure 1: Double chain-star graph consisting of a star-shaped graph around  $n_1$  and a star-shaped graph around  $n_2$ .

union of the properties that appear in the subject  $t_s$  and the object  $t_o$  of a triple  $t$ , i.e. the union of the Characteristic Sets of  $t_s$  and  $t_o$ :

$$E_c(t) = \{p_1 \cup p_2 \mid \exists o_1 : (t_s, p_1, o_1) \in D \text{ and } \exists o_2 : (t_o, p_2, o_2) \in D\}$$

or simply:

$$E_c(t) = \{S_c(t_s) \cup S_c(t_o)\}$$

The set of all ECS in  $D$  is given by:

$$E_c(D) = \{E_c(t) \mid \exists p : (t_s, p, t_o) \in D\}$$

Essentially,  $E_c(t)$  helps to quickly identify the largest superset of graph patterns that contain *double chain-stars* consisting of a star pattern around  $t_s$ , a star pattern around  $t_o$ , and a subject-object join between  $t_s$  and  $t_o$ , with  $t_s$  being the subject and  $t_o$  the object in a common triple. An example double chain-star graph pattern is shown in Figure 1, where nodes  $n_1$  and  $n_2$  are present in the same triple  $t_{n_1, n_2}$  as subject and object respectively, and descriptive star patterns are present for each of the two nodes. A Characteristic Set  $S_c(s)$  is also an *ECS*, meaning that *leaf* star patterns are also parts of the ECS space. Preliminary experiments have shown  $|E_c(D)|$  to be low, specifically for LUBM100 with  $\sim 15m$  triples, this number is 109, for 27 distinct CSs.

## 4. THE DOUBLE CHAIN-STAR INDEX

The Extended Characteristic Sets of a given dataset can act as filters for collections of triples that fulfil a certain query pattern, reducing costly subject-object joins to mere index scans. For this to be feasible, we propose the Double Chain-Star (DCS) index, which is essentially an inverted index that maps collections of triples to ECSs. A triple  $t$  is mapped to an ECS if the ECS contains properties that appear in triples of either the subject, or the object of  $t$ , and  $t$  cannot belong to more than one ECS. Because  $ECS(t)$  contains all properties (outgoing edges) from nodes  $t_s$  and  $t_o$ , it also contains all subsets of properties for these two nodes. Therefore, we can check if an incoming query pattern  $q$  is a subset of  $ECS(t)$ , in which case the nodes of  $ECS(t)$  are potential candidates for evaluating  $q$ . Physically, each *ECS* can be represented as a bit vector, where each bit represents a property in an (ordered) set of properties  $P$  appearing in  $D$ . Assuming a dictionary of properties and their position in  $P$ , an incoming query  $q$  can be split to a set of chain-star sub-queries  $q_1, q_2, \dots, q_n$ , and each  $q_i$  in  $q$  will be represented as a bit vector that instantiates the bits corresponding to the properties in  $q_i$ . Therefore, if  $q_i \subseteq ECS(t)$ , the triples

Table 1: Query execution runtime in seconds.

	Jena	RDF-3X	DCS
LUBM10	248.66	8.24	0.56
LUBM100	timeout	timeout	29.58

mapped to  $ECS(t)$  effectively contribute to the evaluation of  $q_i$ . An example can be seen in Figure 1. The properties need to maintain their interesting order throughout sequential updates, which is left as future work. Patterns where a subject is joined with more than one objects with their own star-shaped graphs, are subject to cost estimation, in order to determine the order in which the DCS index should be accessed.

We implemented a naive version on top of Jena TDB, with the DCS index kept in-memory, and compared with Jena TDB (default stats-based optimizer) and RDF-3X, measuring wall-clock time (time out after 6 hours), for LUBM10 and LUBM100. For a *DISTINCT \** query with three double chain-star patterns, DCS outperforms the rest by orders of magnitude. For LUBM100 the other two approaches failed to answer completely, while ours needed a few seconds. The results can be seen in Table 1.

## 5. CONCLUSIONS AND FUTURE WORK

In this paper, we introduce the notion of *Extended Characteristic Sets* and propose the DCS indexing scheme, an inverted index for fast retrieval of double chain-star patterns that are present in many types of queries. Through these structures, we intend to reduce the time spent in sequential (self-)joins of stars forming around subject-object joins into mere index scans, and provide an implementation for storing and querying RDF data that provides faster query answering even for complex types of queries.

**Acknowledgements.** This work is supported by the EU-funded ICT project "DIACHRON" (agreement no 601043).

## 6. REFERENCES

- [1] A. Brodt, O. Schiller, and B. Mitschang. Efficient resource attribute retrieval in rdf triple stores. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1445–1454. ACM, 2011.
- [2] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient sql-based rdf querying scheme. In *Proceedings of the 31st international conference on Very large data bases*, pages 1216–1227. VLDB Endowment, 2005.
- [3] O. Erling and I. Mikhailov. *Virtuoso: RDF support in a native RDBMS*. Springer, 2010.
- [4] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 984–994. IEEE, 2011.
- [5] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113, 2010.
- [6] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *VLDB Endowment*, 1(1):1008–1019, 2008.
- [7] K. Wilkinson, C. Sayers, H. A. Kuno, D. Reynolds, et al. Efficient rdf storage and retrieval in jena2. In *SWDB*, volume 3, pages 131–150. Citeseer, 2003.

# Minoan ER: Progressive Entity Resolution in the Web of Data

Vasilis Efthymiou  
Univ. of Crete &  
ICS-FORTH  
vefthym@ics.forth.gr

Kostas Stefanidis  
ICS-FORTH  
kstef@ics.forth.gr

Vassilis Christophides  
Univ. of Crete &  
INRIA Paris-Rocquencourt  
Vassilis.Christophides@inria.fr

## ABSTRACT

Entity resolution aims to identify descriptions of the same entity within or across knowledge bases. In this work, we present the Minoan ER platform for resolving entities described by linked data in the Web (e.g., in RDF). To reduce the required number of comparisons, Minoan ER performs blocking to place similar descriptions into blocks and executes comparisons to identify matches only between descriptions within the same block. Moreover, it explores in a pay-as-you-go fashion any intermediate results of matching to obtain similarity evidence of entity neighbors and discover new candidate description pairs for resolution.

## 1. DESCRIPTION

Over the past decade, numerous knowledge bases (KBs) have been built to power large-scale knowledge sharing, but also an entity-centric Web search, mixing both structured data and text querying. These KBs offer comprehensive, machine-readable descriptions of a large variety of real-world entities (e.g., persons, places, products, events) published on the Web as Linked Data (LD). Although KBs (e.g., DBpedia, Freebase) may be derived from the same data source (e.g., a Wikipedia entry), they may provide multiple, non-identical descriptions of the same real-world entities. This is mainly due to the different information extraction tools and curation policies employed by KBs, resulting to complementary and sometimes conflicting entity descriptions. Entity resolution (ER) aims to identify descriptions that refer to the same real-world entity appearing either within or across KBs [2, 3]. Compared to data warehouses, the new ER challenges stem from the openness of the Web of data in describing entities by an unbounded number of KBs, the semantic and structural diversity of the descriptions provided across domains even for the same real-world entities, as well as the autonomy of KBs in terms of adopted processes for creating and curating entity descriptions. In particular:

- The number of KBs (aka RDF datasets) in the Linking Open Data (LOD) cloud has roughly tripled between

2011 and 2014 (from 295 to 1014), while KBs interlinking dropped by 30%. The main reason is that with more KBs available, it becomes more difficult for data publishers to identify relations between the data they publish and the data already published. Thus, the majority of KBs are sparsely linked, while their popularity in links is heavily skewed. Sparsely interlinked KBs appear in the periphery of the LOD cloud (e.g., Open Food Facts, Bio2RDF), while heavily interlinked ones lie at the center (e.g., DBpedia, GeoNames). Encyclopaedic KBs, such as DBpedia, or widely used geo-referencing KBs, such as GeoNames, are interlinked with the largest number of KBs [6].

- The descriptions contained in these KBs present a high degree of semantic and structural diversity, even for the same entity types. Despite the Linked Data principles, multiple names (e.g., URIs) can be used to refer to the same real-world entity. The majority (58.24%) of the 649 vocabularies currently used by KBs are proprietary, i.e., they are used by only one KB, while diverse sets of properties are commonly used to describe the entities both in terms of types and number of occurrences even in the same KB. Only YAGO contains 350K different types of entities, while Google's Knowledge Graph contains 35K properties, used to describe 600M entities.

The two core ER problems, namely how can we (a) effectively compute similarity of entity descriptions and (b) efficiently resolve sets of entities within or across sources, are challenged by the large scale (both in terms of the number of sources and entity descriptions), the high diversity (both in terms of number of entity types and properties) and the importance of relationships among entity descriptions (not committing to a particular schema defined in advance). In particular, in addition to *highly similar* descriptions that feature many common tokens in values of semantically related attributes, typically met in the center of the LOD cloud and heavily interlinked mostly using owl:sameAs predicates, we are encountering *somehow similar* descriptions with significantly fewer common tokens in attributes not always semantically related, that appear usually in the periphery of the LOD cloud and are sparsely interlinked with various kinds of predicates. Plainly, the coming up of highly and somehow similar semi-structured entity descriptions requires solutions that go beyond those applicable to duplicate detection. A promising area of research in this respect is cross-domain similarity search and mining [8, 7], aiming to exploit similarity of objects described by different modalities (i.e., text,

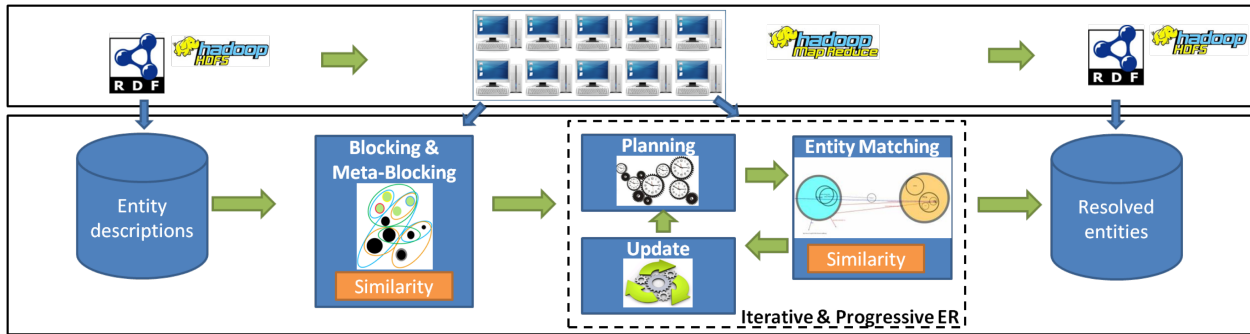


Figure 1: The Minoan ER Framework.

image) and contexts (i.e., facets) and support research by analogy. Such techniques could be also beneficial for matching highly heterogeneous entity descriptions and thus support ER at the Web scale.

We present in this poster the Minoan ER platform for resolving entities described by linked data in the Web (e.g., in RDF). Figure 1 illustrates the general steps involved in our process.

**Blocking and Meta-blocking in Minoan ER:** We use *blocking* as a pre-processing step for ER to reduce the number of required comparisons. Specifically, blocking places similar entity descriptions into blocks, leaving to the entity matching algorithm the comparisons only between descriptions within the same block.

Typically, token-based blocking algorithms place highly similar descriptions (having many common tokens) in many common blocks; intuitively, the more common blocks two descriptions share, the more likely it is that they match. This leads to many repeated comparisons between the same pairs of descriptions. To overcome this problem, we accompany blocking with *meta-blocking*, which prunes such repeated comparisons. Moreover, meta-blocking aims at discarding comparisons between descriptions that share few common blocks and are thus less likely to match. In Minoan ER, to support a Web-scale resolution of heterogeneous and loosely structured entities across domains, we use algorithms for blocking and meta-blocking that disregard strong assumptions about knowledge of the data schema and rely on a minimal number of assumptions about how entities match (e.g., when they feature a common token in their descriptions or URIs) within or across sources. For doing so, we exploit the parallel processing power of a computer cluster via Hadoop MapReduce, as presented in [5, 4].

**Progressive Entity Matching in Minoan ER:** Blocking approaches in the Web of data, especially when handling somehow similar descriptions appearing in the periphery of the LOD cloud, may miss highly heterogeneous matching descriptions featuring few common tokens [5]. To overcome that, we focus on exploiting the partial matching results as a similarity evidence for their neighbor (i.e., linked) descriptions. Since this inherently iterative process entails an additional overhead, we are interested in maximizing its benefit, given a computational cost budget. So, we need to estimate which part of the graph is the most promising to explore in the next iteration, in a *progressive* way.

In this respect, Minoan ER focuses on extending the typical ER workflow with a *scheduling phase*, which is responsible for selecting which pairs of descriptions, that have re-

sulted from blocking, will be compared in the entity matching phase and in what order. The goal of this new phase is to favor more promising comparisons, i.e., those that are more likely to increase the targeted benefit. This way, those comparisons are executed before less promising ones and thus, higher benefit is provided early on in the process. The *update phase* propagates the results of matching, such that a new scheduling phase will promote the comparison of pairs that were influenced by the previous matches. This iterative process continues until the cost budget is consumed.

In contrast to existing works in progressive relational ER (e.g., [1]), which consider the quantity of entity pairs resolved, as the benefit of ER, we explore different aspects of data quality, improved through ER. In particular, we are interested in characterizing the quality of the resolved pairs, with respect to the number of descriptions resolved, corresponding to the same real-world entity (targeting attribute completeness), the number of real-world entities resolved (targeting entity coverage), and the number of real-world entity graphs resolved (targeting relationship completeness).

**Acknowledgements:** This work was partially supported by the EU H2020 PARTHENOS (#654119), FP7 DIACHRON (#601043) and FP7 SemData (#612551) projects.

## 2. REFERENCES

- [1] Y. Altowim, D. V. Kalashnikov, and S. Mehrotra. Progressive approach to relational entity resolution. *PVLDB*, 7(11):999–1010, 2014.
- [2] V. Christophides, V. Eftymiou, and K. Stefanidis. *Entity Resolution in the Web of Data*. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool Publishers, 2015.
- [3] X. L. Dong and D. Srivastava. *Big Data Integration*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2015.
- [4] V. Eftymiou, G. Papadakis, G. Papastefanatos, K. Stefanidis, and T. Palpanas. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. In *IEEE Big Data*, 2015.
- [5] V. Eftymiou, K. Stefanidis, and V. Christophides. Big data entity resolution: From highly to somehow similar entity descriptions in the Web. In *IEEE Big Data*, 2015.
- [6] M. Schmachtenberg, C. Bizer, and H. Paulheim. Adoption of the linked data best practices in different topical domains. In *ISWC*, pages 245–260, 2014.
- [7] A. Shrivastava, T. Malisiewicz, A. Gupta, and A. A. Efros. Data-driven visual similarity for cross-domain image matching. *ACM Trans. Graph.*, 30(6):154, 2011.
- [8] Y. Zhen, P. Rai, H. Zha, and L. Carin. Cross-modal similarity learning via pairs, preferences, and active supervision. In *AAAI*, pages 3203–3209, 2015.

# Proposal of a Database Type and Aggregation Function for Accelerating Medical Genomics Study on RDBMS

Yoshifumi Ujibashi  
Fujitsu Laboratories Ltd.  
Kawasaki, Japan  
ujibashi@jp.fujitsu.com

Motoyuki Kawaba  
Fujitsu Laboratories Ltd.  
Kawasaki, Japan  
kawaba@jp.fujitsu.com

Lilian Harada  
Fujitsu Laboratories Ltd.  
Kawasaki, Japan  
harada.lilian@jp.fujitsu.com

## ABSTRACT

Next generation sequencing (NGS) and the recent development of efficient algorithms for genomic analysis are contributing to the understanding of human genetic variation and thus to personalized medicine. Among those genomic analysis, disease-causal gene analysis that finds genes relevant to specific diseases has received much attention. In this paper, we present our work on extending the PostgreSQL open source relational database management system (RDBMS) to efficiently handle genomic analysis. We introduced a new genome data type and a genome type aggregation function that drastically improved the performance of a typical query for disease-causal gene analysis by a factor of 50 to 360.

## 1. INTRODUCTION

Human beings have a sequence of three billions deoxyribonucleic acid (DNA) molecules which contains some millions of variants called “gene variants” that cause individual variations. Each individual has personal types of gene variants called “genotypes” which are combinations of nucleotide derived from chromosome dipoles. Figure 1 illustrates an example of genotypes. Individual 0 has genotype ‘C/C’ at gene variant 0, and genotype ‘T/C’ at gene variant 1. On the other hand, individual 1 has genotype ‘A/C’ at gene variant 0, and ‘T/T’ at gene variant 1.

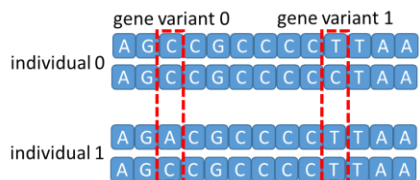


Figure 1: Chromosome dipoles of two individuals

The improvement of processing performance and the reduction of running cost of NGS, and the recent development of efficient algorithms for genomic analysis have resulted in an enormous increase in the amount of data of gene variants like SNP (single nucleotide polymorphism) and INDEL (insertion-deletion polymorphism). These gene variants are leveraged in a variety of studies such as cohort study, inheritance history study and disease-causal gene study. Disease-causal gene study aims to find the genes relevant to specific diseases and clarify the reasons of these diseases. The typical processing in such a study is to first filter the

individuals by some patient clinical condition (e.g. case-control), and then, find the genes whose frequency of its genotype is different between the filtered group and the rest. Normally the genetic data is delivered in flat-files (VCF [4]), and the patient information data, for instance, demographic information as gender/race/age, clinical information, and lifestyle information, are usually stored/managed in RDBMS. Recently, some studies integrate the genetic data and the patient data to be managed in RDBMS [1] [2]. Although the RDBMS approach improves the data manageability and usability, the improvement of the procedure and performance of the analysis processing in finding out the genes of interest remains a big challenge because of the very huge amount of gene variants.

In this paper, we present our work on extending the PostgreSQL [3] open RDBMS with a new data type and a new aggregation function called “genome type” and “genome type aggregation function”, respectively. Some preliminary examination shows that our approach is promising and can improve the execution time of disease causal gene analysis processing by a factor of 50 to 360.

## 2. Conventional Methods

### 2.1 Database Schema

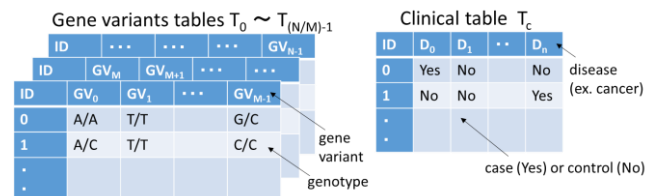


Figure 2: Conventional database schema

Figure 2 shows the database schema composed of tables that contain the N gene variants (GV<sub>0</sub>...N-1) with the associated genotype for each individual. Note that since there is a limit on the number of columns a table can contain, in this case M, there are N/M tables (T<sub>0</sub>...N/M-1) to store all the M gene variants of the individuals. Figure 1 shows also a clinical table (T<sub>c</sub>) with information related to n diseases (D<sub>0</sub>...n-1) for each individual. Other tables containing information about the patients (lifestyle, demographics, etc.) can also be necessary to properly describe the individuals.

### 2.2 Naïve Method

A naïve method counts the number of occurrence of each genotype for all the gene variants of patients with a specified disease. SQL 1 shows the SQL statement that calculates the distribution of genotype of a gene variant (GV<sub>0</sub>) for patients who have a specified disease (D<sub>0</sub>). This query is executed N times (i.e. for each GV<sub>0</sub>...N-1). It takes 1,530ms on PostgreSQL to execute the above query on 150,000 individuals for each gene variant GV<sub>i</sub>. For

```
SELECT count(T0.GV0) FROM T0, Tc
GROUP BY T0.GV0
WHERE T0.ID = Tc.ID and Tc.D0 = 'Yes'
```

**SQL 1: SQL for naïve method**

the usual case of 3,000,000 gene variants, it would take more than 50 days. There are two major reasons for such a long execution time. One is the huge number of gene variants  $N$  and thus, the huge number of corresponding SQL queries that have to be processed. The other is the long processing time of the query for the join between  $T_i$  and  $T_c$ . Note that in real cases the query could contain more joins to include lifestyle and demographics that are usually stored in other tables.

### 2.3 External Count Method

In order to decrease the number of queries and join operations, a method composed of an SQL that first retrieves all the genotypes of the gene variants for the individuals with a specified disease, and then an external application that counts the number of each genotype using the result of the query, is analyzed.

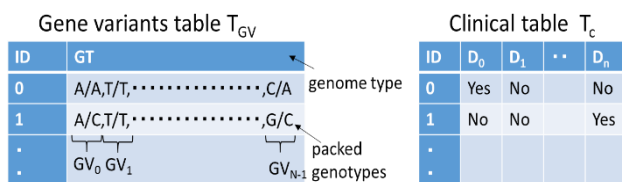
```
SELECT T0.GV0, T0.GV1, ...T0.GVM-1 FROM T0, Tc
WHERE T0.ID = Tc.ID and Tc.D0 = 'Yes';
```

**SQL 2: Pseudo SQL for external count method**

SQL 2 shows the SQL statement that retrieves the genotypes of gene variants ( $GV_0..GV_{M-1}$ ) on Table  $T_0$  for the patients with disease  $D_0$ . Note that a similar query is executed  $N/M$  times, i.e., for all gene variants tables  $T_0..T_{N/M-1}$ . The result of the SQL queries is input into an external program that counts the distribution of each genotype for all gene variants. It still takes about 5,500s, including both the PostgreSQL processing and the external processing, for the case of 1,000 individuals and 3,000,000 variants. This method still has two problems. First, since there is a limit on the number of columns a table can contain, many tables are necessary to store millions of variants and thus, the costly join processing of those tables with clinical and other tables cannot be avoided. Second, external processing causes a huge amount of data flow from the RDBMS to external application and thus, a high cost transfer time is necessary.

### 3. Proposed Method

In order to solve these problems, we proposed a method that integrates all the genotypes of the gene variations in a single table, making possible an efficient counting of the genotypes. We created a special database type and a special aggregation function called “genome type” and “genome aggregation function”, respectively.



**Figure 3: Proposed database schema**

Figure 3 illustrates the gene variants table ( $T_{GV}$ ) with the genome type (GT) column that packs all genotypes of all gene variants for each individual, enabling an efficient storing and scanning of the genotype data for its aggregation. SQL 3 shows the SQL statement with the proposed genome type aggregation function  $f_{\text{geno\_count}}()$ . Since all the genotypes of the gene variations are contained in GT, the genome aggregation function can efficiently count up through all genotypes of each individual at once. And only

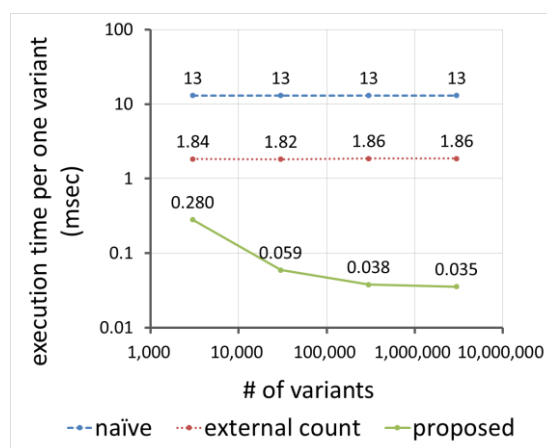
```
SELECT fgeno count(TGV.GT) FROM T0
WHERE TGV.ID = Tc.ID and Tc.D0 = 'Yes';
```

**SQL 3: SQL for proposed method**

a single execution of the join processing with other tables as the clinical table  $T_c$  is necessary.

### 4. Evaluation

We implemented our proposed method on PostgreSQL, and compared its performance with the conventional methods presented in Section 3. We used the machine whose CPU is Xeon CPU E5-2680 0 @2.70GHz x2 and memory is DDR3 128GB.



**Figure 4: Execution time for the three methods**

Figure 4 shows the execution time per gene variant for the case of 1,000 individuals for the naïve, the external count, and our proposed method, when varying the number of gene variants packed in GT. For the naïve method, the execution time per variant is 13ms, and for the external method is 1.86ms. On the other hand, the execution time for our method improves when increasing the number of packed variants, and it is reduced to 0.035ms which represents an improvement factor of about 50 to 360 over the conventional methods.

### 5. Conclusion

We proposed a new database type and new aggregation function as extensions to PostgreSQL for genomic analysis. Our preliminary evaluation showed that it can greatly improve the processing time of a typical query in medical genomics study. We are now working on further performance improvements for the genome aggregation function using dictionary and vectorization techniques, which we plan to report in detail in a future paper.

### 6. REFERENCES

- [1] Ameur, A., Bunkikis, I., Enroth, S., et al. (2014) CanvasDB: a local database infrastructure for analysis of targeted- and whole genome resequencing projects. *Database*, Vol. 2014, Article ID bau098
- [2] Paila, U., Chapman, B.A., Kirchner, R. (2013) GEMINI: integrative exploration of genetic variation and genome annotations. *PLOS Comput. Biol.*, 9, e1003153
- [3] The PostgreSQL Global Development Group. (1996-2015) *PostgreSQL* <http://www.postgresql.org/>
- [4] 1000 Genomes Project (2015) *The Variant Call Format (VCF) Version 4.2 Specification* <http://samtools.github.io/hts-specs/VCFv4.2.pdf>

# The Best Bang for Your Buck

When SQL Debugging and Data Provenance Go Hand in Hand

Benjamin Dietrich    Tobias Müller    Torsten Grust

Universität Tübingen  
Tübingen, Germany


[ b.dietrich, to.mueller, torsten.grust ]@uni-tuebingen.de

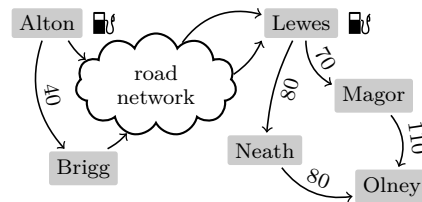
## ABSTRACT


We report on our ongoing effort to develop *observational debuggers* for SQL. This debugging paradigm—in which the evaluation of selected subexpressions may be “spied on”—fits the nature of query languages, but may lead to observations whose size can overwhelm users. Here, we tackle this challenge with the help of *data provenance analysis*. The analysis identifies exactly those input rows that are material in producing suspect query outputs. Running the debugger on such a minimized input will exclusively yield observations that are indeed relevant in understanding the bug.

## 1. SPYING ON SQL EVALUATION

SQL queries are prone to bugs much like code written in conventional programming languages. The present work investigates debugging paradigms that fit the declarative nature of SQL and, in particular, shield users from low-level internals (like execution plans, for example). We argue that *observational debugging* [5], an idea rooted in the logic and functional programming communities, is one such paradigm: users *mark* the “suspect” subexpressions—ranging from simple arithmetics to entire subquery blocks—of a buggy SQL query to *observe* their value at runtime. Seeing the difference between the expected and observed evaluation of a subexpression has turned out to be an effective tool in uncovering subtle SQL bugs [3].

A sample debugging scenario is depicted in Figure 1. Tables `cities` and `roads` jointly model a road network in which only selected cities host fueling stations (label  in Figure 1(a), 0/1 in column `fuel` of table `cities`). Which cities can we reach from `Alton` if our car has a maximum range of 100km before it needs to be refueled? An attempt to answer this question is the recursive SQL query of Figure 3. The query emits table `hops(city, range)` in which a row  $\langle c, r \rangle$  indicates that we can reach city  $c$  with a residual range of  $r$  (see Figure 2). The result looks suspicious, though: we are






(a) Simple road network with travel distances and fueling stations (). Which cities can we reach from Alton?

cities	
city	fuel
Alton	1
Brigg	0
Corby	0
Hedon	0
Lewes	1
Magor	0
Neath	0
Olney	0


(b) Table cities.

roads		
here	dist	there
Alton	40	Brigg
Brigg	30	Corby
Hedon	40	Lewes
Lewes	80	Neath
Lewes	70	Magor
Magor	110	Olney
Neath	80	Olney


(c) Table roads.

Figure 1: A debugging scenario: a relational model of cities and their connecting roads, parts of the instance hidden behind . (Disregard the provenance labels  and  until you reach Section 2.)

able to reach `Olney` although the city is farther from the last fueling station (in `Lewes`) than our maximum reach.

Pursuing the observational debugging paradigm, users mark parts of the buggy query ( in Figure 3) to learn about the evaluation of selected subexpressions. Markings typically start out large and then gradually zoom in on query details until the source of the bug can be observed directly. In keeping with the relational data model, the debugger presents observations in tabular form (Figure 4). Given the particular markings ① to ④ of Figure 3, one row in the observation table shows our current location (column ②) and the range available (possibly after refueling, column ④) before we travel `r.dist` kilometers (column ③)

city	range
Alton	0
Brigg	60
...	...
Magor	90
Neath	80
Magor	50
Olney	0
Neath	40

Figure 2: Final (but incorrect) hops table. The  identifies one questionable output: we did not expect to reach `Olney`.

```

1 WITH RECURSIVE hops(city, range) AS (
2   VALUES ('Alton', 0)
3 UNION ALL
4   SELECT r.there AS city,
5         h.range + c.fuel * 100 - r.dist AS range
6 FROM   cities AS c, roads AS r, hops AS h
7 WHERE  h.city = c.city
8 AND    h.city = r.here
9 AND    h.range + c.fuel * 100 >= r.dist
10 )
11 SELECT *
12 FROM hops;

```

Figure 3: Users place markings (①-④) to observe the evaluation of suspect SQL subexpressions.

to reach the next city (column ①). At recursion depths 9 and 10 we observe suspicious ranges which exceeded the maximum of 100 (see the 160 km range marked by  $\zeta$ , for example). This suggests that the query’s range computation is to blame (see [2] for the complete story behind the hunt for the bug of Figure 3).

## 2. MAKE EVERY OBSERVATION COUNT

Observations may be sizeable, however, and it can be a true challenge to spot enlightening details like  $\zeta$  in Figure 4. The sheer size of the input tables as well as the marking of subexpressions that are *evaluated before* aggregates or filters reduce data volume may lead to huge observations that do not reveal much. Indeed, in Figure 4 the lion’s share of our observations hides behind the ellipses (:), the majority of which contribute nothing to the understanding of the bug.

It is here where we propose to join two strands of work that have evolved independently until now. We build on a variant of *data provenance analysis* [1, 4] as follows:

- (1) In the query *output*, users identify one or more suspect cells or rows (see Figure 2 where we use the mouse to identify the questionable city Olney).
- (2) Provenance analysis infers those *input* table cells that are material in computing the value Olney (*where provenance*, cells labeled   in Figure 1(a)) as well as all rows that were inspected to decide that Olney is part of the query’s result (*why provenance*, label  ).

recursion depth	② hops AS h city range	① SELECT ... city range	③ r.dist	④ h.range...
0	Alton   0	Alton   0		
...	...	...	...	...
9	Iford   30 Lewes   60 Lewes   60 Magor   40 Magor   50 Neath   30 Neath   40	Lewes   20 Magor   90 Neath   80	110 70 80	130 160 $\zeta$ 160
10	Lewes   20 Lewes   20 Magor   90 Neath   80	Magor   50 Neath   40 Olney   0	70 80 80	120 120 80
11	Magor   50 Olney   0 Neath   40			

Figure 4: Excerpt of observations made by markings ① to ④.

- (3) Remove unlabeled input rows and run the observational debugger on the minimized database instance.

This input minimization will, in general, lead to significantly smaller observations: in the road network scenario, any city or road that does not lie on the path from Alton to Olney will be removed (see Figure 5 which features a mere 12 rows and hides nothing). Most importantly, the reduced input will still trigger the bug and *any* observation made will be relevant in identifying the bug’s cause—in a sense, after minimization the query will focus on producing the buggy output. We claim that this focus is just what is needed to effectively debug data-intensive computations.

**Hand in Hand: Debugging and Provenance Analysis.** The practical relevance of this research hinges on the ability to embrace expressive SQL dialects—featuring language constructs like correlation, grouping, window functions, recursion, as well as built-in and user-defined SQL functions. It is these rich queries that are potential sources of obscure bugs.

Our recent work on the efficient value-less interpretation of programs [4] provides a means to derive *where-* and *why-provenance* for such real-world SQL dialects. We are under-way to connect this analysis with the *Habitat* observational debugger for SQL [2] and are positive to be able to make a significant step towards truly declarative and scalable query debugging.

## 3. REFERENCES

- [1] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4), 2007.
- [2] B. Dietrich and T. Grust. A SQL Debugger Built from Spare Parts—Turning a SQL:1999 Database System into its Own Debugger. In *Proc. ACM SIGMOD*, Melbourne, Australia, 2015.
- [3] T. Grust and J. Rittinger. Observing SQL Queries in their Natural Habitat. *ACM TODS*, 38(1), 2013.
- [4] T. Müller and T. Grust. Provenance for SQL Based on Abstract Interpretation: Value-less, but Worthwhile. In *Proc. VLDB*, Hawaii, USA, 2015.
- [5] E. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, USA, 1983.

recursion depth	② hops AS h city range	① SELECT ... city range	③ r.dist	④ h.range...
0	Alton   0	Alton   0		
1	Alton   0	Brigg   60	40	100
2	Brigg   60	Corby   30	30	60
3	Corby   30	Derby   10	20	30
4	Derby   10	Egton   80	30	$\zeta$ 110
5	Egton   80	Filey   10	70	80
6	Filey   10	Goole   50	60	$\zeta$ 110
7	Goole   50	Hedon   100	50	$\zeta$ 150
8	Hedon   100	Lewes   60	40	100
9	Lewes   60	Neath   80	80	$\zeta$ 160
10	Neath   80	Olney   0	80	80
11	Olney   0			

Figure 5: After input minimization: a full observation display reveals the buggy refueling logic of the query in Figure 3.

# A Way to Automatically Enrich Biomedical Ontologies

Juan Antonio Lossio-Ventura<sup>1</sup>, Clement Jonquet<sup>1</sup>, Mathieu Roche<sup>1,2</sup>, Maguelonne Teisseire<sup>1,3</sup>

<sup>1</sup> LIRMM - University of Montpellier, France

<sup>2</sup> Cirad, TETIS, France

<sup>3</sup> Irstea, TETIS, France

juan.lossio@lirmm.fr, jonquet@lirmm.fr, mathieu.roche@cirad.fr, maguelonne.teisseire@teledetection.fr

## ABSTRACT

Biomedical ontologies play an important role for information extraction in the biomedical domain. We present a workflow for updating automatically biomedical ontologies, composed of four steps. We detail two contributions concerning the concept extraction and semantic linkage of extracted terminology.

## 1. INTRODUCTION

Biomedical big data raises a major issue: the analysis of large volumes of heterogeneous data. Ontologies, i.e. conceptual models of the reality, can play a crucial role in biomedical fields for automating data processing, querying, and integration of heterogeneous data. Few semi-automatic methodologies to build ontologies have been proposed in recent years. Semi-automatic construction/enrichment of ontologies are mostly achieved using natural language processing (NLP) [1] techniques to assess text corpus. However, besides the existence of various English tools, there are considerably fewer ontologies and tools available in French and Spanish. This shortcoming is out of line with the huge amount of biomedical data produced for several languages, especially in the clinical world. This paper proposes a workflow to enrich biomedical ontologies or terminologies from texts, addressing the lexical/syntactic and semantic complexity of this process. The lexical/syntactic complexity involves the extraction of biomedical complex terms from a specialized text corpus. The semantic complexity is related to concept induction and semantic linkage of new terms. Our methodology has been applied for English, French, and Spanish.

## 2. PROPOSED APPROACH

Our approach consists of four steps: (I) Term Extraction, (II) Polysemy Detection, (III) Sense Induction, and (IV) Semantic Linkage. The lexical/complexity complexity is tackled by (I), and the semantic complexity is addressed by (II), (III), and (IV).

**(I) Term Extraction:** We use BIOTEX<sup>1</sup>, our application to extract biomedical terms from documents from text databases (e.g.

<sup>1</sup><http://tubo.lirmm.fr/biotex/>

PubMed). This application implements some measures presented in [4] allowing to extract terms that might be added to a biomedical ontology, we called them “candidate terms”.

**(II) Polysemy Detection:** This step seeks to predict if candidate terms are polysemic. We proposed new features based on statistical measures to characterize our text corpus. They are extracted directly from texts and from a graph itself induced from the text corpus. We used several machine learning algorithms to determine if a term is polysemic or not. Totally, 23 features were proposed, 11 direct and 12 from the induced graph. Their effectiveness showed an F-measure of 98%.

**(III) Term Sense Induction:** The objective of this step, is to induce the multiple or unique sense(s) (concept) of polysemic and not polysemic candidate terms. The senses are extracted according to the context of terms. For this, we execute two tasks. First, (a) *Number of senses prediction:* This task is performed only for the candidate terms predicted as polysemic in the previous step. Then, (b) *Clustering for concept induction:* This task executes a clustering algorithm taking as input the predicted  $k$ , then for each cluster it selects the most important features, which represent the induced concept. Note that  $k = 1$  when the candidate term is not polysemic.

The prediction of the sense number of a term falls directly in clustering-based issues. In clustering tasks, one of the most difficult problems is to determine the number of clusters  $k$ , which is a basic input parameter for most clustering algorithms. In the biomedical domain, according to the statistics on UMLS (see Table 1), polysemic terms trend to be linked to only to 2 and 5 senses (i.e. 2 and 5 clusters). Therefore, as we aim at identifying the possible senses for a new biomedical candidate term, we will limit the number of senses between 2 and 5. Table 1 shows the details of polysemic terms statistics in UMLS and MeSH for English, French, and Spanish. The English version of UMLS contains about 9 919 000 distinct terms of which about 54 257 are polysemic. It means that approximately for 200 biomedical terms there exists just 1 polysemic term.

# of Senses $k$	UMLS			MeSH		
	EN	FR	ES	EN	FR	ES
2	54 257	1 292	10 906	178	11	0
3	7 770	36	414	1	0	0
4	1 842	1	56	0	0	0
5+	1 677	1	18	0	0	0

**Table 1:** Details of Polysemic Terms in UMLS and MeSH.

To evaluate the clustering solutions, there exist two kinds of quality indexes [2]: external and internal. External indexes use pre-labelled data sets with “known” cluster configurations. Internal indexes are used to evaluate the “goodness” of a cluster configuration without any priory knowledge of the clusters, in our case, we propose to focus on internal indexes. We use the following measures: (i) the intra-cluster similarity (*ISIM*), and (ii) the inter-cluster



similarity (*ESIM*), in order to create new indexes. They focus on choosing the minimum or maximum value. That allows to have an idea if the reached clusters are homogeneous. New internal indexes are described in Table 2. **Notation:**  $|S_i|$  is the number of objects assigned to the  $i_{th}$  cluster.

1) <b>Average of ISIM:</b> represented as $a_k$ , is the average of the <i>ISIM</i> value of each cluster of a solution clustering with number of clusters $= k$ . $\max(a_k) = \max\left(\frac{\sum_{i=1}^k ISIM_i}{k}\right)$
2) <b>Average of ESIM:</b> represented as $b_k$ , is the average of the <i>ESIM</i> value of each cluster of a solution clustering with number of clusters $= k$ . $\min(b_k) = \min\left(\frac{\sum_{i=1}^k ESIM_i}{k}\right)$
3) <b>Average of the difference between ISIM and ESIM:</b> represented as $c_k$ , is the average of the difference between <i>ISIM</i> and <i>ESIM</i> multiplied by the number of objects in such cluster $ S_i $ . $\max(c_k) = \max\left(\frac{1}{k} \sum_{i=1}^k  S_i  \times (ISIM_i - ESIM_i)\right)$
4) <b>Division between the ISIM sum and ESIM sum:</b> represented as $e_k$ , is the division between the sum of <i>ISIM</i> multiplied by the number of objects in such cluster $ S_i $ , and the sum of <i>ESIM</i> multiplied by the number of objects in such cluster. $\max(e_k) = \max\left(\frac{\sum_{i=1}^k  S_i  \times ISIM_i}{\sum_{i=1}^k  S_i  \times ESIM_i}\right)$
5) <b>Global objective function divided by the logarithm:</b> represented as $f_k$ , is the division between the value of the average of <i>ISIM</i> and the logarithm of $k$ to base 10. $\max(f_k) = \max\left(\frac{\frac{\sum_{i=1}^k ISIM_i}{k}}{\log_{10}(k)}\right)$

**Table 2:** New Internal Indexes.

For this purpose, we represented our text corpus of two different manners: (i) bag-of-words representation, and (ii) graph representation. We used clustering algorithms and computed the new internal indexes.

(IV) **Semantic Linkage:** This step aims to add a candidate term in an existing biomedical ontology, i.e., how to find the correct position in the ontology. (1) Creation of term co-occurrence graph with terms extracted in (I), selecting only the MeSH neighborhood of a candidate term, then (2) we evaluate the semantic similarity of the candidate term with: (i) its MeSH neighbors, and, (ii) the fathers/sons of those neighbors in MeSH ontology. The semantic linkage is based essentially on a context similarity using the cosine measure between the new biomedical candidate term and those appearing in an ontology. At the end, a list of terms is proposed where the new biomedical candidate term could be positioned.

### 3. DATA AND RESULTS

In this section, we report experiments done to evaluate the performance of our proposal for (ii) prediction of sense number, and (ii) semantic linkage.

(i) **Prediction of Sense Number:** We will describe the text database used and the experiments in the following paragraphs.

*Text corpus:* MSH WSD<sup>2</sup> [3], which is composed of 203 polysemic entities in English, linked to a number of concepts (2,3,4,5). This data set is well-known in Word Sense Disambiguation literature applied to the biomedical domain.

*Results:* We use five well-known clustering algorithms implemented in the CLUTO<sup>3</sup> software, such as: *rb*, *rbr*, *direct*, *agglo*, *graph*. In general, bag-of-words and graph representations obtain similar accuracy values. For these two cases, the maximum value is 93.1% obtained by  $\max(f_k)$  index (See Table 2). Which means that for 100 terms, our approach can determine correctly the number of concepts of 93 terms.

(ii) **Semantic Linkage:** *Text corpus:* We collect 60 MeSH terms that have been added between 2009 and 2015, for instance the term “*corneal injuries*”. Each MeSH term will represent a “*biomedical candidate term*”. Then, we retrieve the context of these terms using PubMed, this context is composed of 333 073 311 tokens. Then, we create a co-occurrence graph per term from the retrieved context.

<sup>2</sup><http://wsd.nlm.nih.gov/>

<sup>3</sup><http://glaros.dtc.umn.edu/gkhome/cluto/cluto/overview>

*Results:* We use cosine similarity between contexts and we propose 10 positions to add candidate terms in the MeSH ontology. For instance, we take the term “*corneal injuries*” added in MeSH between 2009 and 2015. Its synonyms in MeSH are *corneal injury*, *corneal damage*, and *corneal trauma*. Its fathers are *corneal diseases* and *eye injuries*. Then, we apply our methodology to locate “*corneal injuries*” in MeSH. Table 3 shows the first 10 best propositions done by our methodology. From our 10 propositions, 5 are correct, i.e. we found the correct synonyms and fathers of “*corneal injuries*” in MeSH version 2015 (yellow rows).

N°	Where	Cosine	N°	Where	Cosine
1	<i>corneal injury</i>	0.4251	6	<i>eye injuries</i>	0.3681
2	<i>corneal damage</i>	0.4181	7	<i>amniotic membrane</i>	0.3639
3	<i>chemical burns</i>	0.4081	8	<i>re-epithelialization</i>	0.3588
4	<i>corneal diseases</i>	0.3696	9	<i>corneal trauma</i>	0.3582
5	<i>corneal ulcer</i>	0.3689	10	<i>wound</i>	0.3472

**Table 3:** Propositions about where to add the term *corneal injuries*.

Table 4 shows the precision of the number of terms which have at least 1 correct proposition with our methodology for the *Top 1*, *Top 2*, *Top 5* and *Top 10* propositions; taking into account the paradigmatic relations, i.e. synonyms, hyperonyms (fathers), and hyponyms (sons). For instance, the yellow cell shows that there exist at least 1 correct proposition (i.e. existent in MeSH ontology) for the 36 of the 60 terms (i.e. 40%).

<i>Top 1</i>	<i>Top 2</i>	<i>Top 5</i>	<i>Top 10</i>
0.333	0.400	0.500	0.583

**Table 4:** Precision of the number of terms which have at least 1 correct proposition with our methodology.

## 4. CONCLUSIONS AND FUTURE WORK

In this paper, we present an entire workflow to enrich biomedical ontologies. We focus on the last two steps of the global process. We presented new internal indexes to predict the number of clusters (number of senses) for a new biomedical candidate term. They are based on the clustering task by using bag-of-words and graph approaches. Another contribution is to find the right position in an already established ontology for new biomedical terms associated with their senses. We extracted the possible relations for a term. Those were based only on the similarity context, using the cosine measure between contexts.

A perspective of this work is to extract the type of relations. This could be performed with the linguistic patterns (e.g. the verbs used between two terms) and the associated contexts.

## Acknowledgments

This work was supported in part by the French National Research Agency under JCJC program, grant ANR-12-JS02-01001, as well as by University of Montpellier, CNRS, IBC of Montpellier project and the FINCyT program, Peru.

## 5. REFERENCES

- [1] A. Gangemi. A comparison of knowledge extraction tools for the semantic web. In *The Semantic Web: Semantics and Big Data, 10th International Conference, ESWC*, pages 351–366. Springer, 2013.
- [2] A. D. Gordon. Classification, (chapman & hall/crc monographs on statistics & applied probability). 1999.
- [3] A. J. Jimeno-Yepes, B. T. McInnes, and A. R. Aronson. Exploiting mesh indexing in medline to generate a data set for word sense disambiguation. *BMC Bioinf*, 12(1):223, 2011.
- [4] J. A. Lossio-Ventura, C. Jonquet, M. Roche, and M. Teisseire. Biomedical term extraction: overview and a new methodology. *Information Retrieval Journal*, to appear 2016.

# A Distributed Mining Framework for Influence in Evolving Entities

Tian Guo  
EPFL, Switzerland  
tian.guo@epfl.ch

Karl Aberer  
EPFL, Switzerland  
karl.aberer@epfl.ch

## ABSTRACT

Mining dynamic influence in evolving entities, which provides insights into the interaction and causal relations among entities, is an important and fundamental data mining task. Meanwhile, nowadays pervasive sensors in a variety of contexts give rise to the development of many distributed real-time computation systems intended for massive time series streams. In this paper, we focus on mining dynamic influence from time series data generated by entities via such a distributed real-time computation system. The proposed D<sup>2</sup>InfMiner framework encompasses a statistical lead-lag correlation based influence detection module and an on-line model for dynamic influence inference. We implement D<sup>2</sup>InfMiner framework based on Apache Storm.

## Categories and Subject Descriptors

H.3 [Information Systems]: Information storage and retrieval

## Keywords

Time series, Influence mining, Distributed data processing

## 1. INTRODUCTION

For our contemporary interconnected and dynamic changing world, dynamic influence in evolving entities described by time series is fundamental knowledge that helps people understand the behaviours of involved entities. For instance, as massive powerful and various sensors are becoming prevalent in our daily life (e.g., mobile phones, sensor networks, smart meters and etc.), influence among the time series generated by these sensors reflects the real-time status of the carriers and their interactions. Moreover, such quickly and continuously increasing amount of real-time data leads to the development of many distributed real-time computation systems [1], analogous to MapReduce ecosystem designed for large-scale static data.

This paper aims at addressing the problem of mining evolving entity influence based on such a new emerging distributed real-time computation paradigm (DisMineInflu problem). DisMineInflu problem is of great value to various applications such as event/anomaly

detection, trend prediction, casualty analysis and so on. For instance, for data-driven event detection in performance monitoring of data centres, when an event is detected from the performance time series (e.g., network I/O) *w.r.t.* a certain server, using our proposed dynamic influence mining framework, operators can quickly identify from large-scale of servers which one(s) are highly probable to be affected by this event in a certain time and then respond in advance. It is also especially applicable in the financial markets and social data analysis [5–7].

Specifically, our proposed distributed data mining framework should be able to tackle the following challenges. Contrary to static influence mining, since new arriving observations of time series from evolving entities are continuously distributed into different computing nodes of a cluster, mining dynamic influence via the distributed real-time computation system requires a computation and communication efficient solution. Otherwise the system would encounter bottlenecks, which lead the influence detection to lag further and further over time and report stale results [1]. Another challenge lies in modeling the dynamic influence through time series. Since the evolving data could be quite volatile during some periods or events [8], the proposed framework should entail a statistical model responsible for providing stable influence inference. How to efficiently on-line maintain this model to capture the dynamic nature of influence is also non-trivial.

**Contributions:** This paper is the first work that proposes a truly distributed and real-time solution for DisMineInflu problem. The approach in [6] assumes that the underlying influence relationships are static. [5, 7, 8] focus on mining influence in a centralized way and do not consider the data communication overhead in the distributed environment as well as correlations among time series of entities. Overall, this paper makes the following concrete contributions: we formally define the problem of continuously mine dynamic influence from time series yielded by evolving entities based on a distributed real-time computation system (DisMineInflu problem). The D<sup>2</sup>InfMiner framework is proposed to optimize both communication and computation cost as well as providing statistical inference of influence for DisMineInflu problem. We implement D<sup>2</sup>InfMiner framework based on Apache Storm.

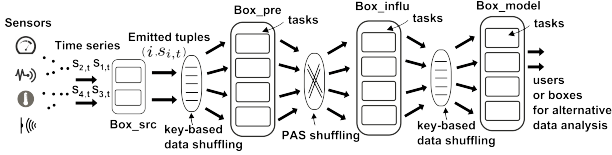
## 2. PROBLEM DEFINITIONS

In this section, we formulate the DisMineInflu problem.

### 2.1 Distributed Real-time Computation Engine

In a typical cluster of a distributed real-time computation engine [1], *Topology* is a job submitted to the cluster, which is a program-described directed acyclic graph (DAG). The vertices are user-defined processing elements denoted by *boxes* and the communication between boxes is dictated by the edges in the topology.

A topology is executed to continuously process tuples. Each box has a user specified number of *tasks* denoted by *parallelism* of the box and such tasks are executed in parallel to process the tuples sent to this box. *Shuffling function* is a function between two boxes, which determines to which task of the connected box a tuple from the preceding box should be sent.



**Figure 1: The topology of dynamic influence mining in a distributed real-time computation system**

We use  $n$  to denote the total number of entities continuously feeding time series streams to the cluster. For an entity  $i$  ( $1 \leq i \leq n$ ), let  $s_i$  denote the sequence of discrete real-valued observations  $s_{i,t}$  ( $t$  represents a time instant) of this entity's attribute. The sliding window of length  $h$  ending at time instant  $t$  of entity  $i$  is denoted by  $s_i^t = (s_{i,t-h+1}, \dots, s_{i,t})$  and  $s_i^t \in \mathbb{R}^h$ .

## 2.2 Problem Statement

In this paper, we utilize statistical lead-lag correlations, namely Pearson correlation and Spearman correlation to measure the influence between time series of entities [4, 8].

**DEFINITION 2.1 (LEAD-LAG CORRELATIONS).** Define a generic correlation function for sliding windows  $s_i^{t_1}$  and  $s_j^{t_2}$  of time series of two entities  $i$  and  $j$  as  $corre(s_i^{t_1}, s_j^{t_2}) = \frac{(s_i^{t_1} - \mu(s_i^{t_1})\mathbb{1}) \cdot (s_j^{t_2} - \mu(s_j^{t_2})\mathbb{1})}{(h-1)\sigma(s_i^{t_1})\sigma(s_j^{t_2})}$

where  $\mathbb{1}$  is all one vector ( $\mathbb{1} \in \mathbb{R}^h$ ),  $\sigma(s_i^{t_1})$  and  $\mu(s_i^{t_1})$  are the sample standard deviation and mean of the elements in  $s_i^{t_1}$ , respectively [4]. Assume  $t_1 < t_2$  and  $corre(s_i^{t_1}, s_j^{t_2})$  measures the correlation between time series  $i$  and  $j$  with lag  $\tau = t_2 - t_1$ .

Pearson correlation coefficient  $\rho_{i,j}^{t_1,\tau}$ , which evaluates the lagged linear relationship, is defined as follows [4]:  $\rho_{i,j}^{t_1,\tau} = corre(s_i^{t_1}, s_j^{t_2})$

Spearman correlation  $\xi_{i,j}^{t_1,\tau}$ , which measures the strength of lagged monotonic relationship is defined as:  $\xi_{i,j}^{t_1,\tau} = corre(r_i^{t_1}, r_j^{t_2})$ , where the entries of  $r_i^{t_1}$  are the ranks of the corresponding entries in original  $s_i^{t_1}$  [4].

For a certain application, users can choose either Pearson or Spearman correlation to measure the influence in evolving entities as defined below:

**DEFINITION 2.2 (CORRELATION BASED INFLUENCE).** Given an application-specific correlation threshold  $\epsilon$ , for the time series of entities  $i$  and  $j$  ( $1 \leq i, j \leq n$ ), entity  $i$  has influence on entity  $j$  at time  $t_1$  with lag  $\tau$  if  $\rho_{i,j}^{t_1,\tau}$  (or  $\xi_{i,j}^{t_1,\tau}$ ) is significantly above  $\epsilon$ .

Intuitively, correlation based influence evaluates to what extent entity  $j$  will have a correlated trend with  $i$  in time  $\ell$  [3, 8]. For simplicity, we call it influence in the rest of paper.

Now the DisMineInflu problem is formulated as:

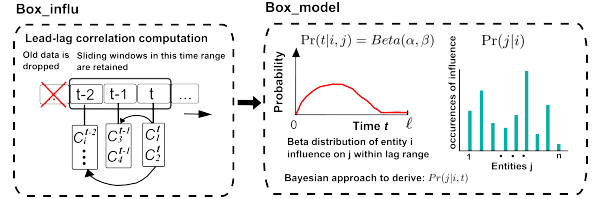
**DEFINITION 2.3 (DISMINEINFLU PROBLEM).** Assume  $n$  time-series streams collected from corresponding  $n$  entities are continuously arriving and distributed to different nodes of a distributed real-time computation system. DisMineInflu problem requires to mine the following information for each entity:

(1) continuously report the entities on which it has influence within a maximum lag  $\ell$ ;

(2) on-line maintain a statistical model over the detected dynamic influence such that the probability that certain entities will be impacted within maximum lag  $\ell$  can be inferred.

The maximum lag  $\ell$  represents the temporal range of users' interest to model the dynamic influence and also indicates how far in advance users want to predict the future based on detected influence. In real applications, due to the dynamic nature of evolving entities and observational errors, the yielded time series often embraces volatile or sudden changed influence [7, 8]. The statistical model built on the real-time detected influence from sub-problem (1) enables to discover significant and stable influence in entities. It can also serve for event and anomaly detection [2], influence prediction and so on [5, 7, 8].

## 3. DISTRIBUTED DYNAMIC INFLUENCE MINER



**Figure 2: Illustration of key components in the topology**

In this section, we briefly describe the proposed D<sup>2</sup>InfMiner framework whose topology is shown in Figure 1. Box\_pre is in charge of maintaining sliding windows and preparing the tuples for PAS-shuffling. Box\_influ collects the data sent by PAS-shuffling and calculates the qualified lead-lag correlations based on hypercube computation pruning. Refer [3] for details of Box\_pre, Box\_influ and PAS-shuffling. Then Box\_model builds a beta-distribution based Bayesian approach to estimate the probability of entity  $i$ 's influence on entity  $j$  at certain time instances. Figure 2 depicts some details of D<sup>2</sup>InfMiner framework.

## 4. ACKNOWLEDGMENTS

This work was supported by Nano-Tera.ch through the OpenSense II project.

## 5. REFERENCES

- [1] Apache Storm. <https://storm.apache.org/>.
- [2] X. C. Chen and et al. Online discovery of group level events in time series. In *SIAM SDM*, 2014.
- [3] T. Guo, J.-P. Calbimonte, H. Zhuang, and K. Aberer. Sigco: Mining significant correlations via a distributed real-time computation engine. In *Big Data (Big Data), 2015 IEEE International Conference on*.
- [4] D. A. Kenny. Correlation and causality.
- [5] C. Liao and et al. Mining influence in evolving entities: A study on stock market. In *Data Science and Advanced Analytics (DSAA), 2014 International Conference on*, pages 244–250. IEEE, 2014.
- [6] X. Shi and et al. Discovering shakers from evolving entities via cascading graph inference. In *Proceedings of the 17th ACM SIGKDD*, pages 1001–1009. ACM, 2011.
- [7] X. Shi, W. Fan, and S. Y. Philip. Dynamic shaker detection from evolving entities. 2011.
- [8] D. Wu, Y. Ke, J. X. Yu, S. Y. Philip, and L. Chen. Leadership discovery when data correlatively evolve. *World Wide Web*, 14(1):1–25, 2011.

# Sweet KIWI: Statistics-Driven OLAP Acceleration using Query Column Sets

Sung-Soo Kim, Taewhi Lee, Moonyoung Chung and Jongho Won  
 Electronics and Telecommunications Research Institute (ETRI)  
 218 Gajeong-ro, Yuseong-gu, Daejeon  
 South Korea  
 {sungsoo, taewhi, mchung, jhwon}@etri.re.kr

## ABSTRACT

KIWI is a SQL-on-Hadoop system enabling batch and interactive analytics for big data. In database systems, materialized views, stored pre-computed results for queries, are one of the most commonly used techniques to improve the query processing speed. However, the key challenge in using materialized views is maintaining their freshness as base data changes. This paper introduces a new approach for accelerating OLAP query processing using query workload statistics and *query column sets* instead of materialized views. We present an architecture of SQL-on-Hadoop system using query column sets of original tables in database. The experimental results demonstrate that our system can provide improved performance by 1.77x on average in terms of TPC-H query processing.

## Keywords

Column Sets; SQL-on-Hadoop; OLAP; Big Data Analytics

## 1. INTRODUCTION

*Data warehouse* (DW) on Hadoop has rapidly gained popularity and is now being used intensively by *business intelligence* (BI) users in enterprises as well as scientific institutions. SQL-on-Hadoop (SoH) is a class of "Big Data" analytics systems that combine established SQL-style querying with Hadoop-based data warehouse [4]. Most of the Online Analytical Processing (OLAP) query workloads in BI applications are long-running batch workloads that are read-mostly and run repeatedly [1]. *Materialized views* are widely used to facilitate fast queries on large datasets. However, one of the most challenging aspects of using materialized views is maintaining their freshness as base data changes.

*Sampling* refers to the commonly used technique of evaluating the queries from a small random sample of the original database [1, 3]. Typical OLAP query processing approaches exploit two sampling methods to construct the samples, such as, *horizontal* sampling (or row sampling) and *vertical* sampling (or column sampling) [2]. Given a table  $T$  with  $r$  rows  $R_1, \dots, R_n$  and  $c$  columns  $C_1, \dots, C_m$ , in horizontal sampling, let  $S_h = \{R_i, R_{i+1}, \dots, R_{i+l}\}$ , where  $i \leq i+l \leq r$ , denote a *row set* that consists of  $l$  rows in  $T$ . In vertical sampling, let  $S_v = \{C_j, C_{j+1}, \dots, C_{j+k}\}$ , where

$j \leq j+k \leq c$ , denote a *column set* that consists of  $k$  columns in  $T$ . A query  $q$  often need to scan fully or partially all data items in a row set or a column set  $S_q$  of  $T$ . If data items in  $S_q$  have been materialized, for  $q$ , need to scan only materialized items instead of full table  $T$ . In case of column sampling, because the number of columns in  $S_q$  is often much smaller than  $c$ , scanning would be done much faster. In our work, we select vertical sampling method to accelerate OLAP query processing on large-scale dataset.

*KIWI*<sup>1</sup> is the proposed SoH system, which runs on hundreds of machines in existing Hadoop cluster. The ultimate goal of our work is to provide a SoH system, which can support interactive analytics as well as deep (batch) analytics. *Sweet KIWI* is a statistics-driven query processing engine in order to support deep analytics at scale.

**Main contributions:** The contributions of our work can be summarized as follows.

- *Dual-Mode Analytics:* The proposed SoH system supports *interactive* analytics as well as MapReduce-based *batch* processing in the unified KIWI architecture.
- *Statistics-Driven OLAP Acceleration:* We introduce a OLAP query acceleration method using query column sets to support deep analytics at scale.

## 2. SYSTEM OVERVIEW

This section focuses on the overall architecture for query processing engine using query column set and describes two main components in the proposed system architecture.

**Query Workload Analyzer:** One common assumption about query workloads is that future queries will be similar to historical queries [2]. This component is responsible for analyzing a set of historical query workloads to classify the frequently used queries in the past. In order to construct the query column sets, we extract the metadata for query column sets over the entire original tables. The set of query column sets are updated both with the arrival of new data, and when the query workloads changes.

**Query Column Sets Constructor:** This block maintains the query column sets as cache tables, and manages the mapping data between the original tables and the cache tables. Query column sets are created, and updated based on statistics collected from the base data and historical queries. When a query arrives at runtime, it is re-written to run against the cache tables instead of the original tables. The KIWI workload manager evaluates the query augmented with cache table selection operations at runtime.

Figure 1 illustrates query processing workflow in our system using the query column sets. In the first step, the query workload

<sup>1</sup>KIWI is the abbreviation for "Key Impact on data Warehouse Infrastructure", which is our project code name.

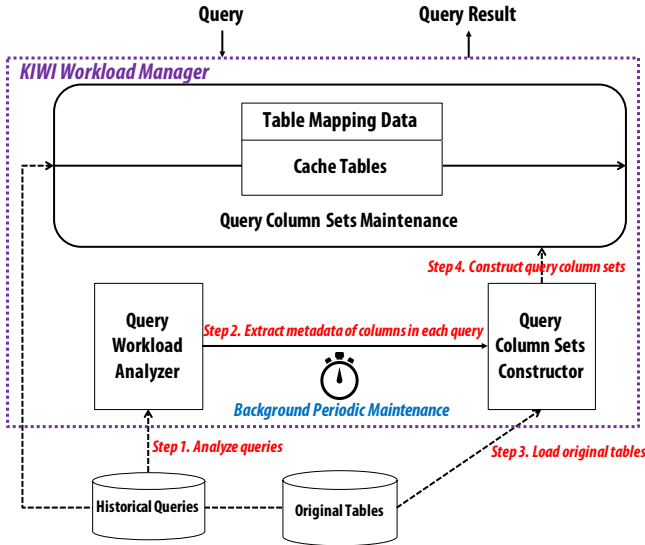


Figure 1: Sweet KIWI Architecture.

analyzer performs historical query analysis. In the second step, it extracts metadata of columns in each query. In the third step, the query column sets (QCS) constructor loads original tables in database to create query column sets. Finally, the QCS constructor inserts the QCS tables and the table mapping data into the database.

## 2.1 Query Column Sets

Let  $\xi(T, S)$  be the memory space needed to store all data items in a column set  $S$  of a table  $T$ . Let  $\varphi$  be the storage system's space limit for materialized column sets. Let  $\omega$  be possible column sets of table  $T$ . The sum of the memory space of possible column sets,  $\sum_{\forall S_i \in \omega} \xi(T, S_i)$  is exponentially large. Let  $Q_p$  is the set of queries issued in the past. Let  $\mathcal{V}(T, S_i)$  be the value obtained for future queries if  $S_i$  is materialized.

**Problem Definition:** Given a table  $T$  and a query  $Q$ , find a collection of optimal column sets,  $S_{opt} = \{S_1, \dots, S_k\}$  consisting of  $k$  column sets, such that  $\sum_{\forall S_i \in \omega} \xi(T, S_i) \leq \omega$  and  $\mathcal{V}_{opt} = \sum_{\forall S_i \in \omega} \mathcal{V}(T, S_i)$  is maximized.

**Algorithm 1** Find optimal column sets ( $S_{opt}$ ).

```

1: procedure FINDOPTIMALCOLUMNSETS( $S_a, Q$ )
2:   List $\langle S \rangle L_s = \text{constructColumnSets}(S_a, Q)$ ;
3:    $L_s.\text{sortByAppearanceFrequency}(\text{DESCENDING})$ ;
4:   for each node  $S_j \in L_s$  do
5:     if  $\sum_{\forall S_i \in S_{opt}} \xi(T, S_i) + \xi(T, S_j) > \omega$  then return
6:     else
7:        $S_{opt}.\text{add}(S_j)$ ;
8:        $S_a.\text{remove}(S_j)$ ;
9:     end if
10:  end for
11: end procedure

```

**Our Approach:** From the set of historical queries  $Q_h$  extracts a set of distinct column sets  $S_a$  that appear in  $Q_h$ .  $\forall S_i \in S_a$ , compute the memory space  $\xi(T, S_i)$ , remove from the column set  $S_a$  if  $\xi(T, S_i) > \omega$ .  $\forall S_i \in S_a$ , compute the appearance frequencies  $f(S_i)$ , remove from the column set  $S_a$  if  $\xi(T, S_i) > \omega$ . Let  $n$  be the number of column sets in  $S_a$ . For an arbitrary column set  $S$ ,

$\xi(T, S)$  can be approximated as:

$$\xi(T, S) = r \times \sum_{i=1}^{|S|} \mathcal{I}(C_i) \quad (1)$$

where  $r$  denotes the number of rows in  $T$ ,  $|S|$  denotes the number of columns in  $S$  and  $\mathcal{I}(C_i)$  denotes the average size of a data item in  $C_i$  (e.g., if data type of  $C_i$  is double, then  $\mathcal{I}(C_i)$  is 8 bytes). **Algorithm 1** shows how optimal column sets,  $S_{opt}$ , can be evaluated progressively for a given query  $Q$ . The time complexity of this algorithm is  $O(n \log n)$ , where  $n$  is the size of the database.

## 3. EXPERIMENTAL RESULTS

To evaluate the performance of the KIWI for analytic workloads, we loaded the industry-standard TPC-H data set at scale factor 10 on a node. The server has dual 2.66GHz Intel Xeon CPUs with 128GB RAM, and runs Mac OS X. We compared the wall time of each TPC-H query between original DB and QCS DB.

Table 1: Runtime for TPC-H queries (unit: seconds)

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Original DB	137	18.6	130	181	28	174	91	176
QCS DB	68.8	11.6	67.9	123	17	84.4	62	87.9

The experimental results demonstrate that our system can provide improved performance in terms of query processing speed on TPC-H 10GB dataset. There were performance improvements of 1.77x on average compared to the original DB as shown in Table 1.

## 4. CONCLUSION

We present a statistics-driven OLAP acceleration in SQL-on-Hadoop system architecture for data-intensive applications. Our main contribution in this work has been to propose a new unified approach for supporting *dual-mode* (interactive and deep) analytics at scale. Our work concludes with the following take-away messages: (1) It is beneficial to have an *unified* query processing engine in the KIWI SQL-on-Hadoop system, (2) *Sweet KIWI* is a general purpose system that constructs the query column sets of historical queries for deep analytics, and (3) the vertical sampling method using *query column sets* is intuitive to use.

**Acknowledgments.** This work was supported by ETRI R&D program ("Development of Big Data Platform for Dual Mode Batch-Query Analytics, 16ZS1400") funded by the government of South Korea.

## 5. REFERENCES

- [1] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when You're Wrong: Building Fast and Reliable Approximate Query Processing Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 481–492. ACM, 2014.
- [2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42. ACM, 2013.
- [3] S. Chaudhuri, G. Das, and V. Narasayya. Optimized Stratified Sampling for Approximate Query Processing. *ACM Trans. Database Syst.*, 32(2), June 2007.
- [4] A. Floratou, U. F. Minhas, and F. Özcan. SQL-on-Hadoop: Full Circle Back to Shared-nothing Database Architectures. *Proc. VLDB Endow.*, 7(12):1295–1306, Aug. 2014.

# On-Line Mobility Pattern Discovering using Trajectory Data

Ticiana Coelho da Silva  
Federal University of Ceará,  
Brazil  
ticianalc@ufc.br

Karine Zeitouni  
Université de  
Versailles-St-Quentin, France  
karine.zeitouni@uvsq.fr

José A. F. de Macêdo  
Federal University of Ceará,  
Brazil  
jose.macedo@lia.ufc.br

Marco A. Casanova  
Department of Informatics -  
PUC-Rio, Brazil  
casanova@inf.puc-rio.br

## ABSTRACT

Mobile location tracking becomes ubiquitous in many applications, which raises great interests in trajectory data analysis and mining. Most existing work tackled the problem of offline trajectory pattern mining. Dynamic discovery and updates of patterns in trajectory data streams in (quasi) real time is a more complex task. In this paper, we propose an incremental algorithm to solve this problem, while maintaining the evolution of the patterns as well as the membership of the moving objects to their patterns.

## 1. INTRODUCTION

The huge volume of collected trajectories opens new opportunities for discovering the hidden patterns about mobility behaviors. These patterns may apply to characterize individual mobility as well as groups sharing similar trajectories for a certain time period. Usually, this analysis is done off-line, i.e., by applying data analysis and mining techniques on the previously collected data [1]. This allows characterizing the past movements of the objects but not the current mobility patterns. Nowadays, many services exist that involve moving objects (e.g., persons, vehicles, animals) to report their trajectory continuously (e.g., every second or every minute). Analyzing these data in real time may bring a real added-value in the comprehension of the city dynamics, and the detection of regularities as well as anomaly, which is essential for decision making. Among these patterns, we consider in this paper the trajectory group constitution and evolution, based on sub-trajectory cluster analysis. Such discovery may help the search for effective re-engineering of traffic, or dynamically detecting events or incidents, e.g., at a city level.

One important property of tracking application is the incremental nature of the data. The data will grow to reach a huge size as time goes. Finding patterns in these data in (quasi) real time is challenging. Since all the tracked moving

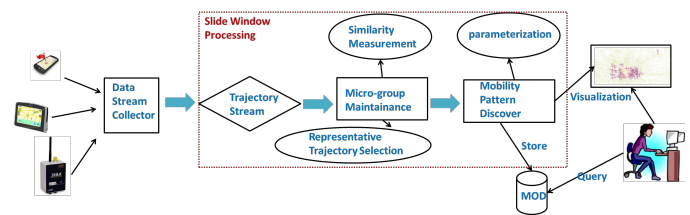


Figure 1: Steps of our proposed framework

objects change their positions over time, movement patterns also evolve in time. Furthermore, new moving objects may start sending their positions while others may stop. There exist approaches for online clustering of moving objects position, but they are restricted to instantaneous positions. Subsequently, they fail to capture the displacement behavior along time. By continuously tracking moving objects sub-trajectories at each time window, rather than just the last position, it becomes possible to gain insight on the current behavior, and potentially detect suspicious behaviors in real time. Although analyzing historical (sub)trajectory data is the majority of mobility patterns approaches today (including trajectory clustering, flocks, convoys, swarms, gathering [1]), no solution exist for clustering and maintaining clusters of sub-trajectories in real time. That is why we believe that our study is relevant.

In this paper, we address the problem of online discovery of mobility patterns and their evolution by tracking the sub-trajectories of moving objects at each time window. To solve this problem, we propose a framework discussed on the next section. Since all the objects change their sub-trajectories data from time to time, new moving objects appear as well as others disappear from the system during a time window. We define a new structure, called micro-group, to incrementally maintain the relationship among moving objects.

## 2. FRAMEWORK

Our framework (see Figure 1) follows these main steps: (i) collect trajectory data stream at each time window, (ii) apply the similarity measure, (iii) maintain the micro-group(s), and (iv) discover the mobility patterns.

A trajectory is a sequence of the locations of a moving object at each time-stamp and is denoted by  $TR_j = p_1 p_2 \dots p_r \dots p_{len_j}$ . Here,  $p_k$  ( $1 \leq k \leq len_j$ ) is a point  $(x_k, y_k, t_k)$  in a three dimensional space, where  $(x_k, y_k)$  indicates the

location of the object at time  $t_k$ . The length  $len_j$  of a trajectory can be different from those of other trajectories.

**First step.** Consider  $i = [t, t + \delta t]$  be the time window observed for the set of moving objects sub-trajectory  $I_i$ . Let  $I_i = \{(o_1, ST_{1,i}), (o_2, ST_{2,i}), \dots, (o_n, ST_{n,i})\}$ , where  $ST_{j,i}$  is the sub-trajectory of the moving object  $o_j$  on  $I_i$ . Therefore,  $I_i$  is the stream at the time window  $i$ .

**Second step.** To measure the similarity between two moving objects sub-trajectories  $ST_{k,i}, ST_{j,i}$  at the time window  $i$ , we implement the synchronous Euclidean distance, which accounts for time, space, and direction. However, our framework is suitable to any distance function for trajectories. Through this paper,  $distance(ST_{k,i}, ST_{j,i})$  denotes the distance function between sub-trajectories.

From a group of moving object sub-trajectories  $S_i = \{(o_1, ST_{1,i}), (o_2, ST_{2,i}), \dots, (o_m, ST_{m,i})\}$  at the time window  $i$ , we define the representative trajectory as a pair composed by one moving object and its sub-trajectory which is similar to the major behavior of  $S_i$ . To choose  $(o_j, ST_{j,i})$  as the representative, our approach checks the number of moving objects that have their sub-trajectory similar to  $ST_{j,i}$  and also uses a Gaussian kernel function (our voting function) to estimate the representativeness of  $ST_{j,i}$ . This voting function is also used on [2], and it has been widely used in a variety of applications of pattern recognition.

**Definition 2.1.** For a set of moving object sub-trajectory  $S_i = \{(o_1, ST_{1,i}), (o_2, ST_{2,i}), \dots, (o_m, ST_{m,i})\}$  at the time window  $i$ ,  $\rho$  be a representativeness threshold,  $\epsilon$  be a given distance threshold and  $\tau$  be a size/density minimum threshold,  $(o_j, ST_{j,i})$  is a **representative trajectory** of  $S_i$  if and only if:

1.  $\forall (o_k, ST_{k,i}) \in S_i$ ,  

$$\text{voting}(ST_{j,i}, ST_{k,i}) = e^{-\frac{\text{distance}^2(ST_{j,i}, ST_{k,i})}{2\sigma^2}} > \rho$$
2.  $N_\epsilon(o_j) = \{(o_k, ST_{k,i}) \in S_i | \text{distance}(ST_{j,i}, ST_{k,i}) \leq \epsilon\}$ , then  $|N_\epsilon(o_j)| \geq \tau$

The parameter  $\sigma$  shows how fast the function "voting" decreases with the distance. The intuition behind the relationship of "distance" and "voting" function is: If "distance" is close to zero, the "voting" is close to its maximum value. This means that if  $ST_{j,i}$  is very close (in time, space and direction, for example) to  $ST_{k,i}$ , then  $(o_j, ST_{j,i})$  is a candidate to be the representative. Otherwise, if the distance is high, the "voting" function is close to its minimum value, meaning that  $ST_{j,i}$  is very far away from  $ST_{k,i}$ , so  $ST_{k,i}$  is ill-represented by  $ST_{j,i}$ .

We define a new structure called micro-group, based on the concept of representative trajectory.

**Definition 2.2.** For a set of moving object sub-trajectory  $S_i = \{(o_1, ST_{1,i}), (o_2, ST_{2,i}), \dots, (o_m, ST_{m,i})\}$  at the time window  $i$ , let  $O_i$  be the set of moving objects in  $S_i$ ,  $\epsilon$  be a distance threshold,  $\tau$  be a size/density minimum threshold,  $\rho$  be a representativeness threshold. A **micro-group**  $g$  is defined as a set of objects satisfying:

1.  $g \subseteq O_i$
2.  $\exists o_j \in g$ , such that  $R_g^{traj} = (o_j, ST_{j,i})$  is a representative trajectory of  $g$  w.r.t.  $\epsilon, \tau$  and  $\rho$ .

The  $\rho$  value can be chosen according to the maximum allowed distance between the representative sub-trajectory

and the farthest member of a micro-group. The  $\epsilon$  and  $\tau$  thresholds captures the density around the representative trajectory, similarly to DBSCAN.

**Third step.** We propose an algorithm to incrementally maintain each micro-group (since all the moving objects update their sub-trajectories, some moving object may leave or join a micro-group) and to capture its evolution patterns. At the initialization phase (i.e., a cold-start of the clustering) the representative trajectories are randomly chosen among the core objects (as defined in DBSCAN). Micro-groups are first derived as the objects that vote for these representatives. The most important contribution is the maintenance phase. The intuition behind is: (i) to check for each micro-group whether the representative is still valid in the next time window (it survives), otherwise, the micro-group disappears or splits, (ii) to track moving object sub-trajectories that are likely to join the micro-group (e.g., outliers, new objects, and other objects migrating from another micro-group), (iii) for the remaining objects, a similar process to the initialization allows creating new micro-groups.

When a micro-group  $g_i$  survives, the algorithm checks for each moving object  $o_k \in g_i$  if it is still well represented by  $(o_j, ST_{j,i})$ . If it is not,  $o_k$  is deleted from  $g_i$  and either it migrates to another micro-group, or it becomes an outlier, or it forms a new micro-group with other outliers. A micro-group  $g_i$  splits or disappears when it changes its representative trajectory. If  $g_i$  splits, new micro-groups have to be computed using the  $g_i$  data. However, if  $g_i$  is not dense enough to generate micro-group(s), its moving objects become outliers and  $g_i$  disappears.

**Fourth step.** We use the maintained micro-groups to discover mobility patterns, by capturing the evolution of micro-groups over time. Since each micro-group is density based, it is suitable to find sub-trajectory density based clustering (for example, merging micro-groups results in density based sub-trajectory clusters). Furthermore, this paves the way for online discovery of more complex patterns, such as flocks, convoys, leadership. Indeed, flocks could be derived from micro-groups by a light post-processing since it is a subset of the later. The convoys are also similar to the density based clusters generated by our algorithm. The representative is a close notion to leadership. Hence, this information could enrich a Moving Object Database, allowing new query and visualization types.

### 3. CONCLUSION

In this paper, we have present a framework to track and discovery mobility patterns in moving objects trajectory data streams. We also proposed an incremental algorithm to maintain the patterns evolution from time to time. It is noteworthy that we evaluated our approach on real data sets, which shows its effectiveness and its efficiency.

### 4. REFERENCES

- [1] Yu Zheng. Trajectory data mining: an overview. *ACM Transactions on Intelligent Systems and Technology*, page 29, 2015.
- [2] Costas Panagiotakis, Nikos Pelekis, Ioannis Kopanakis, Emmanuel Ramasso, and Yannis Theodoridis. Segmentation and sampling of moving object trajectories based on representativeness. *KDE, IEEE Transactions on*, pages 1328–1343, 2012.

# Summarizing Linked Data RDF Graphs Using Approximate Graph Pattern Mining

Mussab Zneika  
ETIS Lab,  
ENSEA /University of  
Cergy-Pontoise /CNRS,  
Cergy, France  
mussab.zneika@ensea.fr

Claudio Lucchese  
ISTI-CNR,  
Pisa, Italy  
claudio.lucchese@cnr.it

Dan Vodislav  
ETIS Lab,  
ENSEA /University of  
Cergy-Pontoise /CNRS,  
Cergy, France  
Dan.Vodislav@u-  
cergy.fr

Dimitris Kotzinos  
ETIS Lab,  
ENSEA /University of  
Cergy-Pontoise /CNRS,  
Cergy, France  
Dimitrios.Kotzinos@u-  
cergy.fr

## ABSTRACT

The Linked Open Data (LOD) cloud brings together information described in RDF and stored on the web in (possibly distributed) RDF Knowledge Bases (KBs). The data in these KBs are not necessarily described by a known schema and many times it is extremely time consuming to query all the interlinked KBs in order to acquire the necessary information. To tackle this problem, we propose a method of summarizing large RDF KBs using approximate RDF graph patterns and calculating the number of instances covered by each pattern. Then we transform the patterns to an RDF schema that describes the contents of the KB. Thus we can then query the RDF graph summary to identify whether the necessary information is present and if so its size, before deciding to include it in a federated query result.

## Keywords

Linked Open Data; RDF Summarization; Query Processing

## 1. INTRODUCTION

The amount of RDF (Resource Description Framework, [www.w3.org/RDF/](http://www.w3.org/RDF/)) data available on the semantic web is increasing fast both in size and complexity, e.g. more than 1000 datasets are now published as part of the Linked Open Data (LOD) cloud, which contains more than 62 billion RDF triples, forming big and complex RDF data graphs. It is also well established that the size and the complexity of the RDF data graph have a direct impact on the evaluation of the RDF queries we express against these data graphs. Especially on the LOD cloud, we observe that a query against a big and complex RDF Knowledge Base (KB) might retrieve no results at the end because either (a) the association between the different RDF KBs is weak (is based only on a few associative links) or (b) there is an association at the schema level that has never been instantiated at the actual data level. Thus we can conclude that having information

on the content of a KB and statistical information on the number of instances described under various concepts will allow us to decide on whether or not to post a query based on the availability of the necessary information.

By creating summaries of the RDF KBs, we allow the user or the system to decide whether or not to post a query, since (s)he knows whether information is present or not. This would provide significant cost savings in processing time since we will substitute queries on complex RDF KBs with queries first on the summaries (on much simpler structures with no instances) and then with queries only towards the KBs that we know will produce significant results. We need to compute the summaries only once and update them only after significant changes to the KB. Given the (linked) nature of LOD this will speed up the processing of queries in both centralized and distributed settings.

Moreover, many RDF KBs are suffering from a total or partial absence of schema information. By applying RDF summarization techniques, we can extract, at least, a subset of the schema information and thus facilitate the query building for the end users with the additional benefit of categorizing the contents of the KB based on the summary. We can envision similar benefits when KBs are using mixed vocabularies to describe their content. In all these cases we can use the RDF summary to concisely describe the data in the RDF KB. Thus in this work we study the problem of LOD/RDF graph summarization that is: given an input RDF graph, find the summary graph which reduces its size, while preserving the original inherent structure and correctly categorizing the instances included in the KB.

Two main categories of graph summarization have been proposed to date: (1) aggregation and grouping approaches [3, 5], which are based on grouping the nodes of input RDF graph  $G$  into clusters/groups based on the similarity of attributes' values and neighborhood relationships associated with nodes of  $G$  and (2) structural extraction approaches [1, 2] which are based on extracting some kind of schema where the summary graph is obtained based on an equivalence relation on the RDF data graph  $G$ , where a node represents an equivalence class on nodes of  $G$ . To the best of our knowledge, few of these approaches are concentrating on RDF KBs and only one of them [1] is capable of producing a RDF schema as result, which would allow the use of RDF tools (e.g. SPARQL) to query the summary. Our approach provides comparable or better results in most cases.

In summary, our solution is responding to all the require-



ments by extracting the best approximate RDF graph patterns, construct a summary RDF schema out of them and thus concisely describe the RDF input data. We offer the following features: (1) The summary is a RDF graph itself, which allows us to post simplified queries towards the summarizations using the same techniques (e.g. SPARQL), (2) statistical information (number of class and property instances per pattern) is included in our summary graph, which allows us to estimate a query’s expected results’ size, (3) the summary is much smaller than the original RDF graph, contains all the important concepts and their relationships based on the number of instances, (4) schema independence: it summarizes an RDF graph regardless of having or not schema and RDFS triples and (5) heterogeneity independence: it summarizes an RDF graph regardless if it is hetero- or homo-geneous.

## 2. RDF-GRAPH PATTERNS COMPUTATION

We present in this section our approach of RDF graph summarization, based on mining a set of approximate graph patterns (an error-tolerant pattern mining technique). It aims at discovering the smallest set of  $k$  patterns that best describe the input dataset, where the quality of the description is measured by an information theoretic cost function. We use a modified version of the PaNda+ algorithm [4], which uses a greedy strategy to identify the  $k$  patterns that best optimize the given cost function. Even if we do not fix the input parameter  $k$ , PaNda+ can stop producing further patterns when the cost of a new pattern is more than the corresponding noise reduction. Our approach works in three independent steps that are described below and in Figure 1.

**Binary Matrix Mapper:** We transform the RDF graph into a binary matrix  $D$ , where the rows represent the subjects and the columns represent the predicates. We preserve the semantics of the information by capturing distinct types (if present), all attributes and properties and also reverse properties (so as to capture both subject and object of a property). We extend the RDF URI information by adding labels that represent the different predicates carrying this information into the patterns. *No* schema information is required for the algorithm to work adequately well.

$$D[i; j] = \begin{cases} 1, & \text{the } i\text{-th URI has } j\text{-typeof or is } j\text{-property's} \\ & \text{domain/range or is } j\text{-attribute's domain} \\ 0, & \text{otherwise} \end{cases}$$

**Graph Pattern Identification:** The binary matrix created in step 1 is used in a calibrated version of the PaNda+ [4] algorithm, which allows us to experiment with different cost functions while retrieving the best approximate RDF graph patterns. Each extracted pattern identifies a set of subjects (rows) all having approximately the same properties (cols). The patterns are extracted so as to minimize errors and to maximize the coverage (i.e. provide a richer description) of the input data. A pattern thus encompasses a set of concepts (type, property, attribute) of the RDF dataset, holding at the same time information about the number of instances that support this set of concepts.

**Constructing the RDF summary graph:** We have implemented a process, which reconstructs the summary as a valid RDF graph using the extracted patterns. For each pattern in step 2 we generate a node labeled by a URI (minted from a hash function) and we add an attribute with the

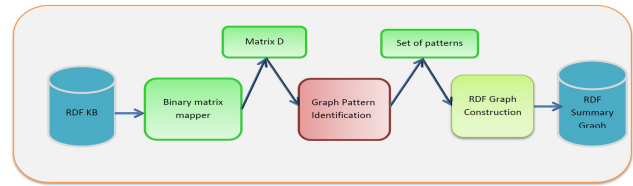


Figure 1: Our RDF Summarization Approach.

*bc:extent* label representing the number of instances for this pattern. Then we use the labels generated at step 1 to understand the type of predicate involved and to generate the proper links. The process exploits information already embedded in the binary matrix and tries to construct a valid RDF schema to represent the KB.

## 3. PRELIMINARY RESULTS

We evaluated so far our approach on variations (e.g. with or without any schema information) over two datasets. An artificial one, which consists of 2000 triples, classified under 8 classes and 9 properties. And a real one, called *Jamendo*, which consists of 11 classes and 25 properties and 1.05 M triples. In both cases 89% of the classes and properties is correctly identified (are the same with the original schema of the dataset even if the schema is not used as a part of the calculation) and the corresponding instances are correctly classified. We produce a summary which is still valid RDF/S and thus can be queried by the same tools.

## 4. CONCLUSIONS AND FUTURE WORK

In this work we apply an approximate graph pattern mining algorithm in order to extract a summary of an RDF KB. The summary is not necessarily the complete schema of the KB but it always remains a valid RDF/S graph. We plan to test our approach on more complex and bigger datasets (billion of triples); the results so far are promising. We plan to integrate additional RDF knowledge into the algorithm and allow for personalized summaries of RDF KBs.

## 5. REFERENCES

- [1] S. Campinas, T. E. Perry, D. Ceccarelli, R. Delbru, and G. Tummarello. Introducing rdf graph summary with application to assisted sparql formulation. In *Database and Expert Systems Applications (DEXA), 2012 23rd International Workshop on*, pages 261–266. IEEE, 2012.
- [2] S. Khatchadourian and M. Consens. Explod: Summary-based exploration of interlinking and rdf usage in the linked open data cloud. *The Semantic Web: Research and Applications*, pages 272–287, 2010.
- [3] A. Louati, M.-A. Aaufaure, Y. Lechevallier, and F. Chatenay-Malabry. Graph aggregation: Application to social networks. In *HSDA*, pages 157–177, 2011.
- [4] C. Lucchese, S. Orlando, and R. Perego. A unifying framework for mining approximate top-binary patterns. *Knowledge and Data Engineering, IEEE Transactions on*, 26(12):2900–2913, 2014.
- [5] N. Zhang, Y. Tian, and J. M. Patel. Discovery-driven graph summarization. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 880–891. IEEE, 2010.

# Understanding Customer Attrition at an Individual Level: a New Model in Grocery Retail Context

Clément Gautrais  
University of Rennes 1  
clement.gautrais@irisa.fr

Peggy Cellier  
INSA Rennes  
peggy.cellier@irisa.fr

Thomas Guyet  
Agrocampus Ouest  
thomas.guyet@irisa.fr

René Quiniou  
Inria Rennes  
rene.quiniou@inria.fr

Alexandre Termier  
University of Rennes 1  
alexandre.termier@irisa.fr

## ABSTRACT

This paper presents a new model to detect and explain customer defection in a grocery retail context. This new model analyzes the evolution of each customer basket content. It therefore provides actionable knowledge for the retailer at an individual scale. In addition, this model is able to identify customers that are likely to defect in the future months.

## CCS Concepts

•Information systems → Data mining;

## Keywords

Data mining; Attrition modeling; Grocery retail

## 1. INTRODUCTION

In grocery retail context, customer defection is partial [1], in the sense that a customer will usually lower his purchases, instead of totally leaving the store. Moreover, no contract binds the customer to the retailer, so customer defection is not clearly signaled through contract ending, like it is in other businesses such as banks or phone operators. Customer defection (also called attrition) is therefore difficult to detect because there is no clear definition of when a customer is defecting.

Attrition in grocery retail environments has mainly been studied through the RFM model [3], based on behavioral variables (recency, frequency and monetary value). RFM demonstrated good performances in partial attrition detection [1] but is limited to building groups of customers which provide few explanations of attrition causes.

Building comprehensive attrition models is of interest because retailers want to lower their retention marketing expenses, by deploying accurate targeted marketing. Models using first and last sequences of purchased products have been proposed [2] and improved attrition detection, while

providing more information about attrition causes. Nevertheless, these models do not explain attrition at an individual level.

Understanding attrition at the customer level is necessary to do efficient targeted marketing. This work presents a model of customer stability that allows for analyzing the evolution of individual customer's purchases to understand attrition causes, at an individual level.

## 2. ATTRITION MODEL

We want to characterize important items, that are bought by a given customer during successive periods. Moreover, we want to detect an evolution in this item set to model and understand customer stability.

Let  $\mathcal{I} = \{i_1 \dots i_n\}$  be the set of all items. The purchases of customer  $i$  can be represented by a chronologically ordered list  $\mathcal{D}_i = \langle (b_1, t_1), \dots, (b_N, t_N) \rangle$ , with  $b_j \subset \mathcal{I}$  being the content of basket  $j$  and  $t_j$  its timestamp.

Let  $w$  be a window. We divide  $\mathcal{D}_i$  in consecutive non overlapping windows of time span  $w$  to define the windowed database of customer  $i$ , denoted  $\mathcal{D}_i^w$ , as an ordered list of tuples  $(t_k^B, t_k^E, u_k)$ .  $k$  is the window number and  $\mathcal{D}_i^w$  is ordered in chronological order on  $t_k^B$ .  $u_k$  is the set of all products bought during window  $k$ , delimited by  $t_k^B$  and  $t_k^E$ .

Let  $p \in \mathcal{I}$  be an item,  $c(k)$  be the number of windows prior to window  $k$  that contain  $p$ ,  $c(k) = |\{u_v | v < k, p \in u_v\}|$  and  $l(k)$  be the number of windows prior to window  $k$  that do not contain  $p$ ,  $l(k) = |\{u_v | v < k, p \notin u_v\}|$ .

We define the **significance** of  $p$  in window  $k$  as  $S(p, k) = \alpha^{c(k)-l(k)}$  if  $c(k) > 0$  and  $S(p, k) = 0$  otherwise.  $\alpha$  is a parameter of the method. The usual expected behavior is to increase the item significance when incrementing  $c(k)$ . Therefore, we generally fix  $\alpha > 1$ .

We define the **stability of customer**  $i$  in window  $k$  as  $Stability_i^k = \sum_{p \in u_k} S(p, k) / \sum_{p \in \mathcal{I}} S(p, k)$ .

If all products are contained in window  $k$ , the stability of the customer is equal to 1. This stability decreases when products are not contained in window  $k$ . This decrease is proportional to the significance of missing products. The more significant a product is, the more the stability will decrease if this product is not present in window  $k$ .

When the stability of some customer decreases, we can identify which product mainly caused this decrease. This product is defined as  $\arg \max_{p \notin u_k} S(p, k)$ , which is the most significant product that was not bought in window  $k$ . This

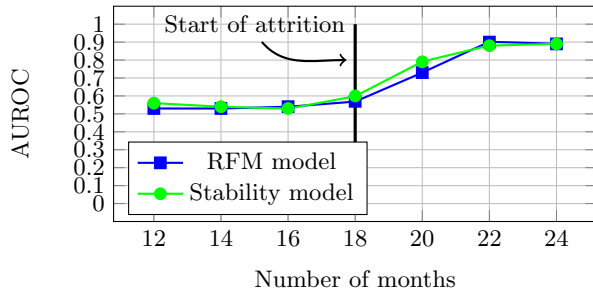


Figure 1: Performance of the attrition detection.

attrition explanation can be easily extended to a set of products.

### 3. EXPERIMENTS

The dataset provided by a major French retailer contains anonymized receipts of 6 millions customers, from May 2012 to August 2014. Each timestamped customer receipt describes a related basket content. A taxonomy is also provided that enables abstracting products in segments. The dataset contains 4 millions products, that are grouped into 3 388 segments.

**Customer selection:** Our main goal is to explain attrition. It is especially important for defecting loyal customers. The retailer provided us with the IDs of loyal customers, and of loyal customers that defected in the last 6 months. The beginning of the defection, given by the retailer, is indicated in Figure 1 by the vertical line at month 18.

#### 3.1 Attrition prediction

The first experiment attempts to validate our model for separating loyal customers from the ones that defected during the last 6 months. To assess the relevance of our model, we use the area under the ROC curve for different window indices. We chose the AUROC because it evaluates the discrimination ability of our model. The points on these curves are obtained using different thresholds  $\beta$  for the customer stability. If  $Stability_i^k > \beta$  the customer is considered loyal. Otherwise, the customer is considered as defecting on window  $k$ . The window length for this experiment is set to two months and the  $\alpha$  parameter is set to 2. These values were chosen after performing a 5-fold cross-validation search. We compare our model to the standard RFM model, that uses recency, frequency and monetary variables to identify defecting customers. This RFM model is built using a logistic regression on these three types of variables. The methodology we used to compute the RFM model is similar to the one presented in [1], but we only used predictors associated to the recency, frequency and monetary variables. For each window, we compute the AUROC of our model and of the RFM model. Their evolution is plotted in Figure 1.

Figure 1 shows that our model accurately identifies customers that are in attrition in the last 6 months. This identification takes place in the first months of the customer defection. Two months after the start of attrition, our model scores an AUROC of 0.79, indicating a rather accurate detection of defecting customers. Our model and the RFM model have similar performances. This shows that our model is not only able to provide information about attrition, but is also able to detect customer attrition, with performances similar

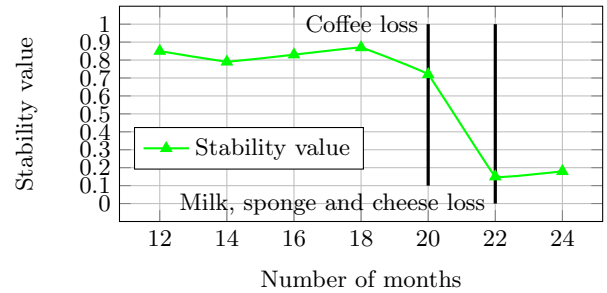


Figure 2: Defecting customer stability value example.

to standard attrition models such as RFM.

#### 3.2 Attrition explanation

The second experiment aims to show the value of our model to explain attrition at an individual level, by illustrating that it provides actionable knowledge about the products responsible for individual customer attrition. We illustrate this on a use-case study of a defecting customer.

In Figure 2 the stability value indicates that the customer is loyal in the first months, and defecting starting from month 20. The upside of the stability value is that it can link each decrease to a loss of significant products.

For example, we can link the decrease in month 20 to the fact that the customer stopped buying coffee during this window. In month 22, the decrease is sharper because the customer lost several significant products: milk, sponge and cheese. We can perform this precise analysis for each window where our model suggests that the customer is not as loyal as he was before. This information is of interest for the retailer because he can then target his marketing on significant products that this customer is not buying anymore.

### 4. CONCLUSION

In this paper, we presented a new model to analyze customer attrition in a grocery retail context. This model is based on the customer basket content evolution and provides precise information about individual defecting customer. It can also reliably detect customer defection.

In the future, we plan to deepen the study of the characterization of significant products that can explain customer defection.

### 5. ACKNOWLEDGMENTS

We would like to thank the french retail company that provided the data and funded this work.

### 6. REFERENCES

- [1] W. Buckinx and D. V. den Poel. Customer base analysis: partial defection of behaviourally loyal clients in a non-contractual FMCG retail setting. *EJOR*, 164(1):252 – 268, 2005.
- [2] V. L. Miguéis, D. Van den Poel, A. S. Camanho, and J. Falcão e Cunha. Modeling partial customer churn: On the value of first product-category purchase sequences. *ESWA*, 39(12):11250–11256, 2012.
- [3] D. Shepard. The new direct marketing. *Business One Irwin, Homewood, IL*, 1990.

# Towards an Efficient Ranking of Interval-Based Patterns

Marwan Hassani      Yifeng Lu      Thomas Seidl

Data Management and Data Exploration Group  
 RWTH Aachen University, Germany  
 {hassani, yifeng.lu, seidl}@cs.rwth-aachen.de

## ABSTRACT

Almost all activities observed in nowadays applications are correlated with a timing sequence. Users are mainly looking for interesting sequences out of such data. Sequential pattern mining algorithms aim at finding frequent sequences. Usually, the mined activities have timing durations that represent time intervals between their starting and ending points. Most sequential pattern mining approaches dealt with such activities as a single point event and thus lost many valuable information in the collected patterns. We present the PIVOTMiner, an efficient interval-based sequential pattern mining algorithm using a geometric representation of intervals. The interestingness level is not necessarily positively correlated with the frequency of the patterns. In many applications, users are seeking for rare patterns that considerably deviate from the majority. Simply delivering the bottom- $k$  patterns does not guarantee their high outlierlieness (or deviation) from the frequent ones. We propose additionally the PIVOTRanker, the first scalable algorithm for ranking rare interval-based sequential patterns based on their outlierlieness. Our experimental results on both synthetic and real-world datasets show that PIVOTMiner spends considerably less time than two state-of-the-art competitors, and that PIVOTRanker delivers a meaningful and useful ranking of rare patterns.

## 1. INTRODUCTION AND PIVOTMINER

Available interval-based approaches (e.g. [2]) employ the Allen's relationship [1] to model the event patterns. They lose the quantitative information and thus the outlierlieness ranking can not be applied. [6] represents interval events as parallel aligned sequences. [4] introduces end point sequence with additional quantitative information. However, these algorithms did not focus on outlier ranking.

We present our PIVOTMiner: **Paradigm of Intervals as Vectors and Origin Transformation**. It is an interval-based frequent pattern mining approach. This approach is introduced here to model the interval-based event pattern for

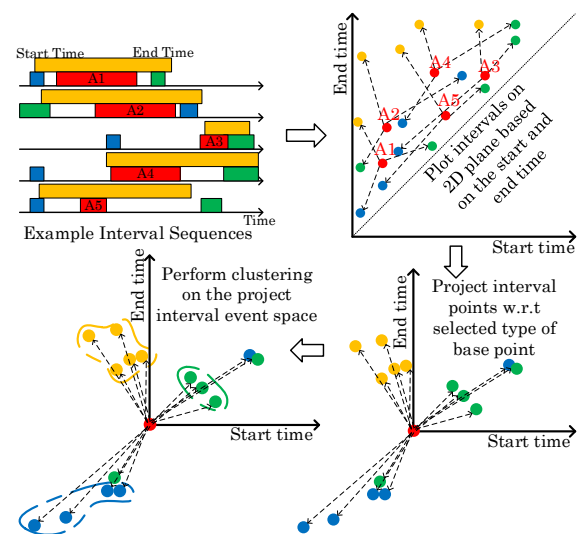


Figure 1: Workflow of PIVOTMiner

outlier ranking. The work flow of the PIVOTMiner is illustrated in Figure 1. Event intervals are modeled as points on 2D plane where the start times and the end times are depicted on the horizontal and vertical axes, respectively. The relationships between events can be considered as vectors. We select one event type as the source type and other event types as the target type. Vectors are constructed within each sequence from point with source type to point with target type. By projecting each source point to the origin, semi-supervised clustering can be applied to group similar patterns that consist of the current source type and each target event type. We repeat the step described above for each event type to generate all binary patterns.

With the idea of PIVOTMiner, we convert the relationship between interval events into vectors. Different distance measure could be employed and normal outlierlieness ranking algorithms can be applied as we show in Section 2. Section 3 presents our first results on running time evaluation of PIVOTMiner and on ranking rare patterns using the PIVOTRanker. Section 4 concludes the paper with an outlook.

## 2. PIVOTRANKER: RANKING PATTERNS

Let the prototype pattern in Figure 2 represent the overlapping of symptoms that leads, in theory, to a certain disease. Specialists would be interested in having an overview of pos-

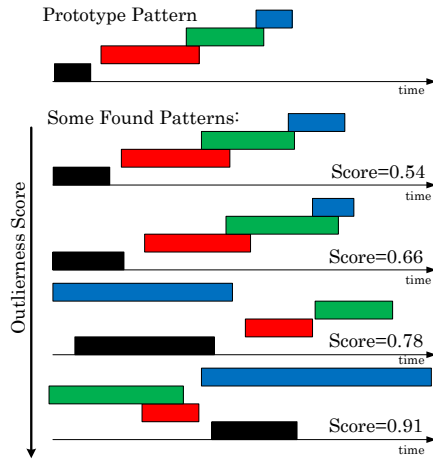


Figure 2: PIVOTRanker output of rare patterns (bottom) ranked according to their outlierness score deviating from the ground truth prototype (top).

itive objects with a deviating pattern, in practice, from the assumed one. The relationship between two interval events is modeled as vectors and a interval-based pattern can be seen as a combination of vectors. In the new representation, and after performing the clustering, some sparse intervals (and thus patterns) deviate from all available clusters and are not dense enough to form a new cluster. These objects are called outliers and they usually do not belong to any of available clusters. The *outlierness* level of those patterns varies according to their densities and to their degree of deviation from available dense clusters, as proposed by [3]. Its value is computed based on the set of vectors with the same source and target event type. We use this outlierness value in our PIVOTRanker to rank rare interval-based patterns.

Since a vector can only describe binary patterns, an overall ranking score needs to be computed for interval-based patterns with more than two events. One method is to use the average value of all vectors belonging to the same sequence. Another method is to select the minimum set of binary vectors that can cover the whole pattern and take the average value as the score for pattern. Since one pattern can be covered by different combinations of binary vectors, we need to find out the set which gives the highest value.

### 3. EXPERIMENTAL EVALUATION

We tested the PIVOTMiner for efficiency and the PIVOTRanker for effectiveness using synthetic and real data sets. To check the capability of the PIVOTRanker realistically, we modeled a data prototype with multiple interval-based noise effects. The real data set is introduced by [5]. Figure 2 illustrates a set of sequences with the corresponding outlier ranking score as an example. The score is computed based on the minimum set of vectors with the highest average score. As shown in the figure, sequences with a higher outlier ranking score deviate much more than lower ones from the original prototype.

We evaluated the efficiency of the PIVOTMiner against the TPrefixSpan [7] algorithm and the QTIPrefixSpan [4] using the real dataset in Figure 3 and a synthetic one in Figure 4. As it is clearly depicted in the two figures, PIVOTMiner scales well with the size of the dataset and is not sensitive to the selection of the minimum support.

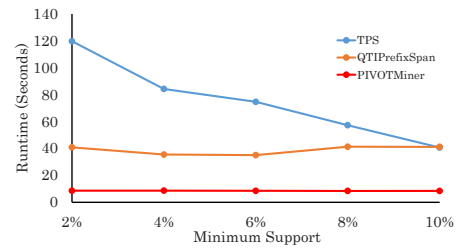


Figure 3: Runtime evaluation w.r.t. *minsup* using the American Sign Language real dataset [5].

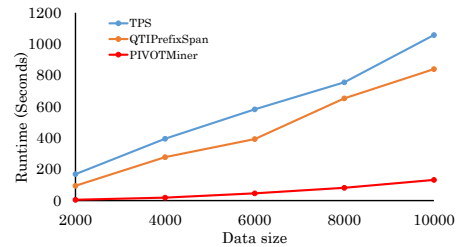


Figure 4: Scalability test using a synthetic dataset.

## 4. CONCLUSION AND OUTLOOK

In this paper we presented our novel efficient algorithm PIVOTMiner for interval-based sequential pattern mining using a geometric representation of intervals. Additionally, we have presented the PIVOTRanker that ranks rare patterns found using the first algorithm using their outlierness score. The source codes and the datasets are available under: <http://dme.rwth-aachen.de/en/PIVOT>. In the future, we will advance the geometrical representation of the PIVOTMiner to include additional information for finding multiple-event patterns. We will also advance the outlierness ranking method to that case.

## 5. REFERENCES

- [1] J. F. Allen. Maintaining knowledge about temporal intervals. *Comm. of the ACM*, 26(11):832–843, 1983.
- [2] Y.-C. Chen, W.-C. Peng, and S.-Y. Lee. Ceminer—an efficient algorithm for mining closed patterns from time interval-based data. In *ICDM*, pages 121–130, 2011.
- [3] E. Müller, I. Assent, U. Steinhausen, and T. Seidl. Outrank: ranking outliers in high dimensional data. In *ICDEW 2008*, pages 600–603, 2008.
- [4] F. Nakagaito, T. Ozaki, and T. Ohkawa. Discovery of quantitative sequential patterns from event sequences. In *ICDMW*, pages 31–36, 2009.
- [5] C. Neidle, A. Thangali, and S. Sclaroff. Challenges in development of the american sign language lexicon video dataset (asllvd) corpus.
- [6] G. Ruan, H. Zhang, and B. Plale. Parallel and quantitative sequential pattern mining for large-scale interval-based temporal data. In *Big Data*, pages 32–39, 2014.
- [7] S.-Y. Wu and Y.-L. Chen. Mining nonambiguous temporal patterns for interval-based events. *TKDE*, 19(6):742–758, 2007.

# SOFYA: Semantic on-the-fly Relation Alignment

Maria Koutraki  
 University of Paris-Saclay  
 45 Av. des États Unis  
 78000, Versailles  
 France  
 kom@prism.uvsq.fr

Nicoleta Preda  
 University of Paris-Saclay  
 45 Av. des États Unis  
 78000, Versailles  
 France  
 preda@prism.uvsq.fr

Dan Vodislav  
 University of Cergy-Pontoise  
 2 Av. Adolphe-Chauvin  
 95302, Cergy-Pontoise  
 France  
 dan.vodislav@u-cergy.fr

## ABSTRACT

Recent years have seen the rise of Web data, in particular Linked Data, with, up to now, more than 1000 datasets in the Linked Open Data Cloud (LOD). These datasets are mostly of entity-centric nature and are highly heterogeneous in terms of domains, language, schema, etc. Hence, the vision of uniformly querying such resources in the LOD has a long way to go. While equivalent *entity instances* across datasets are often linked by *sameAs* links, *relations* from different datasets and schemas are usually not aligned.

In this paper, we propose an on-line *instance-based* relation alignment approach. The alignment may be performed during query execution and requires partial information from the datasets. We align *relations* to a target dataset using association rule mining approaches. We sample for equivalent entity instances with two main sampling strategies. Preliminary experiments, show that we are able to align relations with high accuracy, even if accessing the entire datasets is impossible or impractical.

## 1. INTRODUCTION

As of April 2015, the publicly accessible part of the LOD project counts more than 1000 datasets, which together store more than 30 billion facts. The datasets span across different domains, such as social Web, government data, geographic data, or the life sciences. Moreover, the datasets are highly heterogeneous in terms of schemas, of quality of the data, and only 2% of the schemas are aligned across different datasets [6]. Many of these datasets are accessible through *SPARQL endpoints*, yet uniformly querying them remains a long way to go.

**Motivation.** Successful examples include well known knowledge bases (KB) like DBpedia, YAGO, and Freebase, which comprise factual statements about real world entities. These facts are typically stored as triples  $\langle \text{subject}, \text{relation}, \text{object} \rangle$  (e.g.  $\langle \text{Frank\_Sinatra}, \text{wasBornIn}, \text{USA} \rangle$ ). Yet, even for such KBs, the same entity can have different identifiers (e.g.  $\text{Frank\_Sinatra\_Singer}$  or  $\text{Sinatra}$ ). Similarly, equivalent relations across KBs use different names (e.g.,  $\text{wasBornIn}$  and  $\text{bornInCountry}$ ), hence makes them non-interoperable, such that queries cannot join information across KBs.

**Challenges.** Several approaches have been proposed to align relations across datasets [9, 7, 3], but in all these cases alignment is

performed on the entire KB snapshot. In the real world, however, one may not always have access to the entire dataset. First, KBs are typically quite large (e.g. YAGO, requires 100GB of space on disk), and it is rather impractical to download several entire KBs just to answer a single query. Second, performing relation alignment on KB snapshots, may miss out KB updates. For time-sensitive data, it is better to query the data dynamically. Finally, not all KBs can be freely downloaded. Some providers allow users to issue a limited number of queries to KB via a SPARQL endpoint, but do not allow them to download the entire dataset. In this line, [5] focus on discovering schema alignment on data streams, however, this does not represent any guarantee that one can align any relation given the stream of data.

**Contributions.** In this paper, we propose an *instance-based on-the-fly* approach for relation alignment between two KBs. Our method requires only a SPARQL endpoint for each dataset. Given a relation name in a source KB, e.g. coming from a query on that KB, our method automatically finds corresponding relations in the target dataset, without any need to download the data. Since our method works with few queries, it could be used at query time.

The main idea behind our approach is to use samples of data from both KBs in order to identify candidate relations, then rely on inductive logic programming (ILP) to validate them. Existing works [1, 8], use ILP to mine rules in order to align hierarchies of entities. We go beyond this goal, and want to express more complex mappings, by mining logical rules such as  $\text{kb1}:\text{wasBornIn}(x, y) \Rightarrow \text{kb2}:\text{bornInCountry}(x, y)$ .

In particular, we perform two types of alignments, *subsumption* and *equivalence*, which can be expressed as logical rules. However, as we will show below, such rules cannot be solely mined with standard ILP approaches from small samples of instances. Hence, we develop smart sampling methodologies that are geared to this type of problems. Experiments with real-world datasets show that we can align relations with more than 90% precision, based on only very small samples.

## 2. APPROACH

### 2.1 Rule Mining

Given two KBs  $K$  and  $K'$ , a relation  $r$  in  $K$  and the set  $E$  of *sameAs* entity equivalences, we want to find rules  $r'$  in  $K'$  subsumed by  $r$ , i.e.  $r' \Rightarrow r$ . Candidate relations  $r'$  may be found by sampling  $r(x, y)$ , then considering all  $r'$  such that  $r'(x, y)$  for some sample. Equivalence of relations is expressed as a double subsumption:  $r' \Leftrightarrow r$ , iff  $r' \Rightarrow r$  and  $r \Rightarrow r'$ .

In this work we use two ILP techniques to validate subsumption between relations. A vanilla association rule mining approach [2] could simply regard all absent data as counter-examples (closed world assumption), which yields the following confidence measure:

$$\text{cwaconf}(r' \Rightarrow r) := \frac{\#(x, y) : r'(x, y) \wedge r(x, y)}{\#(x, y) : r'(x, y)} \quad (1)$$

where  $\#(x, y) : A$  is the number of pairs  $(x, y)$  that fulfill  $A$ .

The second technique [4], works under a open world assumption and considers that a KB knows either *all* or *none* of the  $r$ -attributes of some  $x$ . In this case, we count as counter-examples for a rule  $r'(x, y) \Rightarrow r(x, y)$  only instances  $(x, y)$  such that  $x$  has  $r$  relations, but not  $r(x, y)$ . The confidence measure is:

$$pcaconf(r' \Rightarrow r) := \frac{\#(x, y) : r'(x, y) \wedge r(x, y)}{\#(x, y) : \exists y' : r'(x, y) \wedge r(x, y')} \quad (2)$$

## 2.2 Instance Sampling

**Simple Sample Extraction.** We propose a baseline solution that computes a (pseudo-) random set of samples to check if a candidate relation  $r_{sub}$  from  $K'$  satisfies  $r_{sub} \Rightarrow r$ . First, we extract from  $K'$  a set of samples entities that are subjects in  $r_{sub}$  facts:

$$S_{r_{sub}} = \{x_1 \mid r_{sub}(x_1, y_1) \in K', \exists x_2, y_2 \in K : x_1 \equiv x_2 \wedge y_1 \equiv y_2\}$$

The same query extracts the actual  $r_{sub}$  facts where the sample entities occur. More precisely, it extracts the set:

$$K_S^{r_{sub}} = \{r_{sub}(x_1, y_1) \mid x_1 \in S_{r_{sub}} \wedge r_{sub}(x_1, y_1) \in K'\}$$

The actual SPARQL queries that are used to extract the two sets depend on the nature of the relation  $r_{sub}$ . For *entity-entity* relations, we select for a subject  $x_1$  all the facts  $r_{sub}$  for which there are `sameAs` links to entities in  $K$  for both the subject and the object. Since we do not want to punish the score of the alignment because of incomplete information, we ignore the  $r_{sub}$  facts where the `sameAs` links to entities in  $K$  are missing.

In the next step the subject and the object of a  $r_{sub}$  are translated to the equivalent entities in  $K$  and create the set:

$$P_S^{r_{sub}} = \{(x_2, y_2) \mid \exists x_1, y_1 : x_2 \equiv x_1, y_2 \equiv y_1, r_{sub}(x_1, y_1) \in K_S^{r_{sub}}\}$$

then corresponding  $r$  instances are extracted:

$$K_S^{r_{sub}} = \{r(x_2, y_2) \mid r(x_2, y_2) \in K, \exists y'_2 : (x_2, y'_2) \in P_S^{r_{sub}} \wedge r(x_2, y'_2)\}$$

Note that if for some pair  $(x_2, y'_2)$  from  $P_S^{r_{sub}}$  a fact  $r(x_2, y'_2)$  is discovered in  $K$ , then we need to select all the other facts  $r(x_2, y_2)$  of  $x_2$ . This is required by the *pcaconf* measure. For simplicity, in this presentation we assumed that the inverse relations have been added to the two KBs. This is why we only consider direct relations.

If  $r_{sub}$  is an *entity-literal* relation, we retrieve from  $K$  facts of the samples  $S_{r_{sub}}$  and apply string similarity functions to align the literals. Once the sets  $K_S^{r_{sub}}$  and  $K_S^{r_{sub}}$  are retrieved, we can run the *pcaconf* and the *cwaconf* scores on the coalesce of the two sets.

**Unbiased Sample Extraction (UBS).** The random selection of the samples is a fair objective approach, but several cases require a more careful selection of unbiased samples when using *pcaconf*.

*Mining subsumptions that are not equivalences.* Consider the example of a mined subsumption  $K' : composerOf \Rightarrow K : creatorOf$ . When checking the reverse implication to test equivalence, if the sample includes composers that only created musical compositions, we will find that the two relations are equivalent under *pcaconf*, while if a composer is also a writer the reverse implication is false. A way to avoid such missing samples is to discover in  $K'$  a relation subsumed by  $K : creatorOf$  whose domain overlaps with the domain of  $K' : composerOf$ . For instance, we can take the relation  $K' : writerOf$  and consider for sampling the composers that are also writers.

*Mining overlappings that are not subsumptions.* Consider in  $K'$  the relations `hasDirector` for movies and their directors and `hasProducer` for movies and their producers, then in  $K$  the relation `directedBy` for movies and their directors. Since it often happens that the same person directs and produces the same movie, we might wrongly infer that  $K' : hasProducer \Rightarrow K : directedBy$ . To filter out such cases even under *pcaconf*, we may include in the sample movies whose producer and director are different.

To deal with both unbiased samples cases above, our method lays on candidate relations  $K' : r'$  and  $K' : r''$ , subsumed by  $K : r$  for simple samples. Unbiased samples will include facts for  $K' : r'$  and  $K' : r''$  that share the same subjects but have different objects.

More precisely, unbiased samples would contain  $x$  such as  $r'(x, y_1), r''(x, y_2), \neg r'(x, y_2)$ . In the first case, the existence of  $r(x, y_1)$  and  $r(x, y_2)$  filters out the wrong equivalence. In the second one, the condition to filter out the wrong subsumption is to have  $r(x, y_1)$  but not  $r(x, y_2)$ . We used here the same identifiers for equivalent entities in  $K$  and  $K'$ .

## 3. EXPERIMENTAL EVALUATION

**Datasets.** We conduct our experiments on two KBs, with 92 relations from YAGO2 and 1313 relations from DBpedia.

**Baselines.** As baseline solution we consider the (pseudo) random selection of *Simple Sample Extraction* described in Section 2. On the coalesce of the sets of samples retrieved from the two KBs, we have run the two ILP techniques *cwaconf* and *pcaconf*.

We evaluate the algorithms for a sample size of 10 samples (subject entities). Table 1 reports our preliminary results. For the two measures *cwaconf* and *pcaconf*, we have selected the thresholds  $\tau$  that led to the highest average F1 score for both ways implications,  $yago \subset dbpd$  and  $dbpd \subset yago$ .

**Unbiased Sample Extraction.** The method that we propose extends the baseline solution of *pcaconf* by implementing the two strategies for filtering wrong candidates. To eliminate a “wrong” relation we need only one case which shows that there is a contradiction. The results of this method are indicated by the label **UBS** in Table 1. The results suggest that our method consistently prunes wrong candidates.

Table 1: Alignment subsumptions – YAGO and DBpedia relations

	ILP		yago $\subset$ dbpd	dbpd $\subset$ yago
$\tau > 0.3$	pcaconf	P	0.55	0.51
		F1	0.58	0.48
$\tau > 0.1$	cwaconf	P	0.56	0.55
		F1	0.59	0.53
UBS	pcaconf	P	<b>0.95</b>	<b>0.91</b>
		F1	<b>0.97</b>	<b>0.82</b>

## 4. REFERENCES

- [1] J. David, F. Guillet, and H. Briand. Association rule ontology matching approach. *Int. J. Semantic Web Inf. Syst.*, 3(2), 2007.
- [2] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Min. Knowl. Discov.*, 3(1), 1999.
- [3] L. Galárraga, N. Preda, and F. M. Suchanek. Mining rules to align knowledge bases. In *AKBC*, 2013.
- [4] L. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek. Amie: association rule mining under incomplete evidence in ontological knowledge bases. In *WWW*, 2013.
- [5] S. Jaroszewicz, L. Ivantysynova, and T. Scheffer. Schema matching on streams with accuracy guarantees. *Intell. Data Anal.*, 2008.
- [6] M. Schmachtenberg, C. Bizer, and H. Paulheim. Adoption of the linked data best practices in different topical domains. In *The Semantic Web–ISWC 2014*. 2014.
- [7] F. M. Suchanek, S. Abiteboul, and P. Senellart. Paris: Probabilistic alignment of relations, instances, and schema. *PVLDB*, 5(3), 2011.
- [8] C. Tatiopoulos and B. Boutsinas. Ontology mapping based on association rule mining. In *ICEIS (3)*, 2009.
- [9] O. Udrea, L. Getoor, and R. J. Miller. Leveraging data and structure in ontology integration. In *SIGMOD*, 2007.

# Model Kit for Lightweight Data Compression Algorithms

Juliana Hildebrandt, Dirk Habich, Patrick Damme, Wolfgang Lehner  
 Technische Universität Dresden  
 Database Systems Group  
 01189 Dresden, Germany  
 firstname.lastname@tu-dresden.de

## ABSTRACT

Modern database systems are very often in the position to store and efficiently process their entire data in main memory. Aside from increased main memory capacities, a further driver for in-memory database systems has been the shift to a column-oriented storage format in combination with lightweight data compression techniques. In recent years, a lot of lightweight data compression algorithms have been developed to efficiently support different data characteristics. Therefore, database systems should include a large number of these algorithms. To enable this, we introduce our novel modularization concept including our model kit implementation for lightweight data compression algorithms.

## 1. MOTIVATION

With an ever increasing amount of data in almost all application domains, the storage requirements for database systems grow quickly. In the same way, the pressure to achieve the required processing performance increases, too. To tackle both aspects in a uniform way, data compression plays an important role. On the one hand, data compression drastically reduces storage requirements. On the other hand, data compression also is the cornerstone of an efficient processing capability by enabling "in-memory" technologies. As shown in different papers, the performance gain of in-memory data processing is massive because the operations benefit from its higher bandwidth and lower latency [1].

Especially for the use in in-memory database systems, a variety of lightweight compression algorithms have been developed. These algorithms achieve good compression rates with little computational effort for compression as well as decompression. The main classes of lightweight data compression techniques are dictionary compression (DICT), delta coding (DELTA), frame-of-reference (FOR), null suppression (NS), and run-length encoding (RLE) [2, 3]. The algorithms in each class evolve further and the development activities increase over the years, whereas the concept of the classes are interweaved in new algorithms.

Generally, this multitude of algorithms exists because it is impossible to design an algorithm that automatically produces optimal results for any data. In order to support and to implement a wide range of these algorithms in a database system, a unified approach for the specification or engineering is desirable. In our current research, we have focused on that aspect by developing a novel compression scheme consisting of a few number of modules. Our poster presentation at EDBT comprises an in-depth explanation of this compression scheme. As we are going to show, our compression scheme is quite suitable to modularize a variety of lightweight data compression algorithms in a systematic manner. That means, our approach offers an efficient and an easy-to-use way to describe, to compare, and to adapt lightweight data compression techniques. Furthermore, we want to introduce our model kit implementation that is founded on that compression scheme.

## 2. MODULARIZATION CONCEPT

Our novel compression scheme consists of four main modules as shown in Figure 1; the arrows indicate the data flows. The input is a sequence of uncompressed values, the output is a sequence of compressed values. Our whole scheme is a **Recursion** module. The first module in each **Recursion** is a **Tokenizer** splitting the input sequence in finite subsequences or single values. For that, a **Tokenizer** can be parametrized with a calculation rule. Its output is a token, a finite sequence of values that serves as input for our second module, the **Parameter Calculator**. Parameters are often required for the encoding and decoding. This module follows special rules (parameter definitions) for the calculation of several parameters. We summarize different data structures like single values calculated from sequences, dictionaries or vectors as parameters belonging to a token. Our third module is the **Encoder**, which can be parametrized with a

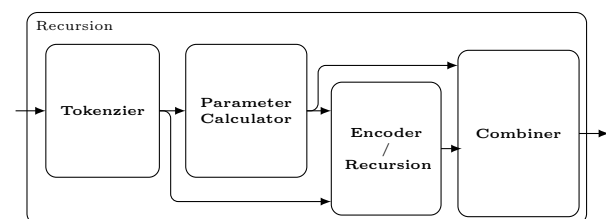


Figure 1: Modularized compression scheme

©2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0



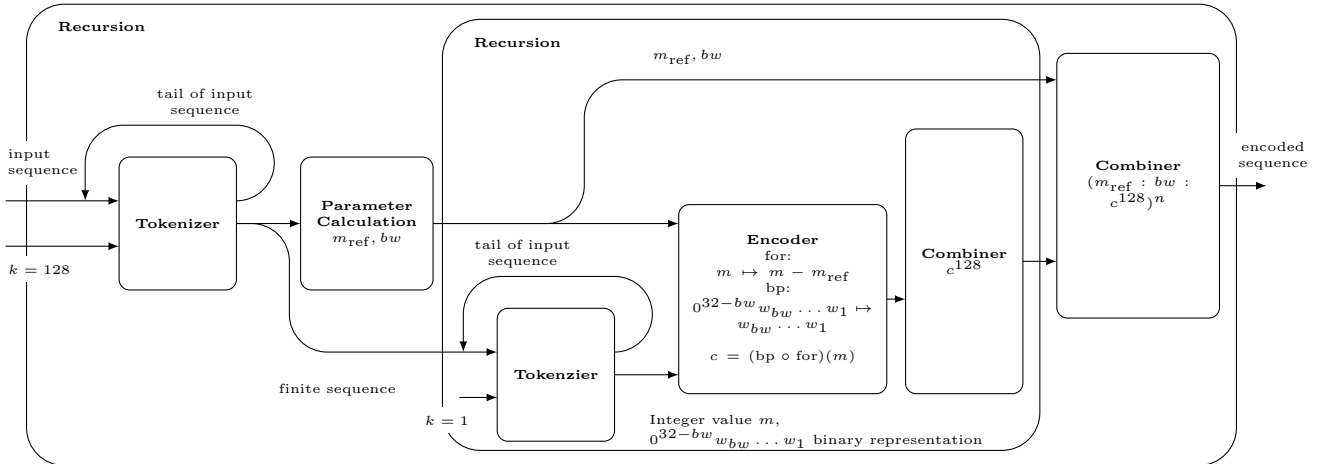


Figure 2: Modularized scheme for the frame-of-reference technique with binary packing

calculation rule for the processing of an atomic input value, whereas the output of the **Parameter Calculator** is an additional set of parameters needed for the calculation inside the **Encoder**. The fourth module is the **Combiner**. It determines how to arrange the output of the **Encoder**. For a more sophisticated hierarchical data partitioning and parameter calculation, we are able to replace the **Encoder** with a **Recursion** module.

### 3. POSTER PRESENTATION DETAILS

Our EDBT poster presentation comprises an in-depth explanation of the compression scheme. In particular, we want to show the applicability by the transcription of a variety of lightweight data compression algorithms. Due to space constraints, we only present one example transcription in this paper: *semi-adaptive frame-of-reference algorithm with binary packing for the compression of positive integer values*. For each set of 128 values, the algorithm calculates the minimum value as reference value. Then, each single integer value is mapped to the offset to the reference value at the logical level. This technique leads to smaller numbers. Then, all 12 offset values are encoded with the same bit width on the physical level. That means, the algorithm has to calculate it. For each compressed sequence of 128 values, we have to store the reference value and the bit width as meta data. Otherwise we are not able to decode the values.

Figure 2 shows the algorithm in our novel compression scheme, which can be directly mapped to our model kit implementation. The first **Tokenizer** is parametrized with a calculation rule that determines that the **Tokenizer** has to output the first 128 32-bit integer values of the input sequence. The tail of the sequence serves as further input for the **Tokenizer** and is processed in the same way in a next step. The **Parameter Calculator** determines the minimum of the 128 values as reference value  $m_{ref}$  and the needed bit width  $bw$  to encode all 128 offset values. Instead of an **Encoder** we use a **Recursion** module. Inside that recursion, we have a **Tokenizer** outputting single integer values. Logically, we have a **Parameter Calculator**. But at that level, we do not need to calculate any parameter here. The **Encoder** manages the logical level of encoding as well as the

bit level. It uses the reference value  $m_{ref}$  and the calculated bit width  $bw$  as parameters. At the logical level it calculates with the function *for* the offset to the common reference value for each of the 128 values. At the bit level, it determines the binary representation of the offset with the help of the bit width. The inner **Combiner** concatenates all physical representations of the 128 offset values to a compressed sequence. Out of the inner **Recursion** the outer **Combiner** concatenates the compressed sequence with the physical representation of  $m_{ref}$  and  $bw$  as long as all input has been processed.

### 4. MODEL KIT AND CONCLUSION

Based on our novel modularized scheme, we also developed an appropriate model kit on the implementation level using C++. Our defined modules are available as building blocks, which can be parameterized with certain calculation rules. The building blocks can be orchestrated to data flows, so that complete lightweight data compression algorithms can be realized. Next, we want to optimize the building blocks, so that the algorithms can be efficiently executed. Furthermore, we want to use the model kit to integrate a large number of algorithms in a database system.

To summarize, in our EDBT poster presentation, we introduce our novel developed compression scheme for lightweight data compression algorithms. In particular, we want to show that our approach offers an efficient and an easy-to-use way to describe, to compare, to adapt, and to implement lightweight data compression techniques.

### 5. REFERENCES

- [1] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *VLDB*, 1999, pp. 54–65.
- [2] H. K. Reghbati, "An overview of data compression techniques." *IEEE Computer*, vol. 14, no. 4, pp. 71–75, 1981.
- [3] D. J. Abadi, S. R. Madden, and M. C. Ferreira, "Integrating compression and execution in column-oriented database systems," in *In SIGMOD*, 2006, pp. 671–682.

# Revisiting DBMS Space Management for Native Flash

Sergey Hardock  
 Databases and Distributed  
 Systems Group  
 TU-Darmstadt, Germany  
 hardock@dvs.tu-  
 darmstadt.de

Iliia Petrov  
 Data Management Lab  
 Reutlingen University,  
 Germany  
 ilia.petrov@reutlingen-  
 university.de

Robert Gottstein  
 Databases and Distributed  
 Systems Group  
 TU-Darmstadt, Germany  
 gottstein@dvs.tu-  
 darmstadt.de

Alejandro Buchmann  
 Databases and Distributed  
 Systems Group  
 TU-Darmstadt, Germany  
 buchmann@dvs.tu-  
 darmstadt.de

## ABSTRACT

In this paper we present our work in progress on revisiting traditional DBMS mechanisms to manage space on native Flash and how it is administered by the DBA. Our observations and initial results show that: the standard logical database structures can be used for physical organization of data on native Flash; at the same time higher DBMS performance is achieved without incurring extra DBA overhead. Initial experimental evaluation indicates a 20% increase in transactional throughput under TPC-C, by performing intelligent data placement on Flash, less erase operations and thus better Flash longevity.

## 1. INTRODUCTION

We argue that the design of the storage architecture is not well suited for new kinds of memory in terms of both software and hardware. Flash memory has significant performance potential, which is underutilized due to the present architecture of Flash SSDs and the way they are used by the DBMS. To provide backwards compatibility with magnetic drives, modern Flash SSDs implement legacy block-device interfaces, supporting *reading* and *writing* at *Flash page granularity* from *immutable device addresses*. As a result a black-box abstraction over the Flash memory is created inside the device by the so called Flash Translation Layer (FTL). Although, this has facilitated the widespread use of SSDs, it results in: (i) significant overhead primarily due to limited on-device resources available to the FTL; (ii) unpredictable performance caused by the background FTL processes (wear-levelling (WL) and garbage collection (GC)) [1]; (iii) inability to optimize the DBMS I/O behavior for new kinds of storage due to an additional level of indirection.

To overcome these disadvantages we recently proposed the NoFTL approach [2], which assumes native Flash as secondary DBMS storage. NoFTL removes all intermediate abstraction layers along

the critical I/O path (block device interface, file system and FTL), and enables the DBMS to control the physical address space of Flash storage directly (see Figure 1). Under NoFTL the DBMS is not confronted with the intricate low-level NAND control. The Flash device is still assumed to have a thin hardware management layer (a low-level controller).

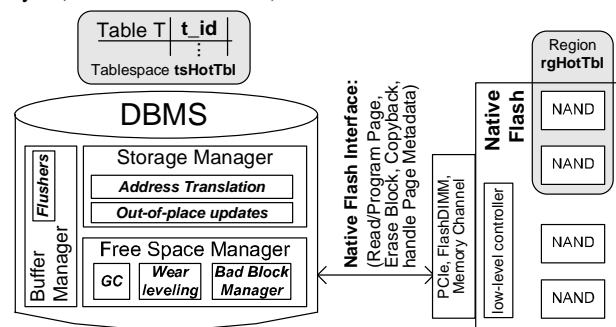


Figure 1: General NoFTL Architecture including Regions.

The major advantages of NoFTL over the traditional FTL-based Flash SSD are the following: (i) usage of the more powerful computational and memory resources of the host system for complex Flash maintenance tasks; (ii) utilization of the DBMS run-time information and knowledge about the stored data and I/O for optimization of GC, WL and the address mapping scheme; (iii) better utilization of available Flash parallelism through intelligent data placement; (iv) direct control over the out-of-place updates, which allows implementing short atomic writes without additional overhead; (v) elimination of redundant functionality along the I/O path.

In the present paper we revisit traditional methods for physical space management on Flash under NoFTL. *The central questions are: how can native Flash comprising a loose set of Flash chips be organized and utilized by the DBMS; do we need new logical storage structures; will they overcomplicate the job of the DBA?*

## 2. LOGICAL STORAGE STRUCTURES AND DATA PLACEMENT ON FLASH

We introduce the concept of **NoFTL regions** as a new physical storage structure, to simplify the organization and management of native Flash storage. A *region* comprises multiple Flash chips or dies, over which the data is evenly distributed. The number of

dies in each region, as well as the structure of their set is dynamic and can change over time depending on different factors: size of objects, required level of I/O parallelism and global wear-levelling.

```
CREATE REGION rgHotTbl (
  MAX_CHIPS=8, MAX_CHANNELS=4, MAX_SIZE=1280M);
CREATE TABLESPACE tsHotTbl (
  REGION=rgHotTbl, EXTENT SIZE 128K );
CREATE TABLE T(t_id NUMBER(3))TABLESPACE tsHotTbl;
```

One or more database objects with similar access properties can be physically placed in a *region*; this holds for complete objects or partitions of them. Objects with different properties are placed in different physically separate *regions* to account and optimize for the specific access characteristics. This gives us two distinct advantages discussed in detail below: (a) coupling of regions and existing logical structures to simplify database administration, and (b) ability to perform intelligent data placement to increase performance and improve Flash longevity.

### Logical Storage Structures and NoFTL Regions.

Existing logical DBMS storage structures can be defined on top of NoFTL regions. Consider the above example: a region of a certain size *rgHotTbl* is defined over 8 chips. A tablespace *tsHotTbl* is defined on top of *rgHotTbl*, where a newly created table *T* is placed. The DBA can continue using established logical storage structures such as *tablespaces* or *extents*, which can be effectively coupled to *regions*. Hence, no new logical structures are needed to manage, organize native Flash storage. *The administration of native Flash does not confront the DBA with additional complexity.*

### Data Placement and NoFTL Regions.

It is well known that Flash memory can perform random access almost as fast as sequential (which is not always true for SSDs). Thus, keeping logically adjacent blocks, physically distributed has negligible performance implications. Furthermore, the distribution over available Flash data channels, dies or planes allows for better I/O parallelism than storing those blocks in sequential order physically on Flash. On the other hand, the database knowledge about the data, about its properties and access patterns can be used to perform smarter data placement and optimize important Flash maintenance algorithms, such as WL, GC and address mapping scheme.

For instance, it is proven that the overhead of garbage collection, which is the major factor for unpredictable performance on SSDs, is highly dependent on the ability to separate between hot and cold data [4, 3]. The limited on-device SSD resources rarely allow for maintaining comprehensive statistics about access patterns and access frequencies over the whole logical address space. At the same time, the DBMS maintains such and other statistics and metadata for each particular database object. Since under NoFTL the DBMS has full control over the physical Flash address space and can perform direct data placement, it becomes easy to utilize the DBMS knowledge. Regions, therefore, allow the DBA and DBMS to control physical data placement on the device in order to optimize Flash management. Intelligent data placement using regions is in the general case an optimal trade off between the provided I/O-parallelism and the overhead of GC.

## 3. PRELIMINARY EVALUATION

We implemented and integrated *regions* in the NoFTL architecture [2] under Shore-MT. Our initial results indicate that the concept of intelligent data placement on Flash has big potential. For instance, under TPC-C we could achieve about 20% increase in transactional throughput (Figure 3) by applying multi-region data place-

ment configuration (Figure 2). In this configuration we have divided database objects of TPC-C based on their I/O properties into 6 regions. Further we have distributed 64 dies of Flash SSD over those regions based on sizes of objects and their I/O rate (required level of I/O parallelism). In addition to performing 20% more transactions than traditional data placement and 20% more READ and WRITE I/Os, the GC performs almost 20% less COPYBACKs and 4.3% less ERASEs. This reduction in write-amplification of multi-region configurations leads to lower I/O latencies and consequently to lower transaction response times. The second effect of decreased write-amplification of multi-region data placement configurations is the better longevity of the Flash devices.

Tablespace/ Region	DB-Objects	Num. of Flash dies
0	DBMS-metadata; HISTORY NEW_ORDER; ORDER	2
1	ORDERLINE	11
2	CUSTOMER	10
3	OL_IDX; STOCK	29
4	C_IDX; I_IDX; S_IDX; W_IDX C_NAME_IDX; ITEM; D_IDX	6
5	WAREHOUSE; DISTRICT NO_IDX; O_IDX; O_CUST_IDX	6

Figure 2: Multi-region data placement configuration for TPC-C

		Traditional data placement	Data placement using Regions
Response Time	TPS	595.42	720.43
	READ 4KB (µs)	531.00	318.63
	WRITE 4KB (µs)	904.00	564.83
	NewOrder TRX (ms)	61.43	58.45
	Payment TRX (ms)	8.88	6.99
	StockLevel TRX (ms)	437.30	293.97
Number of ...	Transactions	359,725	433,192
	Host READ I/Os (4KB)	19,017,255	23,329,310
	Host WRITE I/Os (4KB)	2,740,236	3,259,162
	GC COPYBACKs	4,326,612	3,496,984
	GC ERASEs	110,410	105,564

Figure 3: Performance comparison of traditional and multi-region data placement configuration.

## 4. CONCLUSIONS

The black box architecture of modern Flash SSDs does not allow to utilize the rich DBMS knowledge about stored data and I/O for optimization of complex Flash maintenance functionality such as garbage collection, wear-levelling and address mapping scheme. In pursuit of a solution, we extend our NoFTL approach with the notion of *regions* for allowing the DBMS to control the physical placement based on the properties of database objects. *NoFTL Regions* can be easily mapped to existing DBMS structures such as tablespaces. Our initial results indicate that intelligent data placement can significantly improve the performance of the DBMS as well as the longevity of Flash devices.

## Acknowledgements

This paper was supported by the German BMBF "Software Campus" (01IS12054), the German Research Foundation (DFG) within GRK 1343 "Topology of Technology" and DFG "Flashy-DB" project.

## 5. REFERENCES

- [1] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. SIGMETRICS'09*, 2009.
- [2] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann. Noftl for real: Databases on real native flash storage. In *Proc. EDBT'15*, 2015.
- [3] J. Lee and J.-S. Kim. An empirical study of hot/cold data separation policies in solid state drives. In *Proc. SYSTOR '13*.
- [4] R. Stoica and A. Ailamaki. Improving flash write performance by using update frequency. In *Proc. VLDB'13*, 2013.

# A Two Phase Deep Learning Model for Identifying Discrimination from Tweets

Shuhan Yuan  
Tongji University  
Shanghai, China  
4e66@tongji.edu.cn

Xintao Wu  
University of Arkansas  
Fayetteville, AR, USA  
xintaowu@uark.edu

Yang Xiang  
Tongji University  
Shanghai, China  
shxiangyang@tongji.edu.cn

## ABSTRACT

Discrimination discovery is the data mining problem of unveiling discriminatory practices by analyzing a dataset of historical decision records. In this paper, we focus on discovering discrimination from tweets using deep learning models. One challenge here is that it is difficult to obtain a large well-labeled dataset required by the training of deep learning models for the purpose of discrimination analysis. We develop a two-phase deep learning model to address this challenge. Our model first learns text representations based on weakly-labeled tweets (containing some specific hashtags), then trains the classifier on a small set of well-labeled training data. Experimental results show that: (1) the proposed method can be successfully used for discrimination identification; (2) pre-training text representations, which utilizes weakly-labeled tweets, can significantly improve the accuracy of discrimination detection.

## Keywords

deep learning; discrimination analysis; two phase learning

## 1. INTRODUCTION

Discrimination generally refers to an unjustified distinction of individuals based on gender, race, or religion, and often occurs when the group (e.g., female) is treated less favorably than others. Discrimination discovery and prevention from historical databases has been an active research area recently. In this paper, we are focused on a related but different problem, i.e., how to identify discriminatory tweets. For example, if an individual publishes a tweet saying “*Want to learn photography or how to use photo shop? It’s men’s lifestyle interest. Not for girls!*”, obviously this tweet contains discrimination against female. Identifying discrimination from text is an important task in user-generated content (UGC) mining as discrimination has increasingly become a hotspot of social attention nowadays.

Recent work in natural language processing has shown that deep learning models could learn meaningful represen-

tations (or features) of text and train to classify text on top of text representations with high accuracy in applications like text classification and sentiment analysis. In this paper, we examine the use of deep learning models for discrimination analysis of tweets. However, existing deep learning models require large amounts of training data and it is difficult to obtain such a large well-labeled training dataset (each tweet is clearly marked with discrimination or non-discrimination by domain users) because labeling manually a large number of tweets is time-consuming.

We develop a two-phase deep learning model to detect discrimination from tweets. In the first phase, the model focuses on learning semantic representations of tweets using the large amount of weakly-labeled tweets. In Twitter, users often add hashtags, which mark keywords or topics, in their tweets. We consider tweets containing hashtags like “*#sexism*”, “*#racism*” are weakly-labeled discrimination tweets and those tweets likely contain discrimination information. One example is “*Why are female cabinet members suspect but male ones are not? #bias #sexism*”. However, not all tweets containing such hashtags can be considered as discrimination. For example, the tweet “*#sexism is an important research in behavior research*” is not discriminatory. In general, the tweets that are weakly-labeled by discrimination-related hashtags are likely to be discriminatory than those without discrimination-related hashtags. Hence we train our model to learn the good text representations based on the similarity between the weakly-labeled tweets and well-labeled tweets. In the second phase, we use the representations of tweets trained from the first phase as inputs to train the logistic regression classifier and fine-tune the whole model using the small set of well-labeled tweets.

## 2. THE MODEL

In this section, we describe the two-phase deep learning model to identify discrimination tweets.

### 2.1 Phase One

In the first phase, we first model tweets representations based on semantic composition ideas [4]. Semantic composition aims to understand the text by composing the meaning of each word through a composition function. In our work, we use the Long Shot-Term Memory (LSTM) [3] recurrent neural network as the composition function to model the features of tweets. LSTM is able to model a tweet by sequentially processing each word and mapping a tweet to a low dimensional representation vector. LSTM has various variations. In our work, we adopt a widely used LSTM model

[2] but without peephole connections. In order to learn a tweet representation, the model first maps each word  $w_i$  in a tweet into a  $d$ -dimensional real vector  $x_w \in \mathbb{R}^d$ , also called word embeddings [1]. For a tweet with  $n$  words, a sequence of word embeddings  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  are passed into the LSTM one by one to compute the sequential hidden feature vectors  $\mathbf{h} = (h_1, h_2, \dots, h_n)$ . Then, the model combines the hidden vectors by mean operation  $r = \text{mean}(h_1, h_2, \dots, h_n)$  to get one vector  $r$  as representation of the tweet.

We further need to model the feature of discrimination and non-discrimination category. In order to build the discrimination features, we consider all the discrimination tweets as a document and use the features of each tweet as input to compose the discrimination features of each category. Given a set of discrimination tweets  $T^+ = \{t_1^+, t_2^+, \dots, t_m^+\}$ , after computing the representations of tweets  $R^+ = \{r_1^+, r_2^+, \dots, r_m^+\}$ , we composite the representations of discrimination by Equation 1.

$$Q^+ = \frac{1}{m} \sum_{i \in [1, m]} r_i^+ \quad (1)$$

To build the representations of non-discrimination  $Q^- \in \mathbb{R}^d$ , the framework follows the same procedure.

The objective of our model is to let the representations of weakly-labeled tweets close to the representations of similar category and far away to the representations of their opposite category. For example, if a tweet contains hashtag “#sexism”, we want the representation of this tweet close to the representation of discrimination and far from the representation of non-discrimination. Our model uses cosine function  $\text{sim}(r, Q^+)$  ( $\text{sim}(r, Q^-)$ ) to measure the similarity between a weakly-labeled tweet representation  $r$  and representation of discrimination (non-discrimination) category. If  $r$  is a weakly-labeled discrimination tweet, we set  $\delta = \text{sim}(r, Q^+) - \text{sim}(r, Q^-)$ . If  $r$  is weakly-labeled as non-discrimination tweet, we set  $\delta = \text{sim}(r, Q^-) - \text{sim}(r, Q^+)$ . The loss function is  $L(\delta) = \log(1 + \exp(-\gamma\delta))$ , where  $\gamma$  is a scaling factor. To train the model, we use the back-propagation algorithm by Adadelta [5] to update the parameters of the LSTM model.

## 2.2 Phase Two

In the second phase, we aim to learn the logistic regression classifier to identify discrimination. After the pre-training, the LSTM model which contains word embeddings to the semantic representations of tweets is already well-trained. We stack the logistic regression layer on the LSTM layer and feed the tweets representations as inputs to logistic regression classifier. We use the well-labeled small dataset as training dataset in this phase. The model is to predict whether a tweet contains discrimination  $\hat{y}$ . The logistic regression function is:

$$P(\hat{y}|r, U_l, b_l) = \frac{1}{1 + e^{-(U_l \cdot r + b_l)}}, \quad (2)$$

where  $r$  is the representation of a tweet, and  $U_l, b_l$  are the parameters of logistic regression. We use negative log likelihood as the loss function to train the classifier and fine-tune the whole architecture.

## 3. EXPERIMENTS

We crawled tweets online and labeled 300 discrimination tweets and 300 non-discrimination tweets as the well-labeled

**Table 1: Comparisons of accuracy of our two-phase training deep learning model against other methods**

Methods	Number of training data		
	240	360	480
Our model	<b>0.887</b>	<b>0.901</b>	<b>0.910</b>
Without pre-training	0.870	0.872	0.900
SVM (1-gram)	0.521	0.725	0.713
SVM (2-gram)	0.736	0.756	0.765
Naive Bayes (1-gram)	0.827	0.839	0.860
Naive Bayes (2-gram)	0.852	0.875	0.870

dataset. Meanwhile, we treated 2000 tweets with “#sexism” or “#racism” as weakly-labeled discrimination data and 2000 tweets with “#news” as weakly-labeled non-discrimination data. To evaluate the performance, we split the well-labeled dataset into training data and test data with different sizes and use 5-fold cross validation to evaluate the classification performance. We compare our model with several baselines, which include the deep learning without pre-training the tweets representations, SVM, and Naive Bayes classifiers. We use 1-gram and 2-gram as features of SVM and Naive Bayes classifiers. The prediction results are shown in Table 1. We observe our deep learning model significantly outperforms SVM and Naive Bayes classifiers and the pre-training further improves the accuracy.

## 4. CONCLUSIONS AND FUTURE WORK

We presented a two-phase deep learning model for discrimination analysis of tweets. Our model first learns text representations based on weakly-labeled tweets (containing some specific hashtags), then trains the classifier on a small set of well-labeled training data. The preliminary experiments showed that pre-training text representations by weakly-labeled tweets could improve the accuracy of discrimination detection. Meanwhile, our model can be easily extended to other applications that are restricted by lack of a large amount of training data. In the future, we plan to extend our method to identify more fine-grained discrimination text.

## 5. ACKNOWLEDGMENTS

This work was supported in part by The 973 Program of China (2014CB340404) and China Scholarship Council. This research was conducted while the first author visited University of Arkansas.

## 6. REFERENCES

- [1] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.
- [2] A. Graves. Generating sequences with recurrent neural networks. *arXiv:1308.0850 [cs]*, 2013.
- [3] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [4] J. Mitchell and M. Lapata. Composition in distributional models of semantics. *Cognitive Science*, 34(8):1388–1429, 2010.
- [5] M. D. Zeiler. Adadelta: An adaptive learning rate method. *arXiv:1212.5701 [cs]*, 2012.

# Top- $k$ Dominating Queries, in Parallel, in Memory

Sean Chester, Orestis Gkorgkas, Kjetil Nørnvåg  
 Norwegian University of Science and Technology (NTNU), Trondheim, Norway  
 {sean.chester,orestis,noervaag}@idi.ntnu.no

## ABSTRACT

Top- $k$  dominating queries return the  $k$  points that are better than the largest number of other points. Current methods for answering them focus on indexed data and sequential algorithms. To exploit modern-day parallelism and obtain order-of-magnitude improvements in execution time, we introduce three algorithms, the respective strengths and potential of which are revealed experimentally.

## 1. INTRODUCTION

Top- $k$  dominating queries [6] elegantly fuse top- $k$  queries [4] with skyline queries [2] to produce ranked data without user intervention. Intuitively, a point is ranked highly if it is unequivocally better than many other points. Unsurprisingly, however, it is expensive to evaluate which  $k$  points are better than the most others.

Given the recent emergence of manycore architectures, terabyte-sized RAM, and low-latency, non-volatile memory, it is natural to ask whether top- $k$  dominating queries can be answered more efficiently in a shared, main-memory parallel context. Until now, research has focused on sequential computation in high-latency, disk-based settings [5, 8] and on parallel computation in large, distributed systems [1], or on applications such as web-service ranking [7] and network analysis [5]. This paper investigates how significant efficiency improves can be had on just a single machine.

More formally, a data point *dominates* another, distinct data point if it has equal or better values on every attribute. The *dominance score* of a point  $p$  is simply the number of other points that it dominates. The responses to a top- $k$  dominating query are the  $k$  points with the highest dominance scores. Given an unindexed, memory-resident dataset, we wish to find those  $k$  points as fast as possible.

This paper presents preliminary work towards that goal. In the absence of previous ones, we define three quite distinct algorithms for answering top- $k$  dominating queries on multicore architectures and explore their relative strengths experimentally. We find that each shows promise for maturation, while already executing quickly.

## 2. THREE PROPOSED ALGORITHMS

For multicore top- $k$  dominating queries, we introduce an adaptation of the sequential state-of-the-art and two novel algorithms.

**FILTER** As a baseline, we adapt the sequential, filter-and-refine algorithm for non-indexed data [8]. The algorithm consists of three passes: 1) build a static grid over the data and calculate score bounds for every grid cell; 2) for every cell that cannot be immediately pruned, iterate the points in the cell and filter out those that clearly cannot be in the solution; and 3) refine the solution by computing the exact score for the remaining candidates. The counting and refining steps parallelise readily, since processing is local to each point, but the filtering step would incur a lot of write contention between threads: processing for each point  $p$  will update scores for other points in addition to  $p$ . Therefore, we select the faster but  $\approx 2\times$  less effective filter option (Algorithm 5) of [8] to adapt: within reason, it is better to process excess points in the high-throughput refinement phase than filter them sequentially.

**SORTED** Our next algorithm maximises throughput, but with heuristics to improve efficiency. We first sort the data by Manhattan Norm. Then, for each point in parallel, we iterate the sorted list. For a point  $p$  at index  $i$ , we conduct one-sided dominance tests with points at index  $< i$  to count how often  $p$  is *dominated*. Clearly, if  $p$  is dominated by at least  $k$  other points, it cannot be a top- $k$  solution; so, we break if the count reaches  $k$ . Points that reach their own index are candidates for the solution. Over the remaining indexes  $> i$ , we flip the dominance test and instead count how many points *are dominated by*  $p$  as a score. Finally, we re-sort the data based on the scores; the solution consists of the first  $k$  sorted points.

**PIVOTED** Multicore skyline algorithms benefit from pivot-based partitioning to identify incomparability in addition to dominance [3]; this algorithm attempts it for top- $k$  dominating queries. We will select a series of pivot points (those with the largest dominance area) one-by-one, and use them to globally, dynamically split the data space into a non-uniform grid. We maintain the set of points independently of the grid. Each pivot point newly partitions the entire set of points and the number that end up in the resultant north-east quadrant is exactly the dominance score of that pivot.<sup>1</sup> Meanwhile, the grid is further sub-divided and upper bounds of scores for points in each sub-partition can be updated. Once no cell has an upper bound score better than that of the  $k$ 'th best pivot that we have seen, we terminate. The intuition of the algorithm is that partitioning is easy to parallelise and that the continual fracturing of the data space quickly introduces more knowledge of incomparability and thus very quickly drives down all the upper bound estimates.

## 3. EXPERIMENTS

This section empirically compares our three parallel proposals.

**Setup** We implement the three algorithms in C++ using OpenMP

<sup>1</sup>We describe this in two dimensions for simplicity.

	CORRELATED				INDEPENDENT				ANTICORRELATED			
	1	14	28	56	1	14	28	56	1	14	28	56
FILTER	1929	669	535	568	13412	2672	1935	1861	86369	21056	17762	16745
SORTED	376	313	323	326	5225	624	617	667	54603	4836	2535	1780
PIVOTED	812	715	732	789	964	810	842	833	48008	4870	3115	2579

Table 1: Execution time of each algorithm when run on a single core ( $t = 1$ ), single socket ( $t = 14$ ), all physical cores ( $t = 28$ ), and all virtual cores ( $t = 56$ ). Data distribution varies, but  $n = 10^6$ ,  $d = 3$ , and  $k = 16$  are fixed to match [8]. Times are reported in milliseconds.

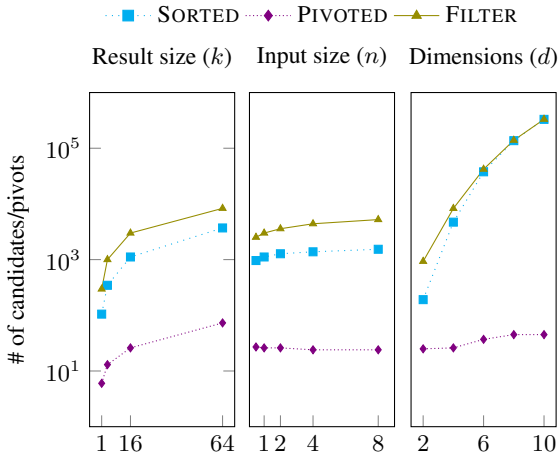


Figure 1: The number of candidates generated by each algorithm as a function of the input data and result size. Independent data.

by extending the openly available SkyBench suite [3] and compile with GNU gcc-4.9.3. We run experiments on a dual socket, 28-core Intel Xeon E5-2683 v3 at 2 GHz that is running Ubuntu 14.04 and has hyperthreading enabled. Time is measured for each algorithm after loading the input from files into a flat array. Ties in dominance score are broken by the order in which candidate points are processed so as not to bias the preferred order of any algorithm.

For comparability with Section 8.3 of [8], we use standard synthetic datasets with defaults of  $k = 16$ ,  $n = 10^6$ , and  $d = 3$ .

**Performance of algorithms** Table 1 shows the execution time of the three algorithms relative to the data distribution and number of threads. In general, we observe the typical pattern that all algorithms perform best on correlated data and worst on anticorrelated data. This is unsurprising because: a) the dominance scores for the top- $k$  points is higher on correlated data, increasing the bounds used for pruning; and b) more points are dominated by at least  $k$  other points and so can be discarded from processing earlier. With additional computational work on anticorrelated data more parallelism can be exposed, thereby achieving more parallel scalability.

The PIVOTED algorithm typically performs best at low thread counts, whereas the SORTED algorithm exhibits very good parallel scalability, even across NUMA nodes, and is consistently the fastest when using all threads. The FILTER method is limited by Amdahl’s Law because of the sequential second phase, so experiences diminishing returns with increasing thread counts. The PIVOTED method struggles to utilise all threads on such low-dimensional correlated and independent data, because there are not enough partitions to iterate and the partitions are quite imbalanced.

The performance of PIVOTED on anticorrelated data is disappointing, given the success that other pivot-based methods (e.g., [3]) have had for standard skyline queries. We will in the future investigate whether better choices of pivot points and the discarding of

some unpromising partitions can improve performance here. Similarly, somehow parallelising the second phase of FILTER could yield sizeable improvements for that algorithm.

**On filters and pivots** Figure 1 internally inspects the performance of the algorithms by counting “important” points. For PIVOTED, these are the *pivots*. For the other methods, these are the *candidate* points that cannot be pruned. The pivots and candidates are exactly those points for which the dominance score is expensively, explicitly computed. We study how variations in  $k$ ,  $n$  (measured in millions), and  $d$  (all on independent data) affect this value.

While it is clear from Figure 1 that PIVOTED computes by far the fewest exact scores, each pivot additionally further partitions the grid, creating much more overhead than evaluating candidates in the other algorithms; so, it is dangerous to compare these values directly across algorithms. Nonetheless, the low number of explicit counts required does suggest that PIVOTED could emerge as the clear best algorithm if the partitioning overhead can be reduced.

The filtering of the other methods is very sensitive to  $d$ , filtering less than 65% of the input at  $d = 10$ , but for  $d \leq 5$ , both prune  $\geq 99\%$ . This indicates a reasonable trade-off for FILTER, because sophisticated pruning replaces parallel work with sequential work.

**Summary and future work** SORTED outperforms the other methods on account of good throughput, but we plan to further develop FILTER (perhaps by trading off more filtering granularity for parallelism) and PIVOTED (trying to bridle the exponential growth of explicitly managed partitions) to see if this result can be overturned. The strong single-threaded performance of PIVOTED in particular, especially on less extreme workloads, suggests that improving its parallel scalability could lead to an extremely fast algorithm.

## 4. REFERENCES

- [1] D. Amagata, Y. Sasaki, T. Hara, and S. Nishio. Efficient processing of top- $k$  dominating queries in distributed environments. In *Proc. of WWW*, 2015.
- [2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. of ICDE*, pages 421–430, 2001.
- [3] S. Chester, D. Šidlauskas, I. Assent, and K. S. Bøgh. Scalable parallelization of skyline computation for multi-core processors. In *Proc. ICDE*, pages 1083–1094, 2015.
- [4] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top- $k$  query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.
- [5] M. Kontaki, A. N. Papadopoulos, and Y. Manolopoulos. Continuous top- $k$  dominating queries. *IEEE Trans. Knowl. Data Eng.*, 24(5):840–853, 2012.
- [6] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1):41–82, 2005.
- [7] D. Skoutas, D. Sacharidis, A. Simitis, V. Kantere, and T. K. Sellis. Top- $k$  dominant web services under multi-criteria matching. In *Proc. of EDBT*, pages 898–909, 2009.
- [8] M. L. Yiu and N. Mamoulis. Multi-dimensional top- $k$  dominating queries. *VLDB J.*, 18(3):695–718, 2009.

# Snapshot Isolation for Neo4j<sup>1</sup>

Marta Patiño-Martínez, Diego Burgos-Sancho  
 Universidad Politécnica de Madrid  
 Madrid, Spain  
 {mpatino,diego.burgos}@fi.upm.es

Ricardo Jiménez-Peris  
 LeanXcale  
 Madrid, Spain  
 rjimenez@leanxcale.com

Iván Brondino, Valerio Vianello, Rohit Dhamane  
 Universidad Politécnica de Madrid  
 Madrid, Spain  
 {ibrondino, vvianello, rdhamane}@fi.upm.es

## ABSTRACT

NoSQL data stores are becoming more and more popular. Graph databases are one of this kind of data stores. Neo4j is a very popular graph database. In Neo4j all operations that access a graph must be performed in a transaction. Transactions in Neo4j use read-committed isolation level. Higher isolation levels are not available. In this paper we present an overview of the implementation of snapshot isolation (SI) for Neo4j. SI provides stronger guarantees than read-committed and provides more concurrency than serializability.

## Keywords

NoSQL, transaction processing, graph databases.

## 1. INTRODUCTION

Graph databases such as Neo4j [1], Titan [2] and Sparksee [3] are being adopted to represent data that is more naturally captured as a graph than with structured or semi-structured data models such as the relational model or key-value models. Graph databases also provide either query languages or APIs that enable for traversing graphs, running the whole query on the data store, therefore, resulting in an efficient traversal of the graph. The use of other data management technology for representing and traversing graphs them becomes very inefficient because it implies executing many iterative queries to extract the adjacent nodes to a given one, what results in a huge overhead.

Some of these graph databases provide transactions, like Neo4j. Neo4j implements a basic isolation level: read-committed. Unfortunately, read committed suffers from many anomalies including unrepeatable reads and phantom reads. Unrepeatable reads allows that a transaction observes different values for a given data item in the same transaction. In the case of graphs, it means that a path that has been traversed might not exist when trying to go through it later in the same transaction. Phantom reads affect to the selection of items with a predicate. This affects a transaction that performs a predicate selection multiple times, since it might observe a different result set each time, resulting in inconsistent behavior. A higher isolation level avoiding these two anomalies is highly recommended.

Snapshot isolation (SI) [4] is an isolation level that has become

very popular since it provides an isolation very close to the one provided by serializability while avoiding read-write conflicts. Snapshot isolation provides a snapshot of the committed state to transactions. SI splits the atomicity of a transaction in two points. The start of the transaction, where logically all reads happen, and the commit of the transaction, where logically all writes happen. SI only can suffer from an anomaly avoided by serializability known as write skew. The anomaly is not exhibited by all applications, for instance, the TPC-C benchmark never observes an anomaly when running on an SI database.

This paper presents how we have designed and implemented a multi-version concurrency control for Neo4j that provides snapshot isolation, avoiding the unrepeatable and phantom reads phenomena that currently affect Neo4j. This work has been performed in the context of the European project CoherentPaaS [5] that provides transactional behavior to NoSQL data stores and global transactions and queries across NoSQL and SQL data stores.

## 2. Neo4j ARCHITECTURE

### Neo4j Architecture

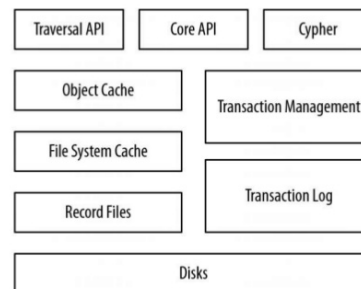


Figure 1: Neo4j Architecture

Neo4j is a graph database, as such, the entities it handles are nodes and relationships (edges in graph jargon) among them. It also allows defining properties and labels. Labels are used to associate a “role” to a node. Properties can be associated to both nodes and relationships.

© 2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0  
 EDBT’16, March 15–18, 2016, Bordeaux, France.

<sup>1</sup> This research has been partially funded by the European Commission under projects CoherentPaaS and LeanBigData (grants FP7-611068, FP7-619606), the Madrid Regional Council, FSE and FEDER, project Cloud4BigData (grant S2013TIC-2894), and the Spanish Research Agency MICIN project BigDataPaaS (grant TIN2013-46883).



Neo4j architecture is similar to the one of a traditional database, although it differs quite a bit in the details (Figure 1). Overall, the architecture has an object cache and a persistent store as a traditional database. However, the internal representation is optimized for graphs.

Nodes are kept in a file whose position is determined by the node identifier. That position in the file contains the ID of the first relationship of the node and the ID of its first property. Relationships are stored in a different file. The source node of the relationship and the destination node are stored. Properties of nodes and relationships are stored in a different file.

Neo4j also uses indexes to optimize some of the accesses. It has two indexes for nodes, one for labels and another one for properties that map them to the set of nodes associated to them. It also maintains one index for relationships, mapping properties to nodes holding those properties.

### 3. SNAPSHOT ISOLATION

Snapshot isolation is typically implemented as a multi-version concurrency control (MVCC). It requires keeping track of multiple versions per entity. This means that updating in place is not especially convenient and a mechanism is needed to maintain multiple versions of each data item, either physically or logically.

SI can be implemented by enforcing two rules. The read rule states that a transaction should observe the most recent committed version of each data item at the time the transaction start. The write rule states that no two concurrent transactions can update the same data item.

SI requires a way to remove obsolete versions of the data items that will never be read by active transactions. Another important issue to take into account is that versions of uncommitted data items should be kept private, but they should be read by the transaction that wrote them to guarantee that a transaction reads its own writes.

### 4. SNAPSHOT ISOLATION FOR Neo4j

Transactions are assigned a start timestamp that corresponds to the most recent committed state. The commit timestamp is given to a transaction when it commits. This commit timestamp is used to tag each item (version) the transaction has updated. We have versioned both nodes and relationships. Versions are implemented as an additional property for both of them. This property stores the commit timestamp. Another property has been added to indicate if a data item has been deleted. A deleted data item has to be kept till no previous version can be read by an active transaction. This mechanism is also called tombstone versions. Versions are kept in the Object Cache of Neo4j. In particular, each object representing a node or relationship stores a list of versions. In that way, when a transaction reads a node, the right version for the reading transaction can be obtained by traversing the list of versions.

Neo4j uses an iterator to traverse the persistent state when needed to answer queries. We have enriched this iterator to take into account the versions kept in the cache in order to guarantee read-your-own writes behavior.

Neo4j implements read-committed isolation with a traditional locking mechanism with short read locks and long write locks. We have removed the short read locks since they are not needed for snapshot isolation. The implementation of long write locks has been modified to perform write-write conflicts implementing a first-updater wins strategy.

Multi-versioning has also been applied to indexes. Properties and labels are never deleted in Neo4j even if no node/relationship is using them. We version them to know whether they should be visible or not for a particular transaction. When a property or label has been created by a transaction with a higher timestamp than the start timestamp of the reader transaction, it can simply discard them. If the timestamp is equal or lower than the start timestamp of the reading transaction then the list of associated nodes/relationships is traversed. The nodes/relationships are tagged with the commit timestamp of the transaction that associated the label/property to the node/relationship. In this way, it is possible to discard those nodes/relationships that do not correspond to the snapshot to be observed by the transaction (those with a higher commit timestamp than the start timestamp of the reading transaction).

The most difficult question to provide snapshot isolation in Neo4j is how to implement multi-versioning in an efficient way. One of the most common inefficiencies introduced by multi-versioning is the version garbage collection process. The approach we have adopted avoids this issue by only writing to the persistent data store the most recent committed version of each data item. The other versions are kept in memory. In order to make the version garbage collection efficient, they are threaded with a double linked list sorted by timestamp to enable to perform the garbage collection just traversing those versions that must be garbage collected. In this way, the cost of garbage collection is reduced to the minimum.

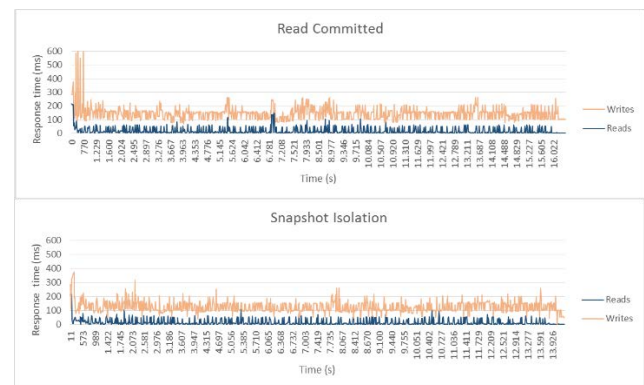


Figure 2: Performance Results

We have performed a preliminary performance evaluation comparing the original implementation of transactions in Neo4j with our SI implementation. The database has 12.3MB and stores movies (from <http://neo4j.com/developer/example-data/>). The workload executes 50% updates and 50% reads. Read transactions read a random node. Update transactions read a random node and modify a random property of the node. In Fig. 2 the results of the micro-benchmark are shown. The response time is similar for both implementations, updates last 100-200 ms and reads below 50 ms.

### 5. REFERENCES

- [1] I. Robinson, et al. Graph Databases. O'Reilly Media. 2015.
- [2] <http://thinkarelius.github.io/titan/>
- [3] <http://sparsity-technologies.com/>
- [4] H. Berenson, et al. A Critique of ANSI SQL Isolation Levels. SIGMOD 1995.
- [5] <http://coherentpaas.eu>

# Maximum Coverage Representative Skyline

Malene S. Søholm  
Aarhus University, Denmark  
soeholm@cs.au.dk

Sean Chester  
NTNU, Norway  
sean.chester@idi.ntnu.no

Ira Assent  
Aarhus University, Denmark  
ira@cs.au.dk

## ABSTRACT

Skyline queries represent a dataset by the points on its pareto frontier, but can become very large. To alleviate this problem, representative skylines select exactly  $k$  skyline points. However, existing approaches are not *scale-invariant*, not *stable*, or must materialise the entire skyline.

We introduce the *maximum coverage representative skyline*, which returns the  $k$  points collectively dominating the largest area of the data space. It satisfies the above properties and reflects a critical property of the skyline itself.

## 1. INTRODUCTION

Grasping large datasets can be overwhelming. The skyline query [2] helps by summarizing a dataset with only those points that represent the pareto frontier of the data. A point  $p$  is on the pareto-frontier (and thus in the skyline) if it is not *dominated* by some other point  $q$ , i.e.,  $p$  is better than all non-equal points in at least one attribute. However, even this is often not enough. The skyline may grow quite large: e.g., on high dimensional data, points have more opportunities (dimensions) on which to be better than other points.

In order to solve this, several approaches have been proposed. Given an integer  $k$ , a *ranking skyline* [3, 9, 10] returns the  $k$  points with the highest score according to a skyline-based utility function. However, the full skyline must be retrieved, which, even using highly parallel computation on a GPU, can still take several seconds [1].

*Regret minimising sets* [4, 7] return the  $k$  points for which a worst-case linear utility function evaluates to a score on the subset as closely as possible to the one on the skyline. Computing such a set also requires knowing the skyline.

Existing approaches for *representative skylines* [5, 6, 8] return the  $k$  skyline points best *representing* the full skyline, but require knowing the skyline to be calculated: the number of skyline points between all pairs of representative skyline points [6]; the maximum distance from any non-representative skyline point to its nearest representative [8]; or the  $k$  skyline points maximising the number of non-skyline

points dominated [5]. Also, [8] is not *scale-invariant*, e.g., scaling miles to kilometres distorts the result, and [5] is not *stable*, i.e., “junk” non-skyline points can be added to manipulate the representative skyline. In this paper, we introduce the first representative skyline to avoid all of these pitfalls.

## 2. MAXIMIZING COVERAGE

The skyline is defined as the subset of non-dominated points [2]. The skyline also has various interesting properties, e.g. that it dominates the rest of the dataset. In [5], this property is emphasized and a representative skyline is developed, but it is not stable as mentioned above.

Another property is that no other subset of points dominates a larger area of the data space. I.e, the skyline captures *the contour of the data space occupied by the data*. This property is agnostic to non-skyline points, suggesting inherent stability. Thus we introduce the *maximum coverage representative skyline* (MCRS): the size- $k$  set that dominates the largest area of the data space. It is the set of  $k$  points that best achieves this critical property of the skyline.

Note that the MCRS is necessarily a subset of the skyline (at least if  $k$  is smaller than the size of the skyline), since every non-skyline point dominates less area than the skyline points that dominate it. The MCRS is also both stable and scale-invariant, since neither adding/removing non-skyline points nor scaling the dataset in any dimension affects the relative size of the dominance area. Perhaps most appealingly, the MCRS is skyline-agnostic: there is no inherent dependence on knowing the skyline to compute the optimal MCRS: the size of the area collectively dominated by any given set of points is unrelated to knowing the full skyline.

Formally, if  $dom\text{-}area(p)$  denotes the area occupied by the (infinitely many) points  $q \in \mathbb{R}^d$  dominated by  $p$ , then the MCRS of size  $k$  on dataset  $\mathcal{D}$  is<sup>1</sup>:

$$MCRS(\mathcal{D}, k) = \operatorname{argmax}_{S \subseteq \mathcal{D}, |S|=k} \left| \bigcup_{p \in S} dom\text{-}area(p) \right| \quad (1)$$

Figure 1 gives an example: the MCRS of size 2 is  $\{p_1, p_3\}$ , since  $p_1$  and  $p_3$  cover an area of 42, whereas  $p_1$  and  $p_2$  only cover 40 and  $p_2$  and  $p_3$  only cover 34.

**An algorithm for 2d** In the following, we show that in two dimensions, the MCRS can be computed in time  $\mathcal{O}(m^3k + n \log n)$  and space  $\mathcal{O}(mk + n)$ , where  $n = |\mathcal{D}|$  and

<sup>1</sup>Note, importantly, that this set formulation avoids multiply counting area dominated by more than one point in a set  $S$ .

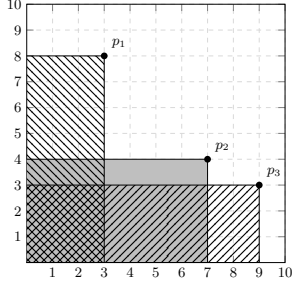


Figure 1: Three skyline points  $p_1$ ,  $p_2$ , and  $p_3$  and their corresponding dominance areas. The optimal size-2 MCRS is indicated by the striped area.

$m = |\text{skyline}(\mathcal{D})|$ . This is a nice asymptotic result given that the naive search space is  $\mathcal{O}(n^k)$ . The algorithm proceeds in three steps: First, sort  $\mathcal{D} \cup \{(1, 0)\}$  so that  $p_0 \leq \dots \leq p_m$ ; then, discard points dominated by their predecessors and relabel points  $p_0, \dots, p_m$ ; the MCRS will now be the value of  $\text{MCRS}(m, m, k + 1) \setminus \{p_m\}$  in the following recursion:

$$\begin{aligned} \text{MCRS}(x, y, 0) &= \{p_y\} \\ \text{MCRS}(x, y, \kappa), y \geq \kappa &= \{p_0, \dots, p_{\kappa-1}\} \\ \text{MCRS}(x, y, \kappa) &= \underset{s \in \{\text{MCRS}(x, \hat{y}, \kappa) \cup \{p_y\}, \text{MCRS}(x-1, y, \kappa)\}}{\text{argmax}} |dom\text{-area}(s)|, \end{aligned}$$

where:

$$\hat{y} = \underset{0 \leq y' < y}{\text{argmax}} \text{area}(\langle p_x \cdot x, p_{y'} \cdot y \rangle, \langle 1, p_y \cdot y \rangle) + |dom\text{-area}(\text{MCRS}(x, y', \kappa - 1))|.$$

The intuition behind the recursion is to sweep through all pairs of skyline points, calculating for each pair the best solution that dominates all the space that it dominates. Because dominance is transitive, the result for each pair of points is very similar to those for nearby pairs of points. One can see this as traversing column-by-column the intersection points of the grid-partitioning induced by the skyline. (In Figure 1 the sequence  $[(9, 0), (9, 3), (7, 0), (7, 3), (7, 4), (3, 0), (3, 3), (3, 4), (3, 8), (0, 0), (0, 3), (0, 4), (0, 8)]$ .)

By adding the sentinel  $p_m = \langle 1, 0 \rangle$  with  $|dom\text{-area}(p_m)| = 0$  to the end of the list, the last column aggregates the best solution from the entire grid. Using dynamic programming to solve the recursion leads to the asymptotic results.

This non-indexing algorithm first computes the skyline to improve efficiency. However, that does not imply computing the skyline first is always more efficient; an index may permit prioritising promising regions of the data space. Also, this algorithm computes the *optimal* solution, not a greedy approximation, thereby allowing us to study the proposed model itself (rather than just the algorithm's efficiency).

### 3. REPRESENTATIVENESS OF AN MCRS

In this section, we evaluate the MCRS concept and algorithm. We generate independent (I) and anti-correlated (A) datasets of 1 million 2-dimensional points as per [2], which have 17 and 64 skyline points each.<sup>2</sup> We implement the algorithm in C++ and execute it on a machine with an Intel Core i7-4770K 3.50GHz CPU and 16GB of memory.

Figure 2a shows the dominance area of the MCRS relative to the skyline and Figure 2b shows execution time, both as

<sup>2</sup>Correlated 2d skylines are already sufficiently small.

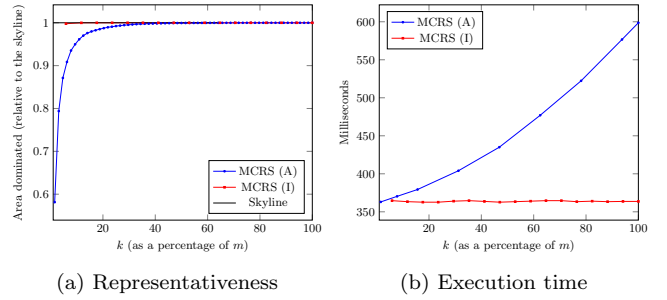


Figure 2: Evaluation of the MCRS

a function of  $k$ . (I): The MCRS almost exactly represents the skyline, even at  $k = 1$ , with stable execution time. (A): It quickly approaches the skyline, dominating  $> 90\%$  with fewer than 10% of the points. Computing representations with  $\leq 60\%$  of the skyline takes less than a half-second.

### 4. CONCLUSION AND FUTURE WORK

We introduced the *maximum coverage representative skyline* (MCRS), a scale-invariant, stable, skyline-agnostic representative skyline, achieving what the skyline achieves. We gave an efficient algorithm to compute an optimal 2d MCRS with which we illustrated that the MCRS covers much of the data space as a full skyline, even for small  $k$ .

We will extend this work with algorithms for  $> 2d$  and multi-dimensional indexes, e.g. R-tree extensions, that can exploit the independence of the MCRS from the skyline.

### 5. REFERENCES

- [1] K. S. Bøgh, S. Chester, and I. Assent. Work-efficient parallel skyline computation for the GPU. *PVLDB*, 8(9):962–973, 2015.
- [2] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.
- [3] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. Tung, and Z. Zhang. On high dimensional skylines. In *EDBT*, pages 478–495. Springer, 2006.
- [4] S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides. Computing k-regret minimizing sets. *PVLDB*, 7(5):389–400, 2014.
- [5] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: The k most representative skyline operator. In *ICDE*, pages 86–95, 2007.
- [6] M. Magnani, I. Assent, and M. L. Mortensen. Taking the big picture: representative skylines based on significance and diversity. *VLDB J*, 23(5):795–815, 2014.
- [7] D. Nanongkai, A. D. Sarma, A. Lall, R. J. Lipton, and J. Xu. Regret-minimizing representative databases. *PVLDB*, 3(1–2):1114–1124, 2010.
- [8] Y. Tao, L. Ding, X. Lin, and J. Pei. Distance-based representative skyline. In *ICDE*, pages 892–903, 2009.
- [9] G. Valkanas, A. N. Papadopoulos, and D. Gunopulos. Skyline ranking à la IR. In *EDBT/ICDT Workshops*, pages 182–187, 2014.
- [10] A. Vlachou and M. Vazirgiannis. Link-based ranking of skyline result sets. In *M-Pref Workshop*, 2007.

# An On-Line Approximation Algorithm for Mining Frequent Closed Itemsets Based on Incremental Intersection

Koji Iwanuma  
University of Yamanashi  
4-4-11 Takeda, Kofu-shi  
Yamanashi, Japan  
iwanuma@yamanashi.ac.jp

Yoshitaka Yamamoto  
University of Yamanashi  
4-4-11 Takeda, Kofu-shi  
Yamanashi, Japan  
yyamamoto@yamanashi.ac.jp

Shoshi Fukuda  
University of Yamanashi  
4-4-11 Takeda, Kofu-shi  
Yamanashi, Japan

## ABSTRACT

We propose a new on-line  $\epsilon$ -approximation algorithm for mining closed itemsets from a transactional data stream, which is also based on the incremental/cumulative intersection principle. The proposed algorithm, called LC-CloStream, is constructed by integrating CloStream algorithm and Lossy Counting algorithm. We investigate some behaviors of the LC-CloStream algorithm. Firstly we show the incompleteness and the semi-completeness for mining all frequent closed itemsets in a stream. Next, we give the completeness of  $\epsilon$ -approximation for extracting frequent itemsets.

## Keywords

On-line algorithm, approximation, closed itemset, intersection, completeness

## 1. INTRODUCTION

Intersecting transactions in a data set is an alternative characterization of closed itemsets [1, 3, 4], which naturally leads to an incremental/cumulative computation of closed itemsets in a transaction data stream. CloStream [6] is an exact-computing on-line mining algorithm, which is a direct implementation of the incremental intersecting approach. Such an incremental intersection approach, however, has great difficulties, in practice, for quitting or breaking intersections in early stages, because it is difficult to predict in advance that current intersection operations never produce any frequent closed itemsets[1].

In this paper, we propose a new on-line  $\epsilon$ -approximation algorithm for mining closed itemsets from a stream, which is also based on the incremental/cumulative intersection principle. The proposed algorithm, called LC-CloStream, is constructed by integrating CloStream [6] algorithm and Lossy Counting algorithm [2]. LC-CloStream succeeded in overcoming the above difficulties using  $\epsilon$ -approximation [2, 5].

We study fundamental properties of LC-CloStream algorithm. Firstly we show the incompleteness and the semi-completeness for mining all frequent closed itemsets in a stream. Next, we give the completeness of  $\epsilon$ -approximation

for extracting frequent itemsets from a transaction streams.

## 2. PRELIMINARIES

Let  $I = \{e_1, e_2, \dots, e_r\}$  be a set of items. A non-empty subset  $A$  of  $I$  is called an *itemset* (or *transaction*). A *transaction stream* of length  $N$  is a sequence of  $N$  transactions  $\langle A_1, A_2, \dots, A_N \rangle$ . In this paper, we denote items as  $a, b, c, \dots$ , and itemsets as  $A, B, C, \dots$ . We also abbreviate an itemset  $\{e_1, e_2, \dots, e_m\}$  as  $e_1 e_2 \dots e_m$ , for simplicity.

Let  $\mathcal{S}$  be a stream  $\langle A_1, \dots, A_N \rangle$  and  $B$  be an itemset. We define a multiset  $\mathcal{K}(B, t)$  at time  $t$  ( $1 \leq t \leq N$ ) as  $\mathcal{K}(B, t) = \{A_j \in \mathcal{S} \mid B \subset A_j, 1 \leq j \leq t\}$ . The *frequency* of  $B$  at time  $t$ , denoted as  $\text{sup}(B, t)$ , is  $|\mathcal{K}(B, t)|$ . Given a minimal frequency threshold  $\sigma$  ( $0 < \sigma < 1$ ),  $B$  is *frequent* at time  $t$  in  $\mathcal{S}$  if  $\text{sup}(B, t) \geq \sigma \cdot t$ . An itemset  $B$  is *closed* at time  $t$  in  $\mathcal{S}$  if there is no itemset  $C$  such that  $B \neq C$  and  $B \subset C$  and  $\text{sup}(B, t) = \text{sup}(C, t)$ .

The following recursive relation makes it possible to incrementally compute closed itemsets in a stream  $\mathcal{S}$ . Let  $CIS(\mathcal{S})$  be a set of all closed itemsets in  $\mathcal{S}$  and  $\circ$  be a well-known concatenation operator of two sequences.

PROPOSITION 1 ([1, 3]). *Let  $\mathcal{S}$  be a stream  $\langle A_1, \dots, A_N \rangle$ . We have:*

$$\begin{aligned} CIS(\langle A_1 \rangle) &= \{A_1\} \\ CIS(\mathcal{S}_k) \circ \langle A_{k+1} \rangle &= CIS(\mathcal{S}_k) \cup \{A_{k+1}\} \cup \\ &\quad \{B \mid \exists C \in CIS(\mathcal{S}_k) : B = C \cap A_{k+1}\}, \end{aligned}$$

where  $\mathcal{S}_k$  is the  $k$  element prefix of  $\mathcal{S}$ , i.e.,  $\langle A_1, \dots, A_k \rangle$ .

CloStream [6] is an on-line exact counting algorithm for mining closed itemsets in a stream, which uses the above recursive relation in a straightforward way, and thus cannot avoid a combinatorial explosion problem caused by  $CIS(\mathcal{S})$ .

## 3. LC-CLOSTREAM

The LC-CloStream algorithm maintains an internal *frequency table*  $TS$ . Formally,  $TS$  is a set of tuples  $\langle B, f(B), \delta(B) \rangle$ , where  $B$  is an itemset,  $f(B)$  is the number of occurrences of  $B$  after the time  $t_B$  when  $B$  was lastly stored in  $TS$ , and  $\delta(B)$  is the maximal error count at time  $t_B$ . We write the frequency table  $TS$  at time  $t$  as  $TS(t)$ , and similarly for  $f(B, t)$  and  $\delta(B, t)$ . Let  $\mathcal{SP}(B, t)$  denote the set of supersets of  $B$  belonging to the frequency table  $TS(t)$ , that is,  $\mathcal{SP}(B, t) = \{C \in TS(t) \mid B \subset C\}$ . We define  $\text{maxSP}(B, t)$  as follows:

$$\text{maxSP}(B, t) = \underset{C \in \mathcal{SP}(B, t)}{\text{argmax}} (f(C, t) + \delta(C, t))$$

The former part of LC-CloStream algorithm, i.e., in lines 5 to 18, performs the incremental intersection and the latter

---

**Algorithm 1** LC-CloStream algorithm

---

**Input:** a stream  $\mathcal{S} = \langle A_1, A_2, \dots, A_N \rangle$ ,  
a relative minimal frequency threshold  $\sigma$  ( $0 < \sigma < 1$ ),  
a maximal permissible error ratio  $\epsilon$  ( $0 < \epsilon < \sigma$ ).

**Output:** a family  $\mathcal{FCS}$  of frequent closed item sets in  $\mathcal{S}$

```
1:  $t \leftarrow 1$  ▷  $t$  is a current time
2: Initialize the frequency table  $TS$ .
3: while  $t \leq N$  do
4:   Read  $A_t$ .
5:   for each  $B \in TS$  do
6:      $C \leftarrow B \cap A_t$ 
7:     if  $C \neq \emptyset$  then ▷ i.e. the case of  $\mathcal{SP}(C, t) \neq \emptyset$ 
8:        $D \leftarrow \max \mathcal{SP}(C)$ 
9:       if  $C \notin TS$  then ▷ register  $C$  as a new entry
10:         $TS \leftarrow TS \cup \{ \langle C, f(D) + 1, \delta(D) \rangle \}$ 
11:       else ▷ increase the frequency vale of  $C$ 
12:         $TS \leftarrow (TS - \{ \langle C, f(C), \delta(C) \rangle \})$   

         $\cup \{ \langle C, f(D) + 1, \delta(D) \rangle \}$ 
13:       end if
14:     end if
15:   end for
16:   if  $A_t \notin TS$  then ▷ register  $A_t$  as a new entry
17:      $TS \leftarrow TS \cup \{ \langle A_t, 1, \epsilon \cdot (t - 1) \rangle \}$ 
18:   end if
19:   for each  $B \in TS$  do ▷  $\epsilon$ -elimination
20:     if  $f(B) + \delta(B) \leq \epsilon \cdot t$  then
21:        $TS \leftarrow TS - \{ \langle B, f(B), \delta(B) \rangle \}$ 
22:     end if
23:   end for
24: end while
25: return  $\mathcal{FCS}(N) = \{ B \in TS \mid f(B) + \delta(B) \geq \sigma \cdot N \}$ 
```

---

part in lines 19 to 23 executes the  $\epsilon$ -elimination operation, which involves an  $\epsilon$ -approximation computation.

Notice that Algorithm 1 is described declaratively for simplicity, thus has the time complexity  $O(k^2)$  where  $k$  is the total number of entries in  $TS$ , while [6] gave an optimized procedural form of the complexity  $O(k)$ .

Unfortunately, LC-CloStream algorithm has a counterexample for the completeness, as shown in Example 1. We can, however, give the semi-completeness for LC-CloStream.

*Example 1.* Let  $\mathcal{S}_1$  be a stream  $\langle a, b, b, b, b, b, ac, ac, ac \rangle$  of length 9. We suppose  $\sigma = 0.3$  and  $\epsilon = 0.2$ . Then, the frequent closed itemsets in  $\mathcal{S}_1$  are three itemsets  $a, b, ac$ . At time  $t = 1$ , LC-CloStream algorithm processes the first transaction  $a$  and store the set  $a$ , as a new closed itemset, into the frequency table  $TS$ . At time  $t = 2$ , LC-CloStream adds the set  $b$  into  $TS$ , and so on. At time  $t = 6$ , LC-CloStream firstly increase the frequency counter  $f(B)$  in  $TS$ , then the table  $TS$  becomes to  $\{ \langle a, 1, 0 \rangle, \langle b, 5, 1 \rangle \}$  at this point. Next LC-CloStream performs the  $\epsilon$ -elimination rule to  $TS$ , and delete the tuple of the closed set  $a$  since  $f(a, 6) + \delta(a, 6) = 1 < 1.2 = \epsilon \cdot 6$  holds. At time  $t = 7$ , LC-CloStream registers the set  $ac$  to  $TS$  as a new closed set, but cannot increase the frequency counter of the set  $a$ , because  $TS$  has the tuple of  $a$  no longer. Thus, LC-CloStream eventually returns the set  $\mathcal{FCS}(9) = \{ b, ac \}$  and fails to produce the frequent closed itemset  $a$ .

Next, we show a semi-completeness theorem which partially overcomes the deficit shown above in LC-CloStream. Furthermore, we give completeness theorem of LC-CloStream for frequent itemsets mining based on  $\epsilon$ -approximation.

*Definition 1.* Let  $S$  be a stream of length  $N$ ,  $B$  be a closed itemset and  $\epsilon$  be a maximal error ratio. We say,  $B$  is  $\epsilon$ -extendable on  $S$  if there is a closed itemset  $C$  such that  $B \subset C$ ,  $B \neq C$  and  $\sup(B) - \sup(C) \leq \epsilon N$

**THEOREM 1 (SEMI-COMPLETENESS FOR CLOSED ITEMSETS).** *Let  $\mathcal{S}$  be a stream of length  $N$  and  $B$  be a frequent closed itemset in  $\mathcal{S}$ . If  $B$  is NOT  $\epsilon$ -extendable, then  $B \in \mathcal{FCS}(N)$ .*

*Definition 2.* Let  $\mathcal{S}$  be a stream of length  $N$ ,  $\sigma$  be a minimal frequency threshold and  $\mathcal{FCS}(N)$  be a output produced from  $\mathcal{S}$  by LC-CloStream algorithm. Then we define  $\mathcal{RS}(N)$  as follows:

$$\mathcal{RS}(N) = \mathcal{FCS}(N) \cup \{ C \mid \exists B \in \mathcal{FCS}(N) : C \subset B, C \neq \emptyset \}$$

**THEOREM 2 (COMPLETENESS FOR ITEMSETS).** *Let  $\mathcal{S}$  be a stream of length  $N$  and  $B$  be a frequent itemset in  $\mathcal{S}$ . Then  $B \in \mathcal{RS}(N)$ .*

*Definition 3.* Let  $\mathcal{S}$  be a stream of length  $N$  and  $\epsilon$  be a maximal error ratio. For any itemset  $B$  at time  $t$  ( $1 \leq t \leq N$ ), we define  $F(B, t)$  and  $\Delta(B, t)$  as follows:

1. if  $\mathcal{SP}(B, t) = \emptyset$ , then  $F(B, t) = 0$ ,  $\Delta(B, t) = \epsilon \cdot t$

2. if  $\mathcal{SP}(B, t) \neq \emptyset$ , then

$$F(B, t) = f(\max \mathcal{SP}(B, t), t), \quad \Delta(B, t) = \delta(\max \mathcal{SP}(B, t), t).$$

We call  $F(B, t) + \Delta(B, t)$  the *estimated frequency* of  $B$  at time  $t$ .

Notice the estimated frequency  $F(B, t) + \Delta(B, t)$  is defined based on  $TS(t)$  of time  $t$ , while the counting frequency  $f(B, t) + \delta(B, t)$  depends just on  $TS(t - 1)$  of the previous time  $t - 1$ .

**THEOREM 3 ( $\epsilon$ -APPROXIMATION OF FREQUENCY).** *Let  $\mathcal{S}$  be a stream of length  $N$  and  $\epsilon$  be a maximal error ratio. For any itemset  $B$ , we have*

$$F(B, N) \leq \sup(B, N) \leq F(B, N) + \epsilon \cdot N$$

## 4. CONCLUSIONS

LC-CloStream can avoid a part of combinational explosion problems in a bursty transactional data stream [5]. In the future, we will study an efficient implementation using a sophisticated data structure, and also have a plan to investigate a more advanced framework where the frequency table has a fixed constant size [5].

## 5. ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number 25730133 and 25330256, and also supported by JST PRESTO (Sakigake).

## 6. REFERENCES

- [1] C. Borgelt, X. Yang, R. Nogaes-Cadenas, P. CarmonaSaez and A. Pascual-Montano: Finding Closed Frequent Item Sets by Intersecting Transactions. *Proc. EDBT 2011*, pp.367–376 (2011)
- [2] G. S Manku and R. Motwani: Approximate Frequency Counts over Data Streams. *VLDM'02*, pp.346–357 (2002)
- [3] T. Mielikäinen: Intersecting Data to Closed Sets with Constraints. *Proc. FIMI 2003*, CEUR WS. Proc. 90 (2003)
- [4] F. Pan, G. Cong, A. Tung, J. Yang and M. Zaki: Carpenter: Finding Closed Patterns in Long Biological Datasets. *ACM SIGKDD 2003*, pp.637–642 (2003)
- [5] Y. Yamamoto, K. Iwanuma, S. Fukuda: Resource-oriented Approximation for Frequent Itemset Mining from Bursty Data Streams. *ACM SIGMOD 2014*, pp.205–216 (2014).
- [6] S. Yen, C. Wu, Y. Lee, V.S. Tseng, C. Hsieh: A Fast Algorithm for Mining Frequent Closed Itemsets over Stream Sliding Window. *IEEE Int'l Conf. on Fuzzy Systems*, pp.27–30, Taipei (2011).

# Extending Database Accelerators for Data Transformations and Predictive Analytics

Knut Stolze  
IBM Germany Research &  
Development GmbH  
Schönaicher Straße 220  
71032 Böblingen, Germany  
stolze@de.ibm.com

Felix Beier  
IBM Germany Research &  
Development GmbH  
Schönaicher Straße 220  
71032 Böblingen, Germany  
febe@de.ibm.com

Daniel Martin  
IBM Germany Research &  
Development GmbH  
Schönaicher Straße 220  
71032 Böblingen, Germany  
danmartin@de.ibm.com

## ABSTRACT

The IBM DB2 Analytics Accelerator (IDAA) integrates the strong OLTP capabilities of DB2 for z/OS with very fast processing of OLAP workloads using Netezza technology. The accelerator is attached to DB2 as analytical processing resource – completely transparent for user applications. But all data modifications must be carried out by DB2 and are replicated to the accelerator internally. However, this behavior is not optimized for ELT processing and predictive analytics or data mining workloads where multi-staged data transformations are involved. We present our work for extending IDAA with accelerator-only tables, which enable direct data transformations without any necessary interventions by DB2. Further, we present a framework for executing arbitrary in-database analytics operations on the accelerator while ensuring data governance aspects like privilege management on DB2 and allowing to ingest data from any other source directly to the accelerator to enrich analytics e.g., with social media data. The evolutionary framework design maintains compatibility with existing infrastructure and applications, a must-have for the majority of customers, while allowing complex analytics beyond read-only reporting.

## Keywords

analytics, data mining, db2, mainframe, idaa

## 1. INTRODUCTION

The IBM DB2 Analytics Accelerator (IDAA) [1] is an extension for IBM's<sup>®</sup> DB2<sup>®</sup> for z/OS<sup>®</sup> database system. Its primary objective is the extremely fast execution of complex, analytical queries on a snapshot of the data copied from DB2. However, when it comes to more complex, multi-staged data analysis tasks like data mining, the accelerator can often provide limited improvements only. Predictive analytics tools like SPSS [4] resort to multiple SQL statements, each implementing a step or stage in a chain of data preparation, transformation, and evaluation tasks. For each stage,

base data needs to be transferred to IDAA before mining algorithms can be run and result data has to be materialized within DB2 before it can be used as input for the next stage or iteration. A key requirement for enhancing these workloads is to minimize data movement while still exploiting the accelerator for this task. We solve this issue with accelerator-only tables (AOTs) which are discussed in Sec. 2. The second use case is the application of the analytic algorithms in the pipeline. A generic framework is required which allows to pass code for arbitrary algorithms to IDAA while still implementing data governance aspects correctly. A seamless approach, completely transparent to user applications is discussed in Sec. 3.

## 2. ACCELERATOR-ONLY TABLES AND DATA INGESTION

Maintaining a copy of the DB2 data in the accelerator for query processing is the main use case of IDAA. However, this design involves a lot of data movements in case of multi-staged algorithms that require result materialization inside DB2 before the next step can be executed. The first building block in our current efforts are accelerator-only tables (AOTs), i.e., tables whose data solely resides inside IDAA (cf. Fig. 1), and DB2 only keeps a *proxy* or *table reference* which is usually named *nickname* in federation contexts [5]. This proxy is used for storing meta data in the DB2 catalog and acts as indicator for delegating any query on the corresponding AOT to IDAA. For creating AOTs the `CREATE TABLE` statement was extended by an additional `IN ACCELERATOR` clause. AOTs are populated with `INSERT` statements comprising a list of values or a sub-select which might invoke arbitrary transformation procedures (cf. Sec. 3), retrieving the data from other regular accelerated tables or AOTs. Likewise, `UPDATE`s and `DELETE`s are handled. The second way for populating AOTs is the new IDAA Loader [2] (cf. Fig. 1). The data to be loaded can originate from a variety of sources, even from applications not running on System z which opens up a wide range of new use cases. Data can be ingested in both, regular DB2 tables and AOTs. In the past, IDAA was not concerned about transactions because only the cursor stability isolation level was supported. Queries were executed under snapshot isolation in Netezza. With AOTs, IDAA has to be aware of the DB2 transaction context so that correct results are guaranteed, i.e., uncommitted data modifications of the own transaction are handled. At the same time, concurrent execution of multiple queries in a single transaction are also supported.

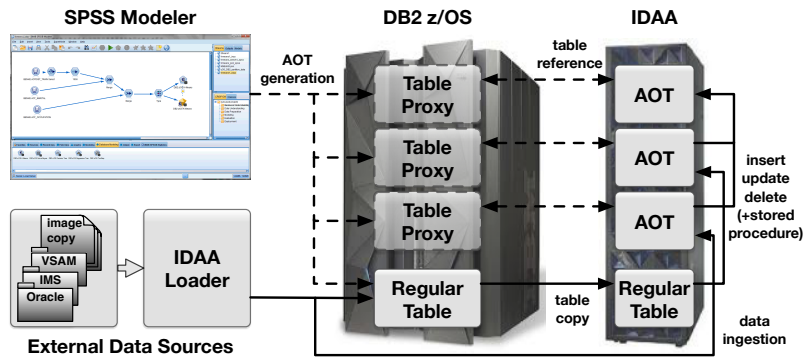


Figure 1: Overview IDAA with Accelerator-Only Tables

### 3. IN-ACCELERATOR ANALYTICS

IDAA has now a framework for invoking generic, customer-specific stored procedures (SPs). A SP can be called in DB2z, and IDAA forwards the procedure execution to the Netezza backend (cf. Fig. 2 and Fig. 3). We integrated the Netezza analytics package [3] which is used by SPSS [4]. The SPSS modeler provides means to define nodes, to create and populate AOTs, e.g., by joining accelerated tables, and then to invoke an analytics SP running in IDAA. Such a scenario is illustrated in Fig. 1 where the k-means clustering algorithm is applied on some filtered, enriched, and sampled input data. Intermediate results are materialized into AOTs and finally fed to the k-means SP. However, two challenges occur here. First, user privileges need to be verified for all data sources referenced by such a black box SP. Therefore, IDAA resolves all dependencies without executing the SP code. These table references are passed to DB2 which in turn validates necessary user privileges before the actual operation is carried out. The second issue arises from views existing on the DB2 side, which are not present on the accelerator. The framework exploits DB2's query acceleration capabilities to extract the view definition and implicitly translate it to the Netezza SQL dialect. A temporary view is created in the Netezza backend using that definition.

### 4. RELATED WORK

Using accelerator-only tables is similar to the concept of pass-through functionality in federated systems based on SQL/MED [5]. However, the overall use case for AOTs is quite different. AOTs are conceptually DB2 tables and can be manipulated using DB2's SQL dialect. It is just that AOTs store all their data in the analytics-optimized query engine and not in the transactional storage engine of DB2

for z/OS itself. Contrary to that, federated systems provide a means to access tables and data residing in another, stand-alone database system. The integration of the different query engines is on a much deeper level in IDAA.

MySQL provides an internal interface to plugin different storage engines with different characteristics [6]. IDAA in DB2 for z/OS implements a similar architecture by combining DB2's and Netezza's characteristics and the respective underlying storage mechanisms. However, the integration of both happens on the SQL dialect and SQL optimizer layer and not on the buffer pool and storage layers.

### 5. TRADEMARKS

IBM, DB2, and z/OS are trademarks of International Business Machines Corporation in USA and/or other countries. Other company, product or service names may be trademarks, or service marks of others. All trademarks are copyright of their respective owners.

### 6. REFERENCES

- [1] P. Bruni et al. *Reliability and Performance with IBM DB2 Analytics Accelerator V4.1*. IBM Redbooks, 2014.
- [2] IBM. *DB2 Analytics Accelerator Loader for z/OS*, 2014. <http://www-03.ibm.com/software/products/en/db2-analytics-accelerator-loader-for-zos>.
- [3] IBM. *IBM Netezza Analytics – In-Database Analytics Developer's Guide, Release 3.0.1*, 2014.
- [4] IBM. *SPSS Software*, 2014. <http://www.ibm.com/software/analytics/spss/>.
- [5] ISO/ IEC 9075-9:2003. *Information Technology – Database Languages – SQL – Part 9: Management of External Data (SQL/ MED)*, 2nd edition, 2003.
- [6] A. Lentz. *MySQL Storage Engine Architecture. MySQL Developer Articles, MySQL AB, May, 2004.*

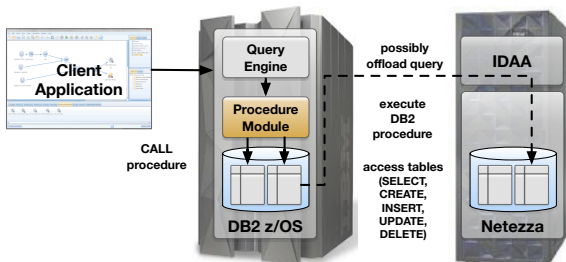


Figure 2: Stored Procedure Execution in DB2

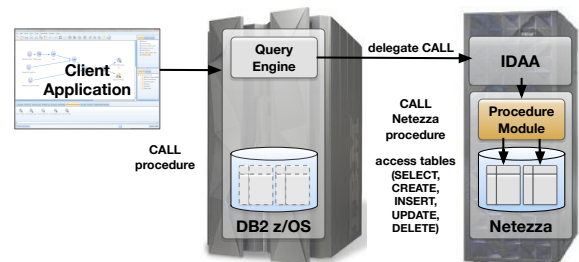


Figure 3: Stored Procedure Execution in Netezza

# Privacy Protection through Query Rewriting in Smart Environments \*

Hannes Grunert  
 Database Research Group  
 University of Rostock  
 18051 Rostock, Germany  
 hg(at)uni-rostock.de

Andreas Heuer  
 Database Research Group  
 University of Rostock  
 18051 Rostock, Germany  
 ah(at)uni-rostock.de

## ABSTRACT

By the events in the past years, the integration of data protection mechanisms into information systems becomes a central research problem again. In this poster, we show how query rewriting can be used to maintain privacy of users in smart (or assistive) environments. We developed a privacy respecting query processing and a vertical fragmentation of queries, processing maximal parts of the query as close to the sources of the data (e.g. sensors) as possible.

## Categories and Subject Descriptors

H.2.4 [Database Management]: Systems—*Query Processing*; K.4.1 [Computer and Society]: Public Policy Issues—*Privacy*

## General Terms

Privacy Enhancing Technologies, Database Systems

## 1. PRIVACY

Smart Metering, Internet surveillance, motion profiles, biometric databases, data retention: In the digital world steadily more and more information about ourselves and our environment is collected. Besides “classical” personal information, such as the name, age or gender, a plurality of sensors records our activities and inclinations. Active and passive RFID tags, cameras, microphones, but also sensors on light switches and power sockets capture the current situation in the ubiquitous environments, up to 100 times per second.

Especially smart environments such as assistive systems using activity and intention recognition [4] are a possible cause of privacy violations, especially if the query realizing the recognition analysis is performed on a cloud server.

To reduce privacy violations, it is necessary

\*A full version of this paper is available as a Technical Report at [www.ls-dbis.de/digbib/dbis-tr-cs-01-16.pdf](http://www.ls-dbis.de/digbib/dbis-tr-cs-01-16.pdf)

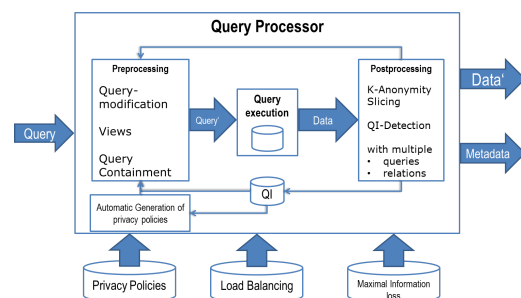


Figure 1: The concept of the privacy-aware query-processor.

- to decrease collected personal information, i.e. to apply the principle of data avoidance (except where data are required),
- to process data with personal references as less as possible or — at least — as close to the local data sources (sensors) as possible and
- to anonymize, pseudonymize and delete personal data, unless it is used for further processing and necessary to realize the aim of the assistive environment.

Data minimization and data avoidance are therefore prescribed, indispensable requirements for the design of smart systems. This requirement can be achieved in databases by transforming both queries and query results, as well as using views [3].

## 2. PRIVACY-AWARE QUERY PROCESSING

The PARADISE<sup>1</sup>-approach aims to withdraw the burden of respecting privacy constraints from the assistive systems by adding privacy protection mechanisms to those systems storing and analyzing the data: database systems on different levels. PARADISE combines performance aspects of big data analytics (by using massively parallel database technology [5]) and privacy protection. Our privacy-aware query-processor (see Figure 1) generates anonymized result sets. These data maintain a high degree of value for the initial query generated by the assistive system. On the opposite, additional knowledge can hardly be derived.

<sup>1</sup>Privacy Aware Assistive Distributed Information System Environment



The preprocessor allows the analysis and the rewriting of database queries regarding user-defined privacy policies [2]. During the execution of the request, it is decided whether the request will be answered and anonymized directly on the current network peer, or is sent to lower nodes (vertical fragmentation, see below). The postprocessor executes the anonymization of the query results, taking into account various criteria of quality and privacy. For this, several data protection metrics and algorithms are provided. The module for the automatic generation of privacy settings produces and adapts existing user-defined privacy policies to new devices and changing requirements and queries.

### 3. QUERY REWRITING BY VERTICAL FRAGMENTATION

The smart environment or assistive system sends a query request  $Q$  to the database  $d$  integrating the entire sensor data recorded in our environment. The result of  $Q$  is needed to perform the activity and intention recognition. The data sources are sensors being located in appliances in apartments and buildings. Instead of shipping  $d$  to the cloud server sending the request, maximal parts of  $Q$  will be evaluated as close to the sensor as possible. As can be seen in Figure 2, instead of performing  $Q(d)$  in the cloud, the maximal subquery  $Q_j$  will be shipped to the next lower node of the processing chain, in the case of the example a PC located in our apartment. While  $Q$  performs an iterative machine learning algorithm implemented in R and SQL, and  $Q_j$  being a complex SQL query with recursion, the lowest node in the processing chain (the sensors) can only compute some filter mechanisms (simple selections) and some simple aggregations over the last values generated (window function: average of last minute). Each of the nodes will ship the query result  $d_j$  to the node sending the request. After a final anonymization step  $A$ , the data “leaving our apartment”  $d'$  will only be a small subset of the original data  $d$ .

We assume that the lower nodes will each have less query computing power than the higher nodes: while sensors are only performing simple filters / selections and aggregations, an appliance like a TV or a smart network music player will be able to perform simple (SQLsuperlight) database queries, and an Android-based home media server even more complex SQL queries.

The whole query processing procedure

$$Q(d) := d_j = Q_j(\dots d'_i = A(d_i = Q_i(\dots (d_1 = Q_1(d)) \dots))$$

is transformed in a way, that the cloud server will perform a remainder query  $Q_\delta$  on  $d'$  instead of performing  $Q$  on  $d$ , hence resulting in the privacy protecting query rewriting  $Q(d) \rightarrow Q_\delta(d')$ . In other words, the query  $Q$  is fragmented in queries  $Q_j$  (that can be performed at a lower node) and a remainder query  $Q_\delta$  that can only be performed at the more powerful node of our vertical fragmentation of query processors.

Since recognizing the maximal SQL queries in an R machine learning algorithm is undecidable in general, we try to detect some larger “SQLable” patterns in the activity and intention recognition procedures described in [4]. Other aspects of our privacy-aware query-processor are described in, e.g. [1].

By rewriting the query  $Q$  into  $Q_j$  and  $Q_\delta$  and only performing  $Q_\delta$  outside our “privacy protected” appliance en-

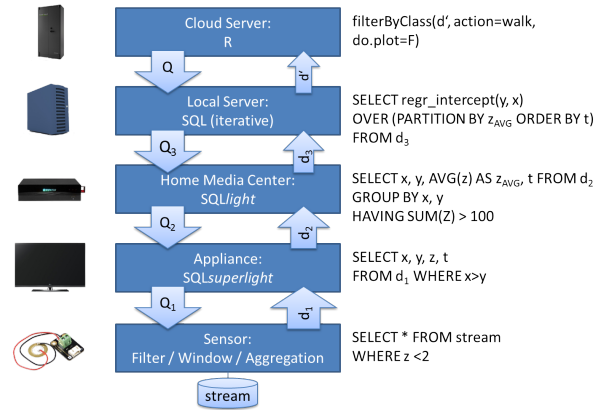


Figure 2: Vertical Query Fragmentation: Query and query result transformation on different peers.

semble in our apartment, we hope to automatically prevent the service provider of our assistive system to use our personal data in a way we did not consider to be possible when starting to use his smart service. A remaining open problem is to decide whether a privacy-violating query  $Q_\downarrow$  can be performed even on  $d'$  instead of  $d$ . In this case, we have to extend the anonymization step  $A$  already performed. This open problem results in a query containment problem of  $Q, Q_\downarrow$  and  $Q_j$  that will be part of our further research.

### 4. ACKNOWLEDGMENTS

Hannes Grunert is funded by the German Research Foundation (DFG), Graduate School 1424 (Multimodal Smart Appliance Ensembles for Mobile Applications - MuSAMA). The authors gratefully acknowledge the constructive comments of the anonymous referees.

### 5. REFERENCES

- [1] H. Grunert. Distributed denial of privacy. In *INFORMATIK 2014: Big Data Komplexität meistern*, pages 2299–2304. Springer, 2014.
- [2] H. Grunert and A. Heuer. Generating privacy constraints for assistive environments. In *Proceedings of the 8th International Conference on Pervasive Technologies Related to Assistive Environments, PETRA 2015*. ACM, 2015.
- [3] A. Heuer and A. Lubinski. Data reduction - an adaptation technique for mobile environments. In *Interactive Applications of Mobile Computing (IMC'98)*, 1998.
- [4] F. Krüger, M. Nyolt, K. Yordanova, A. Hein, and T. Kirste. Computational State Space Models for Activity and Intention Recognition. A Feasibility Study. *PLOS ONE*, Nov. 2014. 9(11): e109381. doi:10.1371/journal.pone.0109381.
- [5] D. Marten and A. Heuer. A framework for self-managing database support and parallel computing for assistive systems. In *Proceedings of the 8th International Conference on Pervasive Technologies Related to Assistive Environments, PETRA 2015*. ACM, 2015.

# DatShA :A Data Sharing Algebra for access control plans

Luc Bouganim  
Inria Saclay  
luc.bouganim@inria.fr

Athanasia Katsouraki  
Inria Saclay  
athanasia.katsouraki@inria.fr

Benjamin Nguyen  
INSA Centre Val de Loire  
benjamin.nguyen@insa-cvl.fr

## 1. INTRODUCTION

Online social networks (OSN) are one of the most successful applications that have been created this last decade. Central to these applications is the problem of sharing data, such as texts, photos, geolocation, etc. In most cases, this data is private, and thus is only shared with “friends”, a loose concept. Some OSN, such as Google+ let you define *circles* in order to categorize your friends: friends, close friends, acquaintances, etc. Data can then be shared on finer grain using these circles. However, there is no automatic way to control the simultaneous sharing of data to several circles, with different data precision granularities, such as in the following scenario: *Alice wants to share a set of photos with her family, photos with no metadata with her close friends, photos without faces (and without metadata) in a reduced definition with her acquaintances, and does not want to share anything with anyone else.*

In this article, we will show how the use of a data sharing algebra to write a variety of *access control plans* (ACP) can overcome these current limitations of OSN access control. Moreover, by using an algebra, it becomes simple to modify, compose, and share these ACPs. Thus less advanced users can easily reuse ACPs shared on a marketplace by more experienced users. A prototype of the DatShA system has been implemented using XQuery 3.0 and is briefly described.

## 2. OVERVIEW OF DatShA

In current OSNs, users have on one side vast quantities of personal data, and on the other side numerous “friends” with whom they wish to share (or sometimes hide) this data. In the current systems, it is not simple to share a specific piece of data while modifying it (e.g. changing its precision or removing some information) depending on the target with whom it is to be shared.

Consider the examples mentioned in the introduction. The ACP related to Alice’s close friends should transform a set of photos to another set where metadata is removed. This could be done by simply specifying a regular expression to identify images files to be shared (FileSearch operator – see Figure 1.e), “type” this file to images (PathToImage operator – see Figure 1.e), then remove metadata (RemoveMeta operator). For Alice’s acquaintances, other operators could be invoked: ExtractFaces, ExtractMeta, Select and ReduceDefinition operators (not detailed here).

Thus the objective of DatShA is to provide the infrastructure and an extensible set of generic operators to describe how users want to process their data before sharing it. The operators must be able to be combined on any sort of (semi-structured) data to form an algebra. Finally, ACP may include user-dependent data (e.g., contact files) such that it can also compute the set of users with whom the data is shared, thus linking a *plan* with its *grantee*.

Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 – Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

## 3. BACKGROUND AND RELATED WORKS

**Access Control.** Many different access control models exist, such as DAC, MAC, or RBAC. Many works exist on enforcing such models in OSN [1]. We adopt a complementary approach: the goal of DatShA can be seen as helping the user to write complex views of her data, on which she can then apply any existing AC model (most often, DAC or RBAC).

**Data Sharing on OSN.** Current works on secure data sharing in OSNs consider various problems such as securing communications, i.e. how to securely share data, once access control has been checked [2], or how to write access control policies over data concerning several users [3].

**XQuery 3.0.** XQuery 3.0. is not only a declarative query language, it is also Turing complete. Rather than using a traditional language such as Java or C, we have chosen to use XQuery and XQuery Update Facility 3.0. Indeed, evaluating an ACP is done through modifications to a structured document (that we chose to code in XML). Generic operators can be completed by snippets of XPath or XQuery code referring to this data structure, which are directly evaluated by the DatShA system.

## 4. THE DATA SHARING ALGEBRA

### 4.1 General principle

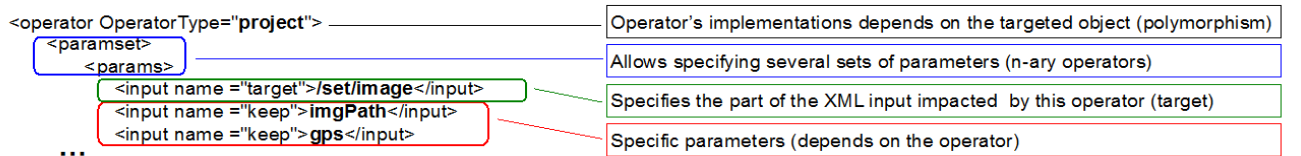
An ACP is seen as a set of sequences of (polymorphic) operators, serialized as an XML file (see Figure 1.a). It takes as input an XML file containing or referencing private sensitive data and produces an XML file containing or referencing data that can be shared or published (See Figure 1.c). Users or sets of users (such as G+ circles) can be given access rights both on atomic data, and on ACPs. As with traditional access control through views, when access rights are given on an ACP, the data accessed during the process is done with the rights of the *grantor*. For example, if Alice grants Bob the right to view the country she is in, which is computed using her precise GPS coordinates, the execution of the ACP will use Alice’s rights, but only return to Bob the final result.

### 4.2 Sharing ACPs through a marketplace

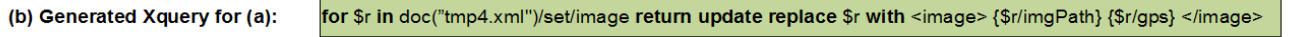
Operators and ACPs can be published on a “marketplace”, and described by a short text explaining their goal. They can be downloaded by users in order to fine tune their data sharing policies. Thus, it is possible, even for non-expert users to apply complex access control policies, by combining existing operators or using existing policies. Search, recommendation, or ranking of ACP or operators based on their level of intrusiveness or their usability is possible within the marketplace. The only complexity is to link groups of users to their ACPs, but as the data shared is defined *intentionally* rather than *extensionally*, we believe this is much easier to do than with current privacy settings in OSN.

### 4.3 ACP Example

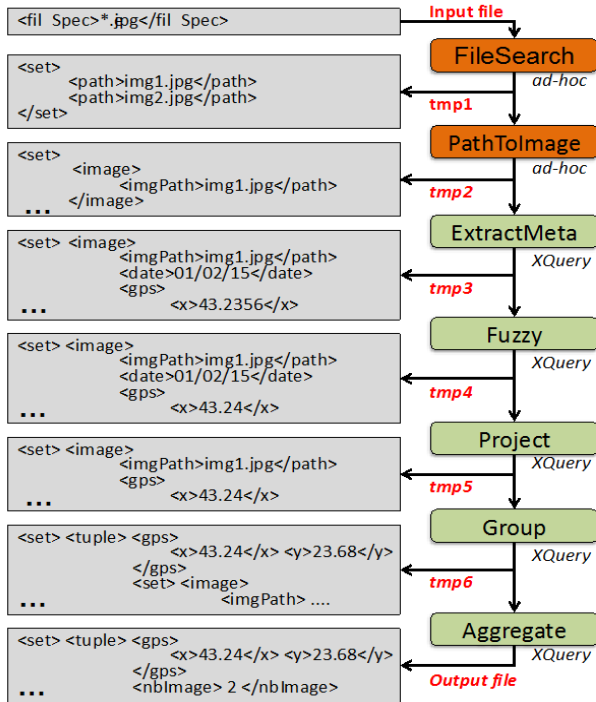
We propose the following example which illustrates well DatShA potential : *Alice wants to participate in a survey to determine the*



(a) Definition of the project operator in the XML ACP



(b) Generated XQuery for (a):



(c) XML input, temporary and result files

(d) ACP tree

**FileSearch:** replaces a `<fileSpec/>` with jokers in a set of file paths looking recursively or not (input "mode") in the directory indicated by input "target" every `<fileSpec/>` should be replaced by a set of path.

**PathToImage:** is an operator that replaces every occurrence of a path by an image (an xsd type). An image is at least a `<imgPath/>` to an "image" file, i.e., a jpg, png, gif, etc.... Initially the image type only includes the `<imgPath/>` but metadata can be added using the ExtractMeta operator.

**ExtractMeta:** replaces every occurrence of an image by the same image (every field is copied), and adds metadata that can be extracted from the actual file (e.g. location information embedded in the image).

**Fuzzy:** is an operator that can be applied to many types. The global behavior is to replace any occurrence of the target by fuzzy values, the precision being informed by the "precision" input, which can be an XPath.

**Project:** this operator is used like the relational algebra  $\Pi$  operator. It replaces the target subtree by the same subtree in which it keeps only the elements (or subtrees) that are mentioned in the "keep" parameters.

**Group:** replaces the "target" subtree by a restructured one which must be a set. It constructs a `<set>` of `<tuple>`s, each containing  $n+1$  elements (where  $n$  is the number of "groupBy" elements in the operator specification, in this example,  $n = 1$ ). The last element of the tuple is a set of elements that share the same value of groupBy (here a set of image having the same GPS value). This operator is implemented by XQuery 3.0. Group By.

**Aggregate :** The aggregate operator replaces a set of elements ("target" input) by an aggregate value having the "AggName" name and applying the "AggOperation", which in this case is the XQuery function `fn:count()`.

(e) Operators details

Figure 1: A detailed example of an ACP delivering statistics (number of photos by fuzzy location)

most photographed place on Earth, which can be done by computing a "fuzzy" location of all her photos, where the "fuzzy" location is defined by GPS coordinate and an error bar e.g.  $X=45.23\pm 0.01$   $Y=27.67\pm 0.01$ . Note that this error bar could also be function of the density of photos in a given area.

We present, in Figure 1, the corresponding ACP.

This ACP can be written as a sequence of operators (Due to space limitations, the actual XML ACP file is available online at: <http://www.benjamin-nguyen.fr/data/ACP.xml>). Each operator takes as input an XML file, and produces as output an XML file. It is possible to type-check the ACP at compile time, given that the operators are typed, but this discussion is beyond our scope here. The sequence is the following: for every image in the file path given in the input file, meta-data of the image is extracted, and an operator to reduce the precision of the GPS coordinates is executed. All meta-data apart from the blurred GPS coordinates is removed, pictures are grouped together by fuzzy GPS location, then counted.

Operators can in most cases be implemented in XQuery 3.0 but they can also be *ad-hoc* operators. Note that DatShA also defines many other operators, including binary (or even n-ary) operators such as the join operator which can be used to join two different sequences, thus needing two input files, and producing a single output file. All these operators have been implemented using XQuery. The framework executing a DatShA ACP has been written in Java, using eXistDB to execute the XQuery fragments.

We believe that the use of operators to build ACPs drastically simplifies the creation, reuse, combination and correctness checking of ACPs.

## 5. CONCLUSION AND FUTURE WORK

DatShA can be used to create ACPs to manage access control to one's data. We believe that the full power of DatShA appears when users start sharing ACPs between each other, either by simply reusing an ACP written by another user, or by integrating such an ACP into a more complex one. Indeed, any ACP can be encapsulated as a DatShA operator. Creating an online marketplace, and testing its usability and adoptability by real users is the next step of our work.

## 6. REFERENCES

- [1] B. Carminati, E., and A. Perego. 2009. Enforcing access control in Web-based social networks. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article 6 (November 2009).
- [2] H. Qinlong, M. Zhaofeng, Y. Yixian, N. Xinxin, F. Jingyi, Improving security and efficiency for encrypted data sharing in online social networks in *IEEE China Communications*, 11(3):104-117, 2014.
- [3] H. Hu, G-J. Ahn, J. Jorgensen: Multiparty Access Control for Online Social Networks: Model and Mechanisms. *IEEE Trans. Knowl. Data Eng.* 25(7): 1614-1627 (2013).

# Cluster-based Contextual Recommendations

Kostas Stefanidis  
ICS-FORTH, Greece  
kstef@ics.forth.gr

Eirini Ntoutsi\*  
Leibniz Universität  
Hannover, Germany  
ntoutsi@kbs.uni-hannover.de

## ABSTRACT

In this work, we address the problem of contextual recommendations by exploiting the concept of subspace clustering. Specifically, we pre-partition users that have rated subsets of data items similarly into clusters and we associate a context situation with each cluster. The cluster context is defined as any internally stored information that can be used to characterize the cluster members per se. Then, given a query context, we identify the clusters with the most similar context, and we use their members for making suggestions in a collaborative filtering manner.

## 1. DESCRIPTION

Recommender systems have become indispensable for several Web sites, such as Amazon, Netflix and Google News, helping users to navigate through the infinite number of available choices. Motivated by the fact that often users have different preferences under different context situations, several approaches, e.g., [1], extend recommender systems beyond the two dimensions of users and items to include further contextual information. Context can be defined as any *external* to the database information that can be used to characterize the situation of a user, such as the location, time or companion of the user, or any internally stored information that can be used to characterize the data per se [6]. In our work, we follow an internal contextualization approach, and infer context from the data itself. A simple way to express an internal context is by specifying conditions for the presence of particular attribute values in the data. For example, for a movies recommender, an internal context can be: *genre=comedy & production-year=2015*. It is clear that such a context characterization cannot be done upon the whole database, as the data display a lot of variability. Rather, we should look for contextual information in smaller, homogeneous subgroups of the data.

\*Work done while with the Ludwig-Maximilians University, Munich.

To extract contextual information, we rely on similarities on the user ratings. Intuitively, users close together in their ratings, share the same context, let it be the preference for similar movie genres, or preferences towards specific directors or actors. Typically the user similarity is evaluated w.r.t. the full dimensional feature space, i.e., all available items. Finding similar users for all different items though is hard, while it is more reasonable to find users similar w.r.t. a subset of the items. A straightforward approach to derive such subsets is to categorize the items based on some domain knowledge. In case of movies, for example, the movie genre can be used and items that belong to the same genre can form a subset. A problem with this approach is that such categories are quite vaguely defined, diverse and also overlapping. For instance, the movies *Ted* and *My big fat Greek wedding* are both classified under *comedy*, the later however can be also found under *romance*. For a user interested in comedies it is not clear whether she would equally appreciate a suggestion on *Ted* and *My big fat Greek wedding*. Such a general item categorization, does not reveal much about the aspects that bring users together. Moreover, such aspects might be beyond some given categorization, like the movie genre and also, they might involve more than one dimension, e.g., movie genre and director. Ideally, we want to find subsets of items which are rated similarly by some users; such a subset implies that these items have something in common which brings these users together. This does not need to be that generic as the genre, but it might be some other common property of the items, like the director, the story, or even a mixture of them.

In [4], we locate such user-item groups by exploiting (fault-tolerant) subspace clustering. Subspace clustering is a popular approach for clustering high dimensional data which discovers, except for the cluster members, the dimensions upon which these members form a cluster. Different subspace clusters might be defined upon different subspaces and member and subspace overlap among the different clusters is allowed. In our case, subspace clustering identifies groups of users with similar behavior w.r.t. a set of items. We employ the items of a subspace cluster to build its context and use it to locate, at query time, clusters with context similar to the query context. In contrast to our prior work [4] that considers all user-related clusters for recommendations, here we define the notion of cluster context and we consider only context-related clusters for the specific user.

**Recommendations Basics:** Assume a recommender system, where  $I$  is the set of items and  $\mathcal{U}$  is the set of users. Each item  $i \in I$  is described as a set of (attribute, value)

pairs; let  $\mathcal{D}$  be the set of all distinct (attribute, value) pairs appearing in all data items. For instance, for a movies application, an attribute can be the director or the production year of a movie. A user  $u$  might rate an item  $i$  with a score  $rating(u, i)$  in  $[0.0, 1.0]$ ; let  $R$  be the set of all ratings recorded in the system. Typically, the cardinality of  $I$  is high and users rate only a few items. For an  $i$ , unrated by  $u$ , with  $N_u$  representing  $u$ 's most similar users (neighbors), its relevance score is computed as:

$$relevance(u, i) = \frac{\sum_{u' \in N_u} simU(u, u') rating(u', i)}{\sum_{u' \in N_u} simU(u, u')} \quad (1)$$

where the similarity function  $simU(u, u')$  evaluates the proximity between  $u$  and  $u'$ . The most prominent items, i.e., those with the higher relevance, are suggested to the user.

**Fault-tolerant Subspace Clustering:** Subspace clustering aims at detecting clusters embedded in subspaces of a high dimensional dataset. Clusters may consist of different combinations of dimensions, while the number of relevant dimensions per cluster may vary strongly. A *subspace*  $S$  describes a subset of items,  $S \subseteq I$ . A subspace cluster  $C$  is then described in terms of both its members  $U \subseteq \mathcal{U}$  and the subspace of dimensions  $S \subseteq I$  upon which it is defined as  $C = (U, S)$ . Typically, subspace clustering does not deal with missing values, which is a key problem for recommendations. Fault tolerant subspace clustering [3] deals with this issue by allowing a certain amount of missing values per items, users and ratings in a subspace cluster.

In [4], we use fault tolerant subspace clustering to locate users with similar preferences to a query user, for computing her recommendations. In particular, for a query user  $u$  we locate its similar users via the subspace clusters where the user belongs to. These are locally similar users, the term “locally” meaning that they are similar w.r.t. a set of dimensions (those in their corresponding subcluster). We refine this set of users based on their common ratings to  $u$ ; this is a “global” evaluation aiming to check their overall proximity, i.e., over all items. This local-global refinement results in a more qualitative set of friends  $N_u$  for recommendations. The new set  $N_u$  is plugged in Formula 1 for issuing recommendations. Our results show that this careful selection of friends, is reflected in more qualitative recommendations.

**Inferring the Cluster Context:** We consider that the context of a subspace cluster  $C = (U, S)$  expresses the most significant parts of the items  $S$  within the cluster; these are captured through the attribute values of the items of  $S$ , upon which  $C$  is defined and are therefore sets of (attribute, value) pairs. Similar to [5], we ground the significance of each (attribute, value) pair on its frequency in the data appearing in the cluster. By post-processing the (attribute,value) pairs in  $S$ , we rank these pairs based on their frequency in  $C$ ; the significance of a pair is normalized taking into account its frequency in the whole database, so as to downgrade global popular pairs corresponding to common trends and focus on cluster-specific context. This way, we define the context of a cluster as an expression containing one or more significant (attribute,value) pairs. For instance, the context of a movie cluster could be: *genre=comedy & actor=Meryl Streep*.

Luckily, our subspace clustering is offline and therefore there is no need to compute at query time the context of the produced clusters. This fact allows us to resorting to non-approximate solutions for context identification.

**Contextual Recommendations:** Given a user  $u$  along with a query with context  $p$ , expressed as a set of (attribute, value) pairs with attributes in  $\mathcal{D}$ , for computing contextual recommendations for  $u$ , we first locate the users that exhibit the most similar behavior to  $u$  under  $p$ . These are the members of the clusters for which  $u$  is also a member; we denote them by  $C_u$ . Due to the context-constraints though, not all clusters are relevant as some of them describe a different context than  $p$ . Therefore, we need a way to evaluate the relevance of a cluster context to  $p$ . We distinguish between:

- *Exact context match:* If there are clusters in  $C_u$  that match exactly the query context  $p$  of  $u$ , i.e.,  $C_u^p$ , then the members of these clusters comprise the set of friends  $N_u$  upon which the recommendations for  $u$  will be computed.

- *Partial context match:* If there is no cluster with context equal to  $p$ , we relax our context relevance evaluation by looking for context-similar clusters, instead of context-identical clusters. To determine how close a context query  $p$  and a cluster context  $c$  are, we rely on a vector-based approach. Let  $\mathcal{D}$  be the set of all  $N$  distinct (attribute, value) pairs appearing in all data items. A vector representation of  $p$  is a binary vector  $V_p$  of size  $N$ , whose  $j$ -th element corresponds to  $\mathcal{D}[j]$ . If  $\mathcal{D}[j]$  appears in  $p$ , then  $V_p[j] = 1$ ; otherwise it is 0. Analogously, the vector representation of a cluster context  $w$  is a binary vector  $V_w$  of size  $N$ , where  $V_w[j] = 1$ , if  $\mathcal{D}[j]$  appears in  $w$ ; otherwise it is 0. The similarity between  $p$  and  $w$  is then defined using their vector representations  $V_p$  and  $V_w$  as:

$$sim(p, w) = \cos(V_p, V_w) = \frac{V_p \cdot V_w}{|V_p| |V_w|} \quad (2)$$

Having located the clusters  $C_u^p$  with the most similar contexts to  $p$ , we employ their members as the set of the most similar users to  $u$  and compute recommendations based on them. Actually, we apply a weighted ranking approach to refine the set of like-mined users according to the similarity of the context of the cluster they belong to, to  $p$ .

**Next Steps:** We are working on improving our cluster context description, by a better aggregation of the attribute values within the cluster and by using item hierarchies, and on more sophisticated methods for context matching and user aggregation. Also, we are working on the scalability aspect to parallelize the subspace cluster and context extraction parts. Preliminary results with MapReduce appear in [2].

## 2. REFERENCES

- [1] G. Adomavicius, R. Sankaranarayanan, S. Sen, and A. Tuzhilin. Incorporating contextual information in recommender systems using a multidimensional approach. *ACM Trans. Inf. Syst.*, 23(1):103–145, 2005.
- [2] V. Efthymiou, K. Stefanidis, and E. Ntoutsis. Top-k computations in MapReduce: A case study on recommendations. In *IEEE Big Data*, 2015.
- [3] S. Günemann, E. Müller, S. Raubach, and T. Seidl. Flexible fault tolerant subspace clustering for data with missing values. In *ICDM*, pages 231–240, 2011.
- [4] E. Ntoutsis, K. Stefanidis, K. Rausch, and H. Kriegel. Strength lies in differences: Diversifying friends for recommendations through subspace clustering. In *CIKM*, 2014.
- [5] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *SIGMOD*, 1996.
- [6] K. Stefanidis, G. Koutrika, and E. Pitoura. A survey on representation, composition and application of preferences in database systems. *ACM Trans. Database Syst.*, 36(3):19, 2011.

# Empirical evaluation of guarded structural indexing

Erik Agterdenbos  
 George H. L. Fletcher  
 Eindhoven University of  
 Technology  
 agterdenbos@gmail.com,  
 g.h.l.fletcher@tue.nl

Chee-Yong Chan  
 National University of  
 Singapore  
 chancy@comp.nus.edu.sg

Stijn Vansummeren  
 Université Libre de Bruxelles  
 svsummer@ulb.ac.be

## ABSTRACT

Traditional indices in relational databases are designed for queries that are *selective by value*. However, queries can also retrieve records on their *relational structure*. In our research, we found that traditional indices are ineffective for structurally selective queries. To accelerate such queries, so-called ‘structural indices’ have been applied in graph databases. These indices group together structurally similar nodes to obtain a compact representation of the graph structure.

We studied how structural indices can be applied in relational databases and evaluated their performance. Guarded bisimulation groups together relational tuples with similar structure, which we use to obtain a guarded structural index. Our solution requires significantly less space than traditional indices. At the same time, it can offer several orders of magnitude faster query evaluation performance.

## 1. PROBLEM DEFINITION

Queries that are selective by value can be accelerated by using B-trees or Hash indices on attributes in the selection condition. The problem is that such indices cannot efficiently answer structurally selective queries. Consider a relation  $customer(id, name, address, phone)$  and a relation  $order(id, status, total\_price, customer\_id)$ , where  $id$  denotes the primary key (PK) in both relations. Further, we have a foreign key (FK) from  $order(customer\_id)$  to  $customer(id)$ .

For example, we might want to retrieve all names of customers that do or do not have an order. To answer these queries, a semijoin or antijoin has to be processed. Semijoins and antijoins require table scans, index scans or index only scans on both relations. These operations are relatively expensive. Moreover, these are *tuple selecting queries*. The result is a subset of a *single* relation: the *customer* relation. No information from the *order* relation occurs in the output. However, the *order* relation or its index must be scanned to determine which tuples must be returned, which impacts the performance when the *order* relation is much larger than the *customer* relation.

Value-based indices are not directly useful in accelerating structurally-sensitive selections such as these.

## 2. BACKGROUND

The main concept of structural indices is to build a summary that is smaller than the original database, while preserving relevant structural properties. Bisimilarity and similarity are two formal notions of structural similarity in graphs which have been used for summarizing semi-structured and graph databases for structural indexing [3, 7]. Andréka et al. [2] and Otto [5] introduced guarded bisimulation and guarded simulation which extends these notions to relational databases. Picalausa et al. characterized query invariance under guarded (bi)simulation (i.e., for which query languages guarded (bi)simulation is the correct structural notion for indexing w.r.t. queries in the language), providing a formal basis for the engineering of guarded structural indices [6]. This abstract summarizes the main results of our empirical study of the practical feasibility of this novel approach to indexing; details can be found in [1].

## 3. APPROACH

Our approach to build a structural index can be summarized as follows: first we represent the relational database instance as a graph. Second, we apply an external memory bisimulation partitioning algorithm to group similar nodes. Third, we map the partitioning of the original tuples. We summarize each step in the following paragraphs.

Relational databases allow joins on any set of attribute pairs with equal types. Because FK constraints are popular candidates for join conditions, we only consider PK-FK joins. The graph is constructed as follows: we create a node for each PK value in the database. Then we create forward edges from PK to FK values and backward edges from FK to PK values. We use relation names as node labels and FK constraint names for edge labels.

We apply a localized version of bisimulation equivalence on the graph, namely,  $k$ -bisimulation. This equivalence relation induces a partitioning of nodes with respect to topological features of their  $k$ -neighborhoods. The  $k$ -neighborhood of a node  $n$  is the subgraph consisting of all nodes at most  $k$  edges away from  $n$ . The partitioning result consists of a mapping from nodes (tuples) to partition blocks, which represent  $k$ -bisimulation equivalence classes, and a reduced graph that summarizes the original structure. The mapping is stored by tagging each tuple in each relation (in an additional attribute) with the distinct identifier of the partition block to which the tuple belongs. The reduced

**Table 1: Queries for TPC-H dataset**

Query	description
FACQ 1	Select partsupp supplycosts having lineitem
FACQ 2	Select part names having a lineitem
FACQ 4	Select region having nation, customer, orders and lineitem
GF 1A	Select partsupp comment without lineitem
GF 1B	Select customers not having an order
GF 4	Select all suppliers that sell parts that are offered but not sold by other suppliers

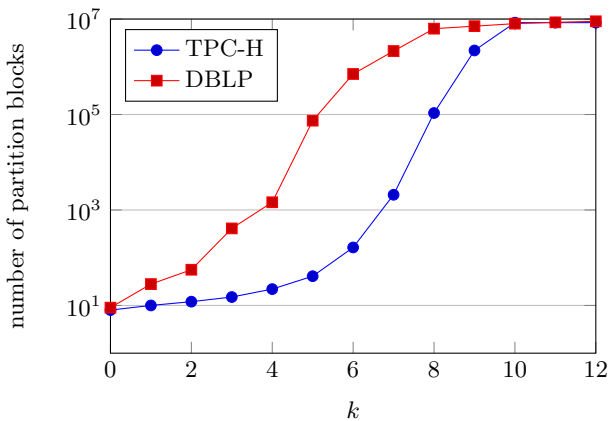
graph is stored in an additional relation  $edge\_label(from\_id, to\_id)$ . After construction of  $edge\_label$ , the PK-FK graph is no longer needed and is discarded.

$k$ -bisimulation structural indices allow the acceleration of semijoins and antijoins with join trees of height  $h \leq k$ . The reduced graph is used to determine which equivalence classes are selected. Then, the projected relation is scanned and tuples that are tagged with those equivalence classes are returned. Queries with join trees of height  $h > k$  can be partially accelerated via query decomposition.

## 4. EXPERIMENTAL STUDY

*Set up.* We used the DBLP<sup>1</sup> data set and the TPC-H<sup>2</sup> data set with scale factor 1 to evaluate guarded structural indexing. PostgreSQL 9.3 and the external memory bisimulation partitioning solution of Luo et al. [4] were used. Table 1 lists the queries used in our evaluation.

*Results.* Figure 1 shows the number of partition blocks that are generated under  $k$ -bisimulation. A higher value of  $k$  leads to more equivalence classes, uses more disk space, and can accelerate higher join (sub)trees. Figure 2 shows the reduced graph under 2-bisimulation. We observe significant compression while preserving non-trivial structure.

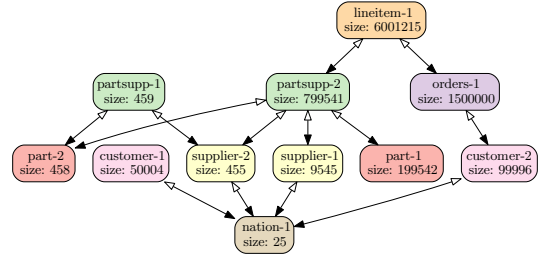


**Figure 1: Partition size under  $k$ -bisimulation**

Figure 3 shows the speed-up of structural indexing over value-based B-trees on foreign keys. For  $4 \leq k \leq 7$ , this is between 4 and 190 times faster. We also observed that our index uses only 1 megabyte of disk space for these  $k$

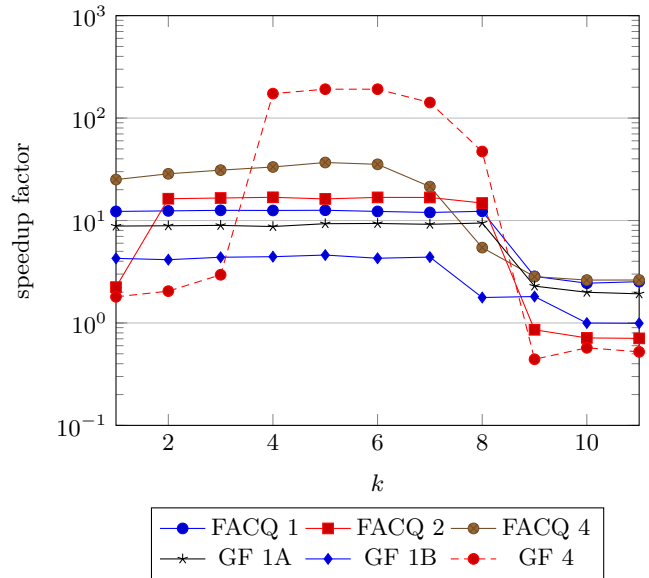
<sup>1</sup><http://dblp.uni-trier.de/xml/>

<sup>2</sup><http://www.tpc.org/tpch/>



**Figure 2: TPC-H partitioning under 2-bisimulation**

values compared to the more than 450 megabytes required for value-based indexes.



**Figure 3: Query running time speedup factor**

*Conclusions.* Our results show that guarded structural indices can be orders of magnitude smaller and faster than traditional indexes. This indicates the significant promise of further study of this new approach to indexing.

## 5. REFERENCES

- [1] E. Agterdenbos. Structural indexing for accelerated join-processing in relational databases. Master's thesis, Eindhoven University of Technology, 2015.
- [2] H. Andr eka, I. N emeti, and J. van Benthem. Modal languages and bounded fragments of predicate logic. *J. Phil. Logic*, 27(3):217–274, 1998.
- [3] Y. Luo et al. Storing and indexing massive RDF datasets. In R. De Virgilio et al, editor, *Semantic Search over the Web*, pages 31–60. Springer, 2012.
- [4] Y. Luo et al. External memory  $k$ -bisimulation reduction of big graphs. In *CIKM*, pages 919–928, 2013.
- [5] M. Otto. Highly acyclic groups, hypergraph covers, and the guarded fragment. *J. ACM*, 59(1), 2012.
- [6] F. Picalausa et al. Principles of guarded structural indexing. In *ICDT*, pages 245–256, 2014.
- [7] D. Sangiorgi and J. Rutten. *Advanced topics in bisimulation and coinduction*. C. U. Press, 2011.

# Context-Dependent Quality-Aware Source Selection for Live Queries on Linked Data

<p>Barbara Catania DIBRIS University of Genova Genova, Italy barbara.catania@unige.it</p>	<p>Giovanna Guerrini DIBRIS University of Genova Genova, Italy giovanna.guerrini@unige.it</p>	<p>Beyza Yaman DIBRIS University of Genova Genova, Italy beyza.yaman@dibris.unige.it</p>
---	---	--

## ABSTRACT

Source selection deserves attention for live query processing over distributed, poorly controlled data sources since it is the key to produce the best available information, in terms of relevance, trustness, and freshness, as query result. In this paper, we present an approach taking into account context-dependent data quality, according to different dimensions, during source selection, with the aim of selecting not only the most relevant but also the highest quality sources.

## 1. INTRODUCTION & BACKGROUND

In the last decade significant amount of Linked Data has been published to construct a global data space. However, it is still difficult to benefit from published data in an effective way because identifying the sources containing the most valuable results for a query is a non-trivial task. We propose an approach to select sources taking into account not only relevance but also quality in a context-dependent way.

Our approach fits in the general vision of Rekatsinas *et al.* [2] for a data source management system that enables users to discover the most valuable data sources for their applications. To characterize data source value, different *quality indicators* [1] can be used to assess the different quality dimensions (accuracy, freshness, completeness, etc), relying on data content, metadata (such as update times) and explicit user feedbacks. Data quality, however, may be different if assessed with reference to a geographical area, historical period, or type of content. As a result, it is not only impossible to assess quality in an absolute way, but it is difficult as well to assess a single quality dimension independently from the *context*. Quality indicators would then be associated with data according to the different contexts.

We adopt the notion of context proposed in [2] in terms of context clusters. A context cluster describes the data domain corresponding to a data collection and it is specified as a conjunction of a set of concept classes and a set of instances. In our approach, context clusters can specify information about *what* (the type of the described information), *where*

(spatial location), *when* (temporal location), and *why* (data motivations).

Each data item (i.e., each RDF triple) is associated with (one or more) context clusters. A source can include data items from different contexts. A (data source, context) pair intensionally characterizes a data collection, consisting of the set of triples in the source associated with the context. Quality indicators are associated, for the relevant quality dimension, with such data collections. A data source can thus exhibit different quality degrees, resulting in different indicator values, according to the different contexts.

Given a user query, the approach consists of four tasks: (i) context-dependent quality aware source selection to devise the most relevant and highest quality sources according to the query and its context; (ii) feedback from the user on the results obtained by the query evaluation; (iii) refinement/update of the data quality indicators according to such feedback; (iv) update of the auxiliary structures employed for source selection according to the refinement in (iii).

The proposed approach relies on two main notions, that are combined in an original way:

1. *Named Graphs*. Named Graphs [1] are useful structures for hierarchically composing subgraphs and building nested graphs. They allow to represent and exchange metadata.

2. *Data Summaries*. Data summaries [3] have been proposed to efficiently determine which sources may contribute answers to a query in live distributed query systems. They approximately describe the data provided by a source in an aggregated form, in much more detail than schema-level indexes. The QTree, specifically, is a data summary over Linked Data sources, seeing the data items (RDF triples) in the sources as points of a three dimensional numerical space, by applying hash functions to the triple components. Like the R-tree, a QTree is a tree structure consisting of nodes defined by minimal bounding boxes (MBBs). An MBB describes the multidimensional region in the data space that is represented by the node the subtree underneath. Leaf nodes in a QTree, however, rather than containing the data items that are contained in their MBBs as in R-trees, are buckets containing statistical information (e.g., count) that approximate the data items contained in their MBBs.

## 2. SOURCE SELECTION EXPLOITING CONTEXT-DEPENDENT QUALITY

Figure 1 provides a graphical overview of the approach. We first briefly describe the steps in the source selection process and then discuss its main components. First, the



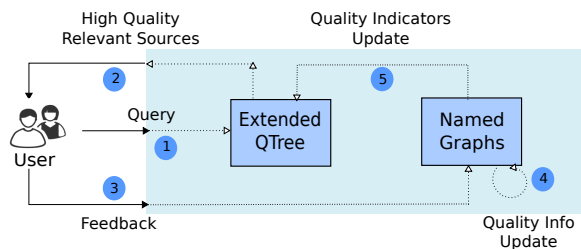


Figure 1: Overview of the Approach

user query is submitted to the system (Edge 1) and a look-up is performed on an extended QTree, returning to the user a ranked list of high quality relevant sources (Edge 2). The extended QTree allows the retrieval of potentially relevant sources with quality indicators, for given quality dimensions in a given context, above a given threshold. Once query results are returned to the user, her feedback (if any, Edge 3) on the obtained results is considered, resulting in an update and refinement of the quality metadata associated with the sources according to the context (Edge 4). Such context-dependent metadata are maintained making use of Named Graphs. Named Graphs maintain detailed quality indicators and metadata, so that this knowledge is shared and easily accessible. Such detail quality information is the basis on which the numeric indicators in the extended QTree are computed. Finally, context-dependent numerical quality indicators are incrementally updated in the extended QTree according to the new metadata (Edge 5).

**Query.** The query is a conjunctive SPARQL query associated with context information and quality thresholds. Conjunctive SPARQL queries consist of so-called basic graph patterns (BGPs), i.e., sets of triple patterns in the *(subject, predicate, object)* form possibly containing variables. We assume that the BGPs come with context information in the form of context clusters. Moreover, thresholds can be specified with respect to one or several among the supported quality dimensions. Only sources associated with indicators with values above the thresholds for the specified dimensions will be considered as result, while the other ones are discarded.

**Named Graph.** We exploit Named Graphs to associate quality related metadata and indicators with data sources, in a context-dependent way, in order to make this knowledge shared and easily accessible. Context clusters are organized in a hierarchical structure enabling the support of different detail levels. For instance, referring to spatial location context clusters, the *{Rome}* cluster is more specific than the *{Central Italy}* cluster. A data source may contain data items from various contexts. For instance, a data source containing information about touristic attractions in central Italy may contain data items associated with the *{Museum}* (*what*) cluster as well as items associated with the *{Rome}* (*where*) cluster as well as items associated with the *{Florence}* (*where*) cluster.

A data source can be associated in the Named Graphs with different quality profiles corresponding to the different contexts of its triples. Inside Named Graphs, indeed, quality-related metadata w.r.t. different quality dimensions

are attached to each data collections characterized by data source and context. This allows a context-dependent quality assessment. Referring to the above mentioned data source, its freshness may be different according to the *{Rome}* or to the *{Florence}* context, and, similarly its trustness may be different according to the *{Museum}* or *{Rome}* context. We rely on metadata standard vocabularies to describe quality-related metadata [1]. For instance, freshness can be described by using access (*dc:date*) and creation dates (*dc:created*), while trustness can be related to the creator of the source (*dc:publisher*).

**Extended QTree.** The QTree index is extended to consider context as a fourth component associated with triples and to associate context-dependent numerical quality indicators with data collections. Contextualized data items are now seen as point of a four dimensional numerical space, since context clusters are hashed to a numerical value as the other triple components. Each four dimensional MBB, moreover, is now associated with a quality range for each of the quality dimension. Indicators are separately computed for each quality dimension from the quality information (metadata and indicators) in the Named Graphs. The approach is flexible, in that the employed indicators may be different from source to source.

Indicator ranges are used during source selection to prune the data sources with quality below the requested thresholds. Specifically, given a four dimensional region and the thresholds for quality dimensions resulting from a query, the extended QTree is employed by descending in a child node if the region is contained in the MBB and the quality ranges intersect the corresponding thresholds. Indicators are finally also employed to rank the retrieved data sources according to their quality w.r.t. the query context.

### 3. CONCLUSION

In this paper we propose an approach to select relevant sources from an arbitrary, unrestricted set of distributed, poorly controlled Linked Data sources, so that queries can be processed on these sources taking into account not only their relevance to the query but also their quality, in terms of a number of dimensions, with respect to the query context.

Technically, the proposed approach relies on the use of nested Named Graphs to associate quality metadata with data source according to different contexts and at different granularity levels, and on extended QTree enabling efficient source selection, not only relying on relevance, but also on context-based quality indicators.

### 4. REFERENCES

- [1] C. Bizer. *Quality Driven Information Filtering: In the Context of Web Based Information Systems*. VDM Publishing, 2007.
- [2] T. Rekatsinas, X. L. Dong, L. Getoor, and D. Srivastava. Finding quality in quantity: The challenge of discovering valuable sources for integration. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2015.
- [3] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres. Comparing Data Summaries for Processing Live Queries over Linked Data. *World Wide Web*, 14(5-6):495–544, 2011.

# Data, Responsibly: Fairness, Neutrality and Transparency in Data Analysis

Julia Stoyanovich  
Drexel University  
stoyanovich@drexel.edu

Serge Abiteboul  
INRIA Saclay & ENS Cachan  
serge.abiteboul@inria.fr

Gerome Miklau  
UMass Amherst  
miklau@cs.umass.edu

## ABSTRACT

Big data technology holds incredible promise of improving people's lives, accelerating scientific discovery and innovation, and bringing about positive societal change. Yet, if not used responsibly, this technology can propel economic inequality, destabilize global markets and affirm systemic bias. While the potential benefits of big data are well-accepted, the importance of using these techniques in a fair and transparent manner is rarely considered.

The primary goal of this tutorial is to draw the attention of the data management community to the important emerging subject of responsible data management and analysis. We will offer our perspective on the issue, will give an overview of existing technical work, primarily from the data mining and algorithms communities, and will motivate future research directions.

## 1. RESPONSIBLE DATA ANALYSIS

Big data technology holds incredible promise of improving people's lives, accelerating scientific discovery and innovation, and bringing about positive societal change. Yet, if not used responsibly, this technology can propel economic inequality, destabilize global markets and affirm systemic bias. While the potential benefits of big data are well-accepted, the importance of using these techniques in a fair and transparent manner is rarely considered.

We will start this tutorial with a brief introduction to foundational concepts of bias, positive and negative discrimination, redlining, and disparate impact. These legal and ethical issues have been attracting attention in the context of big data, and have been receiving coverage in the popular press. We will then identify key properties of responsible data analysis [2], outlined next.

The first property of responsible data analysis is **fairness**, by which we mean *lack of bias*. It is incorrect to assume that insights gained from computation on data are unbiased simply because data was gathered automatically or processing was performed algorithmically. Bias may come from the

data, e.g., if a questionnaire contains biased questions, or from the algorithm, reflecting political, commercial, sexual, religious, or other kinds of preferences of its designers.

The second property is **non-discrimination**. When tackling a technically challenging problem such as relevance ranking of Web search results, or news article recommendation, it is rational to first focus on meeting common needs well. However to afford equal advantage to a wide variety of users, it is important to support uncommon information and data analysis needs. Such tasks are said to be “in the tail” — they may not be common individually, yet together constitute the overwhelming majority. For instance, Lerman [22] argues that the use of big data can lead to data exclusion and therefore poses risks to those it overlooks.

The third property of responsible data analysis is **transparency**. Users want to know and control both what is being recorded about them, and how the recorded information is being used, e.g., to recommend content or target advertisement to them. However, while privacy is certainly an important part of the picture, there is far more to transparency than privacy. Transparent data analysis frameworks will require verification and auditing of datasets and algorithms for fairness, robustness, diversity, non-discrimination and privacy. An important ingredient in transparency is availability of provenance meta-data, which describes who created a dataset and how.

## 2. OVERVIEW OF TECHNICAL WORK

In a paper that pre-dates big data, Friedman and Nissenbaum [15] give a systematic account of bias in computer systems. The authors identify several representative examples of bias, and develop a taxonomy, classifying bias as pre-existing (societal), technical and emergent (based on use).

More recently, two kinds of technical approaches have been developed. The first are empirical studies that serve to underscore the lack of fairness and transparency in current data analysis practices. In the second category are proposals from the data mining and machine learning communities that aim to make some common task unbiased.

The empirical study of current data-intensive applications aims to identify fairness violations in data analysis practices. This work is critical for understanding the current practice and for motivating research into responsible data use. We will give an overview of existing studies, including the XRay project [21] and the study by Datta et al. [7]. Both studies point to the lack of transparency in the way personal data is used for online ad targeting. We will also present a study by Sweeney [25], which identifies cases of racial discrimination

in online advertising.

Recently, work is beginning to emerge in the machine learning and data mining communities that concerns detecting and avoiding discrimination in classification. Fairness in classification is understood in terms of two goals, namely, individual fairness and group fairness. Individual fairness states that two individuals who are similar w.r.t. a particular classification task should be classified similarly, while group fairness states that the proportion of members of a protected group who are classified positively should be statistically indistinguishable from the proportion of members of the overall population.

Dwork et al. [11] propose a framework for fair classification, based on identifying a probabilistic mapping from individuals to an intermediate representation that achieves both individual and group fairness. This framework assumes that a distance function in the space of the classification task is given. In a follow-up work, Zemel et al. [26] propose a method for learning a class of distance functions and formulate fairness as an optimization problem that both encodes the data, preserving necessary attributes, and obfuscates membership in a protected group.

Feldman et al. [14] propose a formalization of the legal doctrine of disparate impact in the context of classification, and study the problems of disparate impact certification and removal, linking disparate impact to a particular loss function, namely, to the balanced error rate.

Beyond classification, Pedreschi et al. [23] and Kamiran et al. [20] propose formalizations of discrimination in association rule mining and decision tree learning, respectively. The authors then develop ways to mediate the effects of discrimination in these settings.

### 3. RESEARCH DIRECTIONS

We will conclude the tutorial by surveying works that, while not specifically motivated by responsible data analysis, can be brought to bear on the problem.

We will mention works seeking to provide accurate data mining results about a population while protecting sensitive information about individuals, e.g., [9, 12]. We will also consider some extensive work on provenance [3, 17], especially in the context of data-intensive workflows [5, 8] and in distributed scenarios [18].

In general, the field of program verification is central to the issue of verifying properties such as fairness or non-discrimination. A broad survey of this field is beyond the scope of the tutorial. We will mention zero-knowledge proofs [6, 16], cryptographic techniques by which one party (the prover) can prove to another party (the verifier) that a given statement is true, without conveying any information apart from the fact that the statement is indeed true.

We will briefly discuss several topics related to supporting diverse preferences and information needs of users. This includes works on search result diversification [4] and rank-aware clustering [24]. We will consider another relevant line of work that concerns modeling, interpreting and aggregating user preferences, e.g., [10, 13, 19]. Finally, we will discuss recent work on personal information management [1], where the goal is to empower users to take control of their own data, so as to manage and disseminate it effectively.

### 4. REFERENCES

- [1] S. Abiteboul et al. Managing your digital life. *Commun. ACM*, 58(5), 2015.
- [2] S. Abiteboul and J. Stoyanovich. Plaidoyer pour une analyse "responsable" des données. *Le Monde*, October 12, 2015.
- [3] P. Agrawal et al. Trio: A system for data, uncertainty, and lineage. In *VLDB*, 2006.
- [4] R. Agrawal et al. Diversifying search results. In *WSDM*, 2009.
- [5] Y. Amsterdamer et al. Putting lipstick on pig: Enabling database-style workflow provenance. *PVLDB*, 5(4), 2011.
- [6] V. Cortier and S. Kremer. Formal models and techniques for analyzing security protocols: A tutorial. *Foundations and Trends in Programming Languages*, 1(3), 2014.
- [7] A. Datta et al. Automated experiments on ad privacy settings: A tale of opacity, choice, and discrimination. *CoRR*, abs/1408.6491, 2014.
- [8] D. Deutch et al. Provenance-based analysis of data-centric processes. *VLDB J.*, 24(4), 2015.
- [9] C. Dwork. A firm foundation for privacy. In *CACM*, volume 54, Jan 2011.
- [10] C. Dwork et al. Rank aggregation methods for the web. In *WWW*, 2001.
- [11] C. Dwork et al. Fairness through awareness. In *Innovations in Theoretical Computer Science*, 2012.
- [12] C. Dwork and A. Roth. *The Algorithmic Foundations of Differential Privacy*. Foundations and Trends in Theoretical Computer Science, 2014.
- [13] R. Fagin et al. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
- [14] M. Feldman et al. Certifying and removing disparate impact. In *ACM SIGKDD*, 2015.
- [15] B. Friedman and H. Nissenbaum. Bias in computer systems. *ACM Trans. Inf. Syst.*, 14(3), 1996.
- [16] S. Goldwasser et al. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1), 1989.
- [17] T. J. Green et al. Provenance semirings. In *PODS*, 2007.
- [18] T. J. Green et al. Provenance in ORCHESTRA. *IEEE Data Eng. Bull.*, 33(3), 2010.
- [19] M. Jacob et al. A system for management and analysis of preference data. *PVLDB*, 7(12), 2014.
- [20] F. Kamiran et al. Discrimination aware decision tree learning. In *ICDM*, 2010.
- [21] M. Lécuyer et al. XRay: Enhancing the web's transparency with differential correlation. In *USENIX*, 2014.
- [22] J. Lerman. Big data and its exclusions. *Stanford Law Review Online*, 66, 2013.
- [23] D. Pedreschi et al. Discrimination-aware data mining. In *ACM SIGKDD*, 2008.
- [24] J. Stoyanovich et al. Making interval-based clustering rank-aware. In *EDBT*, 2011.
- [25] L. Sweeney. Discrimination in online ad delivery. *ACM Queue*, 11(3), 2013.
- [26] R. S. Zemel et al. Learning fair representations. In *ICML*, 2013.

# Core Decomposition in Graphs: Concepts, Algorithms and Applications

Fragkiskos D. Malliaros<sup>1</sup>, Apostolos N. Papadopoulos<sup>2</sup>, Michalis Vazirgiannis<sup>1</sup>

<sup>1</sup>Computer Science Laboratory, École Polytechnique, France

<sup>2</sup>Department of Informatics, Aristotle University of Thessaloniki, Greece

{fmalliaros, mvazirg}@lix.polytechnique.fr, papadopo@csd.auth.gr

## ABSTRACT

Graph mining is an important research area with a plethora of practical applications. Core decomposition in networks, is a fundamental operation strongly related to more complex mining tasks such as community detection, dense subgraph discovery, identification of influential nodes, network visualization, text mining, just to name a few. In this tutorial, we present in detail the concept and properties of core decomposition in graphs, the associated algorithms for its efficient computation and some of its most important applications.

## 1. INTRODUCTION

Core decomposition is a well-studied topic in graph mining. Informally, the  $k$ -core decomposition is a threshold-based hierarchical decomposition of a graph into nested subgraphs. The basic idea is that a threshold  $k$  is set on the degree of each node; nodes that do not satisfy the threshold, are excluded from the process. There exists a rich literature studying algorithmic aspects of core decomposition by taking different viewpoints, such as distributed, streaming, disk-resident data, to name a few. In addition, core decomposition has been used successfully in many diverse application domains, including social networks analysis and text analytics tasks.

Next, we formally define the concept of  $k$ -core decomposition in graphs. Let  $G = (V, E)$  be an undirected graph. Let  $H$  be a subgraph of  $G$ , i.e.,  $H \subseteq G$ . Subgraph  $H$  is defined to be a  $k$ -core of  $G$ , denoted by  $G_k$ , if it is a maximal connected subgraph of  $G$  in which all nodes have degree at least  $k$ . The *degeneracy*  $\delta^*(G)$  of a graph  $G$  is defined as the maximum  $k$  for which graph  $G$  contains a non-empty  $k$ -core subgraph. A node  $i$  has *core number*  $c_i = k$ , if it belongs to a  $k$ -core but not to any  $(k + 1)$ -core. The  $k$ -shell is the subgraph defined by the nodes that belong to the  $k$ -core but not to the  $(k + 1)$ -core.

Based on the above definitions, it is evident that if all the nodes of the graph have degree at least one, i.e.,  $d_v \geq 1, \forall v \in V$ , then the 1-core subgraph corresponds to the whole graph, i.e.,  $G_1 \equiv G$ . Furthermore, assuming that  $G_i, i = 0, 1, 2, \dots, \delta^*(G)$  is the  $i$ -core of  $G$ , then the  $k$ -core subgraphs are nested, i.e.,  $G_0 \supseteq G_1 \supseteq G_2 \supseteq \dots \supseteq G_{\delta^*(G)}$ . Typically, subgraph  $G_{\delta^*(G)}$  is called *maximal  $k$ -core subgraph* of  $G$ .

Figure 1 depicts an example of a graph and the corresponding  $k$ -core decomposition. As we observe, the degeneracy of this graph

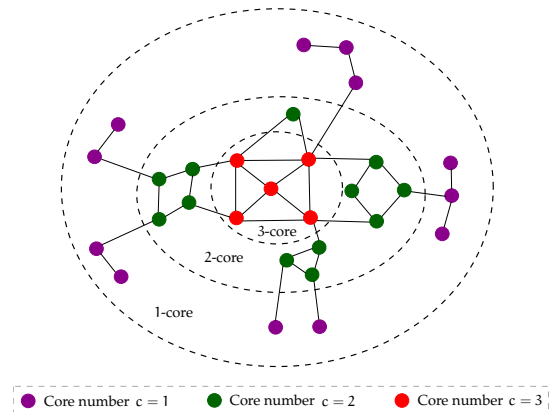


Figure 1: Example of the  $k$ -core decomposition.

is  $\delta^*(G) = 3$ ; thus, the decomposition creates three nested  $k$ -core subgraphs, with the 3-core being the maximal one. The nested structure of the  $k$ -core subgraphs is indicated by the dashed lines. Furthermore, the color on the nodes indicates the core number  $c$  of each node. Lastly, we should note here that the  $k$ -core subgraphs are not necessarily connected.

## 2. GOALS AND OUTLINE

The goal of this tutorial is to present in detail the algorithmic paradigm of core decomposition in graphs. In particular, we will focus on the following points:

- (i) **Fundamental concepts of core decomposition.** We present the notion of  $k$ -core decomposition for unweighted and undirected graphs and then extensions for weighted, directed, probabilistic and signed ones. We also present generalizations of the decomposition to node properties beyond the degree.
- (ii) **Algorithms for core decomposition.** Computing the  $k$ -core decomposition of a graph can be done through a simple process that is based on the following property: to extract the  $k$ -core subgraph, all nodes with degree less than  $k$  and their adjacent edges should be recursively deleted. In the tutorial, we present efficient algorithms for the  $k$ -core decomposition. We also examine several extensions that have been proposed by the databases community for large scale  $k$ -core decomposition under various computation frameworks, including streaming, distributed and disk-based algorithms. We also examine how to estimate the  $k$ -core number of each node using only local information.
- (iii) **Applications.** We demonstrate applications of the  $k$ -core decomposition in various domains, including dense subgraph

detection, graph clustering, modeling of network dynamics and network visualization.

The outline of the tutorial has as follows:

### 1. Introduction

- Social network analysis
- Highlights of core decomposition

### 2. Fundamental Concepts of Core Decomposition

- $k$ -core subgraph,  $k$ -shell subgraph,  $k$ -core number, degeneracy
- Weighted networks, directed networks, signed networks, probabilistic networks
- Generalized cores
- Truss decomposition
- Extensions of the core decomposition

### 3. Algorithms

- Baseline algorithm
- An  $\mathcal{O}(|E|)$  algorithm for  $k$ -core decomposition
- Streaming  $k$ -core decomposition
- Distributed  $k$ -core decomposition
- Disk-based  $k$ -core decomposition
- Local estimation of  $k$ -core numbers

### 4. Applications in Complex Networks

- Dense subgraph discovery
- Community detection and evaluation
- Identification of influential nodes
- Dynamics of networks
- Modeling the Internet topology
- Network visualization
- Text mining

### 5. Open Problems and Future Research

- Algorithms and applications

## 3. ACKNOWLEDGEMENTS

Fragkiskos D. Malliaros is a recipient of the Google Europe Fellowship in Graph Mining, and this research is supported in part by this Google Fellowship.

## 4. REFERENCES

- [1] J. I. Alvarez-Hamelin, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the  $k$ -core decomposition. In *NIPS*, pages 41–50, 2006.
- [2] J. I. Alvarez-Hamelin, L. Dall’Asta, A. Barrat, and A. Vespignani.  $k$ -core decomposition of internet graphs: Hierarchies, self-similarity and measurement biases. *NHM*, 3(2):371, 2008.
- [3] R. Andersen and K. Chellapilla. Finding dense subgraphs with size bounds. In *WAW*, pages 25–37, 2009.
- [4] V. Batagelj and M. Zaversnik. Generalized cores. *CoRR*, 2002.
- [5] V. Batagelj and M. Zaversnik. An  $\mathcal{O}(m)$  algorithm for cores decomposition of networks. *CoRR*, 2003.
- [6] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich. Core decomposition of uncertain graphs. In *KDD*, pages 1316–1325, 2014.
- [7] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir. A model of internet topology using  $k$ -shell decomposition. *PNAS*, 104(27):11150–11154, 2007.
- [8] J. Cheng, Y. Ke, S. Chu, and M. T. Ozsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62, 2011.
- [9] A. Garas, F. Schweitzer, and S. Havlin. A  $k$ -shell decomposition method for weighted networks. *New Journal of Physics*, 14(8), 2012.
- [10] C. Giatsidis, F. D. Malliaros, D. M. Thilikos, and M. Vazirgiannis. CoreCluster: A degeneracy based graph clustering framework. In *AAAI*, pages 44–50, 2014.
- [11] C. Giatsidis, D. Thilikos, and M. Vazirgiannis. Evaluating cooperation in communities with the  $k$ -core structure. In *ASONAM*, pages 87–93, 2011.
- [12] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. D-cores: measuring collaboration of directed graphs based on degeneracy. *Knowl. Inf. Syst.*, 35(2):311–343, 2013.
- [13] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljerosand, L. Muchnik, H. E. Stanley, and H. A. Makse. Identification of influential spreaders in complex networks. *Nature Physics*, 2010.
- [14] R.-H. Li, L. Qin, J. X. Yu, and R. Mao. Influential community search in large networks. *Proc. VLDB Endow.*, 8(5):509–520, 2015.
- [15] F. D. Malliaros, M.-E. G. Rossi, and M. Vazirgiannis. Locating influential nodes in complex networks. *Sci. Rep.*, 2016.
- [16] F. D. Malliaros and M. Vazirgiannis. To stay or not to stay: modeling engagement dynamics in social graphs. In *CIKM*, pages 469–478, 2013.
- [17] F. D. Malliaros and M. Vazirgiannis. Vulnerability assessment in social networks under cascade-based node departures. *EPL (Europhysics Letters)*, 110(6):68006, 2015.
- [18] A. Montesor, F. D. Pellegrini, and D. Miorandi. Distributed  $k$ -core decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 24(2):288–300, 2013.
- [19] M. P. O’Brien and B. D. Sullivan. Locally estimating core numbers. In *ICDM*, pages 460–469, 2014.
- [20] K. Pechlivanidou, D. Katsaros, and L. Tassioulas. MapReduce-based distributed K-shell decomposition for online social networks. In *SERVICES*, pages 30–37, 2014.
- [21] M.-E. G. Rossi, F. D. Malliaros, and M. Vazirgiannis. Spread it good, spread it fast: Identification of influential nodes in social networks. In *WWW*, pages 101–102, 2015.
- [22] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and U. V. Çatalyürek. Streaming algorithms for  $k$ -core decomposition. *Proc. VLDB Endow.*, 6(6):433–444, Apr. 2013.
- [23] A. E. Sariyuce, C. Seshadhri, A. Pinar, and U. V. Catalyurek. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *WWW*, pages 927–937, 2015.
- [24] S. B. Seidman. Network Structure and Minimum Degree. *Social Networks*, 5:269–287, 1983.
- [25] N. Tatti and A. Gionis. Density-friendly graph decomposition. In *WWW*, pages 1089–1099, 2015.
- [26] E. Valari, M. Kontaki, and A. N. Papadopoulos. Discovery of top-k dense subgraphs in dynamic graph collections. In *SSDBM*, pages 213–230, 2012.
- [27] J. Wang and J. Cheng. Truss decomposition in massive networks. *Proc. VLDB Endow.*, 5(9):812–823, 2012.
- [28] G.-Q. Zhang, G.-Q. Zhang, Q.-F. Yang, S.-Q. Cheng, and T. Zhou. Evolution of the Internet and its cores. *New Journal of Physics*, 10(12):123027+, 2008.
- [29] Y. Zhang and S. Parthasarathy. Extracting analyzing and visualizing triangle  $k$ -core motifs within networks. In *ICDE*, pages 1049–1060, 2012.

# Distance-based Multimedia Indexing

Christian Beecks  
Data Management and Data  
Exploration Group  
RWTH Aachen University,  
Germany  
beecks@cs.rwth-  
aachen.de

Merih Seran Uysal  
Data Management and Data  
Exploration Group  
RWTH Aachen University,  
Germany  
uysal@cs.rwth-  
aachen.de

Thomas Seidl  
Database Systems Group  
Ludwig-Maximilians-  
Universität Munich,  
Germany  
seidl@dbs.ifi.lmu.de

## ABSTRACT

This tutorial aims at providing an overview of the state-of-the-art approaches to distance-based multimedia indexing. This tutorial presents the fundamentals of (i) object representation, (ii) distance-based similarity models, (iii) efficient query processing, and (iv) indexing. It is intended for a broad target audience starting from beginners to experts in the domain of distance-based similarity search in multimedia databases and adjacent research fields which utilize distance-based approaches.

## 1. INTRODUCTION

Concomitant with the explosive growth of the digital universe [11], an immensely increasing amount of multimedia data is generated, processed, and finally stored in very large multimedia databases. The rapid expansion of the internet and the extensive spread of mobile devices allow users to generate and share multimedia data everywhere and at any time. As a result, multimedia databases tend to grow continuously without any restriction and are thus no longer manually manageable by humans. Automatic approaches that allow for effective and efficient information access to massive multimedia databases become immensely important.

*Multimedia retrieval approaches* are one class of information access approaches that allow to manage and access multimedia databases with respect to the users' information needs. These approaches deal with the representation, storage, organization of, and access to information items [2]. In fact, they can be thought of approaches allowing users to *search*, *browse*, *explore*, and *analyze* multimedia databases by means of similarity relations among multimedia objects.

One promising and widespread approach to define similarity between multimedia objects consists in automatically extracting inherent properties of multimedia objects and comparing them with each other. For this purpose, the content-based properties of multimedia objects are modeled by feature representations which are comparable by means of *distance-based similarity measures*. This class of similarity

measures follows a rigorous mathematical interpretation [19] and allows domain experts and database experts to address the issues of effectiveness and efficiency simultaneously and independently. In fact, it has become mandatory for current distance-based similarity measures to be indexable in order to facilitate large-scale applicability.

## 2. TUTORIAL OUTLINE

In this tutorial, we aim at providing an overview of the state-of-the-art approaches to distance-based multimedia indexing. We intend to cover a broad target audience starting from beginners to experts in the domain of distance-based similarity search in multimedia databases and adjacent research fields which utilize distance-based approaches. No prerequisite knowledge is needed.

### 2.1 Object Representation

In the first part of this tutorial, we will outline different approaches to object representations in order to answer the question of how to model multimedia data objects in a generic way. We will focus on a unified object representation model including fixed-binning feature histograms and adaptive-binning feature signatures [3]. In addition to these object representations, we will present the idea of probabilistic feature signatures [4, 5] and show how to approximate object representations by means of gradient-based signatures [7].

### 2.2 Distance-based Similarity Models

In the second part of this tutorial, we will provide an overview of state-of-the-art distance-based similarity measures for feature histograms and feature signatures in order to complete our understanding of a similarity model. Among the multitude of distance-based similarity measures applicable to feature signatures, we will present and discuss the Earth Mover's Distance [15], the Signature Quadratic Form Distance [8], and the Signature Matching Distance [6].

### 2.3 Efficient Query Processing

The third part of this tutorial is devoted to techniques and algorithms for efficient query processing. After introducing distance-based similarity queries, we show how to process such queries efficiently by means of multi-step filter-and-refinement algorithms including multi-step range query algorithms [10] and optimal multi-step  $k$ -NN query algorithms [18]. To this end, we elucidate the idea of lower bound approximations and present state-of-the-art lower bound approximations [1, 20, 21] for the Earth Mover's Distance.

## 2.4 Indexing

The last part of this tutorial finally covers indexing approaches for distance-based similarity models where we will give an insight into the fundamentals of spatial access methods [16, 22], metric access methods [9, 14, 17, 23], and ptolemaic access methods [12, 13].

## 3. FURTHER INFORMATION

Further information regarding this tutorial can be found at <http://dme.rwth-aachen.de/en/DBMI>.

## 4. ACKNOWLEDGMENTS

This work is funded by DFG grant SE 1039/7-1. It is partly based on the work of Beecks [3].

## 5. REFERENCES

- [1] I. Assent, A. Wenning, and T. Seidl. Approximation techniques for indexing the earth mover's distance in multimedia databases. In *ICDE*, pages 11:1–12, 2006.
- [2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley Professional, 2011.
- [3] C. Beecks. *Distance-based similarity models for content-based multimedia retrieval*. PhD thesis, RWTH Aachen University, 2013.
- [4] C. Beecks, A. M. Ivanescu, S. Kirchhoff, and T. Seidl. Modeling image similarity by gaussian mixture models and the signature quadratic form distance. In *ICCV*, pages 1754–1761, 2011.
- [5] C. Beecks, A. M. Ivanescu, S. Kirchhoff, and T. Seidl. Modeling multimedia contents through probabilistic feature signatures. In *ACM Multimedia*, pages 1433–1436, 2011.
- [6] C. Beecks, S. Kirchhoff, and T. Seidl. Signature matching distance for content-based image retrieval. In *ICMR*, pages 41–48, 2013.
- [7] C. Beecks, M. S. Uysal, J. Hermanns, and T. Seidl. Gradient-based signatures for efficient similarity search in large-scale multimedia databases. In *CIKM*, pages 1241–1250, 2015.
- [8] C. Beecks, M. S. Uysal, and T. Seidl. Signature quadratic form distance. In *CIVR*, pages 438–445, 2010.
- [9] E. Chávez, G. Navarro, R. A. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
- [10] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, 1994.
- [11] J. F. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva. The diverse and exploding digital universe. *IDC White Paper*, 2, 2008.
- [12] M. L. Hetland. Ptolemaic indexing. *JoCG*, 6(1):165–184, 2015.
- [13] M. L. Hetland, T. Skopal, J. Lokoč, and C. Beecks. Ptolemaic access methods: Challenging the reign of the metric space model. *Information Systems*, 38(7):989 – 1006, 2013.
- [14] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580, 2003.
- [15] Y. Rubner, C. Tomasi, and L. J. Guibas. The earth mover's distance as a metric for image retrieval. *IJCV*, 40(2):99–121, 2000.
- [16] H. Samet. *The design and analysis of spatial data structures*. Addison-Wesley Reading, MA, 1990.
- [17] H. Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.
- [18] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, pages 154–165, 1998.
- [19] R. N. Shepard. Stimulus and response generalization: A stochastic model relating generalization to distance in psychological space. *Psychometrika*, 22:325–345, 1957.
- [20] M. S. Uysal, C. Beecks, D. Sabinasz, J. Schmücking, and T. Seidl. Efficient query processing using the earth mover's distance in video databases. In *EDBT*, 2016.
- [21] M. S. Uysal, C. Beecks, J. Schmücking, and T. Seidl. Efficient filter approximation using the earth mover's distance in very large multimedia databases with feature signatures. In *CIKM*, pages 979–988, 2014.
- [22] P. van Oosterom. Spatial access methods. *Geographical information systems*, 1:385–400, 1999.
- [23] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search - The Metric Space Approach*. Advances in Database Systems. Kluwer, 2006.